An Empirical Study of Supplementary Bug Fixes

Jihun Park,* Miryung Kim,† Baishakhi Ray,† Doo-Hwan Bae*
*Korea Advanced Institute of Science and Technology
{jhpark, bae}@se.kaist.ac.kr
†The University of Texas at Austin
{miryung@ece., rayb@}utexas.edu

***Abstract*—A recent study finds that errors of omission are harder for programmers to detect than errors of commission. While several change recommendation systems already exist to prevent or reduce omission errors during software development, there have been very few studies on why errors of omission occur in practice and how such errors could be prevented. In order to understand the characteristics of omission errors, this paper investigates a group of bugs that were fixed more than once in open source projects—those bugs whose initial patches were later considered incomplete and to which programmers applied supplementary patches.**

Our study on Eclipse JDT core, Eclipse SWT, and Mozilla shows that a significant portion of resolved bugs (22% to 33%) involves more than one fix attempt. Our manual inspection shows that the causes of omission errors are diverse, including missed porting changes, incorrect handling of conditional statements, or incomplete refactorings, etc. While many consider that missed updates to code clones often lead to omission errors, only a very small portion of supplementary patches (12% in JDT, 25% in SWT, and 9% in Mozilla) have a content similar to their initial patches. This implies that supplementary change locations cannot be predicted by code clone analysis alone. Furthermore, 14% to 15% of files in supplementary patches are beyond the scope of immediate neighbors of their initial patch locations—they did not overlap with the initial patch locations nor had direct structural dependencies on them (e.g. calls, accesses, subtyping relations, etc.). These results call for new types of omission error prevention approaches that complement existing change recommendation systems.

***Keywords*-software evolution; empirical study; patches; bug fixes**

I. Introduction

According a recent study, developers are over five times more accurate at detecting errors of commission than errors of omission [1]. To prevent or reduce omission errors, several change recommendation systems [2]–[6] suggest additional change locations given a change set to reduce inconsistent or missing updates. These change recommendation systems make their own assumptions about which types of omission errors are common and how such errors could be reduced. For example, FixWizard suggests code clones of an existing bug fix to reduce potential missed updates [3]. As another example, Robillard uses the dependency structure of a change set to suggest where additional changes need to be made [4]. Zimmermann et al. and Ying et al. predict additional change locations based on historical co-change patterns derived from version histories [6], [7]. Hassan and Holt predict additional change locations based on co-change patterns and the dependency-graph of a change set in conjunction [2].

While these studies make various assumptions about how additional change locations can be recommended, there have been very few studies on how often and why incomplete fixes occur in practice. In this paper, we investigate the extent and characteristics of supplementary patches—patches that were later applied to supplement or correct initial fix attempts. We investigate the percentage of bugs that require supplementary fixes, their severity levels, and the relationship between initial patches and supplementary patches in terms of location and content similarity using the version history of Eclipse JDT core, Eclipse SWT, and Mozilla.

The following summarizes our study findings.

- **How many bugs require supplementary patches?** A considerable portion of resolved bugs (22% in Eclipse JDT core, 24% in Eclipse SWT, and 33% in Mozilla) involve supplementary patches. The bugs with supplementary patches are more likely to have higher severity—they have 35%, 64%, and 13% higher proportion of the Blocker/Critical status, take 56%, 91%, and 36% longer to be resolved, and involve 21%, 37%, and 55% more developers in bug discussions.
- **What are the common causes of incomplete fixes?** We randomly sampled total one hundred supplementary patches and inspected their content and corresponding initial patches. Our manual inspection shows that the common causes of omission errors are diverse, including missed porting changes, incorrect handling of conditional statements, and incomplete refactorings, etc. Furthermore, we found that the size and number of files of incomplete patches are larger than those of regular patches and that the file-level dispersion of incomplete patches is higher than that of regular patches. These results imply that incomplete fixes tend to be larger and more scattered than regular fixes.
- **Are supplementary patches similar to corresponding initial patches?** Several approaches attempt to reduce omission errors by identifying missed or inconsistent updates to similar code fragments [3], [8], [9]. To estimate the utility of such techniques in reducing omission

errors, we compared the content of an initial patch and its supplementary patches using CCFinder [10]. We found that only 12%, 25%, and 9% of supplementary patches have a content that is at least 5 lines similar to its initial patch in Eclipse JDT core, Eclipse SWT, and Mozilla respectively. This indicates that predicting a supplementary fix location using code clone analysis alone is insufficient.

- **Where is the location of supplementary patches with respect to initial patches?**
  48% and 42% of files in a supplementary patch involve changes within 25 lines of the initial patch location in Eclipse JDT core and Eclipse SWT respectively. Furthermore, 32% and 29% of files in supplementary patches have direct structural dependence (e.g., direct calls, accesses, and subtyping relations) on the corresponding initial patches. However, the remaining 15% and 14% of files in a supplementary patch do not have direct dependence to nor overlap with the initial patch location.

These results call for new type of omission error prevention approach that complements existing change recommendation systems, which are based on clone detection or structural dependence analysis.

The rest of this paper is organized as follows. Section II introduces related work, and Section III describes our study subjects and analysis method. Section IV presents empirical results, Section V discusses threats to validity, and Section VI summarizes our study.

II. Related Work

Errors of Omission. Fry and Weimer studied how well a human can localize different types of bugs [1]. They found that humans are over five times more accurate at locating *extra statements* than *missing statements*. This result is aligned with the fact that the omission errors are much more difficult to find than commission errors. While Fry and Weimer focus on measuring the accuracy of fault localization for different types of defects, our study focuses on the characteristics of supplementary patches to derive an insight into the common causes and characteristics of omission errors.

Nguyen et al. find that 17% to 45% of total fixes are recurring bug fixes [3]. Their tool FixWizard suggests additional change locations by finding code clones of an existing bug fix. Our study find that only 9% to 25% of supplementary patches have a content that is at least five lines similar to their initial patches, indicating that clone analysis alone is inadequate for reducing incomplete fixes.

Kim et al. suggest an approach to determine whether null-dereference bug fixes are complete or incomplete, with the concept of bug neighborhood [11]. A bug neighborhood refers to program statements directly related to null-dereference bugs. Zhang et al. propose a method to detect execution omission errors using dynamic slicing [12]. While these techniques focus on detecting omission errors, our study investigates the extent and characteristics of omission errors using the history of incomplete and supplementary bug fixes found in version history.

Empirical Studies of the Extent of Supplementary Bug Fixes. Yin et al. investigated incorrect bug fixes in large operating systems [13]. They found that 14.8% to 24.4% of post release patches are incorrect and that the most difficult type of bugs are related to concurrency. Their results show that a considerable portion of bugs was fixed more than once. Gu et al. measured the number of reopened bug reports in the bug database of Ant, AspectJ, and Rhino and found that such incorrect fixes correspond to as much as 9% of all bugs [14]. They assessed bug fixes with two criteria *coverage*—if the fix correctly handles all inputs triggering a bug, and *disruption*—if the fix unintentionally changes intended behavior. Purushothaman et al. investigated the extent of small changes and found that nearly 40% of bug fixes cause one or more defects in Lucent 5ESS [15]. These results are aligned with our study that incomplete bug fixes are common in practice.

Change Recommendation Systems for Supplementary Fixes. To reduce omission errors, Robillard's approach [4] takes a change set as input and recommends additional change locations based on the dependence structure of the change set. His approach is based on the assumption that additional change locations are likely to have structural dependencies on already changed code. Hassan and Holt propose several change propagation heuristics and find that historical change coupling is a more accurate predictor than structural dependencies such as method call relationships [2]. Padioleau et al. [16] find that Linux device drivers often co-evolve when kernel APIs change, and suggest an approach that infers a generic patch from an example edit [17]. Based on the similar assumption, Wang et al.'s approach automatically find edit locations similar to an existing bug fix using dependence-related queries [18]. While these approaches make individual assumptions about how supplementary fix locations can be predicted based on the content and location of an existing patch, our study investigates the location, content, and characteristics of incomplete and supplementary patches.

III. Study Approach

This section describes our subject programs and the method of identifying supplementary bug fixes.

Study Subjects. We select Eclipse JDT core, Eclipse SWT and Mozilla as our study subjects. Eclipse is an integrated development environment, which has been widely used in various mining software repositories research. Eclipse JDT core and Eclipse SWT are sub-projects of Eclipse written in Java. Mozilla is the open source project for the web. Mozilla contains many different sub-projects, e.g., Firefox web browser, Thunderbird mail client, etc.

Identification of Supplementary Bug Fixes. We study the evolution period of 2001/06 to 2009/02 in Eclipse JDT core, 2001/05 to 2010/05 in Eclipse SWT, and 1998/03 to 2008/05 in Mozilla. Previous studies on bug fixes [19], [20] found that it is possible to extract bug fix revisions by matching keywords against commit logs using the heuristic developed by Mockus and Votta [21]. We parse change logs to look for bug ID numbers using #().,\n\t\r\f as a delimiter and regard all integers as potential bug IDs. We then check those numbers against the corresponding bug databases to ensure that those extracted numbers indeed correspond to bug IDs. All of them use CVS as their configuration management system. In order to group file-revisions checked in the same commit, we convert CVS repositories to SVN repositories using cvs2svn.[1] Using this method, we identify that 30.59%, 31.44%, and 34.56% of commits are bug fixes in Eclipse JDT core, Eclipse SWT, and Mozilla respectively.

We define *supplementary bug fixes* as commits that contain a bug ID that was previously fixed. For example, when the same bug ID appears more than once in a version history, we consider all but the first commit as supplementary fixes for each closed bug ID. For example, the revision 8998 is an initial patch and the revisions 9012 and 9014 are supplementary patches in Table I.

Table I
TWO SUPPLEMENTARY PATCHES ARE APPLIED TO FIX BUG # 52916.

Revision	Date	Author	Comment
8998	2004/07/07	kjohnson	***52916***
8999	2004/07/07	kjohnson	*** empty log message ***
9000	2004/07/07	othomann	HEAD - fix for *69349*
9001	2004/07/07	othomann	HEAD - Fix for *69271*
...	...	...	...
9011	2004/07/08	pmulet	*69375*
9012	2004/07/08	kjohnson	***52916***
9013	2004/07/08	pmulet	*** empty log message ***
9014	2004/07/08	kjohnson	***52916* in 3.1 stream**

To make sure that bugs are *completely resolved* and will not be re-opened later, we consider only the bug reports reported within the period of 2004/07 to 2006/07 in Eclipse JDT core, 2004/07 to 2006/07 in Eclipse SWT, and 2003/04 to 2005/07 in Mozilla, when extracting information from bug databases.

After identifying bug IDs, we categorize bug fix commits into two categories: (1) *initial fix revisions* that refer to first attempts to address a particular bug ID and (2) *supplementary fix revisions* that refer to later attempts to correct, extend, complement, or revert an initial fix. Based on this categorization, we group bug reports into two groups: (1) *Type I bugs*—the bugs that were mentioned in only one commit and (2) *Type II bugs*—the bugs that were mentioned in multiple fix revisions.

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

IV. RESULTS

Section IV-A investigates the number of bugs with supplementary patches and their severity. Section IV-B describes the common cause of incomplete bug fixes that we found through manual inspection of one hundred randomly sampled incomplete patches and corresponding supplementary patches. Section IV-C investigates whether the content of supplementary bug fixes is similar to the content of corresponding initial bug fixes using clone detection analysis. Section IV-D investigates the relationship between an initial patch location and its supplementary patch location.

A. How Many Bugs Require Supplementary Patches?

Table II shows our research subjects and the number of bugs fixed only once *(Type I bugs)* vs. the number of bugs with more than one fix commits *(Type II bugs)*. 22.46% to 32.81% of resolved bugs require supplementary patches. In Figure 1, X axis represents how many times the same bug ID has been occurred in fix commits, and Y axis represents the percentage of bugs with n patches out of all Type II bugs. For example, in the Eclipse JDT core project, more than 70% of Type II bugs are fixed twice.

Figure 2 shows the number of days taken for a supplementary fix to appear since an initial attempt to fix. If there are multiple fix commits, we measure the time gap between the first fix revision and the last fix revision. More than 60% of supplementary fixes in JDT core appear within 24 hours. The majority of supplementary fixes are resolved in a short amount of time. However, some supplementary fixes take a very long time and take a large number of fix attempts. For example, in Mozilla, one bug was fixed 53 times, and one bug report in Eclipse SWT took a total of 1113 days to be resolved.

To investigate the difficulty level of Type I bugs and Type II bugs, we study the severity of bug reports reported in the bug database, the time taken to resolve the individual bugs, and the total number of developers involved in each bug report. Table III shows the results. Severity levels indicate how serious a defect is: Blocker/Critical/Major/Normal/Minor/Trivial/Enhancement in the descending order of severity. The severity distributions for Type I and Type II bugs are different. Overall, Type II bugs are 35.26%, 63.78%, and 13.08% more likely to be Blocker or Critical in Eclipse JDT core, Eclipse SWT, and Mozilla respectively.

Table IV shows the average number of developers involved in the bug discussion and the average time taken to resolve individual bug. In Eclipse JDT core, 3.67 developers are involved in discussing Type I bugs, while 4.44 developers are involved in the discussion of Type II bugs on average. In Eclipse SWT and Mozilla, 3.13 and 4.70 developers are involved in Type I bugs respectively, while 4.29 and 7.28 developers are involved in Type II bugs on average. The time taken to resolve individual bug is measured by the time taken from REPORTED to FIXED or RESOLVED status.

Table II
STUDY SUBJECTS

	Eclipse JDT core	Eclipse SWT	Mozilla Project
Type	IDE	IDE	Several projects associated with Internet
Period of development	2001/06 ∼ 2009/02	2001/05 ∼ 2010/05	1998/03 ∼ 2008/05
Study period	2004/07 ∼ 2006/07	2004/07 ∼ 2006/07	2003/04 ∼ 2005/07
Total revisions	17000 revisions	21530 revisions	200000 revisions
# of bugs	1812	1256	11254
# of Type I bugs	1405 (77.54%)	954 (75.96%)	7562 (67.19%)
# of Type II bugs	407 (22.46%)	302 (24.04%)	3692 (32.81%)

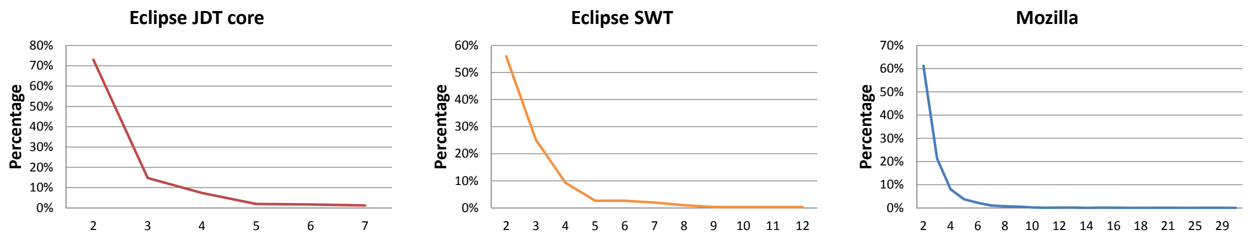

Figure 1. The number of times that the same bug is fixed

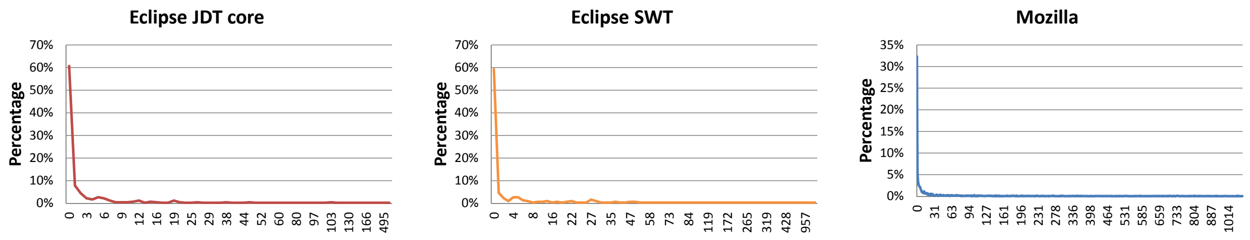

Figure 2. The number of days taken for the supplementary fix to appear since an initial fix

Table III
SEVERITY OF TYPE I AND TYPE II BUGS

	Eclipse JDT core		Eclipse SWT		Mozilla	
	Type I bugs	Type II bugs	Type I bugs	Type II bugs	Type I bugs	Type II bugs
Blocker	1.07%	1.47%	1.99%	2.65%	2.41%	1.79%
Critical	2.56%	3.44%	3.67%	6.62%	8.83%	10.92%
Major	8.33%	8.11%	11.53%	15.23%	8.38%	10.44%
Normal	76.80%	76.90%	74.95%	61.92%	60.92%	61.56%
Minor	5.20%	2.95%	2.41%	1.66%	6.96%	5.39%
Trivial	1.64%	0.00%	1.78%	1.66%	6.87%	3.31%
Enhancement	4.41%	6.88%	3.56%	10.26%	5.61%	6.53%

If a bug is not resolved yet, we measure the time taken from REPORTED to the latest fix attempt. In Eclipse JDT core, Type I bugs take 120.79 days and Type II bugs take 188.27 days to be resolved. The others follow similar trends.

Overall, Type II bugs involve more developers in the bug report discussions and take longer time to be resolved than Type I bugs (p-values from T-test: 1.45e-12, 1.39e-09, 2.05e-84 for the number of developers in the bug report discussion, and 3.84e-04, 2.65e-07, and 8.40e-42 for bug resolution time in Eclipse JDT core, Eclipse SWT, and Mozilla respectively).

A considerable portion of bugs requires supplementary patches. Such bugs take longer to be resolved and involve more developers in the discussion of the bug reports.

B. What Are the Common Causes of Incomplete Bug Fixes?

To understand why omission errors occur in practice, we first contrast the characteristics of *incomplete patches* (initial patches of Type II bugs) against that of *regular patches* (patches of Type I bugs). We measure the average

Table V
THE SIZE AND PHYSICAL DISPERSION OF INCOMPLETE PATCHES VS. REGULAR PATCHES.

	Files		LOC		Added LOC (%)		Dispersion (file)		Dispersion (package)	
	Regular	Incomplete	Regular	Incomplete	Regular	Incomplete	Regular	Incomplete	Regular	Incomplete
JDT	2.58	3.91	92.13	203.38	63.73	66.50	0.27	0.28	0.12	0.11
SWT	1.94	2.16	47.58	73.39	65.90	71.39	0.25	0.25	0.11	0.12
Mozilla	3.62	6.21	170.99	340.31	59.76	62.42	0.33	0.37	0.16	0.20
Total	3.30	5.72	147.98	309.38	60.92	63.42	0.31	0.36	0.15	0.18
p-value	1.15E-18		4.46E-05		4.03E-12		2.05E-09		1.04E-10	

Table IV
THE EFFORT TAKEN TO FIX TYPE I AND TYPE II BUGS

The number of developers required for fixing bugs

	Type I	Type II	p-value
Eclipse JDT core	3.67	4.44	1.45e-12
Eclipse SWT	3.13	4.29	1.39e-09
Mozilla	4.70	7.28	2.05e-84

The time taken to resolve bugs

	Type I	Type II	p-value
Eclipse JDT core	120.79	188.27	3.84e-04
Eclipse SWT	176.99	337.32	2.65e-07
Mozilla	594.50	805.92	8.40e-42

number of files, the total number of changed lines, the portion of added lines among all changed lines, and physical dispersion. To measure the physical dispersion, we compute normalized entropy scores at the file and package level, $entropy = -\sum_{i=1}^{n} p_i log_n(p_i)$, where p_i is the probability that a changed line is made to a particular changed file (or package), using the same method by Hassan [22]. A low entropy score implies that only a few files include most of the modifications. If the entropy is high, changed code is more equally distributed among different changed files. The results are summarized in Table V. Incomplete patches are larger in size than regular patches, and they tend to include more scattered and non-localized edits.

To investigate the common causes of omission errors, we randomly selected one hundred incomplete patches and inspected their patch contents, the structural dependence relations between an incomplete patch and its supplementary patches, and the associated bug reports. The following summarizes a taxonomy of common omission errors we found in the subject programs.

1) **An initial patch is ported to a different component or branch.** For example, to fix the bug 88829 in Eclipse SWT, the patch for Windows is later ported to Mac on 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 Figure 3.

2) **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. Later, the condition was modified to handle the corner case of `AccInterface` as shown in Figure 4.

3) **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.

4) **An initial patch is reverted.** For example, an API added at initial patch of bug 110048 in Eclipse JDT core was deleted during a supplementary fix because the API is turned out to be inadequate, and developers decided to apply a new approach. Its status was changed to `REOPENED`.

5) **Two different parts calling different subclasses of the same type are not updated together.** The initial patch of bug 81244 in Eclipse JDT core is about the use of `ArrayTypeReference`, and the supplementary fix is regarding the use of `ArrayQualifiedType-Reference`. The two classes are subtypes of the same super type. (See Figure 5)

6) **Incomplete refactoring induces a supplementary patch.** The initial patch of bug 104664 in Eclipse JDT core refactored `file` and `zipFile` functions partially. The remaining parts are fixed during a supplementary fix. (See Figure 6)

7) **The locations of incomplete and supplementary fixes are related but cannot be checked using the Java compiler.** The initial patch of bug 83593 in Eclipse JDT core was made to class `CopyResource-ElementsOperation` and the supplementary patch was made to class `DOMFinder`. The two classes do not have direct static call dependencies, as the code uses the Visitor design pattern [23].

8) **An initial patch is refactored during supplementary fixes.** The initial patch of bug 287052 in Mozilla modified the function `CERT_FindCRLReasonExten`, which was later renamed to `CERT_FindCRLEntryReasonExten` in the supplementary patch.

9) **The comment is improved to explain an initial patch in detail.** For example, the comment for bug 243392 in Mozilla is updated to explain the initial fix on `nsContentSink.cpp`.

10) **Others.** Other omission errors include missed value initializations, missing null pointer checks, missing property updates, and forgetting to release run-time resources. For example, the initial patch of bug 114935 in Eclipse JDT core was on `CompilationUnitResolver.java`, to exit a loop early, when there is no need to resolve types further. The remaining resources were not cleaned up, and this problem was fixed during the supplementary fix.

Table VI shows the classification of one hundred incomplete patches into the above ten categories. 28 out of 100 incomplete patches involve missed porting updates, 23 of them involve incorrect handling of conditional statements, and 15 of them involve forgetting to update code that references modified code, etc.

Table VI
CATEGORIZATION OF 100 SAMPLED INCOMPLETE FIX REVISIONS

	#	Bug ID (Project)
1)	28	69152 (JDT), 91709 (JDT), 140879 (JDT), 148010 (JDT), 76185 (SWT), 76391 (SWT), 83408 (SWT), 83819 (SWT), 85666 (SWT), 85962 (SWT), 87554 (SWT), 88829 (SWT), 89785 (SWT), 90856 (SWT), 107777 (SWT), 203041 (Mozilla), 203211 (Mozilla), 220933 (Mozilla), 223111 (Mozilla), 224313 (Mozilla), 225424 (Mozilla), 226600 (Mozilla), 231166 (Mozilla), 235859 (Mozilla), 249337 (Mozilla), 261886 (Mozilla), 271855 (Mozilla), 280740 (Mozilla)
2)	23	80699 (JDT), 92315 (JDT), 92888 (JDT), 96763 (JDT), 105531 (JDT), 110422 (JDT), 111494 (JDT), 119844 (JDT), 139621 (JDT), 142772 (JDT), 70318 (SWT), 72401 (SWT), 81081 (SWT), 85386 (SWT), 85867 (SWT), 200144 (Mozilla), 211470 (Mozilla), 212222 (Mozilla), 216581 (Mozilla), 227432 (Mozilla), 232094 (Mozilla), 258278 (Mozilla), 289558 (Mozilla)
3)	15	79772 (JDT), 108372 (JDT), 120264 (JDT), 120640 (JDT), 123522 (JDT), 133071 (JDT), 76750 (SWT), 81987 (SWT), 83543 (SWT), 88463 (SWT), 90024 (SWT), 92230 (SWT), 93294 (SWT), 94003 (SWT), 94467 (SWT)
4)	7	98906 (JDT), 110048 (JDT), 141289 (JDT), 85069 (SWT), 246966 (Mozilla), 251589 (Mozilla), 272046 (Mozilla)
5)	4	81244 (JDT), 126180 (JDT), 78554 (SWT), 106289 (SWT)
6)	3	104664 (JDT), 125518 (JDT), 263216 (Mozilla)
7)	2	83593 (JDT), 126625 (JDT)
8)	7	100636 (JDT), 117302 (JDT), 75148 (SWT), 110559 (SWT), 223435 (Mozilla), 287052 (Mozilla), 298429 (Mozilla)
9)	5	86580 (JDT), 110797 (JDT), 130390 (JDT), 243392 (Mozilla), 298756 (Mozilla)
10)	6	114935 (JDT), 215587 (Mozilla), 217149 (Mozilla), 217999 (Mozilla), 225570 (Mozilla), 240550 (Mozilla)

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

C. Are Supplementary Bug Fixes Similar to Corresponding Initial Fixes?

It is widely believed that code clones are difficult to maintain and that inconsistent management of code clones

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 [oldOrder [i]];~~
+       TableColumn column = newColumns [i];
        if (!column.isDisposed ()) {
-           ~~if (order [i] != oldOrder [i]) {~~
+           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 [oldOrder [i]];~~
+          TableColumn column = newColumns [i];
          if (!column.isDisposed ()) {
-             ~~if (order [i] != oldOrder [i]) {~~
+             if (newX [i] != oldX [i]) {
                column.sendEvent (SWT.Move);
             }
          }
```

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

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 & AccAnnotation) != 0)~~
-       ~~return IGenericType.ANNOTATION_TYPE_DECL;~~
-   ~~if ((modifiers & AccInterface) != 0)~~
-       ~~return IGenericType.INTERFACE_DECL;~~
+   if ((modifiers & AccInterface) != 0) {
+       if ((modifiers & AccAnnotation) != 0)
            return IGenericType.ANNOTATION_TYPE_DECL;
+       return IGenericType.INTERFACE_DECL;
+   }
...
```

Figure 4. The conditional statement of an initial fix is not correct.

An initial patch (revision 10584 in Eclipse JDT core)

```
Index: .../compiler/problem/ProblemReporter.java
=======================================
@@ -2895,10 +2895,8 @@
...
   } else if (location instanceof ArrayTypeReference) {
~~-       if (!(location instanceof~~
~~-           ParameterizedSingleTypeReference)) {~~
~~-           ArrayTypeReference 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) {
~~-       if (!(location instanceof~~
~~-              ParameterizedQualifiedTypeReference)) {~~
~~-           ArrayQualifiedTypeReference~~
~~-                arrayQualifiedTypeReference =~~
...
+     ArrayQualifiedTypeReference
           arrayQualifiedTypeReference =
           (ArrayQualifiedTypeReference) location;
+     long[] positions = arrayQualifiedTypeReference.
           sourcePositions;
+     end = (int) positions[positions.length - 1];
...
```

Figure 5. Two different parts calling different subclasses of the same type are not updated together.

An initial patch (revision 11796 in Eclipse JDT core)

```
Index: .../compiler/batch/ClasspathJar.java
=======================================
@@ -23,16 +23,17 @@
...
 public class ClasspathJar extends ClasspathLocation {
~~-ZipFile zipFile;~~
~~-boolean closeZipFileAtEnd;~~
~~-Hashtable packageCache;~~
+private File file;
+private ZipFile zipFile;
...
 public ClasspathJar(File file) throws IOException {
~~-         this(new ZipFile(file), true, null);~~
+         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.zipFile.getName(); //$NON-NLS-1$~~
+   return "Classpath for jar file "
          + this.file.getPath(); //$NON-NLS-1$
 }
 public String normalizedPath(){
~~-   String rawName = this.zipFile.getName();~~
+   String rawName = this.file.getPath();
    return rawName.substring(0, rawName.lastIndexOf('.'));
 }
```

Figure 6. Incomplete refactoring induces a supplementary patch.

is a frequent source of omission errors. To investigate how often supplementary bug fixes are induced by inconsistent management of clones, we measure the content similarity between a supplementary patch and its initial patch. The patches are extracted by svndiff. The cloning relationship between an initial patch and its supplementary patch is identified by CCFinder [10] with a minimum token size of 5 tokens. We then exclude some patches if cloned code is less than five lines. As Figure 7 shows, 11.92%, 25.35%, and 8.91% of supplementary patches include at least five lines similar to its initial patch.

Backporting patches are usually identical to original patches, but are applied to different branches. We exclude those from our analysis to accurately estimate the amount of backporting patches out of all supplementary patches that are similar to their initial patches.

Backported patches consist of 17.88%, 6.12%, and 12.34% in Eclipse JDT core, Eclipse SWT, and Mozilla respectively. The majority of supplementary patches, 70.20%, 68.54%, and 78.75%, are not similar to initial patches. The results are against the conventional wisdom that clone management could significantly prevent or reduce omission errors. Our study result reveals that existing change recommendation systems based on clone detection analysis alone [3], [8], [9] are inadequate.

Predicting a supplementary fix location using code clone analysis alone is insufficient.

D. Where Is the Location of Supplementary Bug Fixes in Relation to Initial Fixes?

We investigate how the location of an initial patch is related to the location of a supplementary patch in Java subject programs to draw an insight into how to predict supplementary change locations given an existing change set. The files fixed in an initial patch could be modified again during a supplementary fix, or new files could be fixed instead. Out of 1048 and 1093 files fixed in supplementary patches, 52.96% and 57.46% are applied to the same files in Eclipse JDT core and Eclipse SWT respectively. To determine changes made to similar line locations, we use Yin et al.'s heuristic [13], i.e., changes within 25 lines of the original change. 48.09% and 41.54% are made within 25 lines of initial patch locations respectively.

To identify the structural dependence relationship between the files of initial patches and the files of supplementary patches, we use LSDiff to extract structural dependence among source files. LSDiff infers systematic structural differences as a logic rule [24], [25], and represents each program version using a set of logic facts—e.g., calls (callerMethodName, calleeMethodName), access (fieldName, accessorMethodName), etc. Using this technique, we extract structural facts for every released version of Eclipse

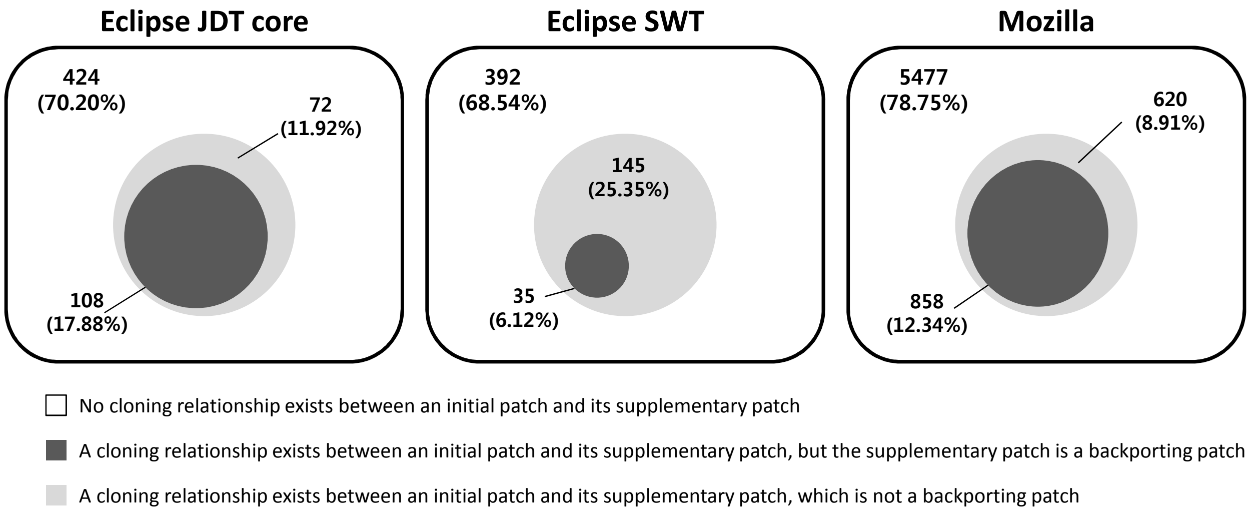

Figure 7. The percentage of cloned patches and backported patches out of all supplementary patches

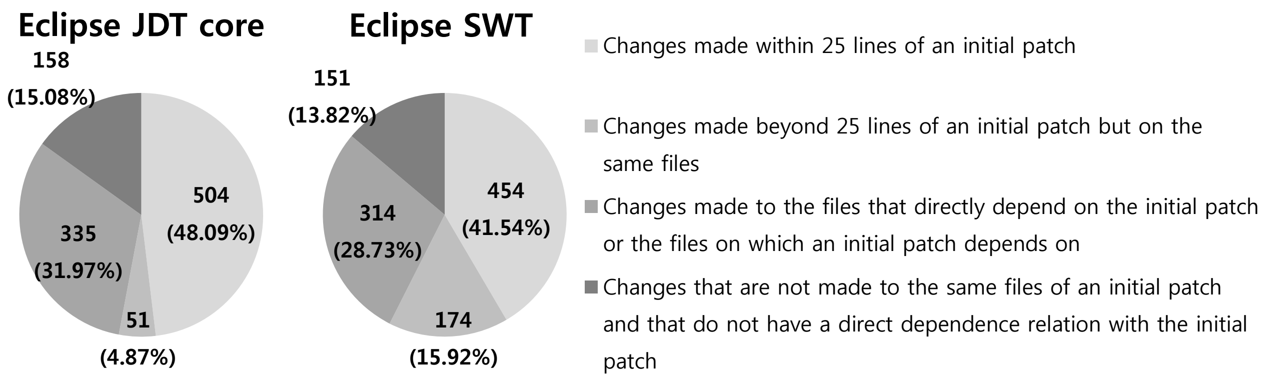

Figure 8. 15.08% and 13.82% of files do not overlap with an initial patch location nor have direct dependencies on them.

JDT core and Eclipse SWT, and then we map the facts to file-level dependence relationships. For example, the first fix of the bug 111618 is on class `ForeachStatement` and the third fix is on class `AbstractVariable-Declaration`. Class `ForeachStatement` calls a method named `printAsExpression` in `Abstract-VariableDeclaration`, producing file-level dependence relation between the initial patch and the supplementary patch. Out of all files of supplementary patches, 31.97% and 28.73% are directly related to at least one file in the initial patch as shown in Figure 8.

We also investigate indirect dependence relationships between the files of supplementary patches and files of initial patches. A sibling relation means the two files extend from the same ancestor type, and indirect call/access means a file is indirectly called or accessed via a different file. Overall, 15.08% and 13.82% of files in supplementary patches do not overlap with nor have any direct relationship to the files modified in the initial patch. This result indicates that change recommendation systems that suggest the direct dependence neighbors of an existing change set is not enough for reducing omission errors.

About 15% of supplementary change locations are beyond the scope of the direct neighbors of initial patch locations.

V. Discussion

Identification of Cloned Patches. In our study, a supplementary fix revision is regarded as a cloned patch when it has similar code chunk from an initial patch, longer than five lines. However, it is possible that only few lines of a supplementary patch is similar to the initial patch.

Figure 9 shows that the percentage of cloned patches while varying the threshold. In Eclipse JDT core, 11.92% of

Table VII
THE STRUCTURAL DEPENDENCE RELATIONSHIPS BETWEEN AN INITIAL PATCH AND ITS SUPPLEMENTARY PATCH AT THE FILE LEVEL

	Relation type	Eclipse JDT core	Eclipse SWT
Direct dependence relations	Call	281 (57.00%)	295 (63.44%)
	Access	244 (49.49%)	257 (55.27%)
	Return	101 (20.49%)	53 (11.40%)
	Fieldoftype	86 (17.44%)	61 (13.12%)
	Extend	42 (8.52%)	26 (5.59%)
	Implement	14 (2.84%)	0 (0.00%)
Indirect dependence relations	Sibling	40 (8.11%)	44 (9.46%)
	Indirect call	85 (17.24%)	47 (10.11%)
	Indirect access	46 (9.33%)	19 (3.09%)
	Other	150 (30.43%)	134 (28.82%)

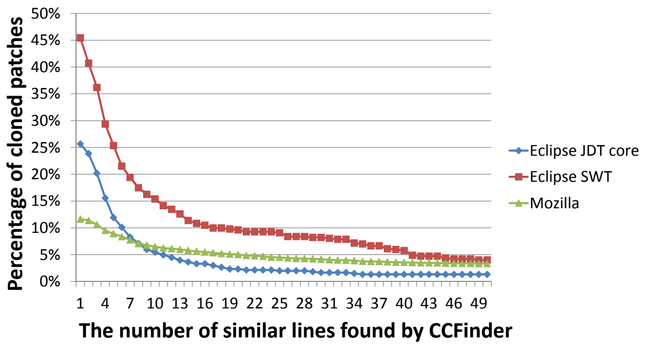

Figure 9. The percentage of supplementary patches that are at least X lines similar to their initial patches according to CCFinder, while varying X.

supplementary patches are classified to cloned patches with the threshold of five lines. When the threshold of ten lines are used, only 5.46% of supplementary patches are identical.

Relationships among Supplementary Patches. In our study, we consider only the relationship between an initial patch and its individual supplementary patches, as opposed to the relationship among supplementary patches. For example, in Eclipse JDT core, a bug 73104 has six fix attempts, while the third patch reverts the second patch and the fourth patch reverts the third one again. We also investigate the time taken to fix a bug as the time from the initial patch to the last supplementary patch, but we do not investigate on the duration between consecutive patches.

The Studied Period of Bug Reports. The studied period of bug reports are around two years in our study, which are relatively short in comparison with the entire evolution period of the subject systems. While the status of a bug report in a bug database could be used to determine whether each bug is resolved or not, even resolved bugs could be re-opened for fix, as we've seen in our study. Thus, we limited our focus to a part of the development period to ensure that each bug is fully resolved. We acknowledge that the limited scope of our study period could be a potential threats to external validity, and we plan to extend the study duration in the future.

New Tools for Reducing Incomplete Bug Fixes. Our results show that a considerable amount of incomplete fixes is caused by patches that do not cover all cases of conditional statements correctly. An analysis tool that represents differences in control flows of changed block might mitigate such difficulties.

Our analysis also shows that an unfinished refactoring often induces a supplementary fix. If a variable is refactored, the methods accessing the variable should be refactored together, and some other methods calling the methods could be refactored. Tools that can detect incomplete refactorings [26] based on pre-defined or common refactoring error patterns may prevent users from introducing omission errors.

VI. CONCLUSIONS

Many approaches have been proposed to prevent or reduce omission errors or to recommend supplementary change locations. These approaches make individual assumptions on the characteristics of omission errors. To investigate why omission errors occur in practice and how such errors can be prevented, we studied the characteristics of incomplete patches and supplementary patches.

Our study on three open source projects shows a considerable portion of bugs requires supplementary bug fixes. Incomplete patches tend to be larger and more scattered than regular patches, and the common causes of omission errors are diverse, including missed porting changes, incorrect handling of conditional statements, or incomplete refactorings, etc. In contrast to the conventional wisdom that missed updates to code clones often cause omission errors, only a small number of supplementary patches have content similar to their initial patches. Recommending the direct neighbors of initially fixed files in terms of structural dependence or recommending clones of an existing change set cannot predict all supplementary change locations. These results call for new omission error reduction approaches that complement existing change recommendation systems.

ACKNOWLEDGEMENTS

This research was sponsored by the Agency for Defense Development, Republic of Korea, under the grant UD100031CD. This work was in part supported by National Science Foundation under the grants CCF-1149391, CCF-1043810, and CCF-1117902, and by a Microsoft SEIF award.

REFERENCES

[1] Z. P. Fry and W. Weimer, “A human study of fault localization accuracy,” in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2010, pp. 1–10.

[2] A. E. Hassan and R. C. Holt, “Predicting change propagation in software systems,” in *ICSM ’04: Proceedings of the 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2004, pp. 284–293.

[3] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, 2010, pp. 315–324.

[4] M. P. Robillard, "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, 2005, pp. 11–20.

[5] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[6] A. T. T. Ying, G. C. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history." *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.

[7] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 563–572.

[8] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Clone-aware configuration management," in *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 123–134.

[9] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 158–167.

[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code." *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[11] M. Kim, S. Sinha, C. Go andrg, H. Shah, M. Harrold, and M. Nanda, "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, 2010, pp. 383 –392.

[12] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards locating execution omission errors," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2007, pp. 415–424.

[13] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "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, 2011.

[14] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su, "Has the bug really been fixed?" in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ACM, 2010, pp. 55–64.

[15] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 511–526, 2005.

[16] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "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, 2008, pp. 247–260.

[17] J. Andersen and J. Lawall, "Generic patch inference," in *ASE '08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008*. IEEE Computer Society, Sept. 2008, pp. 337–346.

[18] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching dependence-related queries in the system dependence graph," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. ACM, 2010, pp. 457–466.

[19] M. Kim, D. Cai, and S. Kim, "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, 2011.

[20] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, March and April 2008.

[21] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *ICSM '00: Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2000, p. 120.

[22] A. E. Hassan, "Predicting faults using the complexity of code changes," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 78–88.

[23] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1994.

[24] A. Loh and M. Kim, "A program differencing tool to identify systematic structural differences," in *ICSE '10: Proceedings of the 2010 ACM and IEEE 32nd International Conference on Software Engineering*. ACM, 2010, pp. 263–266.

[25] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 309–319.

[26] C. Görg and P. Weißgerber, "Error detection by refactoring reconstruction," in *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*. ACM Press, 2005, pp. 1–5.