Enhancing Source Code Refactoring Detection with Explanations from Commit Messages

Rrezarta Krasniqi
Dept. of Computer Science and Eng.
University of Notre Dame
Notre Dame, IN, USA
rkrasniq@nd.edu

Jane Cleland-Huang
Dept. of Computer Science and Eng.
University of Notre Dame
Notre Dame, IN, USA
JaneHuang@nd.edu

***Abstract*—We investigate the extent to which code commit summaries provide rationales and descriptions of code refactorings. We present a refactoring description detection tool *CMMiner* that detects code commit messages containing refactoring information and differentiates between twelve different refactoring types. We further explore whether refactoring information mined from commit messages using CMMiner, can be combined with refactoring descriptions mined from source code using the well-known *RMiner* tool. For six refactoring types covered by both CMMiner and RMiner, we observed 21.96% to 38.59% overlap in refactorings detected across four diverse open-source systems. RMiner identified approximately 49.13% to 60.29% of refactorings missed by CMMiner, primarily because developers often failed to describe code refactorings that occurred alongside other code changes. However, CMMiner identified 10.30% to 19.51% of refactorings missed by RMiner, primarily when refactorings occurred across multiple commits. Our results suggest that integrating both approaches can enhance the completeness of refactoring detection and provide refactoring rationales.**

***Index Terms*—Refactoring, Commit Message, Classification**

I. Introduction

Commit messages convey important information about changes in a software system [1, 11]. Refactoring changes are often intermingled with functional ones [12]; however, there is a tendency for developers to focus on describing the functional aspects of the change in their commit messages [11]. As we show in this paper, developers tend to describe refactorings in just under half of the commits in which they occur and approximately 33.63% of these commits are cases in which refactoring occurred alongside functional changes. Existing refactoring tools can detect many types of low-level refactorings; however, they tend to produce structural explanations about the refactoring, and fail to explain the intent and potential impact of the change. For example, the commit message 'Derby-6180', depicted at the top of Figure 1, describes a 'move method' refactoring and includes the context for the refactoring. In contrast, the RMiner [21] tool detects structural changes in the code for the same commit; however, as denoted in the bottom half of the same figure, it does not explain the intent or context of the change including *why* the method *assertDatabaseMetaDataColumns* was moved from one class to another. Enhancing current refactoring tools with descriptions of refactoring code changes could improve maintainers' comprehension of the context of the change and its broader impact. In this paper, we explore the extent to which different types of refactoring changes are explained in commit messages and investigate whether detected explanations align with refactorings detected by RMiner.

Refactoring Commit Message Detected by CMMiner

Derby-6180. DatabaseMetaDataTest should not fail if there are extra columns but only check the expected ones Moving the assertDatabaseMetaDataColumns method to junit.JDBC and making jdbc4.TestDbMetaData use it.
Refactoring Type: [Move Method]

Code Refactoring Change Detected by RMiner

Derby-6180. Move method assertDatabaseMetaDataColumns(..) from the class dbcapi.DatabaseMetaDataTest to public assertDatabaseMetaDataColumn from class Testing.junit.JDBC
Refactoring Type: [Move Method]

Fig. 1. Refactoring detected in a Commit Message and Code

We have developed a commit message refactoring detection tool, that we refer to as the *CMMiner*. CMMiner extracts contextual refactoring descriptions from commit messages, distinguishes commit messages containing refactoring information (CMR) from those not containing refactoring information (non-CMR), and differentiates between diverse refactoring types. We compare results from CMMiner with those of *RMiner*, a state-of-the-art structural-based refactoring detector [21] which detects numerous code refactoring types from source code changes. We therefore explore the overlap of detected refactorings for six types detected by both tools.

Our experimental results indicate that for six refactoring types across four diverse open-source systems, RMiner identified approximately 49.13% to 58.53% of refactoring types missed by CMMiner, while CMMiner identified approximately 10.30% to 19.51% of refactoring types missed by RMiner. We discuss reasons for these differences later in the paper. We also found 21.96% to 38.59% overlapping commits identified by both miners. We then explore the interplay of refactoring information found in source code and commit messages in order to address the following research questions:

RQ_1: To what extent do code commit messages describe code refactorings?

RQ_2: To what extent can structurally detected code refactoring descriptions be enriched with contextual information from commit messages?

II. Methodology

A. Refactoring Commit Message Detector (CMMiner)

Developing CMMiner involved five distinct steps, as depicted in Figure 2: ❶ collecting commit data from four open-source systems, ❷ manually annotating commit messages into two major classes (i.e., CMR and non-CMR) and labeling each CMR commit message according to its specific refactoring type (e.g., 'move method'), ❸ using the Stanford parser to automatically tag commit messages with their parts-of-speech (POS), ❹ extracting dependancy triplets in a structured form as: 'subject (noun)–relation (verb)–objects (noun)', and finally ❺ constructing several models to train different classifiers.

B. Data Collection and Data Cleansing

❶ Our corpus is comprised of $10,463$ commit messages collected from four diverse domains. The four projects are: Derby and Infinispan (databases), Drools (rule engine), and Groovy (languages). Table I depicts the characteristics of the selected projects. The size of the projects are relatively large with thousands of lines of code (kLOC). The four projects were selected because they utilized both Git and Jira and each of these projects had a non-trivial number of commits and issues [17, 18]. For evaluation purposes, we randomly selected roughly $1/6$ of the commit messages from each project. For the data cleansing step, we applied standard preprocessing techniques for each commit message (i.e., tokenization, normalization, stop-word removal, and stemming).

C. Human Annotation

❷ To identify commit messages containing refactoring information (CMR), three software analysts (two senior software developers—each with more than 7 years of experience in development related tasks and the first author) annotated commit messages. They inspected descriptions in the commit messages to identify the refactoring type, and to label the commits accordingly. However, during this process they also explored the corresponding code changesets only in situations when the commit messages either lacked clarity or provided insufficient information to determine the corresponding label. The process involved: (1) Randomly selecting 1,529 commits from the original set. (2) Using a template of refactoring rules [16] as guidelines to independently label a random set of 100 commit messages. Each message was labeled as either 'CMR' or 'non-CMR'. (3) For those commit messages identified as CMRs, reviewers further labeled them according to their refactoring type (i.e., 'rename method', or 'extract method'). (4) Finally,

TABLE I
Characteristics of the Project Datasets

Project	Interval	kLOC	#Commits	#Commits (sample)
Derby	2004-09–2016-12	170	2382	329
Drools	2005-12–17-03	371	840	115
Groovy	2013-09–2015-04	141	4892	792
Infinispan	2009-03–2016-12	299	2349	293
		Total	**10,463**	**1,529**

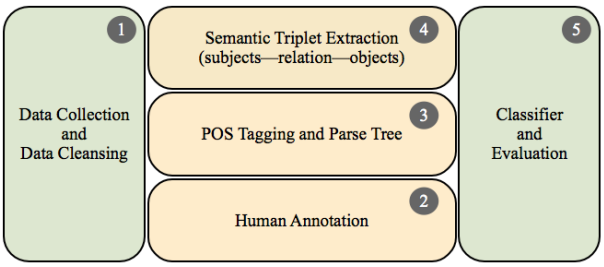

Fig. 2. Commit Message Refactoring Detector (CMMiner)

Fleiss' Kappa [7] statistics (which evaluates the extent to which two or more observers agree on category assignments) was used to measure agreement among the three raters. Fleiss characterizes Kappa range over 0.75 as 'excellent', $[0.40-0.75]$ as 'good', and less than 0.40 as 'poor'. However, following the first round of labeling, the Kappa value was less then satisfactory (i.e., $\kappa = 0.25$), therefore, the annotators discussed the differences, clarified the process, and repeated steps (1)–(4) with a different set of 100 randomly selected commit messages until satisfactory agreement was reached among the three reviewers (i.e., $\kappa = 0.79$). Once the process was established, the first author labeled the remaining commit messages using spot-checking with the original annotators to ensure reliability was maintained. At the end of the annotation process, 12 refactoring types had been identified by human annotators as depicted in Table II. We have made available a gold standard dataset with commit refactoring annotations.[1]

D. POS Tagging and Parse Tree

❸ The commit messages were then tagged with their parts-of-speech (POS) using the Stanford Parser [13], and a dependency tree was generated for each commit message. POS tagging was useful in determining the syntactic role of a word in a commit message, for example, distinguishing between the occurrences of the word when used as a noun or as a verb. After this disambiguation step, we were then able to infer semantic dependencies among other words, e.g., finding the "functional" structure of a text such as (verb-noun-object).

E. Semantic Triplet Extraction

❹ Each commit message was transformed into a triplet [4, 19], represented in the structured form as {subjects–relation–objects}, in order to automatically detect and extract relationships between textual entities (e.g., method names, class names and other contextual dependencies) from the syntactically parsed commit message. Hence, a commit message (S) can be represented as a parsed tree that consists of triplet: two nouns (NP) (i.e., subject and object) and a verb (VP) (i.e., relation). We employed Stanford's Open Information extraction to generate triplets automatically. The benefit of the triplet extraction approach is that it can describe a fact about the subject (i.e., entity that we are trying to know more about) and the object (i.e., entity that we need to discover).

[1] http://doi.org/10.5281/zenodo.3596397

TABLE II
REFACTORING TYPES IDENTIFIED DURING THE ANNOTATION PROCESS OF COMMIT MESSAGES USING TEMPLATE REFACTORING RULES [16]

Project	Add Parameter	Remove Parameter	Rename Method	Move Method	Hide Method	Extract Method	Move Field	Encapsulate Field	Move Class	Extract Class	Rename Class	Extract Subclass	Total
Derby	0	2	17	15	18	24	1	2	1	4	1	1	86
Drools	0	1	7	4	0	8	0	1	4	1	4	0	30
Groovy	7	3	12	7	5	30	0	1	5	3	0	2	75
Infinispan	8	1	7	6	0	23	0	0	8	3	7	0	63
Total	15	7	43	32	23	85	1	4	18	11	12	3	254

TABLE III
BINARY CLASSIFIER WITH CMR CLASS AND NON-CMR CLASS; P—PRECISION, R—RECALL, F_1—F-MEASURE

Classifier	Target Class	Baseline (1-gram) [20]			Baseline (2-gram) [20]			Baseline (3-gram) [20]			CMMiner		
		P	R	F1	P	R	F1	P	R	F1	P	R	F1
NB	CMR	0.602	0.154	0.245	0.704	0.090	0.090	0.893	0.041	0.078	**0.983**	**0.580**	**0.715**
	Non-CMR	0.837	0.977	0.901	0.827	0.991	0.902	0.816	0.999	0.898	0.830	0.996	0.905
	Weighted Avg	0.781	0.793	0.826	0.804	0.824	0.764	0.831	0.817	0.742	**0.878**	**0.857**	**0.844**
LR	CMR	0.746	0.077	0.140	0.964	0.021	0.042	0.899	0.010	0.002	0.996	0.519	0.682
	Non-CMR	0.827	0.994	0.903	0.817	1.000	0.899	0.810	1.000	0.895	0.817	0.999	0.899
	Weighted Avg	0.812	0.825	0.762	0.845	0.818	0.740	0.846	0.810	0.725	0.874	0.847	0.830
SVM	CMR	0.183	0.608	0.281	0.292	0.017	0.032	0.190	0.109	0.138	0.321	0.999	0.486
	Non-CMR	0.814	0.386	0.524	0.815	0.991	0.895	0.810	0.891	0.848	0.979	0.017	0.033
	Weighted Avg	0.698	0..427	0.479	0.718	0.810	0.734	0.692	0.742	0.713	0.770	0.328	0.177
kNN	CMR	0.413	0.151	0.221	0.287	0.158	0.204	0.253	0.128	0.170	0.983	0.556	0.711
	Non-CMR	0.833	0.952	0.888	0.826	0.910	0.866	0.817	0.911	0.861	0.828	0.996	0.904
	Weighted Avg	0.755	0.804	0.765	0.726	0.771	0.743	0.709	0.762	0.730	0.878	0.856	0.843

F. Classifier and Evaluation

We then trained several different classifiers and evaluated their performance using precision, recall and F-measure metrics. As a baseline, we used the n-gram model, which has been shown to perform well for closely related classification problems [2, 15], and has recently been used by Santos and Hindle [20] to detect unusual commit messages as predictors of poor code quality. N-gram based approaches are fundamentally different from CMMiner, where syntactic relations between words are important. We ran three baseline configurations of the n-gram model where $n = \{1, 2, 3\}$ to compute *unigrams*, *bigrams* and *trigrams*. We did not test higher values because predictive performance does not significantly improve for larger values of n [10].

As a first step, we trained a binary classifier to differentiate between CMR commit messages and non-CMR commit messages and then trained a multi-class classifier to differentiate between refactoring types on CMR-detected commit messages. We comparatively evaluated several classification algorithms, namely, Näive Bayes (NB), Logistic Regression (LR), Support Vector Machine (SVM), and k-Nearest Neighbors (kNN) using the data mining framework Orange [5], commonly used for classification problems. Following the labeling process, our dataset included 254 CMRs, representing approximately 16% of commit messages. Then, running a stratified random sampling, we split the dataset into 80% and 20% of training and testing data. To overcome the imbalance problem [9], we applied SMOTE [3] on the training set data. SMOTE generates synthetic minority examples to over-sample the minority class. Finally. we trained different models on the training set and tested the trained models on test set.

III. ANALYSIS OF RESULTS

A. RQ_1: Refactoring Explanations in Commit Messages

To answer RQ_1, we first report the results of using CMMiner to differentiate between CMR and non-CMR commit messages as depicted in Table III. Results indicate that CMMiner with Naive Bayes binary classifier achieved the best performance returning precision of 0.983, recall of 0.580, and an F_1 score of 0.715. The 12 refactoring types that were identified during manual labeling are depicted in Table II. Of these, six frequently occurring types, were also supported by RMiner (see Table IV). We report results achieved when training CMMiner to classify each of these six refactoring types individually. The SVM multiclass classifier performed best returning precision of 0.880, recall of 0.606 and an F_1 score of 0.711. It outperformed the baseline configurations.

B. RQ_2: Refactorings Overlaps Between Commits and Code

As previously stated, we used the existing RMiner tool [21] to detect code refactorings. RMiner takes a commit ID and returns a refactoring type corresponding to that commit.

TABLE IV
PERFORMANCE METRICS BEST RESULTS OF DETECTED CMRS USING *CMMiner* WITH SVM CLASSIFIER

Refactoring Type	P	R	F1	Support
Extract Class	0.941	0.596	0.800	23
Extract Method	0.100	0.350	0.150	26
Move Class	0.760	0.660	0.660	25
Move Method	0.429	0.500	0.462	36
Rename Class	0.920	0.953	0.936	169
Rename Method	0.960	0.709	0.723	38
Other-Refacorings	0.857	0.933	0.894	45
Weighted Avg	0.880	0.606	0.711	362

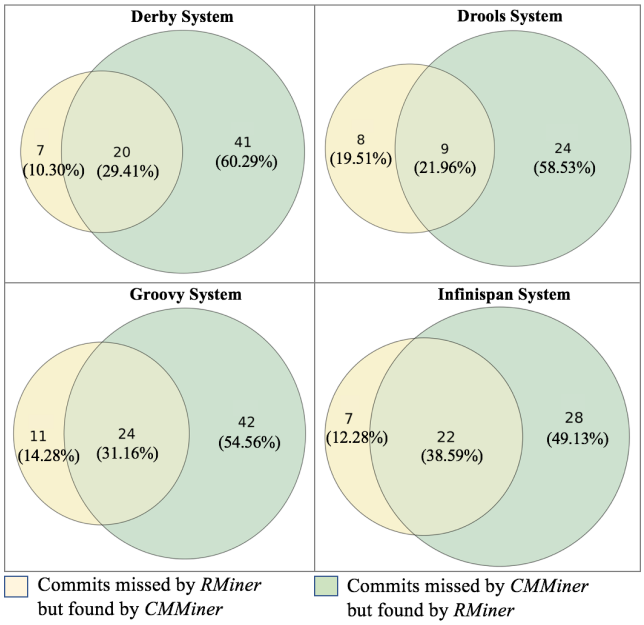

Fig. 3. Overlaps and Missed Refactorings between CMMiner and RMiner across Four Open-Source Domains

RMiner generally returns precision of 98% and recall of 87% [21]. When applied to our four open source projects it detected 23 different types of refactorings.

To answer RQ_2, we compared the refactorings detected by both RMiner and CMMiner for the six common refactoring types reported in RQ_1 as depicted in Table IV. We counted refactorings detected by RMiner and CMMiner and then visualized results in Figure 3 to show their overlaps and differences. For example, in the Derby system, 20 refactorings were found by both tools, 7 were found only by CMMiner, and 41 were found only by RMiner. These overlaps and deltas represent the following three scenarios.

- *Scenario 1 (Refactorings detected by both tools)*: Approximately 21.96% to 38.59% of the detected refactorings were found by both CMMiner (i.e., contextual) and RMiner (i.e., structural) as illustrated in Figure 1. We observed that the explanations often provided rationales and/or rich explanations for the refactoring change. In this case the refactoring developers' explanations can augment the structural explanations generated by the RMiner.
- *Scenario 2 (RMiner Only)*: 35.8% to 60.29% of refactorings were identified only by the RMiner. These represent cases where the developers failed to describe refactorings, in many cases because the refactorings were intermingled with functional changes, and the commit message focused on explaining those changes. An example is found in the commit message: *"DERBY–6189: Fix NPE involving a rollback of temporary table"*, that describes a bug fix for addressing a NullPointerException that occurs when attempting to roll back a transaction. However, RMiner detected the 'Extract Method' refactoring which was performed alongside the bug fix even though it was not mentioned in the commit message.
- *Scenario 3 (CMMiner Only)*: 10.30% to 19.51% of refactorings were identified only by the CMMiner. To inspected the commits and identified two main reasons why these commits were missed by RMiner: (i) CMMiner detected refactorings that occurred across multiple commits, whereas RMiner primarily focused on detecting single-commit refactorings. For example, in *DERBY–6802: Change NetworkServer code to use new MessageUtils class DERBY-6823. Refactor uses of DB2_JCC_MAX_EXCEPTION_PARAM_LENGTH... This change modifies several classes in the Network Server and Engine code bases to use the new features of the MessageUtils class in the shared code. The intent of this change is that no behavior is altered...*"; A similar situation was observed with 'Move Method' instances; (ii) CMMiner was able to detect information containing synchronous refactorings such as both moving and renaming a method, while the RMiner often missed such multiple-type refactorings.

IV. Discussion

A. Other Refactoring Detections

As depicted in Table IV, we also report results of detected CMR-commits containing 'other-refactoring' types that our tool CMMiner identified including: 'Hide Method', 'Extract Subclass', 'Move Field', 'Encapsulate Field', 'Remove Parameter', and 'Add Parameter'. These refactoring type were not covered by RMiner. An illustrative example is a 'Hide Method' CMR commit: *"GROOVY-3326: Obscure method getTimeZone(java.util.Date) in groovy.runtime.TimeCategory."*. Obviously, method getTimeZone(...) has been modified from *public* to *private*; however, this refactoring rule was not covered by RMiner, even though it could be important to detect. In this example, the access modifier for the getTimeZone(...) method was changed from public to private.

B. Implications and Practical Applications

Based on the examples of overlapping commits, we observed that commit messages can provide enhanced rationales for automatically generated code-refactoring descriptions. The scope of our ultimate goal has a broader implication that affects several domains. *Tool Builders* and *Researchers* can (1) integrate refactoring explanations from commit messages into refactoring descriptions that span single or multiple commits, (2) enhance refactoring detection to support changes applied across multiple commits, (3) develop improved static analysis techniques to detect a more diverse set of refactoring types at higher degrees of reliability, and (4) detect source code refactorings at commit-time, determine whether an explanation has been included in the commit message, and where necessary prompt developers to provide refactoring rationales. Finally, *Educators* can teach their students to follow exemplary practices for explaining refactoring changes in commit messages.

C. Threats to Validity

This is a preliminary case study that is designed to explore the extent to which developers explain refactoring changes in their commit messages. However, there are several threats to

validity. First, we could have missed refactoring descriptions that were worded in unexpected ways, thereby incorrectly annotating the answer set. This was partially mitigated through the use of a refactoring catalogue [16] and ensuring acceptable degrees of inter-rater agreement. As an initial study we seek to demonstrate the potential for integrating structural and contextual information for detecting and describing refactorings. To show generalizability we included diverse refactoring types found across the commits of four diverse open source systems.

V. Related Work

Commits and their change aspects have been studied and analysed by several researchers. Hattori and Lanza [8] analyzed the nature of commits. In particular, they analyzed the size of commits. They found that short commits were more related to bug fixes, whereas large commits were more related to new features. Hindle et *al.* [11] classified large commits in terms of their change type. Their results indicated that large commits tend to be perfective while the small ones were more likely to be corrective. Mockus and Votta [14] introduced a heuristic based approach for identifying reasons of changes (e.g., bug fixes) in the log messages. Recently many researchers have focused on untangling commits with bundled changes (e.g., bug fixes and refactorings) [6]. Alali et *al.* [1] analyzed several open-source projects to characterize ways in which developers commit source code. They extracted a vocabulary of terms which denoted actions and activities in commits. While a vocabulary-centered approach could reveal certain characteristics in commit messages, our CMMiner approach extracts semantic information that can reveal clause dependencies and relations between pair of words. Furthermore, while these prior approaches increase our understanding of the role of refactoring in maintenance tasks, they primarily focused on analysing bug fix changes, while other types of changes still lack empirical studies related to code refactoring.

VI. Conclusion and Future Work

We have presented a refactoring detection tool named *CMMiner* for detecting and classifying refactoring descriptions in commit messages. Our analysis has shown that developers described refactorings in 39% or more of their commit messages and that approximately 22% to 39% of commits contain refactoring explanations that could be used to augment refactoring descriptions generated by RMiner. Future work could develop commit message recommender systems and explore ways to integrate developers' refactoring descriptions with automatically generated descriptions.

Acknowledgments

The work in this paper is partially funded by the US National Science Foundation grant SHF-1909007. The authors would also like to thank the senior software developers who manually analyzed and annotated the dataset.

References

[1] A. Alali, H. H. Kagdi, and J. I. Maletic. What's a typical commit? A characterization of open source software repositories. In *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, pages 182–191, 2008.

[2] W. B. Cavnar, J. M. Trenkle, et al. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval*, volume 161175. Citeseer, 1994.

[3] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *J. Artif. Intell. Res.*, 16:321–357, 2002.

[4] L. Dali and B. Fortuna. Triplet extraction from sentences using svm. *Proceedings of SiKDD*, 2008, 2008.

[5] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan. Orange: Data mining toolbox in python. *Journal of Machine Learning*, 14:2349–2353, 2013.

[6] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 341–350, 2015.

[7] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[8] L. Hattori and M. Lanza. On the nature of commits. In *23rd IEEE/ACM International Conference on Automated Software Engineering - Workshop Proceedings (ASE Workshops 2008), 15-16 September 2008, L'Aquila, Italy*, pages 63–71, 2008.

[9] H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge & Data Engineering*, (9):1263–1284, 2008.

[10] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. T. Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, 2016.

[11] A. Hindle, D. M. Germán, and R. C. Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings*, pages 99–108, 2008.

[12] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto. Hey! are you committing tangled changes? In *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, pages 262–265, 2014.

[13] D. Klein and C. D. Manning. Fast exact inference with a factored model for natural language parsing. In *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS, Vancouver, British Columbia, Canada]*, pages 3–10, 2002.

[14] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *2000 International Conference on Software Maintenance, ICSM 2000, San Jose, California, USA, October 11-14, 2000*, pages 120–130, 2000.

[15] K. Ogada, W. Mwangi, and W. Cheruiyot. N-gram based text categorization method for improved data mining. *Journal of Information Engineering and Applications*, 5(8):35–43, 2015.

[16] K. Prete, N. Rachatasumrit, and M. Kim. Catalogue of template refactoring rules. *The University of Texas at Austin, Tech. Rep. UTAUSTINECE-TR-041610*, 2010.

[17] M. Rath, D. Lo, and P. Mäder. Analyzing requirements and traceability information to improve bug localization. In *International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 442–453, 2018.

[18] M. Rath, J. Rendall, J. L. C. Guo, J. Cleland-Huang, and P. Mäder. Traceability in the wild: automatically augmenting incomplete trace links. In *International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 834–845, 2018.

[19] D. Rusu, L. Dali, B. Fortuna, M. Grobelnik, and D. Mladenic. Triplet extraction from sentences. In *Proceedings of the 10th Intl. Multiconference" Information Society*, pages 8–12, 2007.

[20] E. A. Santos and A. Hindle. Judging a commit by its cover: correlating commit message entropy with build status on travis-ci. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 504–507, 2016.

[21] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 483–494, 2018.