Finding Erroneous Components from Change Coupled Relations at Fix-inducing Changes

Ali Zafar Sadiq, Ahmedul Kabir and Kazi Sakib
Institute of Information Technology, University of Dhaka
Email: zafarsadiq120@gmail.com, kabir@iit.du.ac.bd, sakib@iit.du.ac.bd

***Abstract*—During the gradual process of software evolution, errors appear in different components of a software system. These errors are later on fixed by developers as part of corrective maintenance activities. However, if errors appear continuously from a particular component, that may indicate design flaws or code smells. Maintenance cost will greatly reduce if design flaws are treated as early as possible. To find out such flaws it may require time-consuming manual inspections. This paper tries to find out such components using the information of change coupled cluster of files or Java classes at fix-inducing changes. In this proposed approach, information (like class, method, parameter of method and variable names) from change coupled relation of a class at Fix-Inducing Changes (FICs) are used to provide information about erroneous components. Then the error history, of software components, is found by using cosine similarity of information from change coupled cluster of classes found in FICs to see with the architectural information found from authenticated sources. Finally, the error history of components is shown as the percentage of change coupled cluster of a class found in FICs of each 100 commits in the version control system.**

***Index Terms*—Fix-Inducing Change, Software Quality Assurance, Software Change, Software Maintenance, Change Coupling**

I. Introduction

Change is an inevitable part of the evolution of software. Frequently co-changing software artifacts form change coupled relation. Any change in an artifact will influence change in other artifacts which are change coupled with the former. This relation can also be considered to form a cluster of artifacts with respect to a file or class, which may be affected depending on the change of that class. So any class and its co-changing artifacts can be considered to be a part of a module or component which shows close interactions among themselves.

Changes are done to introduce new features into the system or to fix existing errors and any changes can introduce errors, flaws or failures in the system. Various works identified these erroneous changes [1] and analyze their impact [2]. The reasons behind these changes may be improper coding or careless implementation of algorithms. For analyzing these changes, various properties of change like files affected, time, experience of developer and many others taken into consideration [3] [4] [5] [6]. However, none of those explored the architectural components of a software system is affected by those erroneous changes during the process of software development and maintenance.

Continuous appearance of errors from a particular component indicates that either that part has design flaws or it needs redesigning. To find out such components, manual inspection of files from source repository and bugs from bug repository is required. Moreover, the bug repositories will only provide information about reported bugs whereas many unreported bugs fixed by developers will remain hidden. So considering bug repository may give less information than the actual situation.

Various works tried analyze the quality of software systems and condition of architectural components [7] [8] [9]. Evolution radar used change coupling relation to show the condition of software components [7]. This work did not consider the erroneous changes and only focuses on design flaws based on change couple relation. Using the comments from the version control system, sticky notes are seen to provide useful information [8] but the concept of error is not seen there. Furthermore, the relationship between the evolution of software artifacts and the way they are affected by problems is visualized by D'Ambros et al but it did not consider component based analysis using commit history [9]. To the best of author's knowledge, none of the existing works explored by combining the information of change coupling relation and Fix-Inducing Changes of source repositories to find software components error history.

To find the erroneous components, firstly the fix-inducing changes are found by tracking the modified and deleted of error fixing changes. Then the entire history is traversed using a commit window of 100 commits. In every 100 commits, the classes (only Java classes are considered in this work) found in the FICs are noted. Then for each of those classes, that class itself and the cluster of other classes forming change coupled relation with that class are considered. Then for each of these cluster classes, information about class, method, method's parameter, and variable names are collected. Using the cosine similarity of the information obtained from each cluster with the architectural information from the authenticated source, the most probable component for each cluster is found. Then of the total clusters, what percentage of clusters belongs to which component is graphically represented. So, the main contribution of this paper is to propose a methodology to generate the error history of software components for the entire lifetime of any source repository.

II. Methodology

This work tried to utilize the change coupling information of classes found in fix-inducing change. Unlike [7] [8] [9],

DOI reference number: 10.18293/SEKE2019-087

this work focuses on components error history. Analyzing this history might play an important role in the fields of software architecture, evolution, decay, and similar others.

The entire process of the proposed methodology consists of three steps. Firstly, fix-inducing changes are extracted by using historical information. Then clusters of change coupled classes are identified by observing their changing relation. Lastly, the error history is analyzed by using cosine similarity between obtained change coupled cluster of classes and architectural information.

A. Finding Fix-Inducing Changes (FIC):

Bugs are errors of the system that causes the system to behave in unintended ways. The origin of the bugs are FICs which introduce errors in the software system. These are also known as bug introducing change, Figure 1 explains the entire process of finding fix-inducing changes. This process starts with finding fixing commits which are mainly committed in the version control system by developers with a comment containing keywords like “Fix”, “Bug”, “Patch” or their past and gerund form. Any number with hashtags indicating bug number along with those keywords were also considered to represent fixing commits. These commits contain the change to correct errors. This change is done by modifying or deleting some lines of code that is present in the immediate commit parent to a fixing commit. To obtain those lines that were modified or deleted, Diffj tool’s [10] source code is used after modifying it according to our need. Since Diffj ignores white space and other format changes, so it ignores the possibility of finding false FICs as mentioned by [11].

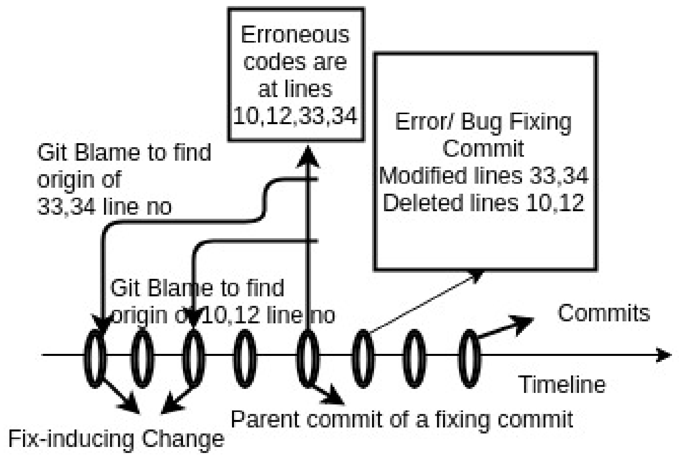

Erroneous codes are at lines 10,12,33,34
Error/ Bug Fixing Commit Modified lines 33,34 Deleted lines 10,12
Git Blame to find origin of 33,34 line no
Git Blame to find origin of 10,12 line no
Commits
Timeline
Fix-inducing Change
Parent commit of a fixing commit

Figure 1: Finding Fix-Inducing Changes.

To track the origin of those lines in the parent commit of fixing change, Git blame command [1] is used [12]. The commits which were found are the FICs or changes introducing errors in the system. All FICs of the repository are found in this way.

[1]git -c core.abbrev=40 blame -L(line number),+1 (FCParentHash) ˆ – (filename)

B. Finding Cluster of Change Coupled (CC) classes:

CC classes are found by constructing a co-change matrix [13]. In this symmetric matrix, any cell, [A, B] and A$\neq$B, represents of total changes of artifacts A or B, how many those changed together. Besides, in the cell [A, A] keeps track of how many times artifact A changed. Using appropriate support and confidence, the change coupled relation can be found among Java classes. Support represents how many times a class changed and confidence represents the likelihood, which means if there are 2 coupled artifacts, if one artifact changes, the probability that another artifact is going to change or not. In co-change matrix [A, A] represents support and confidence are represented by following Equation 1.

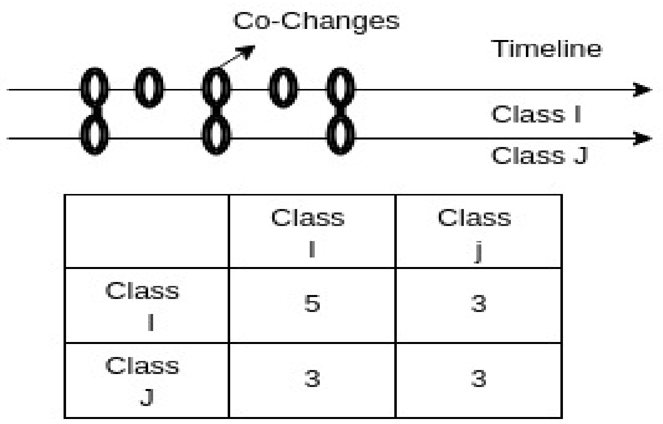
In Figure 2, there are 2 classes I and J. Among 5 changes of A and 3 changes of B, both of the co-changed 3 times. So the probability that if class A changes then class B will change or that confidence is obtained from equation 1 as $\frac{3}{5}$ or 60%. But if B changes then the confidence that A will change is $\frac{3}{3}$ or 100%.

CC classes were found by using different support and confidence in different works. Zimmermann et al used the support greater than 1 and confidence level 0.5 in their work [14]. However, Bavota et al considered elements that co-changing in at least 2% of the commits along with a confidence level of 0.8 [15]. Since, 0.8 confidence is high, in this paper 0.7 confidence level is used along with support 2 or more is considered. So in the considered 70% confidence and according to Figure 2, class J will have change coupled relation with I but not the opposite. This relationship is considered for classes found in FICs and the time period is taken from the 1st commit to the FIC. The source of finding this relationship, the co-change matrix is constructed by considering a n x n array, where n is the number of files or classes and co-changes of Java classes within the considered period is stored in that array. Then for each Java class, the change coupled relation with others are found based on considered support and confidence. Those other Java classes that have change coupled relation with a Java class is considered to form a cluster.

$$\begin{aligned} Confidence(A \rightarrow B) &= \frac{support(A \rightarrow B)}{support(A)} \\ &= \frac{support(A \cup B)}{support(A)} \end{aligned} \qquad (1)$$

C. Commit History Analysis:

In this work, fixing changes are found by searching comments which later on leads to fix-inducing changes. Then using a commit window of 100 commits the entire commit history of source repository is traversed. However, the initial 20 commits were omitted for repository setup issues. After that commit window is used to traverse commits, i.e i.e 20-120, 120-220 and so on. For 100 commits in each window, FICs are analyzed for their contents. Figure 3 shows the entire process. After getting all FICs, these are sorted. Then traversing with commit



	Class I	Class j
Class I	5	3
Class J	3	3

Figure 2: Example of Co-Change Matrix and Commit Timeline

frame of 100 commits, all FICs that falls within the frame can be found out.

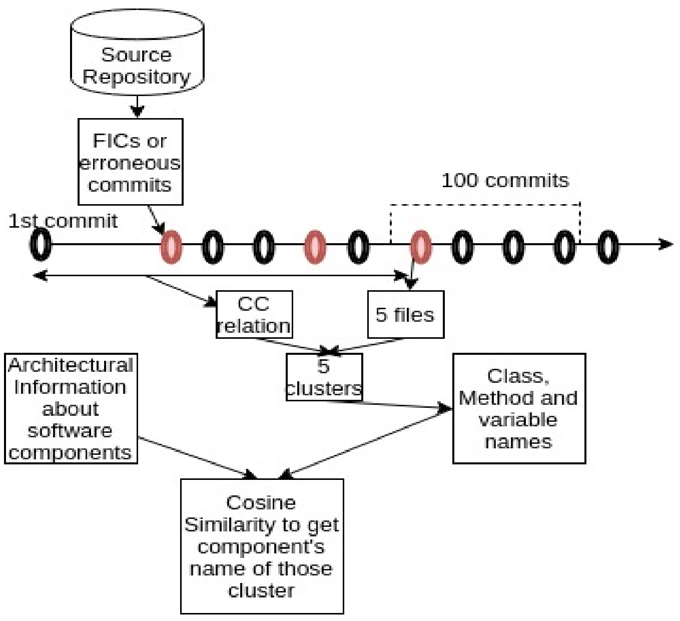

Source Repository
FICs or erroneous commits
100 commits
1st commit
CC relation
5 files
5 clusters
Architectural Information about software components
Class, Method and variable names
Cosine Similarity to get component's name of those cluster

Figure 3: Commit History Analysis process to show the erroneous components.

In the contents of those FICs, firstly files with .java extension were searched. Then those files or classes were taken. For each of those class, cluster of classes found from the change coupled relation (with at least 2 support and 70% confidence) formed from 1st commit to that FIC are taken. Then that class in FIC along with its classes in change couple cluster are explored for their class, variable and method names. Then architectural information are taken from authenticated source and those information were first cleared of their stop words and then by applying the porter stemming algorithm their root words are taken. After that cosine similarity between those architectural information of different components and change coupled cluster information is taken to find out the most probable component for the cluster. After getting names for each of these clusters, what percentage of total cluster belongs to which cluster can be easily known. Frequent appearance of the same component in commit history found by traversing with a commit window of 100 commits means that component is vulnerable to errors and responsible for costly corrective maintenance of the software system.

III. Experimentation

This experiment is carried out in a virtual machine where operating system is Ubuntu 18.04 with 64 GB memory and 16 core CPU. For this experiment, among the popular Java repositories, 2 java repositories are selected for the study. These are as follows:

Repository Name	Source	Total Commits	Commits Analyzed	Number of Java classes	Lines of Code
Google Guava [16]	Github	4798	4020	3170	768858
jEdit [17]	Github	8000	8000	600	196194

TABLE I: General description of repositories.

Of the used 2 repositories, Google Guava is a source repository of library classes maintained by Google developers. This provides more functionalities than existing java collection framework and contains other features like hashing, graph, range etc. The commits of this repository started from June 2009 to the last commit updated in August 2018. In the case of jEdit, it is a text editor written in Java. In its source repository, commits started in 2001 and the last one is a patch commit in August 2019. By looking at commit history it can be said that the gradual development and maintenance is very slow in recent times.

Using the above mentioned repositories, the experiment is methodologically conducted. Firstly, Fixing changes were found by analyzing commit comments. After that, diffj [10] is used to find the lines that were modified or deleted from FCs parent commit. Since diffj ignores cosmetic and format changes, possibilities of finding false FICs are thus reduced. Then those identified lines are tracked by using Git Blame command to find the FIC commits where the last modifications are made. The total number of FCs and FICs that are found in both repositories are showed in Table II.

Repository Name	Fixing Commits	Fix-Inducing Commits
Google Guava	597	486
jEdit	2752	2270

TABLE II: Total Fix-Inducing Commits and Fixing Commits of each repository

The FICs are then sorted to find according to commit timeline. Then using a commit window of 100 commits, the entire history is traversed. For each FICs within the commit window, the .java files or classes are being collected. For each class in FIC, a cluster of CC formed through the gradual development of the Change Couple relation throughout the lifespan of a project was analyzed. From the CC cluster of class in FIC and that class, information about classes, methods, and variables were collected. These are used to find cosine similarity with the description of components available from authenticated sources. Then of total clusters what percent belongs to a particular component for a particular 100 commit is found out.

Architectural information for google guava is collected from authenticated sources like in GitHub or their main website. In the case of jEdit, their main website is used to collect information about components. The obtained result mainly depends on this architectural information as cosine similarity is performed on this information.

IV. Result Analysis

The results are obtained by conducting an experiment on the first 4000 commits of Google guava and 8000 commits of commits of jEdit. From the experiment, the name of components of google guava repository is obtained from [16] and that of jEdit is obtained from [17]. Table III and IV contains the name of the components considered.

Components	Description
Basic utilities	Deals with nulls, preconditions, common object methods, ordering, and throwables
Collections	It is an extension of JDK's collection system. It deals with immutable collections, new collection types, powerful collection utilities, and throwables.
Graphs	It mainly represents a graph, network and has structured data.
Caches	Local caching and support a wide variety of behaviors
Functional idioms	It is used to simplify the code greatly.
Concurrency	It contains powerful abstractions to write correct code.
Strings	It has useful string operations like joining, splitting and padding.
Primitives	It contains operations on primitives like int or char which is not provided by JDK
Ranges	It is an API to deal with both continuous and discrete ranges on comparable types.
I/O	It contains simplified I/O operations which specifically deals with I/O streams or files.
Hashing	It deals with hashing problems.
EventBus	It deals with publish-subscribe-style communication between components.
Math	It contains more optimized and tested math utilities, not provided by the JDK.
Reflection	It contains guava utilities for Java's reflective capabilities.

TABLE III: Information about components of Google Guava repository from github [16].

Components	Description
General	General features of the jEdit text editor providing writing and correcting facilities.
Source Code Editing	Dealing with source codes of different programming language.
Search and Replace	Different functionalities dealing with search.
File Management	It consists of everything related to files, like opening, editing, renaming and other such actions.
Customization	It consists of configurations to deal with users preference like customizable keyboard shortcuts etc.
Extensibility	Various plugins can extend the current abilities of jEdit to provide more functionalities.

TABLE IV: Information about components of jEdit repository from github [17].

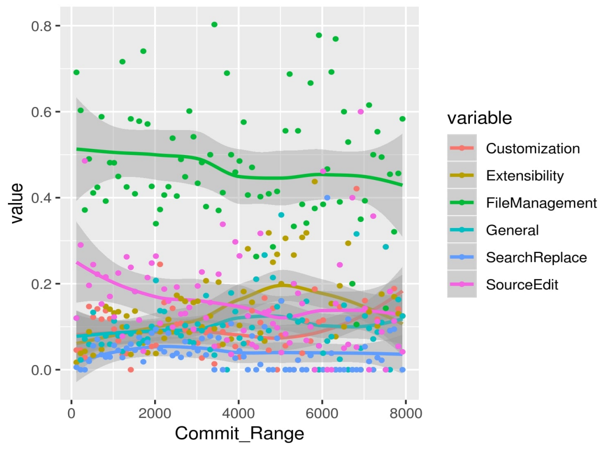
The name of the components are the features or modules found from the architectural information of corresponding sources. In Figure 4a the x-axis represents the commit number and y-axis represents the percentage of clusters obtained from FICs within the commit window of 100 commits. In Figure 4a, it is clearly visible that the file management part contributed a dominant portion to introduce errors into the software system. After it is found that file management, SourceEdit, and extensibility related classes are responsible for introducing errors into the system. jEdit being a text editor, surely works on file management will be more and it is expected to produce more errors. Next comes doing programming using jEdit, classes which manage it was producing more errors in the earlier phase of development which later on is replaced by extensibility. This might be because works on extensibility feature increased in later phase.

From Figure 4b, it is seen that seen that different components of the software system are affected at different times. However dominant top three are related to Math, Ranges, and Primitives. This may be associated with the nature of the project which is a library project. So mainly fixing occurs in classes when there are problems with continuous or discrete ranges, primitives with float or int values and calculations related to math.

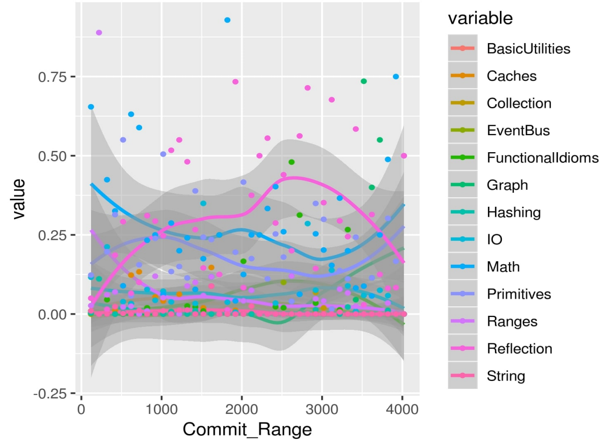
Through Figure 4a and 4b, any developer can understand classes of which components are producing more errors and thus can try to identify design flaws or code smells so that these can be addressed to improve the quality of software.

V. Threats to Validity

The main factors for which construct validity might be threatened are described here. Firstly, only commit comments are searched to identify fixing changes without linking those to bug repositories. To create a link between fixing changes with bug report of the bug repository becomes a problem when the bug ids are not available. So for those cases linking with fixing changes might lead to unfair situation [18]. Besides, the main goal of this study is to find erroneous components. Secondly,



(a) Error History of components in jEdit Repository



(b) Error History of components in Google Guava Repository

Figure 4: Error History of components

there can be varying behavior in the contents of commits. Unrelated classes in bug fix can lead to wrong fix-inducing changes. But it is found that related works are committed together and 15% of all bug fixes to consist of multiple tangled changes [19]. Thirdly, all fixes may not be actually corrective maintenance [20]. However, using 10 random searches in FCs, only bug fixes are found. The main purpose of those FCs was corrective maintenance,

This work only considers java classes as Diffj, which is used for differencing modified and deleted source code lines between files of two commit version, can work only in java repositories. In the future extension, different types of project in different languages will be analyzed. Again, rather than error history, fixing history can also be obtained by considering the fixing changes. All of this information will be used to conduct further research in the fields of software architecture, decay and quality assurance.

VI. Related Work

Evolution of software cannot be explained solely by structural dependency [21]. It is found that rather than structural dependency, change coupling plays a more effective role in fault proneness and are more relevant [22]. Historical information about CC classes can be used to predict further changes based on CC relation [23]. Similarly, based on CC relation, Zhou et al used Bayesian Network to predict changes [24]. Furthermore, Fluri et al used tree edit operations in AST to classify changes depending on how the change is made [25]. Rather than considering static measures, Arisholm et al proposed dynamic coupling measures by taking into account inheritance, polymorphism, and dynamic binding [26]. Moreover, whether or not frequent code changes represent code smell or design issues was investigated by Ratzinger et al and it is found that those changing software parts may be candidates for refactoring [27].

Frequent changes indicate unstable situations as changes do not satisfy the requirements and correctness expected from the software system. Due to these frequent changes errors may be introduced in the system which is shown by DAmbros et al [28] as change coupling measures have a strong relationship with software defects. These erroneous changes which introduced a bug or error in the system are called Fix-Inducing Change by Sliwerski et al [29]. Moreover, information of FICs and Fixing changes can be used for bug prediction [30] [31], localization [32] as well as to find out affected parts [33].

Very few works considered combining information from both change coupling and Bug/Errors. It is found that the change coupling relationship in recent commits is more correlated with recent FICs compared to commits from origin [34]. Furthermore, works of D'Ambros [7] [9] focused on analyzing software evolution and quality, and did not use the combined information to understand the error or maintenance history of software components.

VII. Conclusion

The main achievement of this work is to propose a methodology to analyze the erroneous components by using the change coupled relation at fix-inducing changes. Having knowledge of this information will help the software developers to find out which part of the system is continuously responsible for change and producing bugs. To do this, if separate information about architectural components is available then cosine similarity can be used. Otherwise, Latent Dirichlet Allocation can be used to find topics or probable components. However, in that case, manual labeling is required. From the obtained information, error-prone components can be easily identified. Then quality can be further enhanced by refactoring and re-engineering these error-prone components.

Acknowledgment

This research is supported by the fellowship from ICT Division, Ministry of Posts, Telecommunications and Information Technology, Bangladesh. No- 56.00.0000.028.33.002.19.3; Dated 09.01.2019. The virtual machine facilities used in this research is provided by Bangladesh Research and Education Network (BdREN).

References

[1] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[2] G. Antoniol, V. F. Rollo, and G. Venturi, "Detecting groups of co-changing files in CVS repositories," in *8th International Workshop on Principles of Software Evolution (IWPSE 2005), 5-7 September 2005, Lisbon, Portugal*, 2005, pp. 23–32. [Online]. Available: https://doi.org/10.1109/IWPSE.2005.11

[3] S. Levin and A. Yehudai, "Boosting automatic commit classification into maintenance activities by utilizing source code changes," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2017, Toronto, Canada, November 8, 2017*, 2017, pp. 97–106. [Online]. Available: http://doi.acm.org/10.1145/3127005.3127016

[4] S. Kim, E. J. W. Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Software Eng.*, vol. 34, no. 2, pp. 181–196, 2008. [Online]. Available: https://doi.org/10.1109/TSE.2007.70773

[5] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Software Eng.*, vol. 33, no. 1, pp. 2–13, 2007. [Online]. Available: https://doi.org/10.1109/TSE.2007.256941

[6] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, 2014, pp. 172–181. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597075

[7] M. D'Ambros, M. Lanza, and M. Lungu, "Visualizing co-change information with the evolution radar," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 720–735, 2009.

[8] A. E. Hassan and R. C. Holt, "Using development history sticky notes to understand software architecture," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 2004, pp. 183–192.

[9] M. D'Ambros and M. Lanza, "Software bugs and evolution: A visual approach to uncover their relationship," in *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE, 2006, pp. 10–pp.

[10] Diffj. [Online]. Available: https://github.com/jpace/diffj

[11] S. Kim, T. Zimmermann, K. Pan, and E. J. W. Jr., "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, 2006, pp. 81–90. [Online]. Available: https://doi.org/10.1109/ASE.2006.23

[12] Z. Fabk, "Learn more about the history of a line with git blame," https://zsoltfabok.com/blog/2012/02/git-blame-line-history/, February 2012. [Online]. Available: https://zsoltfabok.com/blog/2012/02/git-blame-line-history/

[13] T. Menzies, L. L. Minku, and F. Peters, "The art and science of analyzing software data; quantitative methods," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, 2015, pp. 959–960. [Online]. Available: https://doi.org/10.1109/ICSE.2015.306

[14] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of.* IEEE, 2003, pp. 73–83.

[15] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 692–701.

[16] Guava architectural information. [Online]. Available: https://github.com/google/guava/wiki

[17] jedit features. [Online]. Available: http://www.jedit.org/index.php?page=features

[18] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 121–130.

[19] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 121–130.

[20] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. ACM, 2008, p. 23.

[21] M. M. Geipel and F. Schweitzer, "The link between dependency and cochange: Empirical evidence," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1432–1444, 2012.

[22] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb, "Software dependencies, work dependencies, and their impact on failures," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 864–878, 2009.

[23] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Trans. Software Eng.*, vol. 31, no. 6, pp. 429–445, 2005. [Online]. Available: https://doi.org/10.1109/TSE.2005.72

[24] Y. Zhou, M. Würsch, E. Giger, H. C. Gall, and J. Lu, "A bayesian network based approach for change coupling prediction," in *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008*, 2008, pp. 27–36. [Online]. Available: https://doi.org/10.1109/WCRE.2008.39

[25] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *14th International Conference on Program Comprehension (ICPC 2006), pages = 35–45, year = 2006, crossref = DBLP:conf/iwpc/2006, url = https://doi.org/10.1109/ICPC.2006.16, doi = 10.1109/ICPC.2006.16, timestamp = Mon, 22 May 2017 17:11:18 +0200, biburl = https://dblp.org/rec/bib/conf/iwpc/FluriG06, bibsource = dblp computer science bibliography, https://dblp.org*.

[26] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Transactions on software engineering*, vol. 30, no. 8, pp. 491–506, 2004.

[27] J. Ratzinger, M. Fischer, and H. C. Gall, "Improving evolvability through refactoring," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083155

[28] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, 2009, pp. 135–144. [Online]. Available: https://doi.org/10.1109/WCRE.2009.19

[29] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[30] D. D. Nucci, F. Palomba, G. D. Rosa, G. Bavota, R. Oliveto, and A. D. Lucia, "A developer centered bug prediction model," *IEEE Trans. Software Eng.*, vol. 44, no. 1, pp. 5–24, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2659747

[31] S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *IEEE Trans. Software Eng.*, vol. 39, no. 4, pp. 552–569, 2013. [Online]. Available: https://doi.org/10.1109/TSE.2012.43

[32] M. Wen, R. Wu, and S. Cheung, "Locus: locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 262–273. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970359

[33] A. T. Misirli, E. Shihab, and Y. Kamei, "Studying high impact fix-inducing changes," *Empirical Software Engineering*, vol. 21, no. 2, pp. 605–641, 2016.

[34] A. Z. Sadiq, M. J. I. Mostafa, and K. Sakib, "On the evolutionary relationship between change coupling and fix-inducing changes," in *15th International Conference on Evalution of Novel Approaches to Software Engineering, ENSAE 2019*, in press.