An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution

Miryung Kim
The University of Texas at Austin
Austin TX
miryung@ece.utexas.edu

Dongxiang Cai,* Sunghun Kim
Hong Kong University of Science and Technology, Hong Kong, China
{caidx, hunkim}@cse.ust.hk

ABSTRACT

It is widely believed that refactoring improves software quality and programmer productivity by making it easier to maintain and understand software systems. However, the role of refactorings has not been systematically investigated using fine-grained evolution history. We quantitatively and qualitatively studied API-level refactorings and bug fixes in three large open source projects, totaling 26523 revisions of evolution.

The study found several surprising results: One, there is an increase in the number of bug fixes after API-level refactorings. Two, the time taken to fix bugs is shorter after API-level refactorings than before. Three, a large number of refactoring revisions include bug fixes at the same time or are related to later bug fix revisions. Four, API-level refactorings occur more frequently before than after major software releases. These results call for re-thinking refactoring's true benefits. Furthermore, frequent *floss refactoring* mistakes observed in this study call for new software engineering tools to support safe application of refactoring and behavior modifying edits together.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring*

General Terms

Measurement, Experimentation

Keywords

Software evolution, empirical study, refactoring, defects, release cycle

*A part of the research was conducted while Dongxiang Cai was an intern at the University of Texas at Austin.

ICSE'11 May 21-28, Waikiki, Honolulu, HI, USA

1. INTRODUCTION

Refactoring is the process of changing a program's design structure without modifying its external functional behavior in order to improve program readability, maintainability, and extensibility [19].

It has been widely believed that refactoring improves software quality and developer productivity by making it easier to maintain and understand software systems [11] and that a lack of refactorings incurs technical debt to be repaid in the future to reduce increasing system complexity [18]. There has been a conventional wisdom that software engineers often avoid refactorings when they are constrained by a lack of resources (e.g., right before major software releases), as refactorings do not provide immediate benefit unlike new features or bug fixes. Some questioned the benefit of refactorings, since refactorings often introduce a large amount of structural changes to the system, creating code churns shown to be correlated with defect density [22]. Weißgerber and Diehl found that a high ratio of refactorings is sometimes followed by an increasing ratio of bug reports [30]. They found that bugs are introduced by incomplete or incorrectly done refactorings [12], even though the original intent of refactoring was to improve software maintainability. Ratzinger et al. [25] found contradicting evidence that the number of defects decreases, if the number of refactorings increases in the preceding time period.

The goal of this paper is to systematically investigate the role of refactorings during software evolution by examining the relationships between refactorings, bug fixes, the time to resolve bugs, and release cycles using fine-grained version histories. First, we applied M. Kim et al.'s refactoring reconstruction technique to version histories to find revisions that underwent *rename*, *move*, and *signature changes* at or above the level of method headers [16]. We used this technique because the documentation about past refactorings is often unavailable in most version histories. Second, we applied S. Kim et al.'s bug history extraction technique to identify bug fix revisions [17]. To mitigate construct validity concerns, we sampled a total of one hundred revisions and manually investigated their commit messages, associated bug reports, and corresponding code changes to measure the accuracy of the tools.

We then investigated the number of bug fixes and the time taken to fix bugs within a sliding window of K revisions before and after each refactoring revision and measured refactoring density and fix density within a sliding window of K revisions before and after each major release.

The following paragraphs summarizes our findings for each

hypothesis about API-level refactorings—move, rename, and signature changes at the method declaration level.

- **H1: Are there more bug fixes after API-level refactorings?** We found that a bug fix rate is higher after API-level refactorings than the preceding period, e.g., from 26.1% to 30.3% when examining 5 revisions before and after API-level refactorings in Eclipse JDT. We manually investigated bug fixes that follow after refactorings and found that a fix rate increase is often caused by mistakes in applying refactorings and behavior modifying edits together.

- **H2: Do API-level refactorings improve developer productivity?** We compared the time taken to fix bugs for those that were closed within 100 revisions before and after refactorings. The time taken to fix those bugs decreases by about 35.0% to 63.1% after refactorings.

- **H3: Do API-level refactorings facilitate bug fixes?** We found that 29.1% to 44.4% of refactoring revisions also include bug fixes in the same revision. Furthermore, 32% of refactoring revisions were related to fix revisions that follow within 20 revisions, as opposed to 14% of non-refactoring revisions being related to later bug fixes within 20 revisions. This implies that, in many cases, either refactorings created new bugs or refactorings may have been applied to facilitate bug fixes that were hard to implement without them.

- **H4: Are there relatively fewer API-level refactorings before major releases?** There are 39.1% more refactoring revisions within 30 revisions before major releases than 30 revisions after the releases.

  This result is surprising because developers do not avoid refactorings even when they have a pressure to meet deadlines. In conjunction with the fact that many revisions include both refactorings and bug fixes, we speculate that refactorings were done to facilitate bug fixes that needed to be implemented before major releases.

These results call for an in-depth investigation of refactoring's true benefits and the economic implications of refactoring investment. Furthermore, frequent *floss refactoring* mistakes observed in the study call for new software engineering tools to help developers apply systematic, coordinated refactorings safely.[1]

The rest of the paper is organized as follows. Section 2 discusses related work and Section 3 describes our data collection and analysis method. Section 4 presents our results, Section 5 describes threats to validity, and Section 6 presents future directions and summarizes the implication of our results.

2. RELATED WORK

Empirical studies on refactoring. Xing and Stroulia studied Eclipse's evolution history and found that 70% of structural changes are due to refactorings and existing IDEs lack support for complex refactorings [33]. Dig et al. studied the role of refactorings in API evolution and found that 80% of the changes that break client applications are API-level refactorings [7]. While these studies focus on the frequency and types of refactorings, our study focuses on the relationship between API-level refactorings, bug fixes, and release cycles.

Hindle et al. found that large commits are more perfective (refactorings) while small commits are more corrective (bug fixes) [14]. Purushothaman and Perry found that nearly 10% of changes involved only a single line of code, which has less than a 4% chance to result in error, while a change of 500 lines or more has nearly a 50% chance of causing at least one defect. Though the focus of these studies is different from our study, the results are somewhat aligned with ours: large commits, which tend to include refactorings, have a higher chance of inducing bugs.

Weißgerber and Diehl found that refactorings often occur together with other types of changes and that refactorings are followed by an increasing number of bugs [30]. Carriere et al.'s case study found that the productivity measure manifested by the average time taken to resolve tickets decreases after re-architecting the system [3]. Ratzinger et al. developed defect prediction models based on software evolution attributes and found that refactoring related features and defects have an inverse correlation [25]—if the number of refactorings increases in the preceding time period, the number of defects decreases. Our results are aligned with some of these findings, yet improve upon these studies. Our study method not only relates refactorings and bug fixes based on their temporal proximity using a K-revision sliding window but also considers method-level location of refactorings and bug fixes to examine whether bug fixes are related to a preceding refactoring.

Though the intent of refactoring is to improve software maintainability, refactoring could be potentially error-prone as it often requires coordinated edits across different parts of a system. Several researchers found such evidence from open source project histories—M. Kim et.al.'s program differencing technique [15, 16] identifies exceptions to systematic change patterns, which often arise from the failure to complete coordinated refactorings. Görg and Weißgerber detect errors caused by incomplete refactorings by relating API-level refactorings to the corresponding class hierarchy [30].

Because manual refactoring is often tedious and error-prone, modern IDEs provide features that automate the application of refactorings [13, 27]. However, recent research found several limitations of tool-assisted refactorings. Daniel et al. found dozens of bugs in the refactoring tools in popular IDEs [6]. Murphy-Hill et al. found that many refactoring tools do a poor job of communicating errors and programmers do not leverage them as effectively as they could [20]. They also found that programmers frequently intersperse refactorings with other program changes—*floss* refactorings and these are not well supported by existing refactoring tools [21]. This need for safe floss refactoring application is also confirmed by our study—refactoring often overlap with bug fixes, *behavior correcting* transformations. *Program metamorphosis* relaxes behavior-preservation checks to safely support floss refactorings [26].

Refactoring reconstruction. A number of existing tech-

[1] *Floss refactoring* is a term coined by Murphy-Hill et al. to describe refactorings interleaved with behavior modifying edits [21].

Table 1: Study subjects

	Eclipse JDT core	jEdit	Columba
type	IDE	text editor	email client
period	June 5, 2001 - May 16, 2007	Sep 2, 2001 - Apr 2, 2009	Jul 9, 2006 - Jan 31, 2010
# revisions	15000	11102	421
# of API-level refactorings	6755	3557	424
# of bug fix revisions	3752	1073	150
# of refactoring revisions	1089	423	36

niques address the problem of automatically inferring refactorings from two program versions. These techniques compare code elements in terms of their name [23,32] and structure similarity [8,23,31] to identify move and rename refactorings. M. Kim's technique used in the study is broadly in the same category [16]; its median precision and recall are in the ranges 93% to 98% and 93% to 99%, and the comparison with five other approaches shows that its recall is higher while its precision is comparable to others. A survey of existing refactoring reconstruction techniques is described elsewhere [24].

Bug history extraction and analysis. There are two well-known techniques to extract bug fix data from version control systems. Fischer et al. search for specific keywords such as `bug` or `fix` in revision logs to identify bug fix revisions [9]. Another well-known technique is to use the links between commits and bug reports [28,29], because developers often leave a corresponding bug report id when resolving a bug. Since leaving special keywords or bug ids in change logs is optional for developers, fix revision data could include noise. For example, Aranda and Venolia et al. manually inspected ten bug reports and interviewed developers who resolved them [1]. They found important information about bug fix process is often missing in software repositories. Bird et al. studied the quality of change logs and bug reports and found that it is difficult to reliably link all commits and bug reports [2].

However, the quality of automatically collected fix revision data depends on projects and their change log quality. In this paper, we selected projects with high quality change logs based on our experience of mining bug repository data.

3. STUDY APPROACH

We selected Eclipse JDT, jEdit, and Columba as study subjects because we studied these subjects in the past and found that they have high quality change logs [16,17]. This enabled us to extract fix revision data reliably.

Identification of refactoring revisions. We used M. Kim et al.'s refactoring reconstruction technique (MK) to identify systematic changes to API names and signatures at or above the level of method-headers [16]. MK takes two program versions as input, first identifies seed matches based on the method-header name similarity and generalizes the transformation witnessed in a seed match to a high-level systematic change pattern. MK identifies *rename* refactoring at the level of packages/classes/methods, *add/delete parameter* refactoring, *move* refactoring at the level of packages/classes/methods and changes to the return type of a method. It does not analyze the internal content of method bodies, and thus is limited to API-level refactorings. The details of the algorithm and evaluation are described elsewhere [16].

Using this technique, we found 6755, 3557, and 424 refactorings in Eclipse, jEdit, and Columba respectively (see Table 1). We considered that a revision is a refactoring revision if the revision contains at least one API-level refactoring.

Identification of bug fix revisions. We identified fix revisions by mining check-in comments (change logs), which is a widely used heuristic in mining software repository research. We first searched for keywords such as `fixed` or `bug` [17] and identified a reference to a bug report number such as 'id #42233' [28, 29]. This heuristic woks well with projects which have high quality change logs such as Columba, Eclipse, and JEdit. For example, Columba developers usually write at least one predefined tag, e.g., `[bug]`, `[feature]`, and `[ui]` in change logs. Eclipse and jEdit have strict guidelines for writing check-in messages. If a commit message contains such fix revision indicators, we considered it as a fix revision.

Identification of bug-introducing changes. When a revision is determined to contain bug fixes, it is possible to trace backward in the revision history to determine the corresponding bug-introducing change [28]. We first use `diff` to determine what changed in each fix revision. `Diff` returns a list of consecutive deleted or added lines, called *hunks*. Using a built-in annotation functionality of a version control system such as *SVN blame*, we track down the origin of deleted or modified source lines in each hunk, which we call as bug-introducing changes.

Change distilling. For each revision, we used Change Distiller to compute syntactic program differences [10]. This has two benefits: (1) This distilling process filters out meaningless changes in a revision because the revision may contain changes to comments, license information, white spaces, etc. (2) Change Distiller maps the line-level location of a bug fix to its container method to allow easy comparison with an API-level refactoring location, e.g., method `foo` was renamed to method `bar`.

Manual inspection of automatically collected data. Since our study approach relies on automatic refactoring and fix identification techniques, it is important to ensure the automatic techniques are accurate enough for our study. To evaluate the techniques, we randomly sampled 100 revisions from Eclipse JDT core's revision 1000 to 15000. For each sampled revision, one of the authors manually investigated the revision and determined whether it is a refactoring revision and/or a fix revision based on three sources: (1) check-in comment, (2) an associated bug report linked by a bug id, and (3) code modification, i.e., *diff* associated with the revision. Based on this investigation, we manually identified 14 API-level refactoring revisions and 62 fix revisions out of 100 revisions (see column Δ in Table 2). We then compared this set with the automated tools' results. The evaluation shows that the MK has 0.93 precision and 0.93 recall (row MK [16]) in identifying refactoring revisions. The bug

Table 2: Accuracy comparison: automated techniques vs. manual inspection

	Δ	source	#	prec.	recall
refactoring revisions	14	MK [16]	14	0.93	0.93
		manual-C	2	0.50	0.07
		manual-B	5	0.75	0.21
fix revisions	62	SK [17]	49	0.96	0.76
		manual-C	55	0.92	0.81
		manual-B	49	0.94	0.74

fix revision identification technique has 0.96 precision and 0.76 recall (row SK [17]) in identifying bug fix revisions. We also compared the tools' results with the data set labeled using an associated check-in comment alone (row manual-C) or using a bug report alone (row manual-B). Though the automated techniques' accuracy (0.93 and 0.96) is not as good as manually inspecting all three information sources including code modification, their accuracy are much higher than manually inspecting check-in comments or bug-report description alone. Overall, we conclude that our fix revision data is accurate enough to base our investigation on, and our refactoring revision data has a high precision and a recall about API-level refactorings.

4. RESULTS

Section 4.1 investigates changes to the number of bug fixes after refactorings, and Section 4.2 investigates changes to the time taken to resolve bugs after refactorings. Section 4.3 investigates the probability of refactorings and bug fixes to appear in the same revision and the probability of a refactoring revision to be related to later bug fixes, and Section 4.4 investigates fix density and refactoring density with respect to software release cycles.

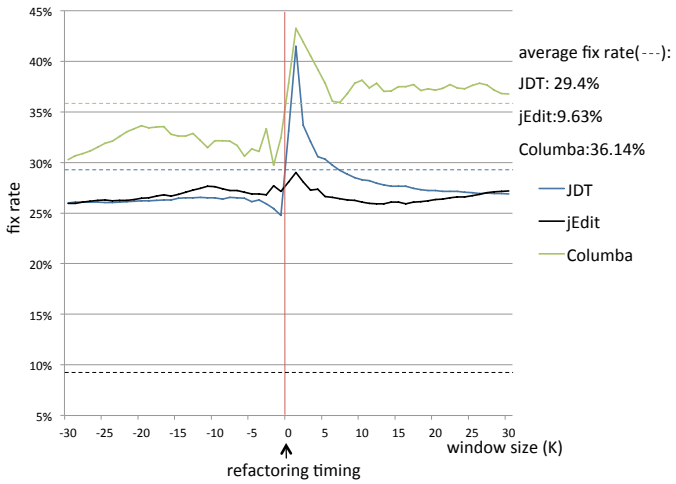

Figure 1: Fix rate before and after refactorings (varying K from 1 to 30)

4.1 Are there more bug fixes after API-level refactorings?

To understand the relationship between refactorings and the number of bug fixes, we find all fix revisions within K revisions before and after for each refactoring revision. Then we compute a *fix rate*, $\frac{|fix\ revisions|}{K}$. Figure 1 shows average fix rates of K-revisions before and after refactorings by varying K from 1 to 30. Horizontal dotted lines show the average bug fix rate for each subject over its entire life time we observed. In all three subjects, the fix rate increases after refactorings. For example, the fix rate of 5 revisions before refactoring (noted as −5 revision in Figure 1) is around 26.1%, while the fix rate of 5 revisions after refactorings is 30.3% for Eclipse JDT. For Columba, the fix rate of the −5 revision is around 31.4%, and the +5 revision is around 37.8%.

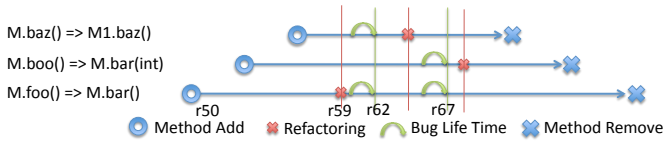

Figure 2: An example method evolution history

We also investigated a fix rate at the method granularity. This technique computes fix rates more accurately by considering refactorings and bug fixes per method. For each method, we reconstructed its revision level history which includes information about when method-level rename and move refactorings occurred and in which revision bug fixes were applied to the method. Figure 2 shows an example change history: method `M.foo()` was created in revision 50, it was renamed to method `M.bar()` in the revision 59, and a bug fix was applied to it at revision 62 and 67. Using the same sliding window method, we computed the fix rate for each method, by varying the window size K from 1 to 30. Figure 3 describes the results at the method granularity. The fix rate at the method level is much lower than the one at the revision level, since most fixes did not happen in the same method location of a refactoring. Its trend is almost the same as Figure 1.

The method-level fix rate increases from 0.101% to 0.162% in Eclipse JDT, from 0.367% to 0.675% in jEdit, and from 0% to 1.454% in Columba within 5 revisions after API-level refactorings compared to the preceding period. To show the statistical significance of these rate differences, the t-test is used [5]. Our null hypothesis is the bug fix rates before and after API-level refactorings are the same at the method level. We reject the null hypothesis and accept the alternative hypothesis if the p-value is smaller than 0.05. The changes in fix rates are statistically significant with a p-value of 8.70e-09 in Eclipse JDT, 0.011 in jEdit, and 1.12e-05 in Columba.

Both at the revision and method level, we observe that the fix rate after refactoring is higher than the fix rate before refactoring. There are several possible explanations. Refactorings may introduce bugs, and developers fix those new bugs after the refactorings. Or refactorings help developers identify and fix previously introduced latent bugs.

In order to investigate the fix rate increase after refactorings both at the revision and at the method level, we conducted an in-depth case study of the refactoring revisions in Eclipse JDT followed by at least one bug fix to the same method location within 20 revisions after the initial refactorings. This process found 208 revisions in Eclipse JDT, out

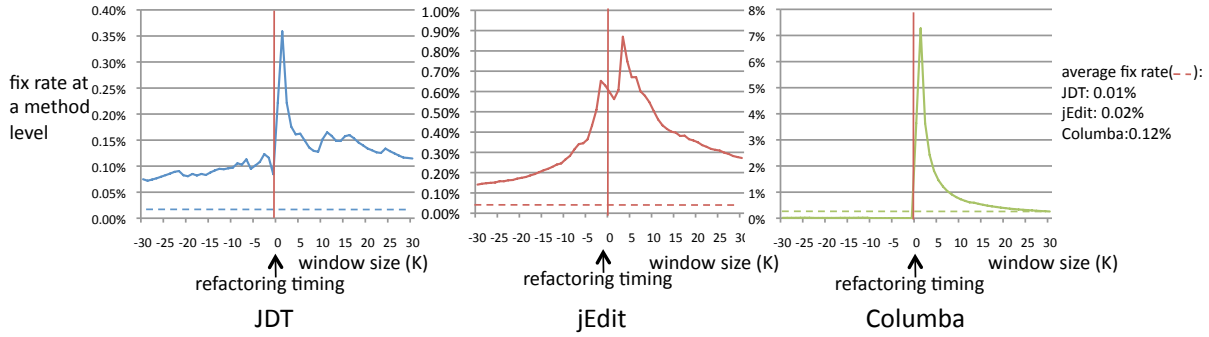

Figure 3: Fix rate before and after refactorings at the method level (varying K from 1 to 30)

of which we randomly selected 50 revisions for manual investigation. We inspected the refactoring and associated fix revisions' check-in comments, associated bug reports, and code modifications. The following five categories emerged from the investigation results.

I. Floss refactoring—refactoring combined with a bug fix—is incomplete, thus inducing a supplementary fix later. For example, Olivier Thomann tried to fix the bug id 111494 at revision 12200. Later he fixed the same bug again at revision 12201 because the original bug fix was incorrect.

II. In order to fix several related bugs that are very hard to fix, a developer first refactors code to enable bug fixes. For example, Jerome Lanneluc decided to fix bug ids 73669 and 73675, which are all related to the AST functionality. He first refactored code at revision 9445, and then bug 73669 and bug 73675 are fixed at revision 9454.

III. Incorrect refactorings cause a bug, which induces a later fix. For example, Olivier Thomann renamed method `unnecessaryNONNLSTags()` to `unnecessaryNLSTags()` but forgot to rename the related XML tag at revision # 11849. He later renamed the related tag at revision 11859.

IV. After a developer performs refactorings, she discovers a new bug or a design problem, which induces a later fix. For example, Jerome Lanneluc tried to speed up a path look-up feature by hashing on container paths at revision 11004. As he realized a design problem in the code, he reported this issue as a bug report id 90431 and fixed it at revision 11019.

V. A bug fix happened after a refactoring in the same location but they are not related. For example, David Audel fixed bug id 106450 with a refactoring at revision 13428, and then at revision 13440 Philippe Mulet fixed another bug (id 104293), which was introduced before revision 13428.

Table 3 shows the categorization of inspected 50 samples, and lists (refactoring revision number, fix revision number) pairs per category. A pair followed by * means that the two revisions are completed by the same author. In 32 cases out of 50, the refactorings are *indeed* related to bug fixes. In 25 cases (50% of inspected samples), later bug fixes are caused by floss refactoring mistakes during a bug fix. This implies that though developers apply refactorings to facilitate bug fixes, such refactorings often cause bugs as well. This result calls for new software engineering tools that repair refactoring mistakes and support developers in applying coordinated refactorings consistently.

Table 3: Categorization of 50 inspected sample pairs of (refactoring revision, fix revision). The * mark indicates that the two revisions are completed by the same author.

	#	(refactoring revision, fix revision)
I	25	(12200,12201)*, (4316,4333)*, (4650,4657)*, (10391,10397)*, (14082,14097)*, (9970,9972)*, (9967,9968)*, (4333,4342)*, (1728,1734)*, (14872,14877)*, (14680,14684)*, (14388,14392)*, (10853,10858)*, (14196,14201)*, (10512,10516)*, (12170,12171)*, (11964,11966)*, (10180,10198)*, (11232,11238)*, (10613,10614)*, (11041,11046)*, (12519,12527)*, (10816,10820)*, (12469,12473)*, (13199,13207)*.
II	3	(9445,9454)*, (1100,1101)*, (12331,12338)*
III	3	(6115,6116), (2383,2402)*, (11849,11859)*.
IV	1	(11004,11019)*.
V	18	(13428,13440), (3931,3947)*, (3678,3689)*, (8955,8965), (7828,7831), (9605,9613), (9898,9913), (6404,6415), (13492,13499)*, (10240,10244)*, (12170,12187)*, (12394,12409)*, (12316,12331)*, (12003,12009)*, (11178,11191)*, (10442,10453)*, (10819,10826)*, (12805,12815).

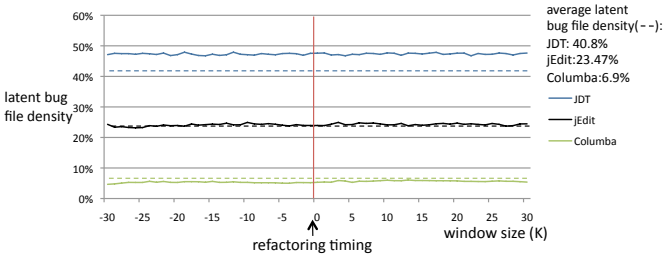

Figure 4: Latent bug density before and after refactorings (K from 1 to 30)

Furthermore, to investigate whether refactorings increase the number of latent bugs, we calculated the number of files

with bug-introducing changes [28]. This technique traces the line-level locations of bug fixes to previous revisions per file to identify which revision created the buggy code that was later modified by the fix. We then computed *latent bug density*, the number of files with bug-introducing changes out of the total number of files at that revision. Figure 4 shows the latent bug density computed by the same sliding window method, varying K from 1 to 30. For all three subjects, the bug density remains stable without much change after refactorings. This implies that while some refactorings help developers identify and fix bugs, some also introduce new bugs.

There is a short-term increase in the number of bug fixes after refactorings.

4.2 Do API-level refactorings reduce the time taken to fix bugs?

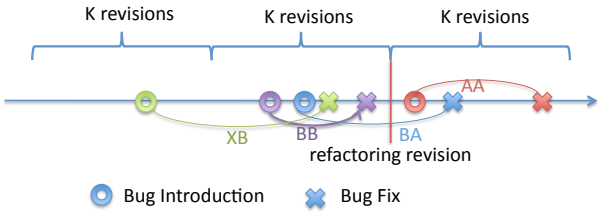

Figure 5: Four categories of bug fixes that were introduced and resolved near the timing of a refactoring

To examine whether refactorings improve developer productivity, we first identified bugs that were both *introduced* and *resolved* near the timing of refactorings and estimated the time taken to fix those bugs by measuring the timing of a bug fix minus the timing of a corresponding bug-introducing change. To compare the difference in productivity before and after refactorings, for each refactoring revision, r, we further categorized nearby bug fixes (closed within [-K, K]) into four categories based on their introduction and resolution time in relation to the timing of refactorings (see Figure 5):

- **BB:** bugs that were introduced *before* r and fixed *before* r. (i.e., open ∈ [-K, 0) and closed ∈ [-K, 0))
- **AA:** bugs that were introduced *after* r and fixed *after* r. (i.e., open ∈ (0, K] and closed ∈ (0, K])
- **BA:** bugs that were introduced *before* r and fixed *after* r. (i.e., open ∈ [-K, 0) and close ∈ (0, K])
- **XB:** bugs that were introduced at least K revisions *before* r and fixed *before* r. (i.e., open ∈ [-2K, -K) and closed ∈ (-K, 0]). Category XB was added for comparison with category BA because the maximum bug life time in category BA is 2K while BB is only K.

Figure 6 shows an average time taken to fix bugs in each category when using K=100. When comparing XB with BA, there is 63.1% decrease in JDT, 56.7% decrease in jEdit and 35.0% decrease in Columba in the time taken to fix bugs after refactorings. When comparing BB and AA, there is 20.2% and 43.1% decrease in JDT and jEdit respectively, but a 26.1% increase (from 38.45 to 48.47) in Columba.

Overall, we observed the bug fix time decrease after refactorings. Based on the results in Section 4.1, we speculate that developers discover bugs during refactorings and quickly fix them as they are already making changes to the involved code or they quickly fix incomplete refactorings as they recognize them. Another explanation is that refactorings combined with bug fixes often incur supplementary fixes, which are usually smaller and easier to implement than main bug fixes.

When it comes to fixing bugs introduced near the time of refactorings, the average fix time tends to decrease after refactorings.

4.3 Do API-level refactorings facilitate bug fixes?

The results in Sections 4.1 and 4.2 motivated us to further investigate whether refactorings were done to facilitate bug fixes.

To examine how many refactorings were done as a part of a bug fix, we first measured the extent of revisions that include both refactorings and bug fixes. In Table 4, $P(F)$ is the probability of a revision to include a bug fix, $P(R)$ is the probability of a revision to include an API-level refactoring, $P(R|F)$ is the conditional probability of including a refactoring given a bug fix revision. $P(R|\neg F)$ is the conditional probability of including a refactoring given that it is a not a fix revision. We used the entire population of our refactoring revision and fix revision data to measure the proportions.

In Eclipse JDT, we found that more than 41.5% refactoring revisions were associated with bug fixes (29.1% in jEdit and 44.4% in Columba). Furthermore, the probability to include a refactoring given a fix revision is much higher than the probability to include a refactoring given a non-fix revision (12.0% vs. 5.7% in Eclipse, 11.5% vs. 3.0% in jEdit, and 10.7% vs. 7.4% in Columba). The probability to include a fix given a refactoring revision is 62.8% while the probability to include a fix given a non-refactoring revision 51.7% in Eclipse. Interestingly, jEdit and Columba show the opposite trends: 33.0% vs. 46.9% in jEdit, and 24.3% vs. 36.3% in Columba.

We also conducted a similar experiment at the method level because a fix revision may include more than one delta where refactorings were applied to only a subset of the deltas. The results at the method-level are slightly different from the revision level analysis, because refactoring tends to involve decentralized, crosscutting changes to more than one methods, making the probability of a method to include a refactoring much higher than the probability of a revision to include a refactoring. To examine whether bug fixes resolved after refactorings indeed are *related* to the preceding refactorings, we measured the percentage of refactoring revisions that have at least one bug fix applied to the same method-level refactoring location within 20 revisions.

As a control group, we measured the percentage of *non-refactoring* revisions that have at least one bug fix applied to the same change location within 20 revisions of the change.

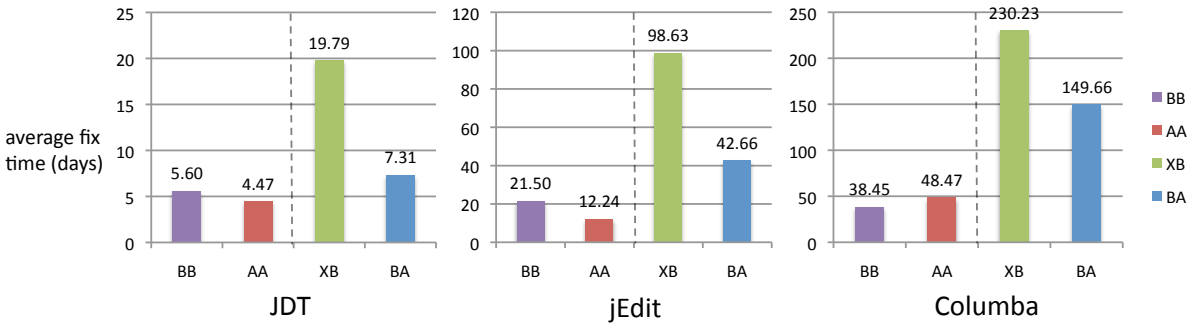

Figure 6: An average time taken to fix bugs in each category

Table 4: Probability of fixes and refactorings at the revision level and at the method level

At the revision level										
Project	**# revisions**	$\|F\|$	$\|R\|$	$\|F \cap R\|$	$P(F)$	$P(R)$	$P(R\|F)$	$P(R\|\neg F)$	$P(F\|R)$	$P(F\|\neg R)$
Eclipse JDT	15,000	3752	1089	452	0.250	0.073	0.120	0.057	0.415	0.237
JEdit	11,102	1073	423	123	0.097	0.038	0.115	0.030	0.291	0.089
Columba	421	150	36	16	0.357	0.086	0.107	0.074	0.444	0.348
Total	26,523	4975	1548	591	0.188	0.058	0.119	0.044	0.382	0.176
At the method level										
Project	**# methods**	$\|F\|$	$\|R\|$	$\|F \cap R\|$	$P(F)$	$P(R)$	$P(R\|F)$	$P(R\|\neg F)$	$P(F\|R)$	$P(F\|\neg R)$
Eclipse JDT	21,938	12049	6278	3945	0.549	0.286	0.327	0.236	0.628	0.517
JEdit	8,938	3704	3529	1166	0.414	0.395	0.315	0.451	0.330	0.469
Columba	2,191	745	424	103	0.340	0.194	0.138	0.222	0.243	0.363
Total	33,067	16498	10231	5214	0.499	0.309	0.316	0.303	0.510	0.494

In Figure 7, the left hand side is for a treatment group, and the right hand side for a control group.

While 32.0% of refactoring revisions have at least one fix revision ∈ (0,20] applied to the same method locations, only 14.0% of non-refactoring revisions have at least one fix revision ∈ (0,20] overlap with the same change locations. When K is 30, the results are 36.5% vs. 15.6% and when K is 100, the results are 49.5% vs. 22.1%.

This result implies that it is more likely for refactoring revisions to be followed by related bug fixes than non-refactoring revisions to be followed by related bug fixes. In conjunction with the manual investigation results in Section 4.1, we find that after a refactoring, usually *the same developer* applies related bug fixes. We speculate that it is because developers apply refactorings first to fix several hard-to-fix bugs *on purpose* or apply supplementary-fixes later to correct *accidental* refactoring mistakes.

Fixes and refactorings often appear in the same revision. Furthermore, it is more likely for a refactoring revision to be followed by related bug fixes than for a non-refactoring revision.

4.4 Are there relatively fewer API-level refactorings before release dates?

Some practitioners believe that refactorings do not have immediate benefits and thus developers often postpone refactorings when they are constrained by time [4]. We compared the number of refactorings before and after major release dates to see whether *feature freeze* before major software releases discourages developers from introducing refactorings

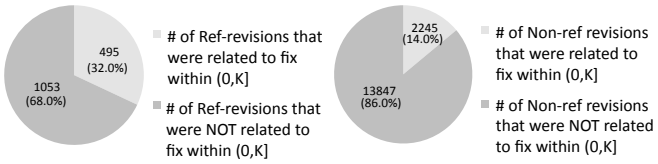

Figure 7: % of refactoring revisions that have at least one fix revision in a 20 revision window applied to the same method location

to the system.

In our study, we analyzed 19 major releases: 15 in Eclipse JDT and 4 in jEdit whose release dates are identified based on their websites.[2] We excluded Columba in this analysis as it had only one release during the period.

We compute a refactoring rate and a fix rate before and after each release using the same K sliding window method in Section 4.1. Figures 8 and 9 show the box plot (1st quartile, median, and 3rd quartile) for each subject for K=+/- 20, 30, and 40.

The results show that there are more refactorings before releases than right after, and there are more bug fixes before releases than after. Combined with Section 4.3's results that refactorings and bug fixes often occur in the same revision, we speculate that developers apply refactorings to implement bug fixes that must be shipped with a new software release.

[2]http://archive.eclipse.org/eclipse/downloads/ and http://sourceforge.net/projects/jedit/files

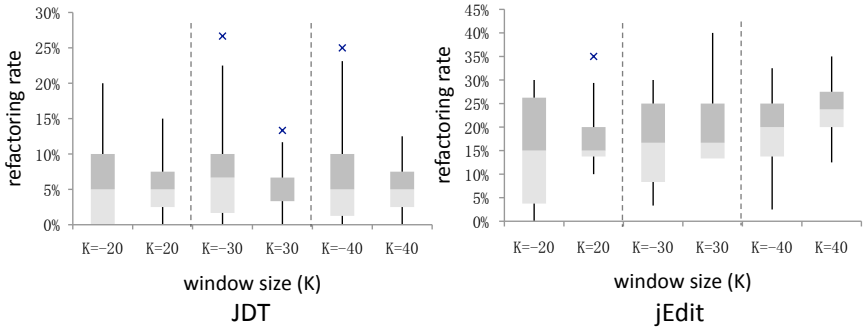

Figure 8: Refactoring rate in relation to release timing

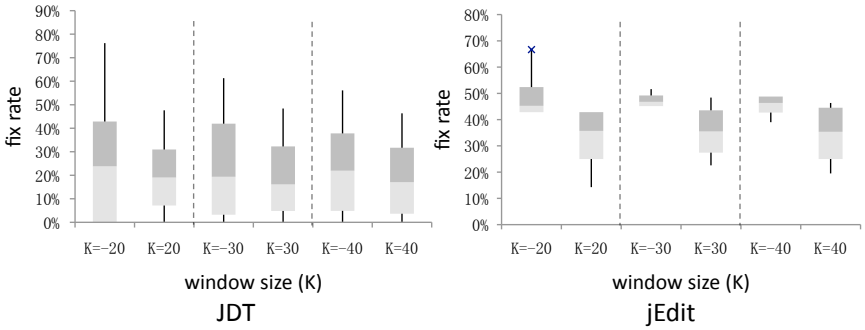

Figure 9: Fix rate in relation to release timing

There are more refactorings and bug fixes prior to major version releases.

5. DISCUSSION

This section discusses the limitation of our study method and the implication of our results.

Refactoring reconstruction's coverage and accuracy. As we discussed in Section 3, the refactoring revision data we used may not accurately represent the population of refactoring revisions as API-level refactoring reconstruction only captures a subset of refactorings—rename, move, and changes to API signatures at or above the level of method headers. Thus, our study results may not generalizable to *intra-method* refactorings or complex refactorings that do not involve any changes to method-headers. Furthermore, some may disagree with our definition of refactoring in this paper, since a refactoring technically cannot overlap with a bug fix, behavior-correcting transformations. Nevertheless, we believe that our results shed light on the relationship between API-level refactorings and bug fixes during software evolution. A recent work by Prete et al. encodes 63 out of 72 refactoring types in Fowler's catalog as template logic rules and uses a logic-query approach to infer concrete refactoring instances [24]. We plan to use this technique to collect refactoring revision data more accurately and comprehensively in the future. In addition, it is possible to collect refactoring revision data from different sources: mining commit logs, observing programmers, logging refactoring tool use, etc.

In our study, we considered a revision is a refactoring revision if it includes at least one API-level refactoring. Thus, a large commit, mostly feature addition with a single API-level refactoring, is still categorized as a refactoring revision according to our definition. A further study that accounts for the size of edits as well as the number of refactorings remains as future work. In our study, due to a large number of detected refactorings, we did not check with open source developers regarding whether they were performed manually or automatically using refactoring engines. It is possible that automatically applied refactorings do not have much correlation with bug fixes.

Phases and activity levels during a software life cycle. We did not investigate other confounding factors such as activity levels, task types, or the phases of a software life cycle (e.g., requirements analysis, design, testing, etc.) For example, a refactoring rate increase before major releases can be interpreted by organization shifting its focus to beautifying code before releases. Figure 10 shows the number of revisions within the same time frame during Eclipse JDT's life time. It shows that there are specific periods of very high-level of activities, indicating that developers may perform different types of tasks during such high activity period. Furthermore, the K-sliding window method in the study does not always map to the same length of absolute time, as 5 revisions during a high activity period could be equivalent to 1 day while it could be equivalent to 10 days during a low activity period. A revision may not be a meaningful time unit as developers could accumulate several logical changes in a single revision.

Development practices. Our study results may be strongly influenced by a few developers' practices. For example, the results on refactorings and bug fixes may be symptoms of micro-commits where developers commit a single logical

change in multiple revisions just as a habit. It is also possible that some organizations assign refactorings and bug fix tasks together to a few developers who drive a majority of refactoring and bug fix commits.

Bug-introducing change identification method. When measuring the extent of latent bugs, we used a bug-introducing change detection method which only tracks deleted or modified code lines. Thus, our method of tracking the origin of a bug is very limited especially when the bug fix involves code addition. This issue in the study's construct validity may have affected our results on bug fix time and the extent of latent bugs. In addition, we calculated the time to fix a bug based using the difference between bug introduction and resolution time, which is not always a reliable measure.

Software maintainability and developer productivity. Though our study reveals interesting relationships between refactorings and bug fixes, further investigation into the impact of refactoring on software maintainability and productivity is needed. The study found that refactoring often serves the role of both facilitating bug fixes and inducing bugs. This result calls for re-thinking the true benefits of refactorings and quantitatively assessing the cost and benefit of refactoring investment. Further research on economics-based quantitative assessment of refactorings remains as an open problem.

New software engineering tools. Empirical evidence from this study resonates with the limitations of refactoring features in modern IDEs reported by previous research [15, 20, 21]—refactorings occur as a part of other behavior enhancing or correcting transformations, and manual application of coordinated systematic refactoring is often error-prone. This result calls for new software engineering analyses such as a tool that detects refactoring mistakes and repairs them.

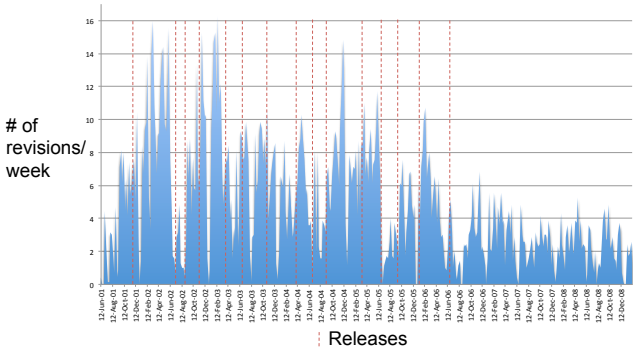

Figure 10: Activity density during Eclipse JDT evolution

6. CONCLUSIONS

It is believed that refactoring improves software maintainability and a lack of refactorings incurs technical debt in the form of increased maintenance cost. This paper presents an empirical investigation into the role of API-level refactorings during software evolution. The study found that the number of bug fixes increases after refactorings while the time taken to fix bugs decreases after refactorings. Refactorings often serve the role of both facilitating bug fixes and inducing bugs.

The study results calls for an in-depth quantitative investigation into the cost-benefit analysis of refactoring investment. The results also suggest the need of new software engineering tools that *detect* and *correct* inconsistent program updates when developers apply refactorings.

Acknowledgments

The authors thank anonymous reviewers, Danny Dig, and the participants of software engineering seminar at UIUC for their helpful comments on our draft. This work was supported in part by National Science Foundation under grant CCF-1043810.

7. REFERENCES

[1] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 298–308, Washington, DC, USA, 2009. IEEE Computer Society.

[2] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130, New York, NY, USA, 2009. ACM.

[3] J. Carriere, R. Kazman, and I. Ozkaya. A cost-benefit framework for making architectural decisions in a business context. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 149–157, New York, NY, USA, 2010. ACM.

[4] W. Cunningham. The wycash portfolio management system. In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, pages 29–30, New York, NY, USA, 1992. ACM.

[5] P. Dalgaard. *Introductory statistics with R*. Springer, New York, 2. ed. edition, 2008.

[6] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194, New York, NY, USA, 2007. ACM.

[7] D. Dig and R. Johnson. The role of refactorings in api evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[8] D. Dig and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP '06: Proceedings of European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.

[9] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug

tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[10] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, August 2007.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[12] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[13] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, WA, USA, 1991.

[14] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108, New York, NY, USA, 2008. ACM.

[15] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.

[16] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[17] S. Kim, T. Zimmermann, E. J. Whitehead, Jr., and A. Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[18] M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[19] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[20] E. Murphy-hill and A. P. Black. Why don't people use refactoring tools. In *ECOOP '07: Proceedings of the 1st Workshop on Refactoring Tools*, TU Berlin, Germany, 2007.

[21] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[22] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, New York, NY, USA, 2005. ACM.

[23] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 302–321, New York, NY, USA, 2010. ACM.

[24] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1 –10, 2010.

[25] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 35–38, New York, NY, USA, 2008. ACM.

[26] C. Reichenbach, D. Coughlin, and A. Diwan. Program metamorphosis. In *Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 394–418, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[28] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[29] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[30] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 112–118, New York, NY, USA, 2006. ACM.

[31] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. Aura: a hybrid approach to identify framework evolution. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 325–334, New York, NY, USA, 2010. ACM.

[32] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[33] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society.