Investigating the Relationship between Evolutionary Coupling and Software Bug-proneness

Manishankar Mondal Banani Roy Chanchal K. Roy Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
{mshankar.mondal, banani.roy, chanchal.roy, kevin.schneider}@usask.ca

ABSTRACT

While designing a software system, minimizing coupling among program entities (such as files, classes, methods) is always desirable. If a software entity is coupled with many other entities, this might be an indication of poor software design because changing that entity will likely have ripple change effects on the other coupled entities. Evolutionary coupling, also known as change coupling, is a well investigated way of identifying coupling among program entities. Existing studies have investigated whether file level or class level evolutionary couplings are related with software bug-proneness. While these existing studies have mixed findings regarding the relationship between bug-proneness and evolutionary coupling, none of these studies investigated whether method level (i.e., function level for procedural languages) evolutionary coupling is correlated with bug-proneness. Investigation considering a finer granularity (i.e., such as method level granularity) can help us pinpoint which methods in the files or classes are actually responsible for coupling as well as bug-proneness.

In this research, we investigate method level evolutionary coupling through mining association rules and analyze whether this coupling is correlated with software bug-proneness. According to our investigation on thousands of commit operations from the evolutionary history of six open-source subject systems, method level evolutionary coupling generally has a good positive correlation with software bug-proneness. Our regression analysis indicates that evolutionary coupling and bug-proneness mostly have a linear relationship between them. We also observe that methods that experience bug-fixes during evolution generally have a significantly higher number of evolutionary coupling links than the methods that do not experience bug-fixes. We realize that minimizing method level evolutionary coupling links can help us minimize bugs in software systems. Our prototype tool is capable of identifying highly coupled methods along with their coupling links so that programmers can find possibilities of minimizing those links for reducing bug-proneness of software systems.

KEYWORDS

Evolutionary Coupling, Software Bug-proneness, Correlation, Regression, Software Maintenance and Evolution

CASCON '19, Nov. 4–6, 2019, Toronto, ON, Canada
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference format:
Manishankar Mondal Banani Roy Chanchal K. Roy Kevin A. Schneider. 2019. Investigating the Relationship between Evolutionary Coupling and Software Bug-proneness . In *Proceedings of Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, Toronto, ON, Canada, Nov. 4–6, 2019 (CASCON '19),* 10 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software systems will undergo changes during evolution and maintenance. However, coupling among program entities (such as files, classes, or methods) in a software system often introduces challenges in making changes to the software system's code-base [43]. If a target program entity (an entity which is going to be changed by a programmer) is coupled with several other entities, a change in that target entity might have ripple change effects on the coupled entities [41]. Changing a target program entity without properly analyzing its impacts on the coupled entities is likely to introduce bugs in a software system [40]. As entity coupling often introduces challenges in making changes to the code-base, lower coupling among entities is always desirable [42, 43]. Our research in this paper investigates a particular type of coupling called *evolutionary coupling* [5, 14, 33] and its relationship with software bug-proneness.

Evolutionary coupling is a well investigated phenomenon in the realm of software engineering research and practice. If two or more program entities appear to co-change (change together) frequently during software evolution, it is expected that the entities have coupling. Such a coupling is termed as *evolutionary coupling* (or change coupling) in the literature [4, 5, 7, 8]. If a group of program entities exhibited evolutionary coupling in the past, a change in one of the entities in future may require corresponding changes to the other entities in the group. According to the existing studies [15, 29], evolutionary coupling can discover those couplings among entities which are missed by the other coupling measures. In our research, we investigate method level evolutionary coupling and its relationship with bug-proneness.

A number of existing studies [31, 34, 35, 37] have investigated relationship between evolutionary coupling and software fault-proneness. For example, Kirbas et al. [35] investigated if file level evolutionary coupling is correlated with bug-proneness. Graves et al. [34] found that module level (a higher granularity than file level) evolutionary coupling is a poor predictor of software defects. Knab et al. [37] showed that file level evolutionary coupling cannot be a predictor of defects in Mozilla project. However, D'Ambros et al. [31] found class level evolutionary coupling to be correlated with software fault-proneness. We see that all these studies were conducted considering coarse grained (file level, class level, or module level) evolutionary coupling and they report mixed findings.

Moreover, analysis using coarse grained couplings makes it difficult to realize which finer grained entities, such as methods (functions in procedural language), are actually responsible for coupling and bug-proneness. In our study, we investigate whether method level evolutionary coupling is correlated with bug-proneness in software systems and whether such a fine-grained coupling information can provide us insights towards minimizing bug-proneness.

We mine method level association rules from thousands of commit operations of our subject systems. We propose a measure to quantify the evolutionary coupling among methods. We also measure the bug-proneness of our candidate systems and analyze whether this measure of bug-proneness is correlated with the method level evolutionary coupling measure. We answer the following important research questions through our research.

- **RQ 1:** Is method level evolutionary coupling of a software system related with its bug-proneness?
- **RQ 2:** Do methods that experience bug-fixes have a higher number of evolutionary coupling links than the others that do not experience bug-fixes?

While the first research question analyzes the correlation and regression between method level evolutionary coupling and software bug-proneness, the second question investigates how method level evolutionary coupling contributes to software bug-proneness. According to our investigation on thousands of commits of six open source subject systems written in three programming languages (Java, C, and C#), we have the following findings:

- Method level evolutionary coupling generally has a very good positive correlation with software bug-proneness. This correlation is statistically significant.
- Our regression analysis reveals that bug-proneness and method level coupling mostly have a good linear relationship between them. Thus, it is expected that an increase in method level coupling will generally be associated with a corresponding increase in the bug-proneness of a software system.
- The average number of evolutionary coupling links per buggy method (i.e., methods that experienced bug-fixes) is significantly higher (according to our significance test) than the average number of coupling links per non-buggy method (i.e., methods that did not experience bug-fixes).

We believe that our in-depth investigation on fine-grained (method level) evolutionary coupling as well as the findings can be of significant importance for better maintenance and evolution of software systems. The techniques that we have used in our study can help programmers in identifying methods with a large number of evolutionary coupling links. When making decisions about restructuring a software system for better maintenance, the programmers should primarily focus on the highly coupled methods and their coupling links in order to find possibilities of minimizing the coupling links.

The rest of the paper is organized as follows. Section 2 describes the terminology, Section 3 discusses our experiment setup and steps, Section 4 defines our metrics regarding software bug-proneness and method level evolutionary coupling, Sections 5 and 6 answer our research questions by analyzing our experiment results, Section 7 discusses the implications of our findings, Section 8 describes the related work, Section 9 mentions some possible threats to validity, and finally, Section 10 makes a concluding remark.

2 TERMINOLOGY

In the following subsections we describe evolutionary coupling and method genealogy. Method genealogy detection is essential for mining evolutionary coupling among methods.

2.1 Evolutionary coupling

Evolutionary coupling among program entities is a well investigated phenomenon in software engineering research and practice. This coupling can be realized using *association rules* [12, 15] with two related measures: *Support* and *Confidence*.

Association Rule. An association rule [12] is formally defined as an expression of the form $X \; => \; Y$. Here, X is known as the antecedent and Y is the consequent. Each of X and Y is a set of one or more program entities. In the context of software engineering, such an association rule implies that if X gets changed in a particular commit operation, Y also has the tendency of getting changed in that commit operation.

Support and confidence of an association rule. According to Zimmermann et al. [15], *support is the number of commits in which an entity or a group of entities changed together.* Let X is the set of one or more program entities. C_X is the set of commits such that all the program entities in X changed together in each of these commits. Then, support of X can be calculated using Eq. 1.

$$Support(X) = |C_X| \tag{1}$$

Now, the support of the association rule $X => Y$ where each of X and Y is a set of one or more program entities can be calculated using the following equation.

$$Support(X => Y) = Support(X \cup Y) \tag{2}$$

Here, $X \cup Y$ is the union of the sets X and Y. *Confidence of an association rule, $X => Y$, determines the conditional probability that Y will change in a commit operation provided that X changed in that commit operation.* We determine the confidence of the association rule, $X => Y$, using Eq. 3.

$$Confidence(X => Y) = Support(X => Y)/Support(X) \tag{3}$$

Let us now consider an example of two program entities E_1 and E_2. These entities can be files, classes, or methods. If E_1 and E_2 have ever changed together (co-changed), we can assume two association rules, $E_1 => E_2$ and $E_2 => E_1$ from these. Suppose, E_1 was changed in four commits: 2, 5, 6, and 10 and E_2 was changed in six commits: 4, 6, 7, 8, 10, and 13. Thus, according to the definition, *Support(E_1)* = 4 and *Support(E_2)* = 6. *Support(E_1, E_2)* = 2, because E_1 and E_2 changed together (co-changed) in two commits: 6 and 10. Again, $Support(E_1 => E_2) = Support(E_2 => E_1) = Support(E_1, E_2) = 2$. Now, *confidence ($E_1 => E_2$) = support($E_1$, E_2) / support(E_1) = 2 / 4 = 0.5* and *confidence($E_2 => E_1$) = 2 / 6 = 0.33.*

In our experiment, we consider method level association rules. The antecedents and consequents of such rules are methods.

2.2 Method Genealogy

During the evolution of a software system, a particular method might be created in a particular revision and the method might remain alive in a number of consecutive revisions. Each of these

Table 1: Investigated subject systems

Systems	Lang.	Domains	LLR	SRev	LRev
Ctags	C	Code Definition Generator	33,270	1	774
jEdit	Java	Text Editor	191,804	3791	4000
SqlBuddy	C#	A tool for using with Microsoft SQL Server and MSDE	18,387	1	945
Camellia	C	Image Processing Library	93,396	1	170
Carol	Java	Game	25,091	1	1700
DNSJava	Java	DNS protocol	23,373	1	1635

LLR = LOC in the Last Revision
SRev = Starting Revision LRev = Last Revision

revisions contains a snap-shot of the method. The genealogy of the method consists of the snap-shots of it from all those revisions where it was alive. Thus, the genealogy of a particular method can help us analyze how it was changed over the evolution. If two or more methods have ever co-changed (changed together) during evolution, we can identify it by examining their genealogies.

3 EXPERIMENT SETUP AND STEPS

We conduct our experiment on six open-source subject systems that we download from an on-line SVN repository called Source-Forge.net [11]. Table 1 lists these systems along with their attributes such as sizes, application domains, starting and ending revisions, and implementation languages. While the systems, jEdit, SqlBuddy, Carol, and DNSJava are object oriented systems (written in Java and C#), the remaining systems, Ctags and Camellia, are non-object-oriented systems (written in C). We also see that the subject systems differ in their sizes, application domains and revision history lengths. We select our subject systems in this way in order to generalize our findings. We see that the starting revision of our subject system jEdit is 3791, while the starting revision for the other systems is 1. In the case of jEdit, we did not get the former revisions (1 to 3790) in the on-line repository possibly because it was considered for storing in the on-line repository from its 3791th revision. We perform the following experimental steps on each subject system.

(1) Downloading all the revisions of the subject system from the on-line SVN repository.

(2) Extracting source code changes between every two consecutive revisions using the UNIX diff operation.

(3) Detecting methods from each of the revisions by applying the tool called Ctags [19].

(4) Detecting method genealogies by considering all the methods detected from all the revisions using the genealogy detection technique proposed by Lozano and Wermelinger [38].

(5) Mapping the previously extracted changes to the already detected methods of different revisions so that we can identify which methods were changed during software evolution. Let us assume that we have extracted source-code changes between the revisions r_i and r_{i+1} using diff. From the diff output, we determine which changes occurred at which lines of which source code files in revision r_i. We also detect all the methods along with their starting and ending line numbers from all the source code files of revision r_i. Using the line number of the changes and methods, we determine which methods in r_i intersect with the changes that occurred in r_i. In other words, we determine which methods were affected by the changes that occurred in r_i.

(6) Detecting bug-fix changes (i.e., bug-fix commit operations) by automatically analyzing the commit messages. We will elaborate this step in Section 3.1.

(7) Detecting evolutionary coupling among methods by extracting and analyzing association rules from the entire evolutionary history of our subject systems. Section 3.2 will discuss the details.

(8) Calculating a method level evolutionary coupling metric and a bug-proneness metric for each of our subject systems and answering the research questions through correlation and regression analysis. Section 4 will elaborate on the metrics that we calculate. Sections 5 and 6 will answer our research questions.

In the following two subsections we describe how we detect bug-fix commit operations and mine evolutionary coupling (i.e., change coupling) among methods.

3.1 Detecting bug-fix commits

For a particular subject system, we retrieve the commit messages by applying the 'SVN log' command. A commit message describes the purpose of the corresponding commit operation. We automatically examine the commit messages using the heuristic proposed by Mockus and Votta [39] to identify those commits that occurred for the purpose of fixing bugs. The way we detect the bug-fix commits was also followed by Barbour et al. [27]. They investigated whether late propagation in code clones [45] are related to bugs. Our study is different because we investigate whether method level evolutionary coupling is related with software bug-proneness.

3.2 Detecting evolutionary coupling among methods

Evolutionary coupling among methods is realized by detecting association rules among them. Let us assume that the commit operation c_i was applied on the revision r_i of a software system. The immediate next revision r_{i+1} was created because of the commit. We detect the changes that occurred to the code-base of revision r_i by detecting the differences between the corresponding source code files of revisions r_i and r_{i+1} using UNIX *diff*. We then map these changes to the methods of revision r_i to determine which methods in r_i were affected by the commit operation c_i. As we detect method genealogies from the entire history of evolution of a software system, we can determine which methods changed together in which commit operations. We identify all possible pairs of methods such that the two methods in each pair changed together in one or more commit operations. Let us assume that the methods x and y make such a method-pair. For this pair we determine three measures: (i) the number of commits where x changed, (ii) the number of commits where y changed, and (iii) the number of commits where both of the methods changed together. From this pair we determine two association rules: $x \Rightarrow y$ and $y \Rightarrow x$. For each of these rules we determine the support and confidence values using the above three measures by following the equations in Section 2.1.

In our experiment we disregard those association rules where the support is less than 2. Such a discarding of the lowest support (a support of 1) rules was previously done by the existing studies [4, 29] for the purpose of avoiding insignificant rules.

4 EVOLUTIONARY COUPLING AND BUG-PRONENESS METRICS

We determine the following two metrics in order to quantify software bug-proneness and evolutionary coupling among methods.

4.1 The metric for quantifying method level evolutionary coupling

In order to measure the method level evolutionary coupling of a software system, we calculate a metric called **Percentage of Coupled Methods (PCM)**. This is the percentage of changed methods that exhibited evolutionary coupling for a particular period of evolution. Let us assume that M is the set of all those methods that were changed during a period of evolution of a software system. The set of changed methods that exhibited evolutionary coupling in that period is M_c. For detecting which methods exhibited evolutionary coupling in that period, we mine association rules among the methods. We calculate PCM according to the following equation.

$$PCM = (|M_c| \times 100)/|M| \tag{4}$$

Here, |M| is the number of changed methods, and $|M_c|$ is the number of those changed methods that exhibited evolutionary coupling during a period of evolution of a software system.

4.2 The metric for quantifying bug-proneness

Bug-proneness of a software system is the possibility that the system will undergo a bug-fix during its evolution. For measuring the bug-proneness of a software system, we calculate a metric called **Percentage of Bug-fix Commits (PBC)**. This is the percentage of bug-fix commits with respect to all the commits during a particular period of evolution. We determine PBC using Eq. 5.

$$PBC = (NBC \times 100)/TNC \tag{5}$$

Here, TNC is the total number of commits during a period of evolution, and NBC is the number of bug-fix commits that occurred in that period. We should note that in a previous study, Sliwerski et al. [46] proposed an algorithm for identifying fix introducing changes by linking the version history of a software system with its bug-database. However, identifying which changes caused the occurrence of a bug is not the goal of our research. The bug-proneness metric (Eq. 5) that we calculate requires how many bug-fixes occurred during software evolution. As we mentioned previously, for detecting which commits occurred for fixing bugs we use the approach proposed by Mockus and Votta [39]. This approach involves automatically analyzing the commit log that we obtain from a version control system such as SVN or GitHub.

4.3 Rationale behind these metrics

We define these metrics in such a way that they allow us to perform evolutionary analysis on the correlation between bug-proneness and evolutionary coupling. We see that these metrics measure method level evolutionary coupling and bug-proneness considering a particular period of evolution of a software system. Thus, we can determine these metric values for different periods of evolution of our subject systems and can analyze whether the values of bug-proneness in these periods are correlated with the values of evolutionary coupling in the corresponding periods. The next section will describe how we analyze the correlation between bug-proneness and method level evolutionary coupling.

Table 2: Interpretation of correlation coefficient

Range of Correlation Coefficient	Interpretation
Correlation Coefficient is less than 0.1	Trivial Correlation
Correlation Coefficient is between 0.1 and 0.3	Low Correlation
Correlation Coefficient is between 0.3 and 0.5	Moderate Correlation
Correlation Coefficient is between 0.5 and 0.7	High Correlation
Correlation Coefficient is between 0.7 and 0.9	Very High Correlation
Correlation Coefficient is greater than 0.9	Almost perfect correlation

We should also note that as we consider method level (i.e., function level for subject systems written in procedural programming languages) evolutionary coupling, our measure PCM is applicable to subject systems written in both object oriented and non-object-oriented programming languages. Table 1 mentions the subject systems that we have studied in our research. We see that four subject systems: jEdit, SqlBuddy, Carol, and DNSJava were written in the object oriented languages (Java and C#). The remaining two systems Ctags and Camellia were written in C which is a procedural programming language.

Finally, as we consider fine grained evolutionary coupling (i.e., method level coupling) in our study, we can easily pin point which methods in the files or in the classes are actually responsible for the coupling as well as for the bug-proneness. The existing studies [31, 34, 35, 37] that have considered file level, class level, or module level evolutionary coupling cannot identify methods that are responsible for the coupling.

5 RELATION BETWEEN BUG-PRONENESS AND EVOLUTIONARY COUPLING

This section answers our first research question through analyzing the correlation and regression between software bug-proneness and evolutionary coupling.

RQ 1: *Is method level evolutionary coupling of a software system related with its bug-proneness?*

We answer this research question (RQ 1) by dividing it into the following two smaller and more specific questions:

RQ 1.1 *Is method level evolutionary coupling correlated with software bug-proneness?*

RQ 1.2 *Is there a linear relationship between method level evolutionary coupling and software bug-proneness?*

We answer RQ 1.1 through analyzing correlation between the two metrics: PCM (percentage of coupled methods) and PBC (percentage of bug-fix commits). For answering RQ 1.2, we analyze the regression between the two metrics.

5.1 Correlation Analysis

For each of our subject systems, we determine the correlation between bug-proneness and evolutionary coupling in the following way. We first identify the bug-fix commit operations that were experienced by the subject system by following the procedure described in Section 3.1. For different intervals of commit operations, we determine the two measures PCM (Percentage of Coupled Methods) and PBC (Percentage of Bug-fix Commits). Then we analyze whether the PCM values are correlated with PBC values. We explain this with a simple example as follows.

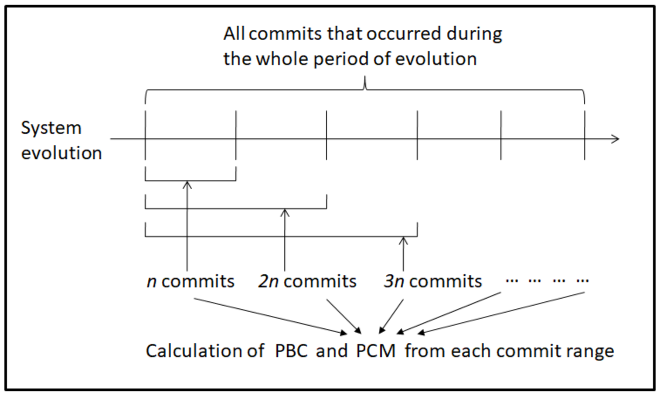

All commits that occurred during
the whole period of evolution
System
evolution
n commits
2*n* commits
3*n* commits
... ...
Calculation of PBC and PCM from each commit range

Figure 1: Calculation of PBC and PCM considering a commit interval of *n*.

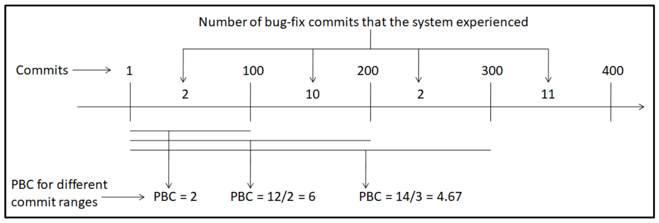


Figure 2: Explanation of calculating PBC considering a commit interval of 100.

Let us consider an interval of n commit operations. We determine the PCM and PBC values from the entire period of evolution on the basis of this interval. We first consider the evolutionary period consisting of the first n commits of a subject system. We determine PCM and PBC for this period using the equations Eq. 4, and 5. Next, we consider the $2n$ commits starting from the first commit. For this period of evolution, we again determine PCM and PBC. Then, we consider the first $3n$ commits and calculate the corresponding PCM and PBC values and so on. In this way, for an interval n, the total number of PCM values that we get from the entire evolutionary history of a subject system consisting of C commits is C/n. We also obtain the same number of corresponding PBC values.

Fig. 1 explains our calculation strategy considering an interval of n commits. We can see the evolutionary periods consisting of n, $2n$, $3n$ commits and so on. From each of these periods, we get a PBC value and a corresponding PCM value. Here, it is important to explain the following two things:

(1) Although the number of commits in the commit ranges is constantly increasing by n in each step as demonstrated in Fig. 1, the values of PCM and PBC are not always increasing because these two values are percentages. PCM is the percentage of changed methods that have exhibited evolutionary coupling and PBC is the percentage of commits that occurred for fixing bugs. Let us assume that the interval $n = 100$. This might be the case that a software system experienced more bug-fixes during its second 100 commits than its third 100 commits as shown Fig. 2. According to Fig. 2, a software system experienced respectively 2, 10, and 2 bug-fix commits during its first, second, and third 100 commits. We can see that the PBC values for the three commit ranges: 1 to 100, 1 to 200, and 1 to 300 are respectively 2, 6, and 4.67. The PBC value decreased in the third commit range. A similar scenario can also occur for PCM.

(2) We see that the ranges of commits that we consider are of different lengths such as n, $2n$, $3n$ and so on, however, each of these ranges starts from the very first commit operation in the history. One can argue that we could determine the coupling and bug-proneness values from the subsequent commit ranges each having n commits. More elaborately, the starting commit in a particular range could be the next commit just after the ending of the previous range. However, we did not follow this way, because the couplings (i.e., method coupling links) introduced in a particular commit range might be the reason behind the bug-proneness exhibited in a later range. In other words, the bug-proneness exhibited in a particular commit range might be the result of couplings introduced in a previous range. With this fact, we should not exclude a former commit range when measuring bug-proneness from newer commits. Finally, we believe that our strategy of measuring PBC and PCM measures by considering different commit ranges starting from the beginning one is reasonable.

For a particular commit interval, n, we determine the commit ranges as shown in Fig. 1 from the entire evolutionary history of a subject system. For each range, we calculate the bug-proneness (PBC) and coupling (PCM) measures. We analyze whether the PBC values obtained from the commit ranges are correlated with the corresponding PCM values obtained from those ranges. We investigate correlation considering different commit intervals (i.e., considering different values of n).

Spearman's Rank Correlation. We determine Spearman's Rank correlation coefficient [24] between bug-proneness (PBC) and evolutionary coupling (PCM) data. We choose Spearman correlation because this is non-parametric, and thus, it does not require the samples to be normally distributed. Table 3 shows the results of correlation analysis between PBC and PCM for different commit intervals from 10 to 100. We can see the correlation coefficient, sample size, and significance of correlation (p-value) considering a significance level of 0.05. As was considered by Kirbas et al. [35], we interpret the values of correlation coefficient according to Table 2. If the p-value regarding a correlation is less than the significance level (0.05), the correlation is considered significant. For our subject systems, jEdit and Camellia, we did not get enough samples for analyzing correlation for higher commit intervals. We have marked these cases as N/A in Table 3.

From Table 3 we see that our subject system Ctags shows almost perfect correlation between PBC and PCM for most of the commit intervals except the intervals of 30 and 90. For these two intervals, the correlation is very high according to Table 2. Moreover, the p-value of correlation considering each commit interval is less that 0.05. Thus, the correlation between PBC and PCM is significant for each commit interval of Ctags. For jEdit, we could perform correlation analysis for the first five commit intervals (10 to 50). For each of these intervals, the correlation between bug-proneness and method coupling is almost perfect and significant. In the case of our subject system, SqlBuddy, the correlation between the two measures (PBC and PCM) is very high for each of the commit intervals. From the p-values we realize that the correlations are significant as well. For the subject system called Camellia, we could find correlation for the first four commit intervals (10 to 40). While the correlation

Table 3: Correlation between PBC (percentage of bug-fix commits) and PCM (percentage of coupled methods)

	Commit Interval	10	20	30	40	50	60	70	80	90	100
Ctags	Rho	0.91015	0.91655	0.87231	0.93158	0.91786	0.93007	0.94546	0.98333	0.80952	0.92857
	p value	0	0	0	0	0	1.00E-05	1.00E-05	0	0.0149	0.00252
	sample size	77	38	25	19	15	12	11	9	8	7
jEdit	Rho	0.94698	0.96364	0.92857	1	0.9					
	p value	0	0	0.00252	0	0.03739	N/A	N/A	N/A	N/A	N/A
	sample size	21	11	7	6	5					
SqlBuddy	Rho	0.84986	0.86716	0.839	0.82481	0.86171	0.86786	0.87912	0.87273	0.87879	0.8
	p value	0	0	0	0	0	3.00E-05	8.00E-05	0.00045	0.00081	0.00963
	sample size	94	47	31	23	18	15	13	11	10	9
Camellia	Rho	0.85177	0.79394	0.81168	0.7						
	p value	0	0.0061	0.04986	0.18812	N/A	N/A	N/A	N/A	N/A	N/A
	sample size	20	10	6	5						
Carol	Rho	0.67285	0.71245	0.68098	0.65414	0.609	0.75192	0.87144	0.75909	0.71304	0.88041
	p value	0	0	0	0	0.00013	0	0	7.00E-05	0.00089	0
	sample size	170	85	56	42	34	28	24	21	18	17
DNSJava	Rho	0.49466	0.50695	0.46773	0.52889	0.44062	0.42918	0.34091	0.46616	0.40557	0.5
	p value	0	0	0.00036	0.00045	0.0116	0.02549	0.11141	0.03829	0.09495	0.04858
	sample size	163	81	54	40	32	27	23	20	18	16

Rho = Spearman's Rank Correlation Coefficient
N/A = Correlation analysis could not be applied because of low sample size (i.e., sample size < 5)
p-value = Significance of Correlation, Sample size = The number of paired samples in the correlation analysis

Table 4: Regression between bug-proneness and method level evolutionary coupling

Subject system	R square	$S_{y,x}$	p-value
Ctags	0.7765	2.648	< 0.0001
jEdit	0.8756	2.196	< 0.0001
SqlBuddy	0.7886	0.6961	< 0.0001
Camellia	0.8275	3.054	< 0.0001
Carol	0.2366	3.18	< 0.0001
DNSJava	0.1088	1.545	< 0.0001

R square = Coefficient of determination
$S_{y,x}$ = Standard deviation of residuals
The highest possible value of R square is 1.

for each of these intervals is very high, the *p*-value regarding the fourth interval (40) is not significant because it is greater than 0.05. For the remaining three intervals, Camellia shows a very high as well as significant correlation between bug-proneness (PBC) and method coupling (PCM). Our subject system, Carol, shows a high correlation between PBC and PCM for each of the commit intervals. In the case of DNSJava, we can see a moderate correlation for each of the commit intervals where the correlations for the intervals, 70 and 90, are not statistically significant (*p*-value is greater than 0.05).

Finally, according to our investigated subject systems, bug-proneness of a software system generally has a good positive correlation with method level evolutionary coupling of that system. We answer RQ 1.1 in the following way.

Answer to RQ 1.1: Our correlation analysis on the subject systems implies that bug-proneness of a software system generally has a good positive correlation with its method level evolutionary coupling.

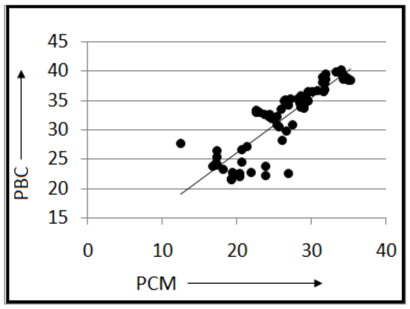
5.2 Regression Analysis

Although we know (from our answer to RQ 1.1) that bug-proneness and method level evolutionary coupling have a good positive correlation between them, we still do not know about the nature of their relationship. In order to better understand their relationship we analyze the linear regression [22] between PBC and PCM measures. For each subject system, we perform regression analysis considering the PBC and PCM values for the commit interval of 10.

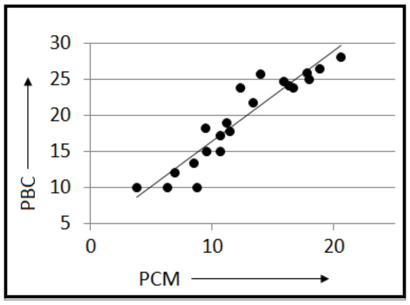
Table 4 shows three measures: **(i)** linear regression coefficient (R square), **(ii)** Standard deviation of residuals ($S_{y,x}$), and **(iii)** *p*-value for the regression regarding each of our subject systems. We see that the coefficient of determination (R square) is good for most of the subject systems except Carol and DNSJava. Such a scenario is also present in the graphs of Fig. 3a, 3b, 3c, 3d, 3e, and 3f. In each of these graphs, we have plotted PCM along the X-axis and PBC along the Y-axis. For the graphs regarding our subject systems: Ctags, jEdit, SqlBuddy, and Camellia, the points are mostly arranged around the trend line. The remaining two subject systems, Carol and DNSJava, do not show a good fit of the points around the trend line. Finally, according to the majority (four out of six systems) of our investigated subject systems, PBC and PCM have a good linear relationship between them. We finally answer our RQ 1.2 in the following way from our regression analysis.

Answer to RQ 1.2: According to the regression analysis on our subject systems, bug-proneness and method level evolutionary coupling of a software system generally have a good linear relationship between them.

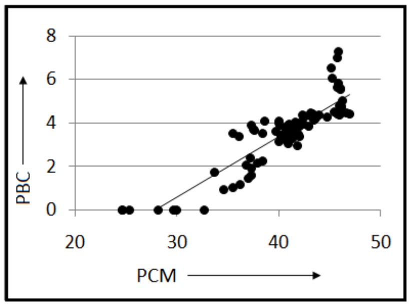
As method level evolutionary coupling and bug-proneness generally have a good positive correlation and they have a good linear



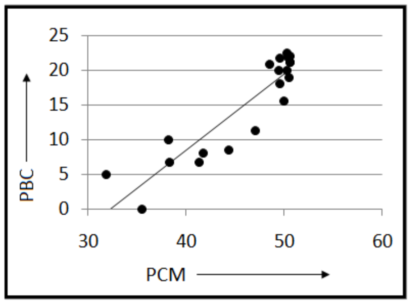
(a) Regression for Ctags



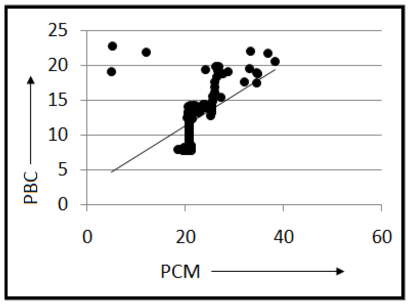
(b) Regression for jEdit



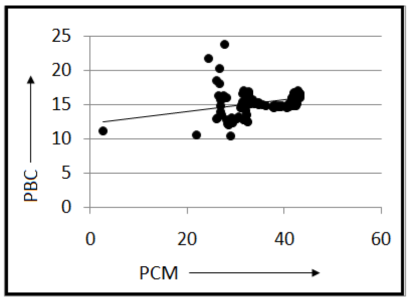
(c) Regression for SqlBuddy



(d) Regression for Camellia



(e) Regression for Carol



(f) Regression for DNSJava

Figure 3: Linear regression between bug-proneness (PBC) and method level evolutionary coupling (PCM)

relationship, it is expected that an increase or decrease in method level coupling will be associated with a corresponding increase or decrease in the bug-proneness of a software system.

6 ANALYZING THE COUPLING OF BUGGY AND NON-BUGGY METHODS

Our answer to the previous research question implies that method level evolutionary coupling has a good correlation with software bug-proneness. However, we still do not know whether the methods that experience bug-fixes have a higher extent of evolutionary coupling compared to the methods that do not experience bug-fixes. We investigate it in this section and answer our second research question (RQ 2) through our analysis.

RQ 2: *Do methods that experience bug-fixes have a higher number of evolutionary coupling links than the methods that do not experience bug-fixes?*

Investigation procedure. For answering this research question, we first identify which methods in a software system were changed during its entire period of evolution. We then separate these methods into two groups. While one group contains those methods that experienced bug-fix changes, the other group contains those methods that never experienced bug-fixes. For identifying the bug-fix method group, we first identify the bug-fix commit operations by analyzing the commit logs as we did in Section 3.1 and then identify which methods were changed in these bug-fix commits. Whether a particular method was changed in a bug-fix commit can be determined by examining its genealogy. We have described method genealogy in Section 2.2. After obtaining the bug-fix and non-bug-fix method groups, we determine the following measures for each of our subject systems.

(1) Total no. of methods that were created during the entire period of evolution
(2) Total no. of methods that were changed during evolution
(3) The no. of methods that experienced bug-fixes
(4) The no. of methods that never experienced bug-fixes
(5) Total no. of coupling links of all the methods that experienced bug-fixes
(6) Total no. of coupling links of all the methods that never experienced bug-fixes

Table 5 shows the above measures for our subject systems. We also determine the following two measures from the last four measures in the above list.

- **ACLB** (Average no. of evolutionary coupling links per method from the group of methods that experienced bug-fixes): We determine this measure by dividing the fifth measure in the above list by the third measure.
- **ACLN** (Average no. of evolutionary coupling links per method from the group of methods that never had bug-fixes): We calculate this measure by dividing the sixth measure by the fourth measure in the above list.

These two measures for each of the subject systems have been shown in Fig. 4. From the figure we realize that the average number of evolutionary coupling links per buggy method is always higher than the average number of links per non-buggy method. While in the case of our subject system Ctags, the difference between the two average numbers is very small, for each of the five remaining systems, we can see a large difference. For jEdit, Camellia, and

Table 5: No. of evolutionary coupling links of the methods that experienced or did not experience bug-fix commits

Measure	CTAGS	jEdit	SQLB.	Camellia	Carol	DNSJ.
Total number of methods that were created during the entire period of evolution	918	24913	2258	686	7937	4445
Total number of methods that were changed during entire evolution	437	689	880	178	1269	1443
Total number of methods that experienced bug-fix changes	169	311	102	34	333	251
Total number of methods that never experienced bug-fixes	268	378	778	144	936	1192
Total number of coupling links of all the methods that experienced bug-fixes	295	466	327	151	856	807
Total number of coupling links of all the methods that never experienced bug-fixes	465	140	1667	285	822	3379

Carol, the average number of coupling links per buggy method is more than twice as much as the average number of coupling links per non-buggy method.

Statistical Significance Test. We wanted to determine whether the average number of coupling links per buggy method (ACLB) is significantly higher than the average number of coupling links per non-buggy method (ACLN). We conduct Wilcoxon Signed Rank Test [16, 17] for this purpose. We use this test because our samples are paired. For every subject system, we get an ACLB value and a corresponding ACLN value. Moreover, Wilcoxon Signed Rank test is non-parametric, and thus, the samples do not need be normally distributed for applying this test [16]. This test can be applied to both small and large data samples [16]. We apply this test considering a significance level of 5%. According to our test result, ACLB (average number of coupling links per buggy method) is significantly different than ACLN (average number of coupling links per non-buggy method) for the 2-tailed test case with a p-value of 0.001 which is smaller than 0.05. The *Cohen's d* effect size [18] of the difference between ACLB and ACLN is 1.129 which is considered to be a large effect size [20, 23]. As ACLB is always greater than ACLN, the average number of coupling links per buggy method (ACLB) is significantly higher than the average number of coupling links per non-buggy method (ACLN) with a large effect size.

Answer to RQ 2: According to our investigation and analysis, the average number of evolutionary coupling links per buggy method is significantly higher (according to our statistical significance test) than the average number of coupling links per non-buggy method.

Such an observation is expected because a large number of coupling links from a method makes it difficult to be changed during evolution. Each change in such a method is likely to be associated with ripple changes in the other methods that are linked to it through evolutionary coupling. If changes are made without properly analyzing their impacts, bugs might get introduced to the code-base. Although our finding from RQ 2 is not a surprise, it is important in the context of the existing studies. The existing studies [34, 37] on file level or module level evolutionary coupling found that evolutionary coupling cannot be a good indicator of bug-proneness. However, our finding from RQ 2 associated with the findings from RQ 1 indicates that fine grained evolutionary coupling (method level coupling in our case) can be a good indicator of software bug-proneness.

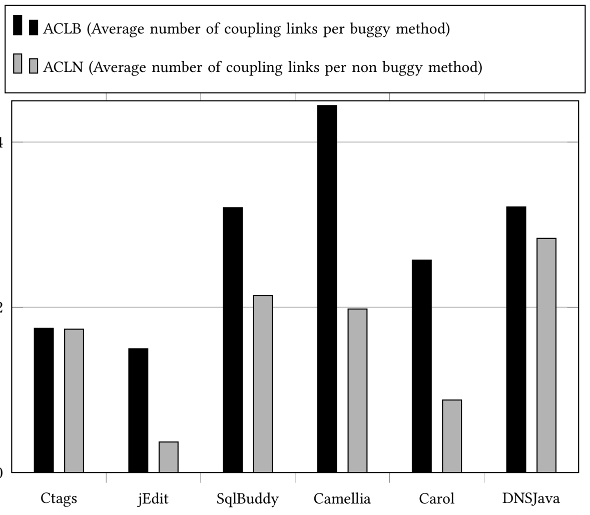


Figure 4: Comparing the number of coupling links per buggy and non-buggy methods

7 DISCUSSION

This section discusses the implications of our answers to the research questions. From our answer to RQ 1 we realize that our evolutionary coupling measure called PCM (percentage of coupled methods) shows a good positive correlation with software bug-proneness (PBC) for most of our subject systems. Also, the regression between our bug-proneness and method coupling metrics shows that these two metrics are linearly related in general. Such findings make us realize that higher bug-proneness of a software system is associated with higher change coupling among the methods (or functions) in that system. According to our answer to RQ 2, buggy methods have a higher number of evolutionary coupling links on an average than the non-buggy methods. Thus, identifying methods with higher number of evolutionary coupling links and taking measures towards minimizing these links can be beneficial for minimizing software bug-proneness.

The techniques that we have used in our study are useful for identifying methods with large number of evolutionary coupling links. When making decisions about restructuring a software system for better maintenance and evolution, our techniques can help programmers identify highly coupled methods as well as their coupling links so that they can find possibilities of minimizing these coupling links for better maintenance and evolution of software systems. Our implementation and data are available on-line [21].

8 RELATED WORK

Evolutionary coupling has been investigated a lot by the existing studies [1–5, 7–10, 13, 14, 29]. However, only a few studies [31, 34, 35, 37] investigated whether evolutionary among program entities is related with software bug-proneness. These studies were conducted considering coarse grained evolutionary coupling such as, coupling among modules, files, or classes. We discuss these studies in the following paragraphs.

Graves et al. [34] detected evolutionary coupling among different modules of one software system, and found that evolutionary coupling among modules cannot be a predictor of software defects. Graves et al. defined a module as a collection of source code files. Thus, they considered a coarse grained evolutionary coupling in their study. In our study, we investigate a fine grained evolutionary coupling (method level evolutionary coupling). Our investigation on six subject systems reveals that method level evolutionary coupling has a good positive correlation with software bug-proneness.

D'Ambros et al. [31] investigated the relationship between change coupling (i.e., evolutionary coupling) and software bug-proneness. They analyzed a number of change coupling metrics and found that software bug-proneness has a stronger correlation with change coupling metrics than with other coupling metrics. However, D'Ambros et al. investigated class level evolutionary coupling, and thus, their study is limited to subject systems that are developed in object oriented languages. Also, class level coupling cannot pin point which methods in the coupled classes are actually responsible for the coupling. In our study, we investigated method level evolutionary coupling, and thus, we can pin point which methods in the classes are really responsible for the coupling as well as for the bug-proneness. Moreover, as we study considering method level granularity (i.e., function granularity for procedural languages), our study is even applicable to subject systems that are developped in non-object-oriented languages. Table 1 demonstrates that two of our six subject systems are written in C which is a procedural language. We find that method level evolutionary coupling has a good correlation with software bug-proneness.

Knab et al. [37] investigated file level evolutionary coupling of Mozilla project and found that this coupling cannot predict fault-proneness of the project. We investigate method level evolutionary coupling (which is a finer granularity than the file level coupling) in six subject systems and find that this coupling has a good relationship with software bug-proneness.

Kirbas et al. [35] investigated file level evolutionary coupling in two industrial software systems. They analyzed the correlation between evolutionary coupling and bug-proneness for different modules. According to their findings, file level evolutionary coupling generally has a positive correlation with software bug-proneness. However, for modules with smaller number of source code files, evolutionary coupling is less likely to be related with bug-proneness. In another study [36], they investigated a banking software system considering file level evolutionary coupling and had similar findings. In our study, we investigate a fine grained evolutionary coupling (method level coupling) in six open-source subject systems and find that such a fine-grained coupling has a good positive correlation with software bug-proneness. Our regression analysis reveals that method level evolutionary coupling has a good linear relationship with bug-proneness of the software systems.

Mondal et al. [40] investigated whether method level evolutionary coupling is related with software change-proneness and found that change-proneness is positively correlated with evolutionary coupling. Our study is different, because we investigate whether evolutionary coupling among methods is related with software bug-proneness. We find a good positive correlation between these.

A number of existing studies have investigated evolutionary coupling for finding architectural weaknesses and modularity issues [26], structural shortcomings [5, 33], and important modules for refactoring [44]. Some studies [6, 14, 15] have investigated suggesting co-change candidates for program entities through analyzing evolutionary coupling. Some other studies [25, 30, 32] have also investigated detecting cross-cutting concerns using evolutionary coupling. Our study is different from all these existing studies, because we analyze whether method level evolutionary coupling is related with bug-proneness of software systems.

Sliwerski et al. [46] investigated finding fix-introducing changes by linking the version archive of a software system to its bug-database. They proposed an algorithm for this. While fix introducing changes can help us analyze how a bug was previously introduced, spotting the changes that caused the occurrence of a bug is not our goal. We calculate a bug-proneness metric which requires the information regarding how many bug-fixes occurred during evolution. We used the approach proposed by Mockus and Votta [39] for identifying the bug-fix commits from the commit log obtained from a software system's version control system. After calculating the bug-proneness metric, we investigate whether it is correlated with the method level evolutionary coupling metric.

We see that while the previous studies investigated module level, file level, or class level evolutionary coupling, we investigate evolutionary coupling considering a finer granularity (method granularity). Without such a fine-grained analysis, it is difficult to identify which methods in the modules, files, or classes are really responsible for the coupling. For the purpose of investigating method level evolutionary coupling, we extracted method genealogies from the entire evolutionary history of our subject systems. We find that method level evolutionary coupling has a good positive correlation with software bug-proneness. Our research reveals unknown facts that can be beneficial for better maintenance of software systems.

9 THREATS TO VALIDITY

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique followed by Barbour et al. [28]. Such a technique proposed by Mocus and Votta [39] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [28] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

In our experiment we did not study enough subject systems to be able to generalize our findings regarding the relation between method level evolutionary coupling and software bug-proneness. However, we selected our candidate systems emphasizing their diversity in sizes, application domains, and revision history lengths. Thus, we believe that our findings cannot be attributed to a chance.

10 CONCLUSION

In this study, we investigate whether method level evolutionary coupling of a software system is related with the bug-proneness of that system. While the existing studies have analyzed module level, file level, or class level evolutionary couplings and found mixed results regarding the relationship between bug-proneness and evolutionary coupling, our fine grained analysis (i.e., considering method granularity) reveals that method level coupling generally has a good positive correlation with software bug-proneness. Our regression analysis shows that software bug-proneness and method level evolutionary coupling are linearly related. According to our investigation on thousands of commit operations of six open-source subject systems, the buggy methods in a software system generally have a higher number of evolutionary coupling links than the non-buggy methods in that system. We realize that minimizing method level evolutionary coupling links can help us minimize software bug-proneness. Our research reveals unknown facts regarding the relationship between bug-proneness and evolutionary coupling and our findings are important for better maintenance and evolution of software systems. Our implemented prototype tool can help programmers automatically identify the highly coupled methods in a software system along with their coupling links. Programmers can then analyze these links in order to find possible ways of minimizing those. In the future, we would like to investigate if we can devise automatic mechanisms for minimizing method level evolutionary coupling links. The implementation and data from our research are available on-line [21].

ACKNOWLEDGEMENT

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by two Canada First Research Excellence Fund (CFREF) grants coordinated by the Global Institute for Food Security (GIFS) and the Global Institute for Water Security (GIWS).

REFERENCES

[1] A. Alali, B. Bartman, C. D. Newman, J. I. Maletic. A Preliminary Investigation of Using Age and Distance Measures in the Detection of Evolutionary Couplings. In *MSR*, pages 169 – 172, 2013.
[2] F. Bantelay, M. B. Zanjani, H. Kagdi. Comparing and Combining Evolutionary Couplings from Interactions and Commits. In *WCRE*, pages 311 – 320, 2013.
[3] F. Jaafar, Y. Gueheneuc, S. Hamel, and G. Antoniol. An Exploratory Study of Macro Co-changes. In *WCRE*, pages 325 – 334, 2011 .
[4] G. Canfora, M. Ceccarelli, L. Cerulo, and M. D. Penta. Using Multivariate Time Series and Association Rules to Detect Logical Change Coupling: an Empirical Study. In *ICSM*, pages 1 – 10, 2010.
[5] H. Gall, M. Jazayeri, and J. Krajewski. CVS Release History Data for Detecting Logical Couplings. In *IWPSE*, pages 13 – 23, 2003.
[6] H. Kagdi, M. Gethers, D. Poshyvanyk, M. L. Collard. Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code. In *WCRE*, pages 119 – 128, 2010.
[7] M. A. Islam, M. M. Islam, M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider. Detecting Evolutionary Coupling Using Transitive Association Rules. In *SCAM*, pages 113 – 122, 2018.
[8] M. D'Ambros and M. Lanza. Reverse Engineering with Logical Coupling. In *WCRE*, pages 189 –198, 2006.
[9] M. Mondal, C. K. Roy, and K. A. Schneider. Improving the detection accuracy of evolutionary coupling by measuring change correspondence. In *WCRE-CSMR*, pages 358 –362, 2014 .
[10] M. Mondal, C. K. Roy, and K. A. Schneider. Improving the Detection Accuracy of Evolutionary Coupling. In *ICPC*, pages 223 – 226, 2013.
[11] Online SVN repository: http://sourceforge.net/.
[12] R. Agrawal, T. Imielinski, A. Swami, "Mining Association Rules between Sets of Items in Large Databases", *ACM SIGMOD*, 1993, 22(2):207 – 216.
[13] R. Robbes, D. Pollet, and M. Lanza. Logical Coupling Based on Fine-Grained Change Information. In *WCRE*, pages 42 – 46, 2008.
[14] T. Rolfsnes, S. D. Alesio, R. Behjati, L. Moonen and D. W. Binkley. Generalizing the Analysis of Evolutionary Coupling for Software Change Impact Analysis. In *SANER*, pages 201 – 212, 2016.
[15] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572, 2004.
[16] Wilcoxon Signed Rank Test. https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test.
[17] Wilcoxon Signed Rank Test. http://www.socscistatistics.com/tests/signedranks/Default2.aspx.
[18] Cohen's d effect size calculator: https://www.ai-therapy.com/psychology-statistics/effect-size-calculator.
[19] CTAGS: http://ctags.sourceforge.net/.
[20] Effect size: https://en.wikipedia.org/wiki/Effect_size.
[21] Implementation and data regarding our research: https://drive.google.com/open?id=1WaWUH1nFq67HH0vIdkwTZCBwy0yO_0Sw.
[22] Linear regression calculator: https://www.graphpad.com/quickcalcs/linear1/.
[23] Power analysis, statistical significance, & effect size: http://meera.snre.umich.edu/power-analysis-statistical-significance-effect-size.
[24] Spearman's rank correlation: https://www.wessa.net/rwasp_spearman.wasp.
[25] B. Adams, Z. M. Jiang, and A. E. Hassan. Identifying crosscutting concerns using historical code changes. In *ICSE*, pages 305 – 314, 2010.
[26] T. Ball, J. M. Kim, and H. P. Siy A. A. Porter. If your version control system could talk. In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.
[27] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *ICSM*, pages 273 – 282, 2011.
[28] L. Barbour, F. Khomh, and Y. Zou. An empirical study of faults in late propagation clone genealogies. *Journal of Software: Evolution and Process*, 25(11):1139 - 1165, 2013.
[29] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and De Lucia. An empirical study on the developers' perception of software coupling. In *ICSE*, pages 692 – 701, 2013.
[30] S. Breu and T. Zimmermann. Mining aspects from version history. In *ASE*, pages 221 – 230, 2006.
[31] M. D'Ambros, M. Lanza, and R. Robbes. On the relationship between change coupling and software defects. In *WCRE*, pages 135 – 144, 2009.
[32] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497 - 515, 2008.
[33] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM*, pages 190 – 199, 1998.
[34] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7): 653 - 661, 2000.
[35] S. Kirbas, B. Caglayan, T. Hall, S. Counsell, D. Bowes, A. Sen, and A. Bener. The relationship between evolutionary coupling and defects in large industrial software. *Journal of Software: Evolution and Process*, 29(4): 1 - 19, 2017.
[36] S. Kirbas, A. Sen, B. Caglayan, A. Bener, and R. Mahmutogullari. The effect of evolutionary coupling on software defects: an industrial case study on a legacy system. In *ESEM*, pages 6:1 – 6:7, 2014.
[37] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *MSR*, pages 119 – 125, 2006.
[38] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *ICSM*, pages 227 – 236, 2008.
[39] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM*, pages 120 – 130, 2000.
[40] M. Mondal, C. K. Roy, and K. A. Schneider. Connectivity of co-changed method groups: A case study on open source systems. In *CASCON*, pages 205 – 219, 2012.
[41] M. Mondal, C. K. Roy, and K. A. Schneider. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In *ICPC*, pages 103 – 112, 2013.
[42] A. J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *Journal of Systems and Software*, 20(3):295-308, 1993.
[43] M. Page-Jones. The practical guide to structured systems design. YOURDON Press, New York, NY, 1980.
[44] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *ACM symposium on Software visualization*, pages 67 – 75, 2005.
[45] C. K. Roy, M. F. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *CSMR-WCRE*, pages 18 – 33, 2014.
[46] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1-5, 2005.