Partitioning Composite Code Changes to Facilitate Code Review

Yida Tao and Sunghun Kim
The Hong Kong University of Science and Technology
Department of Computer Science and Engineering
{idagoo, hunkim}@cse.ust.hk

***Abstract*—Developers expend significant effort on reviewing source code changes. Hence, the comprehensibility of code changes directly affects development productivity. Our prior study has suggested that *composite* code changes, which mix multiple development issues together, are typically difficult to review. Unfortunately, our manual inspection of 453 open source code changes reveals a non-trivial occurrence (up to 29%) of such composite changes.**

In this paper, we propose a heuristic-based approach to automatically partition composite changes, such that each sub-change in the partition is more cohesive and self-contained. Our quantitative and qualitative evaluation results are promising in demonstrating the potential benefits of our approach for facilitating code review of composite code changes.

I. Introduction

Code review is an important practice in software maintenance, and it reportedly takes up to 60% of the software engineering effort [1], [2]. In modern software development, code review has become less formal and more lightweight [3], [4]. Instead of reviewing the entire software system from scratch, developers are more often required to review only the incremental code changes [3], [5].

A small change that tackles one single issue (e.g., fixing one bug) might not be too difficult to review. When multiple issues are addressed in a single change, however, reviewing it becomes time-consuming and error-prone since developers have to figure out which part of the change addresses which issue [5]. We refer to such code changes that address multiple development issues as *composite* changes, as opposed to *atomic* changes that address one single issue.

In this paper, we attempt to address the following research questions regarding the challenges of code review induced by composite code changes:

- **RQ1**: Are composite code changes prevalent?
- **RQ2**: Can we propose an approach to improve the semantic atomicity of composite code changes?
- **RQ3**: Can our approach help developers better review composite code changes?

We first conducted a manual investigation of the occurrence of composite code changes. Among 453 changes extracted from the development history of four open source projects — Ant, Commons Math, Xerces, and JFreeChart, up to 29% and on average 17% are composite. This nonnegligible proportion confirms the prevalence of composite changes.

We then propose a heuristic-based approach to improve the atomicity of composite code changes. Our approach leverages program slicing [6] to identify semantically dependent changes and pattern matching to identify logically related changes. Related changes are then merged together and isolated from other non-related changes. In this way, a composite change can be partitioned into a set of *change-slices*, each is more cohesive and self-contained in terms of the issue being addressed.

We applied our approach to the 78 composite changes identified from our manual investigation. In the evaluation, 69% of them were automatically partitioned the same way as humans manually did. We further conducted a comparative user study to explore the value of change partitioning for code review. Results showed that within a similar period of time, participants understood partitioned changes significantly better than the original composite ones.

In summary, this paper makes the following contributions:

- We provide empirical evidence of the prevalence of composite changes in software evolution.
- We propose an approach to automatically partition composite changes.
- We conduct a user study to illustrate how the proposed change-partition approach facilitates code review.

The remainder of this paper is organized as follows. Section II motivates our work with manual inspections of composite changes. Section III introduces our change-partition approach, followed by an evaluation in Section IV. Section V describes our user study. Section VI discusses future improvements and potential applications of our approach. Section VII reports threats to validity. Section VIII presents related work and Section IX concludes the paper.

II. The Problem of Composite Code Changes

A. Occurrence

To investigate the occurrence of composite code changes (**RQ1**), we manually inspected changes from four open source projects: Ant, Commons Math, Xerces, and JFreeChart. Ant is a tool for automating software build process. Commons Math is a library of mathematics and statistics components. Xerces is a library for parsing and manipulating XML while JFreeChart is a library for creating various charts. The size of these projects ranges from small to medium.

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.24

TABLE I: Study subjects and their total number of revisions in the observed time period. The fourth column shows the number of revisions that change at least two lines of code. We use these selected revisions for our manual inspection and later experiments. The fifth column shows the average changed lines of code (cLOC) per revision, while cLOC is the sum of added, modified, and deleted lines of code in a revision. The last column shows the average number of changed files per revision.

	Time period	Total revisions	Selected revisions	Avg. cLOC per revision	Avg. files per revision
Ant	2010-04-27 ∼ 2012-03-05	503	137	26.1	2.0
Commons Math	2011-11-28 ∼ 2012-04-12	331	107	84.7	3.5
Xerces	2008-11-03 ∼ 2012-03-13	341	116	63.6	3.0
JFreeChart	2008-07-02 ∼ 2010-03-30	301	93	144.9	4.1

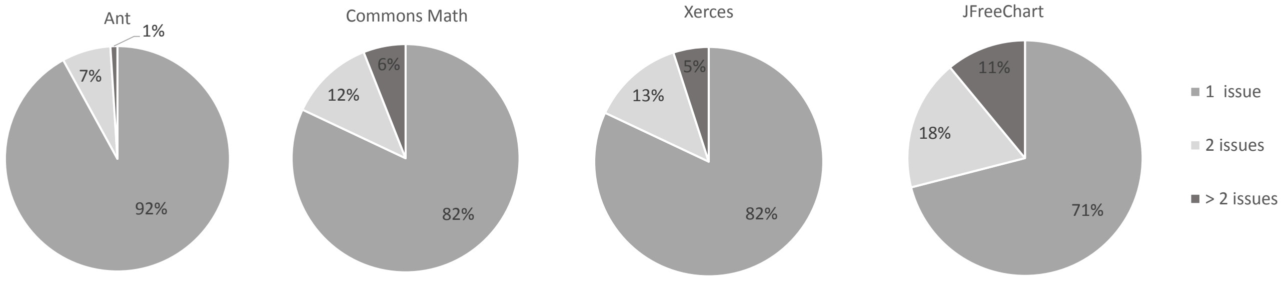
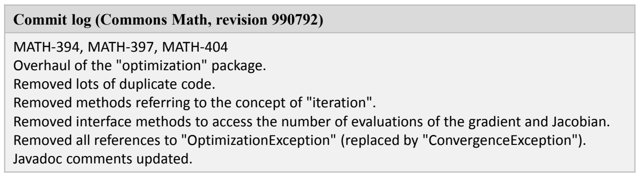


Fig. 1: Percentage of code changes that address one, two, or more than two development issues.

From each project's SVN repository, we checked out revisions (i.e., changes) that have at least two modified source code lines, since it is pointless to partition a change if it had changed only one line of code. As shown in Table I, 453 revisions from these four projects were finally inspected. We inspected the commit log written by developers and manually marked the number of issues addressed in each revision. For example, the commit log of Commons Math revision 990792 suggests that this revision addresses three issues: MATH-394, MATH-397, and MATH-404 (Figure 2). Specifically, MATH-394 removes duplicate code, MATH-397 handles inconsistencies between two packages, and MATH-404 fixes an interface bug. We also manually inspected the source code of revisions when their commit logs were not clear.

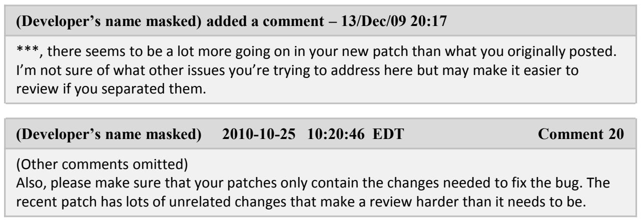
Figure 1 shows the manual inspection results. The majority of revisions (71% ∼ 92%) are atomic, that is, they address only one development issue. Yet, 78 (17%) and up to 29% (JFreeChart) of the revisions are composite changes addressing more than one issue.



Commit log (Commons Math, revision 990792)

MATH-394, MATH-397, MATH-404
Overhaul of the "optimization" package.
Removed lots of duplicate code.
Removed methods referring to the concept of "iteration".
Removed interface methods to access the number of evaluations of the gradient and Jacobian.
Removed all references to "OptimizationException" (replaced by "ConvergenceException").
Javadoc comments updated.

Fig. 2: The commit log of Commons Math revision 990792.



(Developer's name masked) added a comment – 13/Dec/09 20:17

***, there seems to be a lot more going on in your new patch than what you originally posted. I'm not sure of what other issues you're trying to address here but may make it easier to review if you separated them.

(Developer's name masked) 2010-10-25 10:20:46 EDT Comment 20

(Other comments omitted)
Also, please make sure that your patches only contain the changes needed to fix the bug. The recent patch has lots of unrelated changes that make a review harder than it needs to be.

Fig. 3: Review comments on the patches for issues Xerces-1407 and Eclipse-86576. Both patches were rejected since the code reviewers required the removal of unrelated changes from the submitted patches.

B. Difficulty in Code Review and Maintenance

Developers sometimes use "chunky changes" or even "code bombs"[1] to describe changes that touch many places and bundle various unrelated changes together. One of their primary complaints is that "*... some changes are combined with other changes (e.g., multiple bug fixes). It is hard and error-prone to figure out whether a specific change is related to one bug or another*" [5]. For the same reason, code reviewers typically prefer a clear separation of a change if it addresses multiple issues. For example, a code reviewer commented on Gson revision 1154 saying "*I would have preferred to have two different commits: one for adding the new getFieldNamingPolicy method, and another for allowing overriding of primitives.*"[2] Violations of this recommended practice often result in the change being rejected for integration, as shown in Figure 3.

In the next section, we propose an approach to alleviate this issue, by partitioning a composite code change so that each sub-change in the partition is more semantically cohesive.

[1]http://darkforge.blogspot.hk/2010/02/code-bomb-or-newbie-with-big-ideas.html

[2]https://code.google.com/p/google-gson/source/detail?r=1154

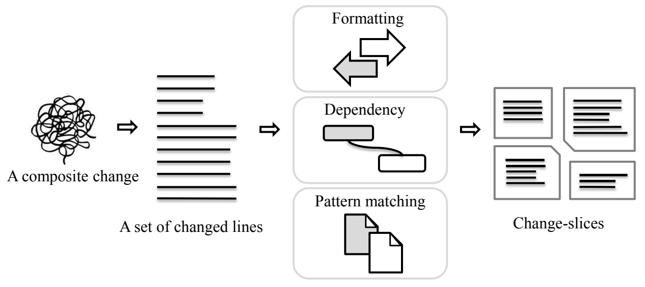


Fig. 4: A composite change is represented by a set of changed lines. Three heuristics are used to identify related lines and merge them into change-slices.

RQ1: We observed 78 out of 453 (17%) code changes to be composite. For each project, from 8% up to 29% code changes are composite.

III. Approach

We present the problem statement of change partition in this section. We then detail our partition approach following the overview of Figure 4 and introduce the implementation.

A. The Change Partitioning Problem

The problem of change partitioning is modeled as follows.
Input: A program change C, which is a set of deleted, modified, and added source code lines.
Output: The partition P of the set C. More formally, P is a division of C into non-empty, non-overlapping, and collectively exhaustive subsets (S).

- $\forall S \in P, S \neq \emptyset$
- $\forall_{i \neq j} S_i, S_j \in P, S_i \cap S_j = \emptyset$
- $\bigcup_{S \in P} S = C$

We use the term *change-slice* to refer to such a subset of C. A change-slice consists of only changed lines that are *related*, which is defined next.

B. Identifying Related Changes

We consider two changed code lines *related* if 1) both are formatting-only changes; or 2) they are semantically related for having static dependencies, or 3) they are logically related for having similar change patterns. We detail these heuristics as below.

Formatting-only changes: Developers format code for better readability. However, when formatting is performed together with bug-fixes or new feature implementations, code changes become composite. Hence, we isolate formatting-only changes into one single change-slice.

Changes with static dependencies: *Program slicing* refers to the computation of program-slice, which is the part of a program that potentially affects the value at a given program point [6]. We use program slicing to identify statically dependent lines that are likely to address the same issue. This assumption is similar to several previous studies [7], [8], [9], [10], which leveraged program slicing to isolate consistent concerns.

static double[] loadExpIntA() {	static double[] loadExpIntA() {
return EXP_INT_A;	return EXP_INT_A.clone();
}	}
static double[] loadExpIntB() {	static double[] loadExpIntB() {
return EXP_INT_B;	return EXP_INT_B.clone();
}	}
static double[] loadExpFracA() {	static double[] loadExpFracA() {
return EXP_FRAC_A;	return EXP_FRAC_A.clone();
}	}
static double[] loadExpFracB() {	static double[] loadExpFracB() {
return EXP_FRAC_B;	return EXP_FRAC_B.clone();
}	}
static double[][] loadLnMant() {	static double[][] loadLnMant() {
return LN_MANT;	return LN_MANT.clone();
}	}

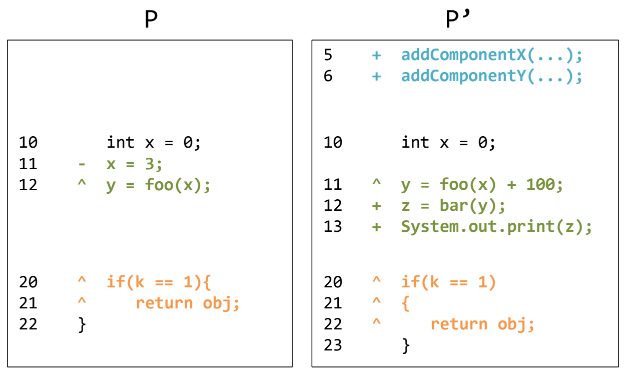
Fig. 5: Commons Math revision 1235784. The five highlighted statements, although not statically dependent, are changed in the same way and serve one single purpose.

Changes with similar patterns: Some code changes are related, even though no static dependency exists between them. Figure 5 shows an example, Commons Math revision 1235784, whose commit log indicates its single intention is to "*protect array entries against malicious or accidental corruption by returning a clone.*" Accordingly, this revision modifies five *return* statements in the same way by invoking *clone()*. Even though no static dependency exists between these five statements, they indeed serve one single purpose. Another example is that multiple methods with similar names are added in a change. In Commons Math revision 1206475, four overloaded methods *populationVariance* are added. JFreeChart revision 1363 adds three methods *clearSectionPaints*, *clearSectionOutlinePaints*, and *clearSectionOutlineStrokes*. The similarity between these methods' names indicates that they are likely to serve the same purpose, even though they have no static dependency.

We use pattern matching to identify such logically related changes. Specifically, two changes are considered similar if 1) they are newly added methods whose names' similarity is above a threshold t_m, or 2) they have the same change type and the similarity between their string *delta* is above a threshold t_s. In Figure 5, all changes have the same type "return statement update" and the delta for each updated statement is "*.clone()*".

C. Partition

Figure 6 shows a sample code change. Using our partition approach, three change-slices are produced. One change-slice contains line 5 and 6 in P', since two methods with similar names are added. Another change-slice contains the formatting changes in line 20–22. The remaining changed lines form the third change-slice, since line 11 and 12 in P are statically



```
10       int x = 0;
11   -   x = 3;
12   ^   y = foo(x);

20   ^   if(k == 1){
21   ^      return obj;
22       }
```

```
5    +   addComponentX(...);
6    +   addComponentY(...);

10       int x = 0;

11   ^   y = foo(x) + 100;
12   +   z = bar(y);
13   +   System.out.print(z);

20   ^   if(k == 1)
21   ^   {
22   ^      return obj;
23       }
```

Fig. 6: An example of a code change, which includes deletions (-), modifications (∧) and additions (+). This change is partitioned into three change-slices, shown in three colors.

dependent, while line 12 in P is modified and has static dependency with the newly added line 12 and 13 in P'.

Suppose we construct an undirected graph G whose nodes represent changed lines and edges represent whether two nodes are related based on our heuristics explained in Section III-B. Then, the connected components of G can be found and mapped to each change-slice. In other words, a change C is partitioned into a set of change-slices exactly the same way G is divided into a set of connected components.

D. Implementation

We leveraged a code differencing tool called ChangeDistiller [11] to identify formatting changes. ChangeDistiller performs a tree differencing algorithm on source code, which is represented by Abstract Syntax Tree (AST). Since ChangeDistiller works on a structured representation of the program (i.e., AST) rather than a flat representation (i.e., textual code) as Unix *diff* does, it effectively ignores formatting changes. Hence, we compared the results computed by Unix diff and ChangeDistiller. Changed lines that are presented in Unix diff but not presented in AST diff are considered to be formatting-only changes.

To identify static dependency between changed lines, we used the IBM T.J. Watson Libraries for Analysis (WALA) and specifically its built-in ZeroOneCFA pointer analysis policy to compute the backward program slice of each changed line [12].

To identify change patterns in method names and statements, we used the python module *difflib*, specifically, its *Differ.compare()* and *get_close_matches* functionalities to compute string delta and match similar strings. The threshold t_m for method name matching and t_s for statement delta matching were set to 0.6 and 0.8, respectively.

IV. Evaluation

We evaluate our change-partition approach in this section (**RQ2**). Section IV-A and IV-B report the quantitative results and highlight useful examples of change partitioning. Section IV-C explains reasons for unsatisfactory partitions.

A. Results

As reported in Section II-A, 78 out of 453 code changes we observed address multiple development issues. We used these 78 composite changes to build "ground truth". Since code semantics are subject to the understanding and interpretation of human developers, the solution space of partitioning code changes is potentially infinite. However, our ultimate goal of partitioning composite changes is to facilitate code review. In this regard, if a change-partition result of our automatic approach matched one of the (many) ways humans will partition the change, then this result can be considered *acceptable*.

Based on this assumption, three human evaluators, which included the first author and two external computer science graduate students, manually partitioned these 78 changes. Each evaluator independently identified the number of issues addressed in each change and the corresponding code. Then, they discussed to establish an agreement on the partitions. Finally, for each change, we used the manual partition that was agreed by all three evaluators as "ground truth".

We then compared partitions of our automatic approach to the ground truth. An automatic partition is considered to be *acceptable* if it exactly matched the manual partition in terms of the number of change-slices and the content of each change-slice. Table II reports the evaluation results for each project. From 53% to 76% and on average 69% of our partition results are acceptable.

TABLE II: Evaluation results. Among the 78 composite changes (Section II-A), 54 (69%) were automatically partitioned the same way as human evaluators did.

	Acceptable # / Total #
Ant	8 / 11 (72.7%)
Commons Math	10 / 19 (52.6%)
Xerces	16 / 21 (76.2%)
JFreeChart	20 / 27 (74.1%)
Total	54 / 78 (69.2%)

B. Partition Examples

We now highlight a few cases to illustrate how automatic change partition can potentially help developers' code review process.

- Case 1: Highlighting intended fix

JFreeChart revision 1083 touches one file and modifies six lines of code. According to its commit log, this revision fixes a bug by modifying the method *createCopy* to handle an empty range. Our approach partitions this revision into three change-slices, as shown in Figure 7. In particular, the second change-slice contains the intended fix exclusively, which is separated it from a minor removal of the *this* keyword in the first change-slice and a formatting change in the third change-slice.

- Case 2: Isolating inconspicuous but possibly important changes

Using our approach, JFreeChart revisions 1366 and 1801 are partitioned into three and two change-slices, respectively. Figure 8 shows one of the change-slices from both revisions

1st Change-slice	
return this.addOrUpdate(period, new Double(value));	return addOrUpdate(period, new Double(value));
2nd Change-slice	
if (endIndex < 0) {	if ((endIndex < 0) || (endIndex < startIndex)) {
emptyRange = true;	emptyRange = true;
}	}
3rd Change-slice	
if (!ObjectUtilities.equal(	if (!ObjectUtilities.equal(getDomainDescription(),
getDomainDescription(), s.getDomainDescription()	s.getDomainDescription())) {
)){	
return false;	return false;
}	}
if (!ObjectUtilities.equal(	if (!ObjectUtilities.equal(getRangeDescription(),
getRangeDescription(), s.getRangeDescription()	s.getRangeDescription())) {
)){	
return false;	return false;
}	}

Fig. 7: JFreeChart revision 1083 is partitioned into three slices. The second exclusively contains the intended fix.

One of the three change-slices from JFreeChart revision 1366	
this.strokeList = new StrokeList();	this.strokeList = new StrokeList();
this.baseStroke = DEFAULT_STROKE;	this.baseStroke = DEFAULT_STROKE;
this.autoPopulateSeriesStroke = false;	this.autoPopulateSeriesStroke = true;
One of the two change-slices from JFreeChart revision 1801	
if (tick.getValue() != 0.0	if ((tick.getValue() != 0.0
|| !isRangeZeroBaselineVisible() && (paintLine)) {	|| !isRangeZeroBaselineVisible()) && paintLine) {
getRenderer().drawRangeLine(g2, this, getRangeAxis(),	getRenderer().drawRangeLine(g2, this, getRangeAxis(),
area, tick.getValue(), gridPaint, gridStroke);	area, tick.getValue(), gridPaint, gridStroke);
}	}

Fig. 8: Inconspicuous but important changes, which are not mentioned in commit logs but are isolated by our automatic partitioning.

public void setForce(boolean force) {	public void setForce(boolean forceOverwrite) {
this.forceOverwrite = force;	this.forceOverwrite = force; (*)
}	}
	public void setOverwrite(boolean forceOverwrite) {
	setForce(forceOverwrite);
	}

Fig. 9: The second change-slice from Ant revision 943068 reveals an error.

that contains only one-line change. For revision 1366, a field is set to *true* instead of *false*. For revision 1801, the one-line change fixes a subtle bug in an incorrect if-condition where a parenthesis for the "||" operation is missing.

Although both fixes are small, they indeed have a significant impact on the program behavior. For example, the missing parenthesis in revision 1801 messes up the operation precedence and flips the if-condition value, which may cause the program to produce wrong output or even crash. However, due to its small size, such a crucial fix might be missed by code reviewers especially when the change is composite or its commit log is not informative (the commit log for the above two revisions are both "Synchronised with 1.0.x branch"). To increase the awareness of such small but critical changes, automatic partitioning may come in handy by isolating them from a potentially large and complex composite change.

- Case 3: Revealing suspicious changes

Our approach partitions Ant revision 943068 into two change-slices. While the first change-slice faithfully matches the description in the commit log, the second change-slice looks suspicious (Figure 9). In this change-slice, the parameter *force* is renamed as *forceOverwrite* in the method *setForce (boolean)*, which is invoked by a newly added method *setOverwrite(boolean)*. The intention here is probably adding a setter for the field *forceOverwrite*, but the change seems to fail on this purpose: the renamed parameter does not really affect the field assignment at (*). We suspected that *force* at the right hand side of the assignment should also be changed to *forceOverwrite*.

We searched subsequent revisions for more evidence of our speculation. As expected, the same developer of this change fixed the bug later in revision 943070, along with a commit log saying "*wrong assignment after I renamed the parameter. Unfortunately there doesn't seem to be a testcase that catches the error.*[3]" If our approach had been applied in this case, it might have been much easier for code reviewers to spot this sneaky error, which passed tests but was isolated as an individual change-slice.

RQ2: We used our approach to automatically partition 78 composite code changes and 54 (69%) are partitioned the same way as human evaluators did.

C. Reasons for Unsatisfactory Partitions

We now discuss the reasons why our approach produces *unsatisfactory* partitions for the remaining 24 (31%) code changes. First, our approach may incorrectly partition changes that address only one single issue. In Commons Math revision 1230907, two methods *operateTranspose* and *isTransposable* are added to "*support for transposition of linear operators*". However, no static dependency exists between these two methods and the similarity between their names is below the threshold. As a result, our approach partitions this revision into two change-slices. Similarly in Commons Math revision 1206655, two statically-independent statements "*return (long) -magnitude*" and "*return (int) -magnitude*" in different methods are changed to "*return -magnitude*", both removing the unnecessary cast. However, our approach partitions them into separate change-slices since their delta "*(long)*" and "*(int)*" fail to match. We plan to address this limitation by integrating advanced rule mining tools such as LSdiff [13] to capture more complex change patterns.

Second, our approach may fail to partition changes that address different issues. In JFreeChart revision 1370, the *this* keyword is removed in several places, which in fact constitutes trivial or *non-essential* modifications [14] that should be separated from the remaining changes. Our current approach is unable to detect such non-essential changes. In the future, we plan to apply DiffCat [14] to isolate non-essential changes as individual change-slices.

[3]http://svn.apache.org/viewvc?view=revision&revision=943070

TABLE III: Changes used in each session of the user study. This table shows the size of each change and its number of slices after partitioning. The last two columns present the code review questions for participants and the corresponding information covered in each question.

	Change	# files	# cLOC	# slices	Code Review Questions	Information
Session 1	Ant 943068	2	21	2	The commit log for this revision is "deal with read-only dest files in echo and concat." Please briefly explain how this is achieved. Is there any problematic change in this revision? If so, please explain briefly.	Rationale Correctness
	Xerces 730238	14	22	2	The commit log for this revision is "remove schemaType field." Where is this field removed? What is the consequence of this removal?	Location Impact
	JFreeChart 1366	1	12	3	Please briefly summarize this change.	Rationale
	Xerce 779777	11	98	3	There is a similar pattern in this big change. Please briefly explain this pattern. Is there any change(s) in this revision that doesn't match this pattern? If so, please point it out.	Similarity Difference
	JFreeChart 1663	2	29	3	Please briefly describe the issue(s) addressed in this change.	Rationale
	Xerces 810237	2	47	3	How many types of validity checkers are added in this change? Please briefly explain each of them. In addition to adding validity checkers, does this change address other issue(s) (e.g., fixing another bug)? If so, please explain briefly.	Rationale Behavior
	Change	**# files**	**# cLOC**	**# slices**	**Code Review Questions**	**Information**
Session 2	Math 1210359	2	7	3	Please briefly describe the issue(s) addressed in this change.	Rationale Correctness
	Xerces 778245	6	14	2	Please briefly summarize this change.	Rationale
	Xerces 890457	4	56	3	This revision changes three files with the suffix "DocumentImpl" (i.e., HTMLDocumentImpl, WMLDocumentImpl, and CoreDocumentImpl). What do these changes have in common? In addition, the file "ElementImpl" is also changed. Does this change have anything in common with changes in other files? If so, please explain briefly.	Location Similarity Impact
	JFreeChart 1576	10	31	3	Please briefly describe the issue(s) addressed in this revision.	Rationale
	JFreeChart 1596	11	54	5	There is a similar pattern in this big change. Please briefly explain this pattern. Is there any change(s) in this revision that doesn't match this pattern? If so, please point it out.	Similarity Difference
	JFreeChart 1801	1	64	2	Common types of changes include adaptive (e.g., new feature implementation), corrective (e.g., bug fix), and perfective (e.g., refactoring). What types of changes are involved in this revision? Please explain briefly.	Rationale Behavior

In addition, our approach relies on ChangeDistiller's output. However, ChangeDistiller tends to produce inaccurate differencing results when changes occur in a small sub-AST (e.g., a small if-statement) [11]; consequently, the partition results can be affected.

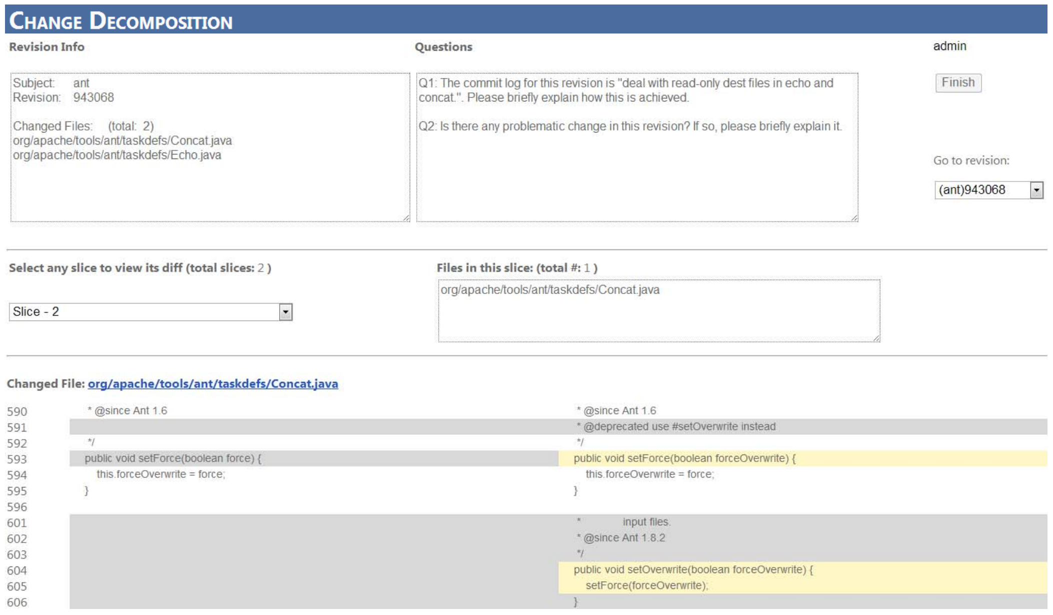
V. Preliminary User Study

In this section, we explore whether our change-partition approach can facilitate code review (**RQ3**). We conducted a preliminary user study, in which participants review composite code changes with or without partitioning and answer a series of code review questions. Similar to previous studies [15], [16], [17], we used the time participants spent on the tasks and the correctness of their answers to quantify their code review performance. Accordingly, we formulated two null hypotheses to be tested in our user study:

- $H1_0$: Partitioning does not affect the time needed for code review.
- $H2_0$: Partitioning does not affect the correctness of code review tasks.

The alternative hypotheses we used in the study are:

- $H1$: Partitioning reduces the time needed for code review.
- $H2$: Partitioning increases the correctness of code review tasks.



Subject: ant
Revision: 943068

Changed Files: (total: 2)
org/apache/tools/ant/taskdefs/Concat.java
org/apache/tools/ant/taskdefs/Echo.java

Q1: The commit log for this revision is "deal with read-only dest files in echo and concat.". Please briefly explain how this is achieved.

Q2: Is there any problematic change in this revision? If so, please briefly explain it.

Select any slice to view its diff (total slices: 2)

Slice - 2

Files in this slice: (total #: 1)

org/apache/tools/ant/taskdefs/Concat.java

Changed File: org/apache/tools/ant/taskdefs/Concat.java

```
 * @since Ant 1.6                                  * @since Ant 1.6
                                                   * @deprecated use #setOverwrite instead
 */                                                */
public void setForce(boolean force) {             public void setForce(boolean forceOverwrite) {
   this.forceOverwrite = force;                      this.forceOverwrite = force;
}                                                 }

                                                   *          input files.
                                                   * @since Ant 1.8.2
                                                   */
                                                  public void setOverwrite(boolean forceOverwrite) {
                                                     setForce(forceOverwrite);
                                                  }
```

Fig. 10: The screenshot of the user study website.

A. Study Design

Developers typically review a code change by navigating the list of changed files and viewing each file's diff. We called this traditional approach *by file* and used it as the control treatment. As the experimental treatment, which we refer to as *by slice*, a code change is represented by a list of change-slices.

We employed a balanced experiment design [18] that involved two lab sessions, each comprised a different set of code review tasks (Section V-B). Study participants were evenly divided into two groups according to their expertise (Section V-C). These two groups worked in the first session under different treatments and worked in the second session with treatments switched (Table IV). In other words, the same set of review tasks was completed by the two groups under different treatments (each column in Table IV) and each group was exposed to both treatments in different sessions (each row in Table IV).

TABLE IV: User Study Design.

	Session 1	**Session 2**
	(changes 1-6)	(changes 7-12)
Group 1	By slice	By file
Group 2	By file	By slice

B. Code Review Tasks

In Section IV-A, we report that our automatic approach produces acceptable partitions for 54 out of 78 composite code changes. As a preliminary investigation of the potential benefits of change partition, we currently selected only from these 54 composite changes with acceptable partitions for our user study. We discuss the potential impact of unsatisfactory change partitions in Section VI-A.

Since participants were exposed to both treatments, changes and review tasks in each session should be different in content but similar in levels of difficulty [19]. In addition, we need to ensure the appropriate length of the user study, which is generally suggested to be 60 to 90 minutes or shorter [20]. To satisfy the above criteria, we first characterized each composite change by its size, complexity, and number of change-slices after partitioning. We then selected twelve composite changes such that each session included six changes with similar characteristics (Table III).

We conceived code review questions for each of the twelve changes (Table III) following guidelines suggested by the literature [5], [21], [13]. We then decided a detailed grading scheme for these twelve changes based on their commit logs and the ground truth we established in Section IV-A.

C. Participants

We recruited 18 computer science students to participate in our user study. Two of them are senior undergraduate students, 10 are Masters students, and 6 are PhD students. Prior to the study, we conducted a survey to measure each participant's expertise. Specifically, we asked participants to rate their familiarity with Java, Source Control Management systems (SCMs) and *diff* tools on a 5-point Likert scale. We then split these participants into two groups with similar expertise.

D. Procedure

For the purpose of this user study, we created a website that allows participants to review code changes and answer questions (Figure 10). The website displays basic information about each change, such as its project name, revision number, list of changed files, and code review questions. Participants

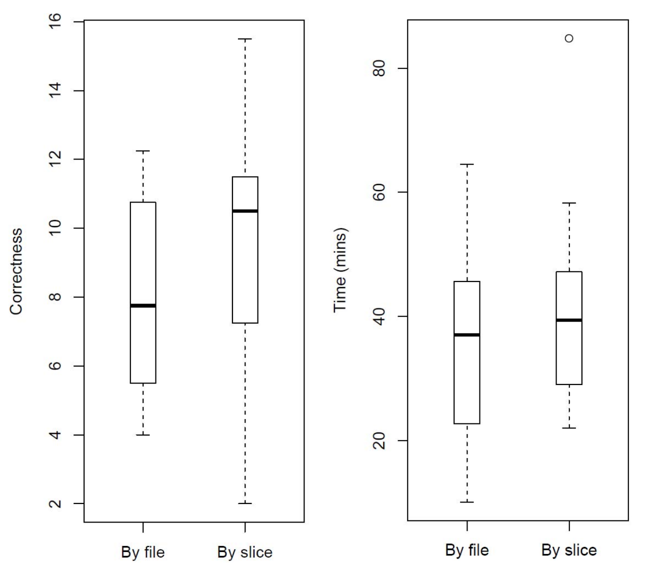


Fig. 11: Boxplot of participants' correctness and time for completing all code review tasks under the two different treatments.

TABLE V: User Study Results.

	Correctness		Time (mins)	
	By file	**By slice**	**By file**	**By slice**
mean	7.94	9.40	35.92	40.60
median	7.75	10.5	37.05	39.41
stdev.	2.86	3.71	14.91	15.50
One-tailed Paired-samples *t*-test (confidence level 95%)				
p-value	0.01		0.81	

review a change by navigating the list of changed files under the *by file* treatment or the list of change-slices under the *by slice* treatment. When each file or change-slice is selected, the website shows its content as a side-by-side diff.

The user study started with a 10-minute introduction, in which the coordinator described the tasks to complete and provided a short tutorial on using the website. Participants then started the first session. The entire study was supervised: the coordinator was responsible for ensuring correct website usage, preventing collaboration, and providing clarifications on task descriptions. Upon the completion of the first session, participants took a 10-minute break before they proceeded to the second session. We recorded the time each participant spent on completing each code review task. Their written answers were collected at the end of the second session and graded according to our marking scheme.

E. User Study Results

Figure 11 shows the boxplot of participants' correctness of answers and the time they spent on all code review tasks under the *by file* and *by slice* treatments. The corresponding descriptive statistics are shown in Table V.

The correctness data passed the Kolmogorov-Smirnov test [22] and Levene's test [23], indicating that they are normally distributed and have equal variances. We used the one-tailed Paired-samples *t*-test to test hypothesis $H2_0$. Table V shows that the *p*-value is $0.01<0.05$, indicating that $H2_0$ can be rejected in favor of the alternative hypothesis $H2$. That is, change-partition significantly increases the correctness of code review tasks.

Similarly, we tested the time difference between the *by file* and *by slice* treatments. As shown in Table V, the *p*-value is 0.81, meaning that we cannot reject $H1_0$. In other words, change-partition does not affect the time needed for code review. Figure 11 shows that participants even spent a slightly longer time under the *by slice* treatment. We conjecture that such a time increase mainly resulted from participants' additional learning cost for using the unfamiliar "change-slices" to review code changes. We further discuss this issue in Section VI-B.

RQ3: In a similar amount of time, participants answered code review questions significantly better when the composite changes under review were partitioned.

VI. Discussions

In this section, we discuss the impact of unsatisfactory partitions and the balance between change-partition's benefits and costs. We also envision other potential applications of our approach.

A. Impact of Unsatisfactory Change Partitions

Section V reports our preliminary user study, in which participants given properly partitioned changes conducted more effective code review compared to those who were given the original composite changes. To obtain a well-rounded evaluation on our approach, we plan to conduct a further user study to evaluate the impact of unsatisfactory change partitions in code review. Specifically, we plan to add one more experimental group to our current study design. Participants in this new experimental group will review code changes that are "over-partitioned" such that changes addressing only one issue are partitioned into multiple change-slices.

B. Balancing between Partitioning Costs and Benefits

Our preliminary user study shows that change-partition can help participants conduct code review more effectively. However, change-partition also comes with a cost. In addition to its execution overhead, code reviewers might spend extra time on getting familiar with the *by-slice* way of review (Section V-E). Code reviewers might also be distracted if they do not agree with the partition results.

For these reasons, how to balance between the costs and benefits of change-partition becomes a crucial question. One important variable in addressing this question could be the size or complexity of code changes, as partition might be more helpful for reviewing large and complex instead of small and straightforward code changes. A systematic evaluation of this assumption, however, remains as future work.

C. Potential Applications

In addition to facilitate code review, we further identify the following potential applications of change partitioning:

Pre-commit inspection: Developers typically double-check their local changes before commit to reduce the risk of integration [24]. We expect that change-partition can also aid this pre-commit inspection. If an about-to-commit change is partitioned into several change-slices, developers may wonder why so many change-slices exist and inspect whether irrelevant changes are accidentally introduced.

Commit assistance: Modern SCMs such as Git allow developers to commit a part of their changes. However, developers have to manually select the part to commit, which is time-consuming and error-prone [5]. If change-partition was applied in this case, developers could simply select from the automatically generated change-slices and perform an atomic commit.

Partial rollback of code changes: Modern SCMs provide a certain degree of support for developers to undo incorrect or imperfect code changes and return to a previous clean state. Nevertheless, the granularity of such support is usually at commit-level or file-level, meaning that developers can only undo either the entire changes in a commit/file or nothing at all. This mechanism may work perfectly when the target change is atomic and addresses only one single issue. However, if the change is composite but a developer wants to undo only a part of it, the current all-or-nothing revert-mechanism seems insufficient. We envision that our change-partition approach can be applied here so that developers can simply specify one or more change-slices to revert.

VII. Threats to Validity

We observed 17% code changes of the four projects to be composite. However, this observation might not be generalizable to other software projects. Yet, we speculate that projects who embrace code review [24] and the "commit-early-and-often" practice [25] should have less composite code changes compared to those who do not.

We considered an automatic partition to be acceptable if it was exactly the same as the manual partition agreed by three human evaluators (Section IV-A). There might be other different partitions that are also acceptable, which, however, are difficult to enumerate. In addition, we used *exact* match instead of *partial* match in our evaluation. If we gave credits to an automatic partition that is partially acceptable (e.g., one of its change-slices exactly matches the manual partition but the other two of its change-slices are different), our evaluation results could be different.

The sizes of the twelve code changes used in our user study are small to medium, with 1 to 14 files and on average 38 changed lines of code. With these changes, we observed that participants answered code review questions more correctly on partitioned changes but with slightly longer time. However, this result may not generalize to larger code changes. As discussed in Section VI-B, the size and complexity of code changes could have affected the benefits (e.g., correctness) and costs (e.g., time) of using change partition.

The questions asked in the user study might not be complete for general code review practice. However, we conceived all code review questions following the guidelines from literature [5], [21], [13] to maximize their representativeness. Another threat is that the participants in our study may not well represent the population of real developers.

Our user study results might have been affected by the differences between individual participants. We minimized this threat by adopting a within-group design [19], such that each participant was exposed to both the *by file* and *by slice* treatments. In addition, we conducted a pilot survey to evenly divide participants according to their expertise.

Finally, the user study results may have been susceptible to the impact of learning effects [19]. We minimized this threat by assigning different orders of treatments to participants (i.e., group-1 used *by slice* in the first session while group-2 used *by file* first, as shown in Table IV). We also provided a tutorial of both treatments prior to the study.

VIII. Related Work

A. Describing Code Changes

Various techniques have been proposed to help developers answer the "*what is changed*" question. ChangeDistiller compares the abstract syntax trees of two program versions to extract fine-grained syntax changes [11]. The DeltaDoc algorithm explains a program change by differentiating the program's symbolic execution before and after the change to generate descriptions readable by humans [26]. These techniques are applied directly to an entire code change, which is considered as an inseparable unit by default. However, we have empirically showed that this is not always the case.

B. Separating Concerns from Code Changes

The existence and prevalence of composite code changes have recently been recognized by the research community. Murphy-Hill et al. provided empirical evidence that the majority of refactorings are in fact *floss refactorings*, which mix refactorings with other types of development activities [27]. Kawrykow and Robillard reported that up to 15.5% of method updates are merely non-essential modifications such as local variable refactorings [14]. Herzig and Zeller reported that between 6% and 15% of bug fixes address multiple concerns [28]. Complementary to their findings, we conducted a manual investigation of code changes that are not limited to bug-fixing and refactorings.

Several studies have aimed at separating concerns from code changes. DiffCat distinguishes non-essensial and non-trivial changes in a single commit [14]. LSdiff characterizes a code change by a list of logical rules and exceptions [13]. Licata et al. characterized a change by its feature signature, which is derived from test suite executions [29]. Collard et al. proposed *difffact*, which allows users to specify the type and location of changes to be filtered out from a large change [30].

Herzig and Zeller developed an approach to untangle non-atomic bug-fixing changes [28]. Their work mainly differs from ours in that their goal was to improve mining-software-archive research by removing noises from bug-fixing changes. Therefore, they evaluated the impact of non-atomic bug-fixes on labeling defect-prone instances. We instead partition changes to facilitate code review, which is mainly evaluated through our user study. In addition, Herzig and Zeller created artificially tangled changes as input and evaluated their approach based on partial match. We instead used real code changes and evaluated our approach using exact match (Section IV-A). In terms of the approach itself, Herzig and Zeller also used heuristics such as file distance and change coupling to untangle changes. We plan to compare their approach with ours and merge useful heuristics if necessary.

Independently from us, Barnett et al. proposed CLUSTERCHANGES that automatically decomposes changesets to help code review [31]. This tool leverages *def-use* information exclusively on the additions and modifications of a changeset [31]. However, our approach uses three heuristics, including static dependency, to partition an entire change including code deletions. In addition, Barnett et al. interviewed developers to assess their tool's usefulness in code review, while we conducted controlled experiments involving actual usages of partition results and code review tasks to this end.

C. Program Slicing on Code Comprehension

Although static program slicing was originally proposed to facilitate debugging [6], it was later found useful in many other applications such as testing, validation, program parallelization, and reverse engineering [32], [33]. Static slicing has also been applied to program comprehension, which is closely related to our work. Gallagher and Lyle introduced *decomposition slice*, which captures all computations on a given variable independent of program points, to help developers understand the semantic contexts of maintenance tasks [34]. De Lucia et al. proposed *conditioned slicing* as a general framework for program comprehension [10]. Komondoor and Horwitz used program dependence graphs and program slicing to identify code duplication, alleviating the difficulty caused by duplicate code in program understanding [35]. Different from theirs, our work leverages program slicing to partition composite code changes and aid code review.

IX. CONCLUSIONS

"*More is not always better.*" When multiple development issues are addressed in a code change, reviewing this change can be difficult and error-prone. We propose an approach to partition such composite code changes into change-slices so that each change-slice is more semantically cohesive. In an evaluation on 78 real composite code changes, our approach partitions 54 (69%) of them the same way as human evaluators manually did. Our user study further shows that when composite code changes are properly partitioned, participants' code review effectiveness is significantly improved.

As future work, we plan to incorporate change rule mining and non-essential change detection techniques into our change-partition approach. Meanwhile, we plan to conduct a more thorough user study to also investigate the potential impact of unsatisfactory change partitions on code review.

ACKNOWLEDGMENT

We thank all the students that participated in our user study. We especially thank Tao He and Hai Wan for their kind help of arranging the user study. We also thank Ananya Kanjilal for her comments on the early draft of this paper.

REFERENCES

[1] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *Software Engineering, IEEE Transactions on*, vol. 35, no. 5, pp. 684–702, 2009.

[2] H. C. Benestad, B. Anda, and E. Arisholm, "Understanding software maintenance and evolution by analyzing individual changes: a literature review," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 6, pp. 349–378, 2009. [Online]. Available: http://dx.doi.org/10.1002/smr.412

[3] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 35th international conference on Software engineering*, ser. ICSE '13, 2013.

[4] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey, "Peer review on open-source software projects: Parameters, statistical models, and theory," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 35:1–35:33, Sep. 2014. [Online]. Available: http://doi.acm.org/10.1145/2594458

[5] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 51:1–51:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393656

[6] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/358557.358577

[7] G. Canfora, A. Cimitile, A. De Lucia, and G. Di Lucca, "Decomposing legacy programs: a first step towards migrating to client-server platforms," in *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, 1998, pp. 136–144.

[8] K. Gallagher and J. Lyle, "Using program slicing in software maintenance," *Software Engineering, IEEE Transactions on*, vol. 17, no. 8, pp. 751–761, 1991.

[9] F. Lanubile and G. Visaggio, "Extracting reusable functions by flow graph based program slicing," *Software Engineering, IEEE Transactions on*, vol. 23, no. 4, pp. 246–259, 1997.

[10] A. De Lucia, A. Fasolino, and M. Munro, "Understanding function behaviors through program slicing," in *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, 1996, pp. 9–18.

[11] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70731

[12] T.J.Watson Libraries for Analysis. [Online]. Available: http://wala.sourceforge.net

[13] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070531

[14] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 351–360. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985842

[15] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *Software Engineering, IEEE Transactions on*, vol. 37, no. 3, pp. 341–355, 2011.

[16] C. Lange and M. R. V. Chaudron, "Interactive views to improve the comprehension of uml models - an experimental validation," in *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, 2007, pp. 221–230.

[17] J. Quante, "Do dynamic object process graphs support program understanding? - a controlled experiment." in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, 2008, pp. 73–82.

[18] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "The effectiveness of source code obfuscation: An experimental assessment," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, 2009, pp. 178–187.

[19] J. Lazar, J. H. Feng, and H. Hochheiser, *Research Methods in Human-Computer Interaction*. John Wiley Sons, 2010.

[20] J.Nielsen. Time budgets for usability sessions. [Online]. Available: http:// www.useit.com/alertbox/usability_sessions.html

[21] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 23–34. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181779

[22] F. E. J. M. R. W. T. Eadie, D. Drijard and B. Sadoulet, *Statistical Methods in Experimental Physics*.

[23] H. Levene, *Robust tests for equality of variances*. Stanford University Press, 1960.

[24] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: a case study of the apache server," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 541–550. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368162

[25] "Commit Often, Perfect Later, Publish Once: Git Best Practices," https://sethrobertson.github.io/GitBestPractices/, 2012.

[26] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859005

[27] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 287–297. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070529

[28] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 121–130. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487113

[29] D. Licata, C. Harris, and S. Krishnamurthi, "The feature signatures of evolving programs," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, 2003, pp. 281–285.

[30] M. L. Collard, H. Kagdi, and J. I. Maletic, "Factoring differences for iterative change management," in *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, ser. SCAM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 217–226. [Online]. Available: http://dx.doi.org/10.1109/SCAM.2006.15

[31] M. Barnett, C. Bird, J. Brunet, and S. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets." in *Proceedings of the 37th International Conference on Software Engineering*, May 2015. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=238937

[32] A. De Lucia, "Program slicing: methods and applications," in *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, 2001, pp. 142–149.

[33] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1050849.1050865

[34] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, pp. 751–761, 1991.

[35] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, ser. SAS '01, London, UK, 2001, pp. 40–56.