Empir Software Eng (2017) 22:436–473
DOI 10.1007/s10664-016-9432-x

An empirical study of supplementary patches in open source projects

Jihun Park[1] · Miryung Kim[2] · Doo-Hwan Bae[1]

Published online: 7 May 2016

Abstract Developers occasionally make more than one patch to fix a bug. The related patches sometimes are intentionally separated, but unintended omission errors require supplementary patches. Several change recommendation systems have been suggested based on clone analysis, structural dependency, and historical change coupling in order to reduce or prevent incomplete patches. However, very few studies have examined the reason that incomplete patches occur and how real-world omission errors could be reduced. This paper systematically studies a group of bugs that were fixed more than once in open source projects in order to understand the characteristics of incomplete patches. Our study on Eclipse JDT core, Eclipse SWT, Mozilla, and Equinox p2 showed that a significant portion of the resolved bugs require more than one attempt to fix. Compared to single-fix bugs, the multi-fix bugs did not have a lower quality of bug reports, but more attribute changes (i.e., cc'ed developers or title) were made to the multi-fix bugs than to the single-fix bugs. Multi-fix bugs are more likely to have high severity levels than single-fix bugs. Hence, more developers have participated in discussions about multi-fix bugs compared to single-fix bugs. Multi-fix bugs take more time to resolve than single-fix bugs do. Incomplete patches are longer and more scattered, and they are related to more files than regular patches are. Our manual inspection showed that the causes of incomplete patches were diverse, including missed porting updates, incorrect handling of conditional statements, and incomplete

Communicated by: Massimiliano Di Penta

✉ Jihun Park
jhpark@se.kaist.ac.kr

Miryung Kim
miryung@cs.ucla.edu

Doo-Hwan Bae
bae@se.kaist.ac.kr

[1] School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, Korea

[2] Computer Science Department, The University of California, Los Angeles, CA, USA

refactoring. Our investigation showed that only 7 % to 17 % of supplementary patches had content similar to their initial patches, which implies that supplementary patch locations cannot be predicted by code clone analysis alone. Furthermore, 16 % to 46 % of supplementary patches were beyond the scope of the immediate structural dependency of their initial patch locations. Historical co-change patterns also showed low precision in predicting supplementary patch locations. Code clones, structural dependencies, and historical co-change analyses predicted different supplementary patch locations, and there was little overlap between them. Combining these analyses did not cover all supplementary patch locations. The present study investigates the characteristics of incomplete patches and multi-fix bugs, which have not been systematically examined in previous research. We reveal that predicting supplementary patch is a difficult problem that existing change recommendation approaches could not cover. New type of approaches should be developed and validated on a supplementary patch data set, which developers failed to make the complete patches at once in practice.

Keywords Software evolution · Empirical study · Patches · Bug fixes

1 Introduction

When developers attempt to fix a bug, they occasionally make more than one patch. Ordinary events, such as coffee breaks, sometimes cause the developers to separate related commits. However, many of these *multi-fix bugs* are resolved long after the initial patch. We observed that a number of initial patches required supplementary patches in open source projects. These kinds of bugs have not been systematically studied yet, and existing change recommendation systems have not been evaluated with these real-world incomplete patches, in which developers failed to apply an appropriate fix to the initial patch.

To prevent incomplete patches, several change recommendation systems (Hassan and Holt 2004; Nguyen et al. 2010; Robillard 2005; Ying et al. 2004; Zimmermann et al. 2005) have been suggested to predict additional change locations for a given change set. These change recommendation systems make their own assumptions about common omission errors and about how such errors could be eliminated. For example, FixWizard suggested the code clones of an existing patch to reduce potential missed updates (Nguyen et al. 2010). Robillard used the dependency structure of a change set to suggest where additional changes need to be made (Robillard 2005). Zimmermann et al. and Ying et al. predicted additional change locations based on historical co-change patterns derived from version histories (Ying et al. 2004; Zimmermann et al. 2004). Hassan and Holt predicted additional change locations based on co-change patterns and the dependency graph of a change set in conjunction with them (Hassan and Holt 2004). These previous studies have made various assumptions about how additional change locations can be recommended, but very few studies have focused on the reason that incomplete patches occur, as well as the prediction of supplementary patch locations for an initial change set.

This study examines *supplementary patches*—patches that are subsequently applied to supplement or correct initial fix attempts and *multi-fix bugs*—the bugs required the supplementary patches to be resolved. The goal of this study is to understand the characteristics of multi-fix bugs and to investigate how to predict the supplementary patch locations based on the initial patch location. We inspect the quality of the bug reports, the level of difficulty, and the characteristics of incomplete patches. To provide insights into the reasons that

incomplete patches occur, and the kinds of tools that are needed to prevent omission errors, we conduct a manual inspection of the common causes of incomplete patches. We also investigate the relationship between the initial patches and supplementary patches in terms of cloning, structural dependency, and historical change coupling using the version histories of Eclipse JDT core, Eclipse SWT, Mozilla, and Equinox p2.

The findings of our study are summarized as follows.

- **How many bugs require supplementary patches?** A considerable portion of resolved bugs (23 % in Eclipse JDT core, 26 % in Eclipse SWT, 33 % in Mozilla, and 26 % in Equinox p2) involves supplementary patches. We define bugs that are fixed with only one commit as *single-fix bugs*, and we define bugs that require supplementary patches later as *multi-fix bugs*. The quality of multi-fix bug reports is not less than that of single-fix bug reports, but more bug report attribute changes are made to multi-fix bugs. Because multi-fix bugs are more challenging than single-fix bugs are in terms of bug severity, more developers participate in the discussion about them. Multi-fix bugs also required more time to resolve compared to single-fix bugs.
- **What are the common causes of incomplete patches?** We randomly sampled a total of 200 supplementary patches and inspected their content and corresponding initial patches. Our manual inspection showed that the common causes of omission errors were diverse, including missed porting updates, the incorrect handling of conditional statements, and incomplete refactoring. Furthermore, we found that both the size of the patches and the number of related files were greater in the incomplete patches than in the regular patches and that the file-level dispersion was higher in incomplete patches than in regular patches. These results imply that incomplete patches tend to be larger and more scattered compared to regular patches.
- **How can we predict supplementary patch locations based on initial patch location?** We estimated the utility of techniques based on a code-clone, structural dependency, and historical co-change analysis to predict supplementary patch locations based on initial patch locations. First, we found that only 7 % to 14 % of supplementary patches had content similar to its initial patch in at least five lines. Second, 16 %, 21 %, and 45 % of the supplementary patch locations did not have direct dependence on nor did they overlap, the initial patch location in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively. Third, only 17 % to 36 % of supplementary patch locations had co-changed with the initial patch locations within 50 days before the date of the initial patch. Fourth, little overlap occurred among code clone, structural dependency, and historical co-change analyses. In addition, even when the three approaches were applied simultaneously, only 39 % to 53 % of the supplementary patch files could be predicted.

These results indicate that applying supplementary patches is a common phenomenon, and multi-fix bugs tend to be more difficult to fix than single-fix bugs. The causes of incomplete patches are diverse, and various approaches to recommend supplementary patches will be required to prevent each type of incomplete patch. Existing change recommendation approaches cannot detect the relationships between initial patches and supplementary patches. New tools to prevent real-world incomplete patches should be developed and validated using a supplementary patch data set.

The rest of this paper is organized as follows. Section 2 introduces related work, and Section 3 describes the subjects and analytical method used in the present study. Section 4 presents the empirical results, Section 5 discusses threats to validity, and Section 6 summarizes the study.

2 Related Work

Errors of Omission Fry and Weimer investigated how well a human could localize different types of bugs (Fry and Weimer 2010). They found that humans were five times more accurate at locating *extra statements* than they were at locating *missing statements*. This demonstrated that finding omission errors is much more difficult than finding commission errors. Fry and Weimer focused on measuring the accuracy of fault localization for different types of defects, whereas the present study focuses on the characteristics of supplementary patches in order to gain insights into the common causes and characteristics of omission errors.

FixWizard by Nguyen et al. (2010) suggests additional change locations by identifying the candidates of recurring bug fixes. Recurring bug fixes are cloned fix code chunks that are similar in terms of structure and function. They found that 17 % to 45 % of all fixing changes could be considered recurring bug fixes. Recurring bug fixes include fixes within the same commit, whereas the supplementary patches of the present study complement the initial patch at different commits. According to Nguyen et al. (2010) only 17 % to 28 % of recurring bug fixes occurred in different commits. Our study found that only 7 % to 16 % of the supplementary patches were similar to the corresponding initial patches, indicating that a clone analysis alone was insufficient to predict supplementary patch locations based on the initial patch location.

Kim et al. studied incomplete patches that are made when developers try to fix null pointer exceptions (Kim et al. 2010). They introduced the concept of the bug neighborhood to represent the program statements that are directly related to null pointer dereference bugs. Zhang et al. proposed a dynamic slicing method that detects execution omission errors (Zhang et al. 2007). These techniques focused on detecting omission errors, whereas the present study investigates the extent and characteristics of omission errors using the incomplete and supplementary patches found in the version history.

This paper is an extended version of our previous work in MSR'12 (Park et al. 2012). The present study extends our previous work in five ways. First, we extend the time frame of our study period from two years on each project to five, six, and nine years in Eclipse JDT core, Eclipse SWT, and Mozilla, respectively, so that our results could be generalized to a period of long-term development. We also add a new subject program, Equinox p2. Second, we investigate the levels of quality and difficulty of bug reports by inspecting their attributes and the history of each bug. Third, we manually inspect 100 more bugs (a total of 200 bugs) than in our previous work, which resulted in new categories, including late updates of test codes and property updates. In this study, two additional graduate students participated in the manual inspection to validate the categorization. Fourth, we study how supplementary patch locations can be predicted based on historical co-change patterns. Fifth, we study the degree to which code clone, structural dependency, and historical co-change analysis overlap.

In another work in ASE'14 (Park et al. 2014), we proposed *the change relationship graph (CRG)* to study how supplementary change locations could be predicted based the initial change locations. In the previous study, we used the supplementary patch data set of Eclipse JDT core, Eclipse SWT, and Equinox p2, which is also used in the present study. We found that it was inherently challenging to predict supplementary change locations.

Empirical Studies of the Extent of Supplementary Patches Yin et al. investigated incorrect bug fixes in large operating systems (Yin et al. 2011). They found that 15 % to 24 % of post release patches were incorrect and that the most difficult type of bug was

related to concurrency. Their results showed that a considerable proportion of bugs underwent more than one fix. Gu et al. studied bad fixes, i.e., cases in which a bug fix failed to fix the bug or created a new bug in Ant, AspectJ, and Rhino (Gu et al. 2010). They found that bad fixes corresponded to as much as 9 % of all bugs. They assessed bug fixes using two criteria: *coverage*—if the fix correctly handled all bug-triggering inputs, i.e., the fixed program passed all failing test cases; and *disruption*—if the fix unintentionally introduced a new bug, i.e., the behavior of fixed program was the same as that of the correct oracle. Gu et al. used the re-opened bug reports as the data set. Of the re-opened bugs, they aimed to identify bad fixes that failed to fix the failing test cases correctly. The present study investigates the characteristics of incomplete patches which require supplementary patches to be completely resolved. Their tool, Fixation, could be used to detect incomplete patches that are related to failing test cases.

Purushothaman et al. investigated the extent of small changes and found that nearly 40 % of bug fixes caused one or more defects in Lucent 5ESS (Purushothaman and Perry 2005). The findings of the present study support these results, which showed that incomplete bug fixes are common in practice.

Change Recommendation Systems for Supplementary Patches Robillard's approach (Robillard 2005) reduced omission errors by using a change set as input and recommending additional change locations based on the dependence structure of the change set. His approach was based on the assumption that additional change locations are likely to have structural dependencies on the previously changed code. Hassan and Holt proposed several change propagation heuristics and found that historical change coupling was more accurate than structural dependencies, such as method call relationships (Hassan and Holt 2004). Zimmermann et al. group changes, and mine co-change patterns in predicting extra change locations given an existing change (Zimmermann et al. 2004). Ying et al. suggested an association-mining technique to predict extra change locations based on change patterns, i.e., the type of files that frequently have been changed simultaneously in the past (Ying et al. 2004). Nagappan et al. investigated the predictive power of consecutive code changes (change bursts) (Nagappan et al. 2010). Padioleau et al. (2008) found that Linux device drivers often co-evolved when kernel APIs changed, suggesting an approach that infers a generic patch from an example edit (Andersen and Lawall 2008). Based on a similar assumption, Wang et al.'s approach automatically found edit locations similar to an existing bug fix using dependence-related queries (Wang et al. 2010). While these approaches make individual assumptions about how supplementary patch locations can be predicted based on the content and location of an existing patch, our study investigates the actual location, content, and characteristics of incomplete and supplementary patches.

Bug Report Quality Hooimeijer and Weimer (2007) suggested a model that predicts whether the triage cost of a bug report is expensive or not. Zimmermann et al. (2010) studied the quality of bug reports based on results of a survey of developers and reporters. They developed the prototype tool, CUEZILLA, which measures the quality of a new bug report. The researchers used bug report attributes as input features for their models. These attributes included the readability of the bug description and the presence of code samples or stack traces. We found that the low quality of the initial bug report was not the main cause of incomplete patches. Instead, attribute changes caused by incorrect information were more common in multi-fix bugs than in single-fix bugs.

Re-opened Bugs In a bug tracking system, developers often re-open a bug after the first closing of the bug. Zimmermann et al. (2012) studied the characteristics of re-opened bugs. Bug re-openings were caused by a bug that was difficult to reproduce when developers misunderstood the root cause or by a bug with insufficient information when the priority of the bug was underestimated. Shihab et al. (2013) also studied re-opened bugs in open source software and predicted whether a bug would be re-opened after the bug was resolved. Their prediction model used bug attributes with four dimensions: 1) work habits, 2) bug report, 3) bug fix, and 4) team. They found that the comment text and the status at the time when the bug was initially closed were the most important factors related to bug re-opening. An et al. (2014) studied the relationship between supplementary bug fixes, as defined in our previous work (Park et al. 2012), and re-opened bugs. They found that 21.6 % to 33.8 % of supplementary bug fixes were related to bug re-opening, and that 33 % to 57.5 % of re-opened bugs were fixed only once in the version history. Most previous studies on re-opened bugs were focused on the information given by the bug reporting system, whereas the current study focuses the characteristics of code changes in incomplete and supplementary patches.

Others Rahman et al. (2012) found that the majority of bugs were not related to code clones. Bettenburg et al. (2012) also found that only a small portion of clone genealogies induced software defects at the release level. These results are aligned with our finding that supplementary patches rarely had content similar to the initial patches. Ray and Kim (2012a) studied cross-system porting in the BSD product family, finding that 11 % to 14 % of all edits were ported from other projects. We found that porting changes were among the most common causes of supplementary bug fixes. 5 % to 10 % of supplementary patches were patches in which the initial patch was ported to a different component or a different branch.

3 Study Approach

This section describes our study subjects and the method used to identify supplementary patches.

Study Subjects We selected Eclipse JDT core, Eclipse SWT, Mozilla, and Equinox p2 as our study subjects. Eclipse is an integrated development environment that has been widely used in several studies on mining software repositories. Eclipse JDT core, Eclipse SWT, and Equinox p2 are sub-projects of Eclipse and are written mainly in Java. Mozilla is an open source project for the web, which contains many different sub-projects written in many different languages (mostly C, C++, and Java). Two examples are the Firefox web browser and the Thunderbird mail client.

When we extracted information from the bug databases, we ensured that the bugs were *completely resolved* and would not be re-opened later by considering only the bug reports reported within the periods of January 2002 to December 2007 in Eclipse JDT core; August 2002 to December 2008 in Eclipse SWT; January 2000 to December 2009 in Mozilla; and October 2006 to January 2010 in Equinox p2. From the bugs in the selected periods, we excluded *invalid* bugs, which were not fixed adequately, that is, did not display FIXED status. Table 1 summarizes our study subjects.

Table 1 Study subjects

	Eclipse JDT core	Eclipse SWT
Type	IDE	IDE
Development period	2001/06 ∼ 2009/02	2001/05 ∼ 2010/05
Study period	2002/01 ∼ 2007/12	2002/01 ∼ 2008/12
Total revisions	17009 revisions	21530 revisions
	Mozilla	**Equinox p2**
Type	Several projects associated with internet	IDE
Development period	1998/03 ∼ 2011/08	2006/01 ∼ 2013/07
Study period	2000/01 ∼ 2009/12	2006/10 ∼ 2010/01
Total revisions	261630 revisions	6761 revisions

Identification of the Commit Transaction Eclipse JDT Core, Eclipse SWT, and Mozilla provide CVS repositories.[1] We grouped file-revisions checked in the same commit, by converting CVS repositories to SVN repositories using cvs2svn.[2] We also converted the Equinox p2 GIT repository to a SVN repository using subgit[3] to permit the use of the same analytical method used with the other study subjects. We disregarded all patches that did not include source file changes. The extensions c, cpp, h, and java were regarded as source files, which accounted for 82 %, 912 %, 53 %, and 75 % of all changed files in the repositories of Eclipse JDT core, Eclipse SWT, Mozilla, and Equinox p2, respectively. We also disregarded patches that involved more than 50 files in order to remove noise from the analyses of very large changes.

Identification of Incomplete and Supplementary Bug Fixes Previous studies of bug fixes (Kim et al. 2011, 2008) found that bug fix revisions could be extracted by matching keywords against commit logs using the heuristic developed by Mockus and Votta (2000). We parsed change logs to look for bug ID numbers and regarded all integers as potential bug IDs. We then checked those numbers against the corresponding bug databases to ensure that those extracted numbers indeed correspond to bug IDs that are of interest (Fischer et al. 2003; Čubranić and Murphy 2003). We disregarded invalid numbers, such as numbers representing dates (e.g., 20050101), or small numbers (e.g., 1, 8, and 6 in 1.8a6) using this method. We identified that 24 % to 33 % of the commits were bug fixes.

After identifying the bug IDs, we grouped the bug reports into two groups: 1) *single-fix bugs*—the bugs that were mentioned in only one fix commit; and 2) *multi-fix bugs*—the bugs that were mentioned in multiple fix commits. Based on this grouping, we divided the commits of multi-fix bugs into two categories: 1) *incomplete (initial) patches* as the first attempt to address a multi-fix bug; and 2) *supplementary patches* as later attempts to correct, extend, complement, or revert an incomplete fix.

We assessed the accuracy of our method of identifying supplementary patches by inspecting sample periods of bug reports (01 January 2007 to 31 January 2007 in Eclipse JDT core

[1] They currently use the GIT (Eclipse sub-projects) and Mercurial (Mozilla) repositories

[2] http://cvs2svn.tigris.org/

[3] http://subgit.com/

Table 2 Accuracy of our incomplete patch identification method

	JDT	SWT	Mozilla	Equinox	Total
# of revisions	107	165	786	193	742
# of incomplete fixes (automated)	12	7	53	15	87
# of incomplete fixes (ground truth)	13	10	55	19	97
Precision	100 %	100 %	100 %	100 %	100 %
Recall	92.3 %	70.0 %	96.4 %	78.9 %	89.7 %

and Eclipse SWT, 01 January 2007 to 14 January 2007 in Mozilla, and 01 January 2008 to 31 January 2008 in Equinox p2).

To compose the ground truth set of incomplete patches, we identified whether the automatically identified incomplete patches were false positives or not. We also manually inspected the incomplete patch candidates to find false negatives by investigating the following: 1) the same file was fixed again within one month; 2) two commit log messages were similar (i.e., the longest common subsequence of them was longer than 80 % of the shorter one), or 3) the contents of the two patches were similar (identified by CCFinderX).

We compared this ground truth set to the results of our incomplete patch identification method. Our incomplete patch identification method had 100 % precision and 90 % recall overall, as shown in Table 2. There were no false positives, but we found that several incomplete patches were not linked to a bug ID. Developers often make a patch without making a bug report because the problem is minor, and the first patch in the minor problem is sometimes complemented by a supplementary patch.

4 Study Results

In this section, we present the characteristics of multi-fix bugs and describe how to predict supplementary patch locations.

4.1 How Many Bugs Require Supplementary Patches?

Table 3 shows the number of bugs fixed only once *(single-fix bugs)* vs. the number of bugs with more than one fix commit *(multi-fix bugs)*. Of the resolved bugs, 22.9 % to 33.7 % required supplementary patches. In Fig. 1, the X axis represents the number of times the same bug ID occurred in the fix commits, and the Y axis represents the percentages of all multi-fix bugs with n patches. Overall, 60 % to 75 % of multi-fix bugs were fixed twice.

To determine whether the supplementary patches were caused by a mistake or were intended separation of related patches, we inspected the commit logs. We investigated whether the developers recorded their mistake in the commit logs. We first observed that 34 % to 80 % of the supplementary patches had the same or similar commit logs as the

Table 3 The number of single-fix bugs and multi-fix bugs

	Eclipse JDT core	Eclipse SWT	Mozilla	Equinox p2
# of bugs	3677	3992	42283	1641
# of single-fix bugs	2836 (77.1 %)	2959 (74.1 %)	28034 (66.3 %)	1203 (73.3 %)
# of multi-fix bugs	841 (22.9 %)	1033 (25.9 %)	14249 (33.7 %)	438 (26.7 %)

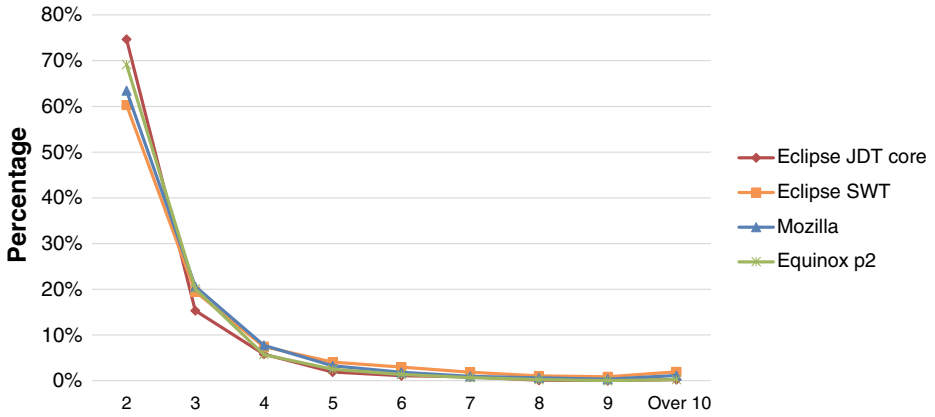

Fig. 1 The number of times that the same bug is fixed

corresponding initial patch had (the longest common subsequence accounted for more than 90 %). For the remaining patches, we looked for keywords that implied that their initial patch was incomplete—*improve, again, complement, complete, mistake, supplementary, incomplete, clarify, workaround, additional, rewrite, better, port, undo, revert, optimize, clean up, redesign, etc.* We found that 10 % to 19 % of the supplementary patches included these keywords. Table 4 shows examples of unintended supplementary patches that were very likely caused by omission errors.

Figure 2 shows the number of days taken for a supplementary patch to appear since an initial fix attempt. We measured the time gap between the first patch and the last patch of the multi-fix bugs. Of the supplementary patches, 31 % to 58 % appeared within one day. In addition, 16 % to 31 % of supplementary patches were made within 1 h from the initial patch. Very short time gap between initial and supplementary patches indicates that the patches could be intentionally separated or caused by ordinary events, such as lunch or coffee break.

On the other hand, some supplementary patches take a very long time and require a large number of fix attempts. For example, in Mozilla, one bug (236613) was resolved with 105 commits over five years to re-license Mozilla for a MPL/LGPL/GPL tri-license. In

Table 4 Examples of supplementary patches caused by omission errors

Revision	Date	Author	Comment
Bug 80904 of Eclipse JDT core			
13468	2006/04/20	Jeromel	HEAD - 80904
13471	2006/04/20	Jeromel	HEAD - *Better* fix for 80904
Bug 135811 of Mozilla			
125980	2002/08/07	Timeless	Bug 135811 Crash after infinite recursion: nsContentTree...
167681	2005/02/01	Bzbarsky	*Undo* the checkin for bug 135811 and refix it *better*...
Bug 207624 of Equinox p2			
321	2007/10/27	Pascal	Bug 207624 - [prov] Logic to make the path relative and ...
337	2007/10/30	Pascal	*Reverting* changes for bug 207624

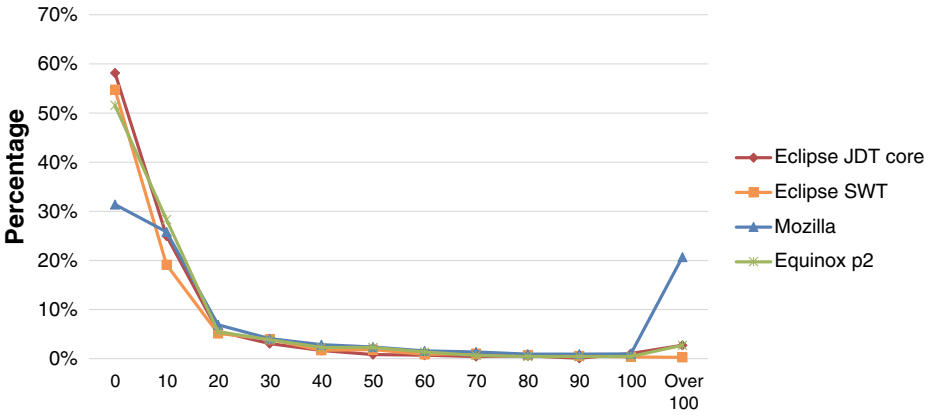

Fig. 2 The number of days taken for the supplementary patch to appear since an initial fix

Eclipse SWT, a supplementary patch for bug 28766 was applied 1938 days after the initial fix attempt. This bug was found in printing a transparent region; the bug was fixed and tested with a gif file. However, the same problem reoccurred in a png file, and the bug was re-opened to resolve it again.

In conclusion, 23 % to 34 % of the bugs required supplementary patches. 16 % to 31 % of the supplementary patches were made within one hour from the initial patch, indicating that they could be intended separations of related commits, and that supplementary patches might not be a totally bad phenomenon. Meanwhile, 10 % to 19 % of the supplementary patches explicitly mentioned that the first patch was incomplete, and 42 % to 69 % take more than 24 hours to be made.

A considerable portion of bugs requires supplementary patches.

4.2 How Different are the Quality Levels of the Bug Reports of Single-Fix Bugs and Multi-fix Bugs?

We hypothesized that a low quality bug report leads to supplementary patches. To assess the initial quality levels of bug reports, we investigated the descriptions and comments made by the reporter within 15 min from the creation of the bug report (Zimmermann et al. 2010). We investigated the presence of code samples, stack traces, patch attachments, screenshot attachments, other type of attachments in the bug description or comment, the readability of the bug description, and the length of the bug description.

We identified the specific patterns of code samples and stack traces in the descriptions of the bugs. For example, code lines end with opening/ending brackets ('{' or '}') or start with a conditional/loop statement (for, while, or if). Attachments were categorized into three types: patches, screenshots, and others. All images were regarded as screenshots (Zimmermann et al. 2010). We used the Coleman-Liau index[4] to calculate the readability of the description.

[4] http://en.wikipedia.org/wiki/Coleman-Liau_index

Table 5 The quality of single-fix bugs and multi-fix bugs

	JDT		SWT		Mozilla		Equinox	
	S-fix	M-fix	S-fix	M-fix	S-fix	M-fix	S-fix	M-fix
Code sample	41.8 %	39.6 %	24.8 %	22.2 %	6.3 %	6.5 %	3.9 %	3.3 %
Stack trace	2.2 %	1.1 %	3.8 %	3.4 %	0.1 %	0.1 %	8.5 %	9.4 %
Patch att.	2.9 %	4.9 %	3.2 %	2.4 %	22.4 %	18.9 %	10.6 %	8.2 %
Screenshot att.	0.6 %	0.7 %	9.0 %	7.5 %	2.4 %	2.0 %	2.8 %	2.6 %
Other att.	5.8 %	7.7 %	9.8 %	9.7 %	11.5 %	11.6 %	4.6 %	5.6 %
Readability	34.8	32.1	17.5	15.2	19.4	18.1	15.4	16.0
Desc. length	1214.8 %	1368.6 %	1120.7 %	1387.9	726.9 %	796.0 %	785.7 %	992.9 %

Gray cell means the difference is *not* statistically significant (*p*-value is greater than 0.05)

To obtain the length of the bug description, we removed html tags and calculated the string length of the description.

Table 5 shows the quality of bug reports for single-fix vs. multi-fix bugs.[5] For the five true/false attributes (the presence of code samples, stack traces, patch attachments, screenshot attachments, and other attachments), we used z-tests for two proportions. We used t-test to determine the difference in readability and description length.

Only a small portion of the results (7 of 28) was statistically significant. Furthermore, the results did not consistently indicate that the quality level of the multi-fix bugs was lower than that of the single-fix bugs. The average length of a bug description was longer for multi-fix bugs than for single-fix bugs, which indicated that the multi-fix bug reports likely contained more information, were of higher quality (Zimmermann et al. 2010), or were more complicated to describe. The inconsistent and statistically insignificant numbers indicated that the quality levels of the single-fix bugs and the multi-fix bugs were not substantially different, and the incomplete patches might not have been caused by low-quality reports.

When a reporter submits a bug, he or she inputs bug report attributes. After the report is filed, the bug report attributes can be changed. We investigated the attribute changes made to single-fix bugs and multi-fix bugs to assess the degree to which the report was initially correct.

Table 6 shows the average number of changes made to single-fix vs. multi-fix bugs within a week after the report. To remove the effect of time, we considered the attribute changes within one week after the reporting of the bug, instead of the whole bug history. On average, 5 % to 20 % more changes were made to multi-fix bugs than to single-fix bugs.

Table 7 shows the average number of changes within a week after the report for each attribute. First, we observed that 12 % to 53 % more developers were added in the CC lists of multi-fix bugs than in those lists of single-fix bugs. This result implies that multi-fix bugs require more attention from developers than single-fix bugs do.

The greater number of changes to title, severity, or keyword indicate that the corresponding information in multi-fix bugs is more likely to be incorrect or inadequate. For example, a change in the title field indicated that the initial title of the bug report was inadequate,

[5]In the remainder of the paper, we will use S-fix and M-fix as abbreviation of single-fix bugs and multi-fix bugs, respectively, in tables because space is limited.

Table 6 The average number of changes within a week since the reporting

	S-fix	M-fix	*p*-value
Eclipse JDT core	3.82	4.35	2.16e-08
Eclipse SWT	3.04	3.39	1.41e-05
Mozilla	6.35	7.60	2.63e-82
Equinox p2	3.62	3.80	2.19e-01

Gray cell means the difference is *not* statistically significant (*p*-value is greater than 0.05)

and a change in the severity field indicates that the initial estimation of the severity was incorrect.

The initial quality of multi-fix bug reports was not lower than that of single-fix bug reports. However, more bug report attribute changes are made to multi-fix bugs, which involve incorrect information.

4.3 How Difficult are Single-Fix Bugs and Multi-fix Bugs to Fix?

We investigated the severity levels and the time-to-fix of single-fix bugs and multi-fix bugs by extracting and comparing the following information from the bug databases; 1) The severity level of a bug, 2) the number of comments within a week of the initial reporting of a bug, 3) the number of developers involved within a week, and 4) the time taken to resolve a bug.

The severity levels indicated the seriousness of a defect: Blocker/Critical/Major/Normal/Minor/Trivial/Enhancement is the descending order of severity. Table 8 shows the severity distributions of single-fix bugs vs. multi-fix bugs. The multi-fix bugs were 26 % to 59 % more likely to be Blocker or Critical.

We conducted a z-test to compare the two proportions of bugs according to Blocker or Critical severity in each project. Our analysis showed that a high level of severity was more likely to occur in multi-fix bugs than in single-fix bugs in each project—Based on the results of the z-tests, the hypotheses that the two proportions are the same was rejected. The

Table 7 The average number of changes on bug history for each type

	JDT		SWT		Mozilla		Equinox	
	S-fix	M-fix	S-fix	M-fix	S-fix	M-fix	S-fix	M-fix
CC	0.47	0.73	0.79	1.00	1.90	2.28	1.27	1.42
Status	1.16	1.17	0.62	0.54	1.09	1.07	0.77	0.75
Resolution	0.71	0.73	0.55	0.47	0.49	0.46	0.60	0.50
Target milestone	0.74	0.73	0.26	0.20	0.34	0.35	0.90	0.80
Assignee	1.00	1.03	0.92	1.00	0.45	0.46	0.62	0.61
Title	0.27	0.35	0.14	0.17	0.22	0.24	0.22	0.26
Component	0.18	0.15	0.14	0.16	0.14	0.15	0.05	0.07
Severity	0.05	0.07	0.05	0.07	0.10	0.11	0.06	0.06
Keyword	0.02	0.04	0.02	0.03	0.44	0.63	0.05	0.07

Gray cell indicates the difference is *not* statistically significant (*p*-value is greater than 0.05)

Table 8 Severity of single-fix bugs and multi-fix bugs

	JDT		SWT		Mozilla		Equinox	
	S-fix	M-fix	S-fix	M-fix	S-fix	M-fix	S-fix	M-fix
Blocker	0.88 %	1.54 %	1.85 %	2.51 %	2.67 %	2.37 %	1.24 %	2.05 %
Critical	3.03 %	4.04 %	4.35 %	7.35 %	12.52 %	16.70 %	1.66 %	2.51 %
Major	8.11 %	8.79 %	12.20 %	15.29 %	9.34 %	11.12 %	5.23 %	7.30 %
Normal	77.36 %	74.43 %	75.39 %	63.98 %	65.90 %	61.08 %	86.36 %	79.90 %
Minor	4.47 %	2.85 %	2.29 %	1.83 %	3.78 %	2.51 %	1.99 %	1.82 %
Trivial	1.65 %	0.71 %	1.14 %	0.67 %	2.12 %	0.99 %	0.58 %	0.00 %
Enhancement	4.47 %	7.60 %	2.73 %	8.32 %	3.63 %	5.19 %	2.90 %	6.39 %

findings showed that multi-fix bugs required urgent fixes by developers, and they had greater effects on programs than the single-fix bugs had.

We also found that the proportion of Enhancement was greater in multi-fix bugs than in single-fix bugs. Multi-fix bugs were 43 % to 204 % more likely to be an Enhancement. Enhancement issues are likely to be fixed more than once because they often require more discussion than usual bugs do. Usual bug reports include faulty behavior, but Enhancement issues are often related to functional or nonfunctional improvements for which the specifications are not clarified.

Table 9 shows the results of the developer participation and the time taken to resolve bugs. We counted the number of comments on a bug report and the number of developers who wrote comments on the bug report within a week of the initial reporting of the bug. The

Table 9 The effort taken to fix single-fix bugs and multi-fix bugs

	S-fix	M-fix	*p*-value
The number of comments within a week			
Eclipse JDT core	3.40	4.63	4.64e-11
Eclipse SWT	2.73	3.30	3.36e-05
Mozilla	5.41	6.90	5.04e-86
Equinox p2	3.53	3.70	4.77e-01
The number of developers involved within a week			
Eclipse JDT core	1.69	1.87	5.28e-04
Eclipse SWT	1.57	1.73	9.61e-04
Mozilla	2.52	2.81	6.20e-33
Equinox p2	1.71	1.73	7.57e-01
The time taken to resolve bugs			
Eclipse JDT core	103.40	165.05	6.86e-08
Eclipse SWT	153.94	292.97	1.74e-16
Mozilla	700.51	822.35	5.25e-34
Equinox p2	73.06	106.33	1.78e-03

Gray cell indicates the difference is *not* statistically significant (*p*-value is greater than 0.05)

results showed that more comments were written and that more developers were involved in multi-fix bugs than in single-fix bugs during the first week of bug reporting. The developers had more interest in multi-fix bugs. The reason might be that multi-fix bugs have higher severity levels, and thus they are more likely to deal with complicated or difficult tasks that require more interest and effort from developers.

The results also showed that the multi-fix bugs took more time to resolve. The time taken to resolve an individual bug was measured as the time taken from the reporting of the bug to the last status change of the bug (i.e., from REPORTED to VERIFIED, RESOLVED, or CLOSED status). Although multi-fix bugs attracted more attention from developers, they took 60 %, 90 %, 17 %, and 46 % more time to resolve in Eclipse JDT core, Eclipse SWT, Mozilla, and Equinox p2, respectively.

Multi-fix bugs are likely to have a greater effect on programs than single-fix bugs. They draw more attention than single-fix bugs from developers, but they take more time to be resolved

4.4 What are the Common Causes of Incomplete Patches?

To understand why omission errors occur in practice, we initially contrasted the characteristics of *incomplete patches* (initial patches of multi-fix bugs) to those of *regular patches* (patches of single-fix bugs). We measured the average number of files, the total number of changed lines, the percentage of added lines of all changed lines, and physical dispersion. We measure the physical dispersion by computing the *Shannon Entropy* at the file and package levels. Shannon (1949) $entropy = -\sum_{i=1}^{n} p_i * log_2(p_i)$. p_i is the probability that a changed line is made to a particular changed file (or package), which is the same method used by Hassan (2009). A low entropy score implied that only a few files include most of the modifications. If the entropy was high, the changed code was more equally distributed among the different changed files. The results are summarized in Table 10. The incomplete patches were larger in size than the regular patches were, and they tended to include a greater number of scattered and non-localized edits.

We investigated the common causes of omission errors by randomly selecting 200 incomplete patches and inspecting their patch contents, the structural dependence relationships

Table 10 The size and physical dispersion of regular patches vs. incomplete patches

		JDT	SWT	Mozilla	Equinox
Files	Regular	2.42	2.12	2.97	3.44
	Incomplete	3.16	2.80	3.51	3.93
LOC	Regular	107.55	95.81	148.86	134.77
	Incomplete	183.23	106.04	208.42	179.40
Added LOC	Regular	63.50	65.89	61.24	65.28
	Incomplete	65.82	69.93	64.86	68.45
Dispersion	Regular	0.64	0.56	0.84	1.03
(file)	Incomplete	0.79	0.83	0.94	1.14
Dispersion	Regular	0.28	0.23	0.36	0.67
(package)	Incomplete	0.33	0.40	0.41	0.70

An initial patch (revision 10911 in Eclipse SWT)

```
Index: .../win32/org/eclipse/swt/widgets/Table.java
=======================================
@@ -2180,10 +2180,12 @@
    OS.InvalidateRect (handle, null, true);
    TableColumn[] newColumns = new TableColumn [count];
    System.arraycopy (columns, 0, newColumns, 0, count);
+   RECT newRect = new RECT ();}
    for (int i=0; i<count; i++) {
+       TableColumn column = newColumns [i];
        if (!column.isDisposed ()) {
+           OS.SendMessage (hwndHeader, OS.HDM_GETITEMRECT, i, newRect);
+           if (newRect.left != oldRects [i].left) {
                column.sendEvent (SWT.Move);
            }
        }
```

A supplementary patch (revision 10915 in Eclipse SWT)

```
Index: .../carbon/org/eclipse/swt/widgets/Table.java
=======================================
@@ -2047,21 +2047,34 @@
...
       TableColumn[] newColumns = new TableColumn[columnCount];
       System.arraycopy (columns, 0, newColumns, 0, columnCount);
       for (int i=0; i<columnCount; i++) {
+         TableColumn column = newColumns [i];
          if (!column.isDisposed ()) {
+            if (newX [i] != oldX [i]) {
                column.sendEvent (SWT.Move);
             }
          }
```

Fig. 3 An initial patch is ported to a different component or branch

between an incomplete patch and its supplementary patches, and the associated bug reports. We identified the causes of incomplete patches, and then devised categories by grouping the incomplete patches according to similar causes. The categorization was initially done by the first author, and then verified by two graduate students. We agreed on 175 incomplete patches. The inter-rater agreement (Cohen's kappa) was 0.85, which indicated almost perfect agreement (Viera et al. 2005). For the 25 incomplete patches on which our opinions initially diverged, we worked in conjunction to find the most appropriate category. The following summarizes the taxonomy of the common omission errors found in the subject programs.

1) **An initial patch is ported to a different component or branch.** For example, to fix bug 88829 in Eclipse SWT, the patch for Windows was subsequently ported to Mac in the same file, `Table.java`. Ported edits are usually identical or very similar to the original edits, but the content is often adjusted for different target components or branches, as shown in Fig. 3.[6]
2) **Code elements referring to or being referenced by changed code (i.e., calls, accesses, or extends) are later updated.** For example, the programmer originally fixed `Display` to fix bug 93294 in SWT. The class `Device`, a super-type of `Display`, was later modified.
3) **The conditional statement of an initial fix is not correct.** For example, the initial patch of bug 80699 in Eclipse JDT core modified control flows when the condition `(modifiers && AccAnnotation)` is true. Subsequently, the condition was

[6] Additional examples are provided in the Appendix

modified to handle the corner case of `AccInterface`, as shown in Fig. 11 in the Appendix.

4) **An initial patch is reverted.** For example, an API that was added to the initial patch of bug 110048 in Eclipse JDT core was deleted during a supplementary patch because the API was inadequate, and the developers decided to apply a new approach. Its status was changed to `REOPENED`.
5) **The test code is added or updated.** For example, a developer fixed a bug by checking whether the variable profile was null (bug 229205 in Equinox p2). The test code for the bug fix, which is done by using testNoSelfProfile, was added to the supplementary patch. (See Fig. 12 in the Appendix)
6) **An initial patch is refactored during the supplementary patch.** The initial patch of bug 287052 in Mozilla modified the function `CERT_FindCRLReasonExten`, which was later renamed as `CERT_FindCRLEntryReasonExten` in the supplementary patch.
7) **The comment is improved to explain an initial patch in detail.** For example, the comment for bug 243392 in Mozilla was updated to explain the initial fix of `nsContentSink.cpp` (see Fig. 13 in the Appendix).
8) **Incomplete refactoring induces a supplementary patch.** The initial patch of bug 104664 in Eclipse JDT core partially refactored the `file` and the `zipFile` functions. The remaining parts were fixed during a supplementary patch (see Fig. 14 in the Appendix).
9) **Properties are updated.** The initial patch of bug 183399 in Equinox p2 was applied to find a default location for `bundles.txt`. The supplementary patch of this bug updated the configuration file to change the default location folder from `simpleConfiguration` to `org.eclipse.equinox.simpleconfigurator` (see Fig. 15 in the Appendix).
10) **Two different parts calling different subclasses of the same type are not updated together.** The initial patch of bug 81244 in the Eclipse JDT core concerned the use of `ArrayTypeReference`, and the supplementary patch concerned the use of `ArrayQualifiedTypeReference`. The two classes are subtypes of the same super type (see Fig. 16 in the Appendix).
11) **The locations of incomplete and supplementary patches are related but cannot be checked using the Java compiler.** The initial patch of bug 83593 in Eclipse JDT core was made to class `CopyResourceElementsOperation` and the supplementary patch was made to class `DOMFinder`. The two classes do not have direct static call dependencies because the code uses the Visitor design pattern (Gamma et al. 1994).
12) **Others.** Other omission errors included missing null pointer checks, missed value initialization, misunderstanding requirements, and forgetting to release run-time resources. For example, the initial patch of bug 114935 in the Eclipse JDT core was in `CompilationUnitResolver.java` to exit a loop early when there was no need to resolve the types further. The remaining resources were not cleaned up, but this problem was fixed during a supplementary patch.

Figure 4 shows the classification of 200 incomplete patches into the 12 categories described above; 36 % of the incomplete patches involved missed porting updates, 20 % involved forgetting to update a code that referenced a modified code, and 15 % involved the incorrect handling of conditional statements. The categories of all sampled incomplete patches are shown in Table 20 in the Appendix.

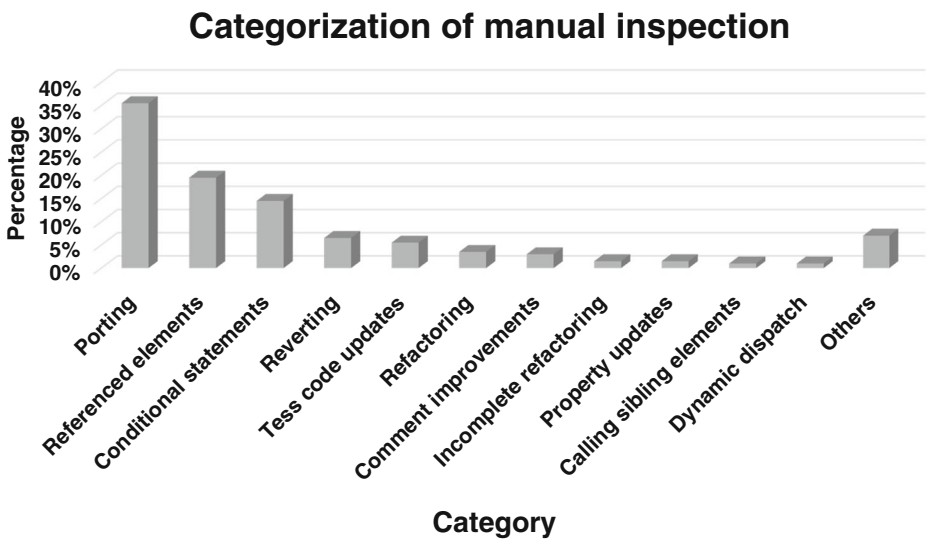

Fig. 4 Categorization of manual inspection results

These categorization of common causes led to insights into several potential approaches to preventing omission errors. For example, developers would find helpful a tool that would automate porting the selected patches to a target branch or component. The existing research on systematic editing, which has applied similar but not identical patches to a different location, could be used. Porting patches often require slight changes to the initial patch, such as the identifier name (Meng et al. 2011, 2013). In another approach, an analysis tool would find the differences in the control flows of a changed block, which could mitigate the errors in detecting all cases of conditional statements. Tools that detect incomplete refactoring based on the class inheritance hierarchy (Görg and Weißgerber 2005) might also prevent users from introducing omission errors when developers miss propagating a refactoring change to related locations, such as subclasses or sibling classes.

The common causes of incomplete fixes are diverse. Incomplete patches are larger in size and more scattered than regular patches.

4.5 Are Supplementary Patches Similar to Corresponding Initial Patches?

It is widely believed that code clones are difficult to maintain and that inconsistency in the management of code clones is a frequent source of omission errors. We determined how often supplementary patches were induced by the inconsistent management of clones by measuring the degree of content similarity between a supplementary patch and its initial patch. The patches were extracted using svndiff. The cloning relationship between an initial patch and its supplementary patch was identified by Repertoire (Ray and Kim 2012b), which was built on CCFinderX (Kamiya et al. 2002) to analyze ported changes with a minimum size of five tokens.

We determined how often supplementary patches involved the porting of patches to different branches or different components. Although our previous work (Park et al. 2012)

considered only backporting, that is, porting the same or similar patches to different branches, this study considers cases involving the porting of the same or similar patches to either different branches *or different components*.

We identified a supplementary patch as a porting patch if the following conditions held. *1) more than 80 % of the lines of the added code were detected as a clone of the corresponding initial patches*; and *2) the files on supplementary patches had the same name as the files on the corresponding initial patches, but the patch was applied to another branch or another component*. We used a high threshold value (80 % of lines should be similar to corresponding initial patches) to reduce false positives and to consider porting patches that were almost identical to only the corresponding initial patches. The ported patches account for 4 % to 11 %, as shown in Fig. 5.

Table 11 shows the top five most frequent source and target branches of porting in each project. In Eclipse JDT core, the developers generally ported patches from the trunk to maintenance branches, while some patches were ported from a maintenance branch to the trunk. In Eclipse SWT, porting from and to different components was relatively more common than in other projects because the Eclipse SWT project consists of several components for different libraries (e.g., win32, carbon, gtk, cocoa, etc.) In Equinox p2, only four source and target branches of porting occurred more than two times.

If we excluded supplementary patches that were involved in porting codes to different branches or different components, only 7 % to 14 % of supplementary patches included at least five lines similar to their initial patches, as shown in Fig. 5. Overall, 75 % to 86 % of the supplementary patches, the majority, were not similar to the initial patches. These results did not support the conventional wisdom that clone management can significantly prevent or reduce omission errors. The results also revealed the inadequacy of existing change recommendation systems based on clone detection analysis alone (Duala-Ekoko and Robillard 2007; Nguyen et al. 2009, 2010).

4% to 11% of supplementary patches involve porting an initial patch to a different branch or component. Only 7% to 14% of supplementary patches have content similar to their corresponding initial patches.

4.6 Where are the Locations of Supplementary Patches with Respect to the Initial Patches?

We determined how the location of an initial patch was related to the location of a supplementary patch at the file level in *Java subject programs* to obtain insights into the prediction

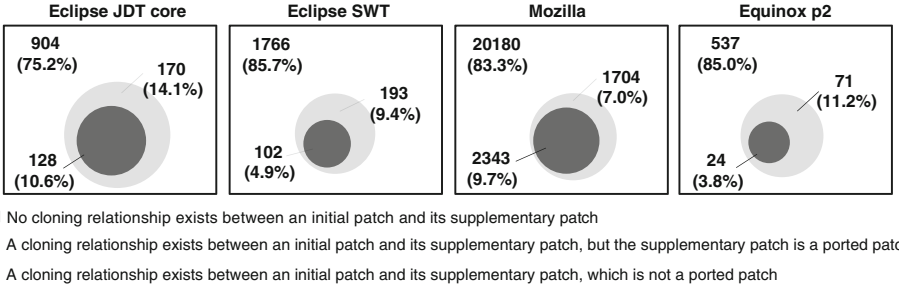

Fig. 5 The percentage of cloned patches and backported patches of all supplementary patches

Table 11 The most common porting source and targets

Eclipse JDT core	trunk → branches/R3_1_maintenance
	trunk → branches/R3_2_maintenance
	trunk → branches/R3_3_maintenance
	branches/R3_1_maintenance → trunk
	trunk → branches/TARGET_321
Eclipse SWT	trunk → branches/R3_4_maintenance
	trunk → branches/R3_2_maintenance
	trunk → branches/R3_3_maintenance
	win32 → carbon
	win32 → gtk
Mozilla	trunk → branches/MOZILLA_1_8_BRANCH
	trunk → branches/MOZILLA_1_0_BRANCH
	trunk → branches/MOZILLA_1_7_BRANCH
	trunk → branches/MOZILLA_0_9_2_BRANCH
	trunk → branches/MOZILLA_1_8_0_BRANCH
Equinox p2	branches/v20080930 → branches/R3_4_1
	branches/v20080722 → branches/v20080930
	branches/R3_4_1 → branches/v20080930
	branches/v20080930 → branches/v20080605-1731

of supplementary patch locations given an existing change set. The files fixed in an initial patch could be modified again during a supplementary patch, or new files could be fixed instead.

As shown in Fig. 6, 29 % to 55 % of the supplementary patches were applied to the same files to which the initial patches were applied. We determined the changes made to similar line locations using the heuristic developed by Yin et al. (2011); i.e., at least one line change was made within 25 lines of at least one changed line in the original patch. Because we used the line number from the patches, blank lines were also included. Of the supplementary patches, 27 % to 50 % were made within 25 lines of the initial patch locations.

We identified the structural dependence relationship between the files of initial patches and the files of supplementary patches by using LSDiff to extract instances of structural dependence among the source files. LSDiff infers systematic structural differences as a logic rule (Kim and Notkin 2009; Loh and Kim 2010), and represents each program version using a set of logic facts, such as call (callerMethodName, calleeMethodName) and

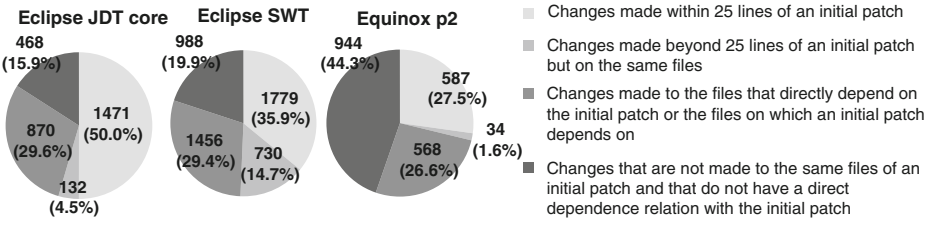

Fig. 6 16 % to 45 % of the files neither overlap an initial patch location nor have direct dependencies on them

access (fieldName, accessorMethodName), etc. Using this technique, we extracted structural facts for every released version of Eclipse JDT core, Eclipse SWT, and Equinox p2. We then mapped the facts to file-level dependence relationships. For example, the first patch of bug 111618 was in the class `ForeachStatement` and the third patch was in the class `AbstractVariableDeclaration.` The class `ForeachStatement` called a method named `printAsExpression` in `AbstractVariableDeclaration`, producing a file-level dependence relationship between the initial patch and the supplementary patch. Of all the files of the supplementary patches, 27 % to 30 % were directly related to at least one file in the initial patch.

We also investigated the indirect dependence relationships between the files of the supplementary patches and the files of the initial patches (see Table 12). A sibling relationship indicated that the two files extended the same ancestor type, and indirect call/access meant that a file was indirectly called or accessed via a different file. Overall, 16 %, 21 %, and 45 % of the files in the supplementary patches in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively, did not overlap with or have any direct relationship to the files modified in the initial patch. This result indicates that change recommendation systems suggesting the direct dependence neighbors of an existing change set are not sufficient to reduce omission errors.

16% to 45% of supplementary patch locations were beyond the scope of the direct neighbors of the initial patch locations.

4.7 Can a Historical Co-change Pattern Analysis Predict Supplementary Patch Locations?

Previous studies found that additional change locations could be identified using historical co-change patterns (Hassan and Holt 2004; Zimmermann et al. 2004). We determined whether supplementary patch locations could be identified by using historical co-change patterns. We first determined whether change coupling existed between supplementary patch locations and their initial patch locations in one of two ways: 1) they were changed together recently *(within n days)*; or 2) they were changed together frequently *(more than n commit transactions)* before each initial fix date. We considered only the files of

Table 12 The structural dependence relationships between an initial patch and its supplementary patch at the file level

	Relation type	Eclipse JDT core		Eclipse SWT		Equinox p2	
Direct dependence relations	Call	746	(25.4 %)	1284	(28.7 %)	527	(25.1 %)
	Access	619	(21.1 %)	1148	(25.6 %)	58	(2.8 %)
	Return	210	(7.1 %)	317	(7.1 %)	18	(0.9 %)
	Fieldoftype	206	(7.0 %)	311	(6.9 %)	110	(5.2 %)
	Extend	95	(3.2 %)	107	(2.4 %)	64	(3.0 %)
	Implement	28	(1.0 %)	4	(0.1 %)	7	(0.3 %)
Indirect dependence relations	Sibling	133	(4.5 %)	165	(3.7 %)	160	(7.6 %)
	Indirect call	160	(5.4 %)	91	(2.0 %)	144	(6.9 %)
	Indirect access	115	(3.9 %)	41	(0.9 %)	3	(0.1 %)
	Other	418	(14.2 %)	427	(9.5 %)	820	(39.0 %)

supplementary patches that did not appear in the files of the corresponding initial patches.

Figure 7 sows the proportion of supplementary patch locations that could be predicted by historical co-change patterns. The X axis represents the parameters, days and the number of times that the files changed together, and the Y axis represents the proportions of supplementary patch locations of all files fixed in the supplementary patches. The results showed that 17 % to 38 % of supplementary patch locations were changed together with the corresponding initial patch locations within 50 days of the initial patch. Overall, 35 % to 80 % of the supplementary patch locations in Eclipse JDT core, Eclipse SWT, Mozilla, and Equinox p2, respectively, were co-changed with the initial patch locations at least once. This finding indicated that about 20 % to 65 % of the supplementary patch locations had not been co-changed with the initial patch locations, and they could not be predicted using historical co-change patterns.

We also investigated the accuracy of predictions based on the historical co-change pattern. We identified a set of files that had been co-changed with the files of an initial patch *2) within n days* and *1) more than n times*. In this set, we identified a set of files that appeared in the supplementary patch. The equation of accuracy is shown below.

$$Accuracy = \frac{|S \cap C|}{|C|}$$

where S = A set of files that appear in supplementary patches, and C = A set of files that were co-changed with the location of the corresponding initial patch.

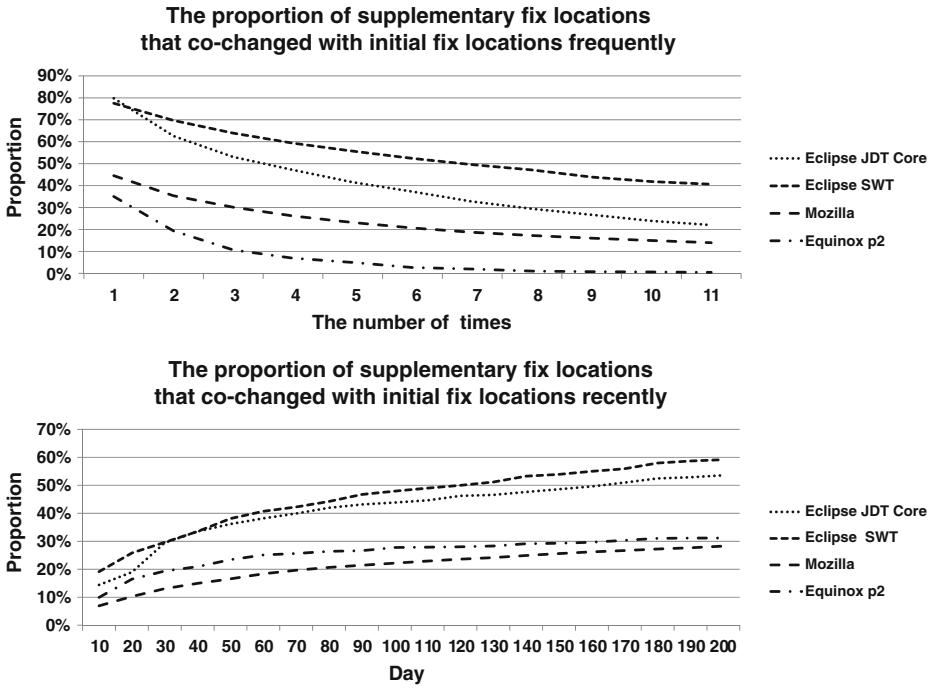

Fig. 7 The proportion of supplementary patch locations that co-changed with initial fix locations

We calculated the average accuracy for multi-fix bugs. Figure 8 shows the results of the accuracy analysis. As the number of suggested entities decreased, the prediction accuracy increased. However, the accuracies were less than 10 %. This low level of accuracy indicated that less than 10 supplementary patch locations would appear among a hundred of files that were suggested by the historical co-change pattern based prediction method.

Some of the supplementary patch locations were identified using historical co-change pattern based prediction, but the accuracy was very low.

4.8 Do Results of the Three Prediction Approaches Overlap?

We use code clone analysis, structural dependency analysis, and historical co-change analysis to predict the supplementary patch locations given the corresponding initial patch locations. We determined whether the results from these three prediction approaches were correlated or whether individual approaches suggested unique locations. At the file level,

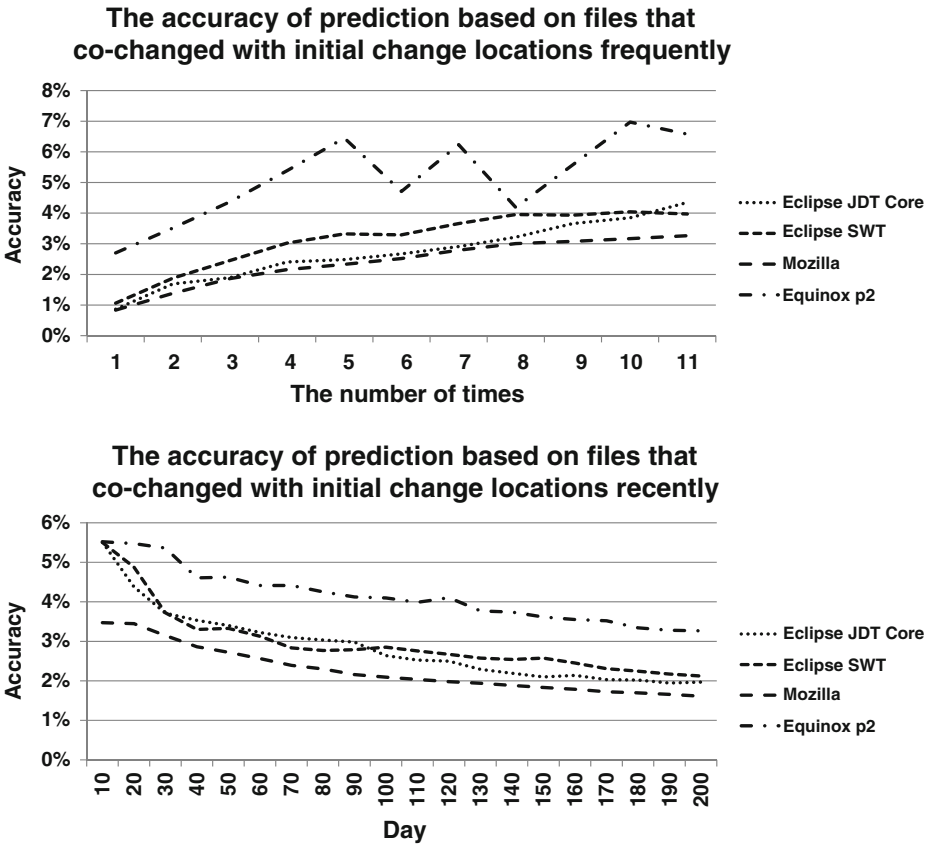

Fig. 8 The accuracy of prediction based on historical co-change patterns

three result sets were identified based on the results of Sections 4.5, 4.6, and 4.7 of Eclipse JDT core, Eclipse SWT, and Equinox p2. We used *five* lines as the code clone threshold, which indicated that the result set from the code clone analysis contained supplementary patch files that were similar to the initial patch in at least five lines. The structural dependency analysis identified a set of files that had a direct structural relationship with an initial patch file. The historical co-change analysis identified the supplementary patch files that had been co-changed with an initial patch file at least *twice* before the initial patch.

Figure 9 shows the number of overlapping results from the code clone, structural dependency, and co-change analyses. The number of files in the supplementary patches that could be identified by each prediction approach are shown in the figure. For example, in Eclipse JDT core, 10 files in the supplementary patches were identified using any of the three prediction approaches, and 257 files were identified by the historical co-change analysis alone. The percentages shown in parentheses indicate the proportion of all the supplementary patch files.

The results of the code clone analysis rarely overlapped the results of the structural dependency or co-change analyses. Because the structural dependency and historical co-change analyses identified the relationship between two distinct locations, the results from these two analyses do not contain the same locations as the initial patch locations. Of the cloning based relationships, 75 % to 95 % appeared in the same file path as the initial patch. Most of the cloning based relationships were related to ported patches (i.e., in the same file path but on a different branch or component) or missed updates of code clones within the same file. A fair amount (10 % to 25 %) of the results of structural dependency analysis and the co-change analysis overlap, but both two techniques alone were responsible for 16 % to 22 % of the supplementary patch files.

Among all supplementary patches, 39 % to 53 % of supplementary patch files were identified by the three prediction approaches. In conclusion, the overlap of each analysis was very small, indicating that each type of analysis was necessary. Further, combining three types of analysis alone could not predict all supplementary patch locations. This result indicates that there is a need for a different type of analysis in addition to the present approach.

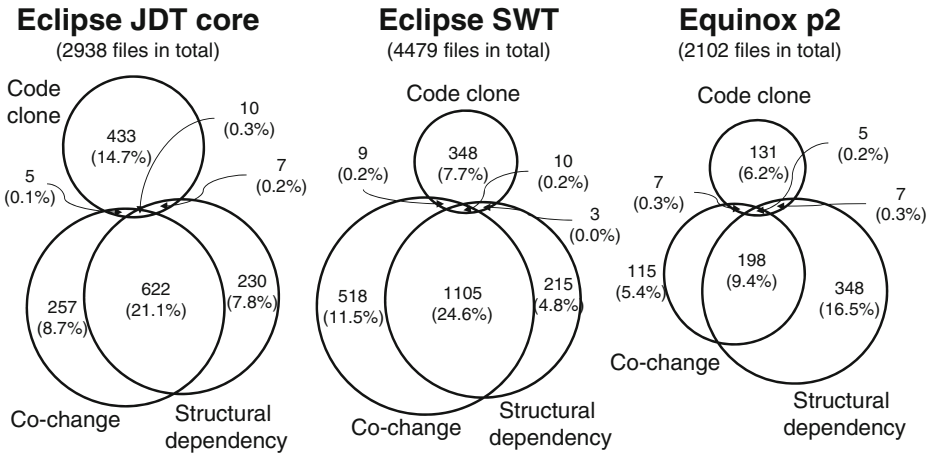

Fig. 9 The number of overlapping results from code clone, structural dependency, and co-change analyses

Table 13 The extent of single-fix and multi-fix bugs in open source projects

Project name	Bug identifier	# of single-fix bugs	# of multi-fix bugs	# of total bugs
activemq	AMQ	2045 (73.1 %)	752 (26.9 %)	2797
ambari	AMBARI	9149 (90.5 %)	959 (9.5 %)	10108
camel	CAMEL	4404 (66.5 %)	2218 (33.5 %)	6622
cassandra	CASSANDRA	4042 (81.2 %)	937 (18.8 %)	4979
commons-lang	LANG	303 (67.9 %)	143 (32.1 %)	446
commons-math	MATH	486 (60.6 %)	316 (39.4 %)	802
flink	FLINK	584 (80.3 %)	143 (19.7 %)	727
hadoop	HADOOP	2321 (86.9 %)	350 (13.1 %)	2671
karaf	KARAF	1460 (76.2 %)	457 (23.8 %)	1917
spark	SPARK	3571 (87.4 %)	514 (12.6 %)	4085
zookeeper	ZOOKEEPER	933 (91.5 %)	87 (8.5 %)	1020
jbosstools-base	JBIDE	1114 (71.5 %)	443 (28.5 %)	1557
spring-framework	SPR	2787 (73.0 %)	1032 (27.0 %)	3819
spring-roo	ROO	1421 (68.3 %)	661 (31.7 %)	2082

Results of code clone, structural dependency, and historical co-change analyses rarely overlapped, and combining these analyses was also insufficient to predict supplementary patch locations.

5 Discussion

External Validity We studied Eclipse JDT core, Eclipse SWT, Mozilla, and Equinox p2 as representatives of open source projects, but the study results may not be generalized to other projects. Table 13 shows the extent of the multi-fix bugs in many open source projects from Apache,[7] JBoss,[8] and Spring,[9] which use JIRA as their bug tracking system. We tracked the bug identifier and the bug identification number for JIRA (e.g., LANG-123 for the Commons-Lang project) in the commit logs. The extent of multi-fix bugs was 9 % to 39 %, an average of 23 %. This result shows that a considerable portion of bugs were not fixed in the first fix attempt, and that applying a supplementary patch was a common phenomenon in the open source projects. However, the other results of this study, such as the severity levels and the relationship between initial and supplementary patches, may not be generalized to other projects. An in-depth study of further projects will be conducted in future work.

Misclassification of Supplementary Patches We assessed our supplementary patch identification method by manually inspecting one-month periods for Eclipse JDT core,

[7] https://projects.apache.org/

[8] http://www.jboss.org/projects/

[9] https://spring.io/projects

Eclipse SWT, and Equinox p2 (107, 165, and 193 commits, respectively), and a two week period for Mozilla (786 commits). We used a relatively short period for the Mozilla project because it had too many commits. This two-week period yielded 786 commits, the most highest number our subject programs. One threat to validity is that in manual inspection, building the ground truth set might not cover all incomplete patches in the period. Our study revealed that the three criteria for the ground truth set were not sufficient to cover the relationship between the incomplete patches and the supplementary patches.

Overall, the precision of our incomplete patch identification method was 100 % in all projects. Hence, the number recorded in the commit log generally indicated a bug ID. However, the recall rates were 90 % on average because the developers did not always include a bug ID when they applied a bug fix. The method used to identify supplementary patch was fairly accurate in terms of precision, but we may have missed some supplementary patches that were not managed by the bug reporting system; several patches may not have been linked to bug reports (Bird et al. 2009). Furthermore, it is possible that some developers did not recognize that they fixed the same bug when they applied a supplementary patch.

In addition, Herzig et al. (2013) also showed that many non-bugs are misclassified as bugs, and they claimed that the automated quantitative analysis of a bug data set should consider the effects of misclassification. According to our definition of a supplementary patch, non-fixing changes could be included in our data set. In Section 4.4, the results of the manual inspection showed that missed porting changes or incomplete refactoring induced supplementary patches. Herzig et al. did not regard these changes as bug fixes. The accuracy of the prediction approaches based on code clone, structural dependency, and historical co-change analyses could be affected by the non-fixing changes, which then would threaten the validity.

Confounding Factors Pertaining to the Difference Between Single-Fix Bugs and Multi-fix Bugs Compared to single-fix bugs, multi-fix bugs have larger patch sizes, involve more people, and require more time to resolve. By sampling several bugs, we found that multi-fix bugs frequently involved incomplete specifications, which could cause a reassignment of the task or involve additional discussions among relevant developers. Reassignment processes and discussion processes to fix the requirement could require the involvement of another developer and additional time and effort for discussion. Because multi-fix bugs are more likely to be difficult to fix compared to single-fix bugs, they may require longer patches than single-fix bugs require.

Identification of Cloned Patches In our study, a supplementary patch is regarded as a cloned patch when its code chunk is similar to the code in the initial patch, and the code chunk is longer than five lines. However, it is possible for only a few lines in a supplementary patch to be similar to the initial patch.

Figure 10 shows the percentage of cloned patches while varying the threshold. In Eclipse JDT core, 15.0 % of the supplementary patches were classified as cloned patches within a threshold of five lines. When the threshold of ten lines was used, only 6.5 % of the supplementary patches were identical. In addition, it is possible that recent clone detection techniques (Jiang et al. 2007; Pham et al. 2009) could be used instead.

Relationships Among Supplementary Patches In our study, we considered only the first commit an incomplete (initial) patch, and we investigated the relationships between the incomplete patch and corresponding supplementary patches. If there were three patches for a multi-fix bug, then only the first patch was regarded as an incomplete patch. However,

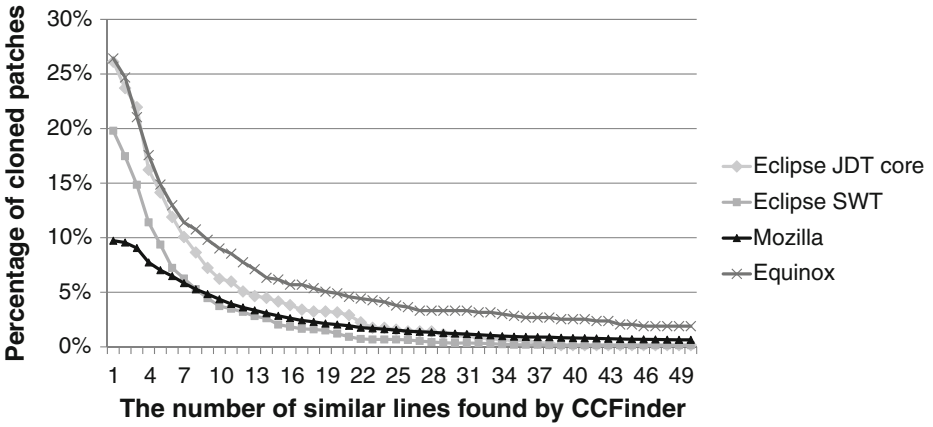

Fig. 10 The percentage of supplementary patches that are at least X lines similar to their initial patches according to Repertoire, while varying X

the second patch also could be regarded as incomplete because it was complemented by the third patch. Developers could make a supplementary patch to complement the previous incomplete patch, but the supplementary patch also can be an incomplete patch. An investigation of the relationships between consecutive supplementary patches will be conducted in future work.

Statistical Significance of the Results We used t-tests to justify the statistical significance of our results. If the sample size is large enough, even when two distributions are similar, a t-test can determine that there is a significant difference. In this case, we used an effect size measure, such as the Cliff's delta, to compare two distributions statistically. Cliff's delta measures the probability that a value from group A (size n) is greater than a value from group B (size m) by comparing each element ($n \times m$ times) (Cliff 1996).

We calculated Cliff's delta between the number of changed files of regular and incomplete patches in Eclipse SWT. As shown in Table 14, there was a probability of 39.2 % that the number of changed files of incomplete patches was greater than that of regular patches. Although incomplete patches are more likely have a greater number of changed files, the Cliff's delta value (0.1517) was relatively *small* (Grissom and Kim 2005).

Validity of Manual Inspection In Section 4.4, we manually inspected the causes of incomplete patches. The first author initially categorized the causes of incomplete patches, after which two graduate students, who were studying software engineering, validated the

Table 14 Cliff's delta between the number of changed files of regular and incomplete patches in Eclipse SWT

Cliff's delta = 0.1517		
$x_r < x_i$	$x_r = x_i$	$x_r > x_i$
39.20 %	36.77 %	24.03 %

x_r: a value from a regular patch

x_i: a value from an incomplete patch

Table 15 An example of a tangled change that a supplementary patch applied later

Bug 81571 of Eclipse JDT core			
Revision	Date	Author	Comment
10294	2005/01/06	Kent	81824 + 81571 + 81568
10301	2005/01/07	Kent	81571

categorization. For bugs where our classifications diverged, we reached a consensus to classify the bug into the correct category. Although we endeavored to obtain accurate categorizations, threats to validity still exist, such as subjectivity and incorrect classifications. In addition, because we selected 200 samples of incomplete patches, categories that were not covered by the samples might have been missed.

Effect of Tangled Changes Herzig and Zeller (2013) addressed the possible effects of tangled changes, in which code changes were related to more than one bug report. Recent researches on change decomposition (Tao and Kim 2015; Barnett et al. 2015) also pointed out that a portion of changes contained composite changes, which were related to more than one issue. If irrelevant issues were tangled into a patch, it could be a noise for identifying the relationship between the initial patches and the supplementary patches. We find that 3 % to 5 % of commits were connected to more than one bug report, and this noise from the tangled changes could be a threat to validity.

An interesting case that a tangled change might affect the incomplete patch is shown in Table 15. A developer first tried to fix three bugs (81824, 81571, and 81568) at the same time, and a supplementary patch for only one bug (81517) was applied later. These tangled changes might cause incomplete patches because the developer tried to handle multiple issues at the same time. The in-depth investigation of the relationship between tangled changes and incomplete patches requires future work.

Roles and Experience of Developers It is possible that the roles of developers are different. Hence, more than one developer should be involved in fixing corresponding parts such as GUI or test codes. We found that supplementary patches were usually written by the same author who wrote initial patch; 73 % to 85 % of multi-fix bugs were written by the same author. The authors usually participated in the discussion of the bug report; however, we could not find evidence that the roles of developers were separate.

We also found that the authors of single fix bugs made 6 % to 8 % more commits before the fix commits in Eclipse JDT core, Eclipse SWT, and Equinox p2, which indicated that the incomplete patches were made by less experienced developers. In Mozilla, however, the authors of the multi-fix bugs were more experienced than the authors of the single-fix bugs. The experience of developers can be measured in several ways, such as the total number of program lines that a developer has made (Mockus and Weiss 2000; Rahman and Devanbu 2011) or the amount of time since a developer's first commit (Eyolfson et al. 2011). Further investigations of the relationship between the roles and experience of developers and supplementary patches will be studied in future work.

6 Conclusions

Developers occasionally make incomplete patches, and unintended omission errors require supplementary patches. Many approaches have been proposed to reduce incomplete patches

by recommending supplementary change locations. These approaches include assumptions about the characteristics of omission errors. In this study, we investigated the reasons that incomplete patches occur in practice and how such errors could be prevented by examining the characteristics of incomplete patches and supplementary patches.

Our study of four open source projects showed that a considerable proportion of bugs required supplementary patches. The quality of the reports on multi-fix bugs were not lower than that of single-fix bugs, but they contained more attribute changes because of the incorrect information about multi-fix bugs. The results of our study indicate that multi-fix bugs are likely to have higher severity levels than single-fix bugs because they are often related to important and complicated parts of a program. Although more developers actively participate in multi-fix bugs than in single-fix bugs, the multi-fix bugs takes more time to resolve than single-fix bugs do. Furthermore, incomplete patches tended to be larger and more scattered than regular patches. The common causes of omission errors are diverse, including missed porting changes, the incorrect handling of conditional statements, or incomplete refactoring. In contrast to the conventional wisdom that missed updates of code clones often cause omission errors, the findings of this study showed that only a small number of supplementary patches had content similar to their initial patches. Moreover, structural dependency analysis alone is insufficient to predict supplementary patch locations, so as historical co-change analysis. The findings showed that the three code clone, structural dependency, and historical co-change analyses predicted different locations with little overlap. However, combining these analyses is still not sufficient to predict all supplementary change locations. Because existing change recommendation approaches are insufficient in identifying the relationship between initial and supplementary patch locations and since the causes of incomplete patches are diverse, new approaches to prevent each type of real-world incomplete patches should be developed and validated using a supplementary patch data set.

Acknowledgments This work was supported in part by the National Science Foundation under grants CCF-1149391, CNS-1239498, and a Google Faculty Award. This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. R0126-16-1101, (SW Star Lab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System)

Appendix

The data set and soucre code for the analysis in this paper are available on the first author's web page [10]. In this appendix section, we compare the results from our previous work published in MSR 2012 (Park et al. 2012), and the extended results for this journal revision.

We extend our study period from two years in each project to five, six, and nine years in Eclipse JDT core, Eclipse SWT, and Mozilla, respectively. All analyses including bug extent analysis, bug severity analysis, the similarity analysis between the initial and supplementary patches, and the location analysis for the initial and supplementary patch locations are redone for the extended periods, allowing us to generalize our results to the long-term history of the development.

[10] http://se.kaist.ac.kr/jhpark

Table 16 Comparison of the results from our MSR paper and this journal paper—the extent of single-fix bugs and multi-fix bugs

		Study period	# of bugs	# of single-fix bugs	# of multi-fix bugs
MSR	Eclipse JDT core	2004/07 ∼ 2006/07	1812	1405 (77.5 %)	407 (22.5 %)
	Eclipse SWT	2004/07 ∼ 2006/07	1256	954 (76.0 %)	302 (24.0 %)
	Mozilla	2003/04 ∼ 2005/07	11254	7562 (67.2 %)	3692 (32.8 %)
Journal	Eclipse JDT core	2002/01 ∼ 2007/12	3677	2836 (77.1 %)	841 (22.9 %)
	Eclipse SWT	2002/01 ∼ 2008/12	3992	2959 (74.1 %)	1033 (25.9 %)
	Mozilla	2000/01 ∼ 2009/12	42283	28034 (66.3 %)	14249 (33.7 %)

Tables 16, 17, 18 and 19 show the comparison results. The number of bugs increases as we extend our study period, but the proportions of single-fix bugs (Type I bugs) and multi-fix bugs (Type II bugs) remain similar. The severity analysis—the percentage of bugs with

Table 17 Comparison of the results from our MSR paper and this journal paper—the percentage of bugs with Blocker/Critical severity, the number of developers involved, and the days taken to resolve each bug

		Blocker/critical bugs (S-fix)	Blocker/critical bugs (M-fix)
MSR	Eclipse JDT core	3.63 %	4.91 %
	Eclipse SWT	5.66 %	9.27 %
	Mozilla	11.24 %	12.71 %
Journal	Eclipse JDT core	3.91 %	5.58 %
	Eclipse SWT	6.20 %	9.86 %
	Mozilla	15.19 %	19.07 %
		# developers involved (S-fix)	# developers involved (M-fix)
MSR	Eclipse JDT core	3.67	4.44
	Eclipse SWT	3.13	4.29
	Mozilla	4.70	7.28
Journal	Eclipse JDT core	3.08	3.97
	Eclipse SWT	2.73	4.06
	Mozilla	5.08	7.45
		# days to resolve bugs (S-fix)	# days to resolve bugs (M-fix)
MSR	Eclipse JDT core	120.79	188.27
	Eclipse SWT	176.99	337.32
	Mozilla	594.50	805.92
Journal	Eclipse JDT core	103.40	165.05
	Eclipse SWT	153.94	292.97
	Mozilla	700.51	822.35

Table 18 Comparison of the results from our MSR paper and this journal paper—the percentage of files in the supplementary patches similar to initial patches

		Class 1	Class 2	Class 3
MSR	Eclipse JDT core	424 (70.2 %)	108 (17.88 %)	72 (11.9 %)
	Eclipse SWT	392 (68.5 %)	35 (6.1 %)	145 (25.4 %)
	Mozilla	5477 (78.8 %)	858 (12.3 %)	620 (8.9 %)
Journal	Eclipse JDT core	904 (75.2 %)	128 (10.6 %)	170 (14.1 %)
	Eclipse SWT	1766 (85.7 %)	102 (4.9 %)	194 (9.4 %)
	Mozilla	20180 (83.3 %)	2343 (9.7 %)	1704 (7.0 %)

Class1: No cloning relationship exists between an initial patch and its supplementary patch

Class2: A cloning relationship exists between an initial patch and its supplementary patch, but the supplementary patch is a ported patch

Class3: A cloning relationship exists between an initial patch and its supplementary patch, which is not a ported patch

Table 19 Comparison of the results from our MSR paper and this journal paper—the relationship between initial patch locations and supplementary patch locations

		Class 1	Class 2	Class 3	Class 4
MSR	Eclipse JDT core	504 (48.1 %)	51 (4.9 %)	335 (32.0 %)	158 (15.1 %)
	Eclipse SWT	454 (41.5 %)	174 (15.9 %)	314 (28.7 %)	151 (13.8 %)
Journal	Eclipse JDT core	1469 (50.0 %)	132 (4.5 %)	869 (29.6 %)	468 (15.9 %)
	Eclipse SWT	1587 (35.4 %)	607 (13.6 %)	1333 (29.8 %)	952 (21.3 %)

Class1: Changes made within 25 lines of an initial patch

Class2: Changes made beyond 25 lines of an initial patch but on the same files

Class3: Changes made to the files that directly depend on the initial patch or the files on which an initial patch depends on

Class4: Changes that are not made to the same files of an initial patch and that do not have a direct dependence relation with the initial patch

Table 20 Categories of 200 sampled incomplete fix revisions

	#	Bug ID (Project)
1)	71	66512 (JDT), 91709 (JDT), 99903 (JDT), 104780 (JDT), 140879 (JDT), 148010 (JDT), 161557 (JDT), 171703 (JDT), 191739 (JDT), 201104 (JDT), 24542 (SWT), 30854 (SWT), 39223 (SWT), 39892 (SWT), 56780 (SWT), 76185 (SWT), 76391 (SWT), 83408 (SWT), 83819 (SWT), 85666 (SWT), 85962 (SWT), 87554 (SWT), 88829 (SWT), 89785 (SWT), 90856 (SWT), 97981 (SWT), 106289 (SWT), 107777 (SWT), 194146 (SWT), 236312 (SWT), 90064 (Mozilla), 118656 (Mozilla), 136580 (Mozilla), 155222 (Mozilla), 174132 (Mozilla), 193372 (Mozilla), 203041 (Mozilla), 203211

Table 20 (continued)

	#	Bug ID (Project)
		(Mozilla), 207673 (Mozilla), 208843 (Mozilla), 212112 (Mozilla), 220933 (Mozilla), 223111 (Mozilla), 224313 (Mozilla), 226600 (Mozilla), 228176 (Mozilla), 231166 (Mozilla), 261886 (Mozilla), 280740 (Mozilla), 301338 (Mozilla), 305041 (Mozilla), 307277 (Mozilla), 335366 (Mozilla), 337110 (Mozilla), 342922 (Mozilla), 358296 (Mozilla), 222969 (Equinox),227189 (Equinox), 227835 (Equinox), 233229 (Equinox),235496 (Equinox),235906 (Equinox), 240062 (Equinox),240254 (Equinox), 241185 (Equinox), 242632 (Equinox), 243422 (Equinox), 244630 (Equinox), 246432 (Equinox), 247177 (Equinox), 257242 (Equinox)
2)	39	73479 (JDT), 79772 (JDT), 80914 (JDT), 108372 (JDT), 120264 (JDT), 120640 (JDT), 123522 (JDT), 133071 (JDT), 23745 (SWT), 29642 (SWT), 57777 (SWT), 76750 (SWT), 81987 (SWT), 88463 (SWT), 90024 (SWT), 92230 (SWT), 93294 (SWT), 94003 (SWT), 94467 (SWT), 210825 (SWT), 220398 (SWT), 225351 (SWT), 255113 (SWT), 34373 (Mozilla), 77234 (Mozilla), 78809 (Mozilla), 82141 (Mozilla), 211267 (Equinox), 212647 (Equinox), 212651 (Equinox), 217492 (Equinox), 221573 (Equinox), 226344 (Equinox), 228730 (Equinox), 232315 (Equinox), 232440 (Equinox), 249215 (Equinox), 252449 (Equinox), 257602 (Equinox)
3)	29	80699 (JDT), 92315 (JDT), 92888 (JDT), 96763 (JDT), 105531 (JDT), 110422 (JDT), 111494 (JDT), 119844 (JDT), 139621 (JDT), 142772 (JDT), 208541 (JDT), 211872 (JDT), 70318 (SWT), 72401 (SWT), 81081 (SWT), 85386 (SWT), 85867 (SWT), 200144 (Mozilla), 211470 (Mozilla), 213910 (Mozilla), 216581 (Mozilla), 227432 (Mozilla), 390275 (Mozilla), 212305 (Equinox), 228406 (Equinox), 251561 (Equinox), 251772 (Equinox), 258130 (Equinox)
8)	3	42839 (JDT), 104664 (JDT), 125518 (JDT)
9)	3	118117 (Mozilla), 183399 (Equinox), 222260 (Equinox)
10)	2	81244 (JDT), 78554 (SWT)
11)	2	83593 (JDT), 126625 (JDT)
12)	14	114935 (JDT), 126180 (JDT), 183211 (JDT), 212222 (Mozilla), 22231 (SWT), 74095 (SWT), 91317 (SWT), 124965 (SWT), 180576 (Mozilla), 191749 (Mozilla), 215587 (Mozilla), 225570 (Mozilla), 206818 (Equinox), 246822 (Equinox)
4)	13	73330 (JDT), 98906 (JDT), 110048 (JDT), 141289 (JDT), 77102 (SWT), 85069 (SWT), 235832 (Mozilla), 246966 (Mozilla), 251337 (Mozilla), 289558 (Mozilla), 99168 (Mozilla), 210484 (Equinox), 236077 (Equinox)
5)	11	212361 (Equinox), 212476 (Equinox), 226929 (Equinox), 229205 (Equinox), 235165 (Equinox), 249318 (Equinox), 249483 (Equinox), 250164 (Equinox), 255623 (Equinox), 255820 (Equinox), 258370 (Equinox)
6)	7	100636 (JDT), 117302 (JDT), 148224 (JDT), 75148 (SWT), 110559 (SWT), 223435 (Mozilla), 287052 (Mozilla)
7)	6	86580 (JDT), 110797 (JDT), 122442 (JDT), 130390 (JDT), 243392 (Mozilla), 208343 (Equinox)

An initial patch (revision 10181 in Eclipse JDT core)

```
Index: .../compiler/classfmt/ClassFileReader.java
===================================================
@@ -584,6 +584,7 @@
  */
 public int getKind() {
    int modifiers = getModifiers();
+   if ((modifiers & AccAnnotation) != 0)
       return IGenericType.ANNOTATION_TYPE_DECL;
    if ((modifiers & AccInterface) != 0)
       return IGenericType.INTERFACE_DECL;
...
```

A supplementary patch (revision 10189 in Eclipse JDT core)

```
Index: .../compiler/classfmt/ClassFileReader.java
===================================================
@@ -584,16 +584,19 @@
  */
 public int getKind() {
    int modifiers = getModifiers();
+   if ((modifiers & AccInterface) != 0) {
+       if ((modifiers & AccAnnotation) != 0)
+           return IGenericType.ANNOTATION_TYPE_DECL;
+       return IGenericType.INTERFACE_DECL;
+   }
...
```

Fig. 11 The conditional statement of an initial fix is not correct

Blocker/Critical severity, the number of developers involved, and the days taken to resolve each bug—also show trends similar to those noted in the previous results. For the similarity analysis, the proportions of cloned patches and ported patches decrease, confirming that

An initial patch (revision 1769 in Equinox p2)

```
Index: .../equinox/internal/p2/engine/SimpleProfileRegistry.java
===================================================
@@ -166,7 +166,7 @@
   if (SELF.equals(id))
      id = self;
   Profile profile = (Profile) getProfileMap().get(id);
+  if (profile == null && self != null && self.equals(id))
      profile = createSurrogateProfile(id);

   return profile;
```

A supplementary patch (revision 1772 in Equinox p2)

```
Index: .../equinox/p2/tests/engine/ProfileRegistryTest.java
===================================================
@@ -82,6 +82,10 @@
...
+  public void testNoSelfProfile() {
+     assertNull(registry.getProfile(IProfileRegistry.SELF));
+  }
```

Fig. 12 The test code is added or updated

An initial patch (revision 156498 in Mozilla)

```
Index: trunk/mozilla/content/base/src/nsContentSink.cpp
===================================================================
@@ -236,7 +236,7 @@
...
+   if (mParser && aWasPending && aResult != NS_BINDING_ABORTED) {
     // Loading external script failed!. So, resume
     // parsing since the parser got blocked when loading
     // external script. - Ref. Bug: 94903
```

A supplementary patch (revision 159594 in Mozilla)

```
Index: trunk/mozilla/content/base/src/nsContentSink.cpp
===================================================================
@@ -238,9 +238,14 @@
...
+     // Loading external script failed!. So, resume parsing since the parser
+     // got blocked when loading external script. See
+     // http://bugzilla.mozilla.org/show_bug.cgi?id=94903.
+     //
+     // XXX We don't resume parsing if we get NS_BINDING_ABORTED from the
+     //     script load, assuming that that error code means that the user
+     //     stopped the load through some action (like clicking a link). See
+     //     http://bugzilla.mozilla.org/show_bug.cgi?id=243392.
```

Fig. 13 The comment is improved to explain an initial patch in detail

code clone analysis is insufficient to identify supplementary patch locations. The results from the location analyses also show similar trends.

Table 20 shows the categories of 200 sampled incomplete fix revisions. Figures 3, 11, 12, 13, 14, 15 and 16 show the code snippets for the categorizations of our manual inspection. For the contexts of these table and figures, please refer to Section 4.4.

An initial patch (revision 11796 in Eclipse JDT core)

```
Index: .../compiler/batch/ClasspathJar.java
===================================================================
@@ -23,16 +23,17 @@
...
 public class ClasspathJar extends ClasspathLocation {
+private File file;
+private ZipFile zipFile;
...
 public ClasspathJar(File file) throws IOException {
+        this(file, true, null);
 }
```

A supplementary patch (revision 11817 in Eclipse JDT core)

```
Index: .../compiler/batch/ClasspathJar.java
===================================================================
@@ -88,10 +88,13 @@
...
    this.packageCache = null;
 }
 public String toString() {
+   return "Classpath for jar file " + this.file.getPath(); //$NON-NLS-1$
 }
 public String normalizedPath(){
+   String rawName = this.file.getPath();
    return rawName.substring(0, rawName.lastIndexOf('.'));
 }
```

Fig. 14 Incomplete refactoring induces a supplementary patch

An initial patch (revision 71 in Equinox p2)

```
Index: .../internal/simpleconfigurator/utils/EquinoxUtils.java
===================================================================
@@ -0,0 +1,52 @@
...
+ URL configURL = new URL(baseURL,SimpleConfiguratorConstants
   CONFIGURATOR_FOLDER + "/" + SimpleConfiguratorConstants.CONFIG_LIST);
+ File configFile = new File(configURL.getFile());
+ if (configFile.exists())
+  return configURL;
...
```

A supplementary patch (revision 75 in Equinox p2)

```
Index: .../internal/simpleconfigurator/utils/SimpleConfiguratorConstants.java
===================================================================
@@ -42,7 +42,7 @@
...

+ public static final String CONFIGURATOR_FOLDER =
+             "org.eclipse.equinox.simpleconfigurator"; //$NON-NLS-1$
...
```

Fig. 15 Properties are updated

An initial patch (revision 10584 in Eclipse JDT core)

```
Index: .../compiler/problem/ProblemReporter.java
===================================================================
@@ -2895,10 +2895,8 @@
...
   } else if (location instanceof ArrayTypeReference) {
...
+     ArrayTypeReference arrayTypeReference = (ArrayTypeReference) location;
+     end = arrayTypeReference.originalSourceEnd;
...
```

A supplementary patch (revision 10585 in Eclipse JDT core)

```
Index: .../compiler/problem/ProblemReporter.java
===================================================================
@@ -2877,11 +2877,9 @@
...
   } else if (location instanceof
                          ArrayQualifiedTypeReference) {
...
+     ArrayQualifiedTypeReference arrayQualifiedTypeReference =
          (ArrayQualifiedTypeReference) location;
+     long[] positions = arrayQualifiedTypeReference.sourcePositions;
+     end = (int) positions[positions.length - 1];
...
```

Fig. 16 Two different parts calling different subclasses of the same type are not updated together

References

An L, Khomh F, Adams B (2014) Supplementary bug fixes vs. re-opened bugs. In: 2014 IEEE 14th international working conference on source code analysis and manipulation (SCAM). IEEE, pp 205–214

Andersen J, Lawall J (2008) Generic patch inference. In: ASE '08: Proceedings of the 23rd IEEE/ACM international conference on automated software engineering. IEEE Computer Society, Washington, DC, pp 337–346. doi:10.1109/ASE.2008.44

Barnett M, Bird C, Brunet J, Lahiri SK (2015) Helping developers help themselves: automatic decomposition of code review changesets. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering (ICSE), vol 1. IEEE, pp 134–144

Bettenburg N, Shang W, Ibrahim WM, Adams B, Zou Y, Hassan AE (2012) An empirical study on inconsistent changes to code clones at the release level. Sci Comput Program 77(6):760–776

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) 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. ACM, New York, pp 121–130. doi:10.1145/1595696.1595716
Cliff N (1996) Ordinal methods for behavioral data analysis. Routledge, Evanston
Čubranić D, Murphy GC (2003) Hipikat: recommending pertinent software development artifacts. In: Proceedings of the 25th international conference on software engineering. IEEE Computer Society, Washington, DC, ICSE '03, pp 408–418. http://portal.acm.org/citation.cfm?id=776816.776866
Duala-Ekoko E, Robillard MP (2007) Tracking code clones in evolving software. In: ICSE '07: Proceedings of the 29th international conference on software engineering. IEEE Computer Society, Washington, DC, pp 158–167. doi:10.1109/ICSE.2007.90
Eyolfson J, Tan L, Lam P (2011) Do time of day and developer experience affect commit bugginess? In: Proceedings of the 8th working conference on mining software repositories. ACM, pp 153–162
Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: ICSM '03: Proceedings of the international conference on software maintenance. IEEE Computer Society, Washington, DC, p 23
Fry ZP, Weimer W (2010) A human study of fault localization accuracy. In: ICSM '10: Proceedings of the 2010 IEEE international conference on software maintenance. IEEE Computer Society, Washington, DC, pp 1–10. doi:10.1109/ICSM.2010.5609691
Gamma E, Helm R, Johnson R, Vlissides JM (1994) Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional
Görg C, Weißgerber P (2005) Error detection by refactoring reconstruction. In: MSR '05: Proceedings of the 2005 international workshop on mining software repositories. ACM Press, New York, pp 1–5. doi:10.1145/1083142.1083148
Grissom RJ, Kim JJ (2005) Effect sizes for research: a broad practical approach. Lawrence Erlbaum Associates Publishers
Gu Z, Barr E, Hamilton D, Su Z (2010) Has the bug really been fixed? In: 2010 ACM/IEEE 32nd international conference on software engineering, vol 1, pp 55–64. doi:10.1145/1806799.1806812
Hassan AE (2009) Predicting faults using the complexity of code changes. In: ICSE '09: Proceedings of the 31st international conference on software engineering, IEEE Computer Society, Washington, DC, pp 78–88. doi:10.1109/ICSE.2009.5070510
Hassan AE, Holt RC (2004) Predicting change propagation in software systems. In: ICSM '04: Proceedings of the 20th IEEE international conference on software maintenance. IEEE Computer Society, Washington, DC, pp 284–293. doi:10.1109/ICSM.2004.1357812
Herzig K, Zeller A (2013) The impact of tangled code changes. In: 2013 10th IEEE working conference on mining software repositories (MSR). IEEE, pp 121–130
Herzig K, Just S, Zeller A (2013) It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 392–401
Hooimeijer P, Weimer W (2007) Modeling bug report quality. In: Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering. ACM, pp 34–43
Jiang L, Misherghi G, Su Z, Glondu S (2007) Deckard: scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th international conference on software engineering. IEEE Computer Society, Washington, DC, ICSE '07, pp 96–105. doi:10.1109/ICSE.2007.30
Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 28(7):654–670. doi:10.1109/TSE.2002.1019480
Kim M, Notkin D (2009) Discovering and representing systematic code changes. In: ICSE '09: Proceedings of the 2009 IEEE 31st international conference on software engineering. IEEE Computer Society, Washington, DC, pp 309–319. doi:10.1109/ICSE.2009.5070531
Kim S, Whitehead EJ Jr, Zhang Y (2008) Classifying software changes: clean or buggy? IEEE Trans Softw Eng 34(2):181–196. doi:10.1109/TSE.2007.70773
Kim M, Sinha S, Go Andrg C, Shah H, Harrold M, Nanda M (2010) Automated bug neighborhood analysis for identifying incomplete bug fixes. In: ICST '10: Proceedings of the third international conference on software testing, verification and validation. IEEE Computer Society, Washington, DC, pp 383–392. doi:10.1109/ICST.2010.63
Kim M, Cai D, Kim S (2011) An empirical investigation into the role of refactorings during software evolution. In: ICSE' 11: Proceedings of the 2011 ACM and IEEE 33rd international conference on software engineering. ACM, New York, pp 151–160. doi:10.1145/1985793.1985815
Loh A, Kim M (2010) Lsdiff: a program differencing tool to identify systematic structural differences. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering—volume 2. ACM, New York, ICSE '10, pp 263–266. doi:10.1145/1810295.1810348

Meng N, Kim M, McKinley KS (2011) Systematic editing: generating program transformations from an example. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation. ACM, New York, PLDI '11, pp 329–342. 10.1145/1993498.1993537

Meng N, Kim M, McKinley KS (2013) Lase: locating and applying systematic edits by learning from examples. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, Piscataway, ICSE '13, pp 502–511. http://dl.acm.org/citation.cfm?id=2486788.2486855

Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. In: ICSM '00: Proceedings of the international conference on software maintenance. IEEE Computer Society, Washington, DC, pp 120–130. doi:10.1109/ICSM.2000.883028

Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Tech J 5(2):169–180

Nagappan N, Zeller A, Zimmermann T, Herzig K, Murphy B (2010) Change bursts as defect predictors. In: Proceedings of the 2010 IEEE 21st international symposium on software reliability engineering. IEEE Computer Society, Washington, DC, ISSRE '10, pp 309–318. doi:10.1109/ISSRE.2010.25

Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi JM, Nguyen TN (2009) Clone-aware configuration management. In: ASE '09: Proceedings of the 2009 IEEE/ACM international conference on automated software engineering. IEEE Computer Society, Washington, DC, pp 123–134. doi:10.1109/ASE.2009.90

Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi J, Nguyen TN (2010) Recurring bug fixes in object-oriented programs. In: ICSE '10: Proceedings of the 32nd ACM/IEEE international conference on software engineering. ACM, New York, pp 315–324. doi:10.1145/1806799.1806847

Padioleau Y, Lawall J, Hansen RR, Muller G (2008) Documenting and automating collateral evolutions in linux device drivers. In: Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European conference on computer systems 2008. ACM, New York, pp 247–260. doi:10.1145/1352592.1352618

Park J, Kim M, Ray B, Bae DH (2012) An empirical study of supplementary bug fixes. In: MSR '12: 9th IEEE working conference on mining software repositories. IEEE Computer Society, Washington, DC, pp 40–49. doi:10.1109/MSR.2012.6224298

Park J, Kim M, Bae DH (2014) An empirical study on reducing omission errors in practice. In: Proceedings of the 29th ACM/IEEE international conference on automated software engineering. ACM, pp 121–126

Pham NH, Nguyen HA, Nguyen TT, Al-Kofahi JM, Nguyen TN (2009) Complete and accurate clone detection in graph-based models. In: Proceedings of the 31st international conference on software engineering. IEEE Computer Society, Washington, DC, ICSE '09, pp 276–286. doi:10.1109/ICSE.2009.5070528

Purushothaman R, Perry DE (2005) Toward understanding the rhetoric of small source code changes. IEEE Trans Softw Eng 31(6):511–526. doi:10.1109/TSE.2005.74

Rahman F, Devanbu P (2011) Ownership, experience and defects: a fine-grained study of authorship. In: Proceedings of the 33rd international conference on software engineering. ACM, pp 491–500

Rahman F, Bird C, Devanbu P (2012) Clones: what is that smell? Empir Softw Eng 17(4–5):503–530

Ray B, Kim M (2012a) A case study of cross-system porting in forked projects. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering. ACM, p 53

Ray B, Kim M (2012b) A case study of cross-system porting in forked projects. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering. ACM, New York, FSE '12, pp 53:1–53:11. doi:10.1145/2393596.2393659

Robillard MP (2005) Automatic generation of suggestions for program investigation. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on foundations of software engineering. ACM, New York, pp 11–20. doi:10.1145/1081706.1081711

Shannon CE (1949) Communication theory of secrecy systems*. Bell Syst Tech J 28(4):656–715

Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Ki M (2013) Studying re-opened bugs in open source software. Empir Softw Eng 18(5):1005–1042

Tao Y, Kim S (2015) Partitioning composite code changes to facilitate code review. In: 2015 IEEE/ACM 12th working conference on mining software repositories (MSR). IEEE, pp 180–190

Viera AJ, Garrett JM, et al. (2005) Understanding interobserver agreement: the kappa statistic. Fam Med 37(5):360–363

Wang X, Lo D, Cheng J, Zhang L, Mei H, Yu JX (2010) Matching dependence-related queries in the system dependence graph. In: ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, New York, ASE '10, pp 457–466. doi:10.1145/1858996.1859091

Yin Z, Yuan D, Zhou Y, Pasupathy S, Bairavasundaram L (2011) How do fixes become bugs? In: ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, New York. doi:10.1145/2025113.2025121

Ying ATT, Murphy GC, Ng R, Chu-Carroll M (2004) Predicting source code changes by mining change history. IEEE Trans Softw Eng 30(9):574–586

Zhang X, Tallam S, Gupta N, Gupta R (2007) Towards locating execution omission errors. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on programming language design and implementation. ACM, New York, pp 415–424. doi:10.1145/1250734.1250782

Zimmermann T, Weisgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: ICSE '04: Proceedings of the 26th international conference on software engineering. IEEE Computer Society, Washington, DC, pp 563–572. doi:10.1109/ICSE.2004.1317478

Zimmermann T, Weißgerber P, Diehl S, Zeller A (2005) Mining version histories to guide software changes. IEEE Trans Softw Eng 31(6):429–445. doi:10.1109/TSE.2005.72

Zimmermann T, Premraj R, Bettenburg N, Just S, Schroter A, Weiss C (2010) What makes a good bug report? IEEE Trans Softw Eng 36(5):618–643

Zimmermann T, Nagappan N, Guo PJ, Murphy B (2012) Characterizing and predicting which bugs get reopened. In: 2012 34th international conference on software engineering (ICSE). IEEE, pp 1074–1083

Jihun Park is a Ph.D. student in School of Computing at Korea Advanced Institute of Science and Technology. His research interests include mining software repositories, software evolution, and fault prediction. Park received B.S. and M.S. in Computer Science from Korea Advanced Institute of Science and Technology in 2009 and 2012 respectively.

Miryung Kim is an associate professor in the Department of Computer Science at the University of California, Los Angeles. She received her B.S. in Computer Science from Korea Advanced Institute of Science and Technology in 2001 and her M.S. and Ph.D. in Computer Science and Engineering from the University of Washington under the supervision of Dr. David Notkin in 2003 and 2008 respectively. Between January 2009 and August 2014, she was an assistant professor in the Department of Electrical and Computer Engineering at the University of Texas at Austin. She also spent time as a visiting researcher at the Research in Software Engineering (RiSE) group at Microsoft Research during the summer of 2011 and 2014. Her research interests are software evolution, and tool support for improving programmer productivity and software correctness.

Doo-Hwan Bae is a Professor in School of Computing at KAIST. He received a B.S. from Seoul National University in 1980, an M.S. degree in Computer Science from the University of Wisconsin-Milwaukee in 1987, and the Ph.D. in Computer and Information Sciences from the University of Florida in 1992. His research interest includes software design and analysis, software quality and process improvement, and model-based software engineering. He has served as General Chair of COMPSAC 2010 and QSIC 2009, PC Chair for APSEC 2004 and PC Co-Chair for COMPSAC 2003, and as President of the Korea Software Engineering Society.