On the Co-Occurrence of Refactoring of Test and Source Code

Nicholas Alexandre Nagy
Concordia University
Montreal, Canada
nicholas.a.nagy@protonmail.com

Rabe Abdalkareem
Carleton University
Ottawa, Canada
rabe.abdalkareem@carleton.ca

ABSTRACT

Refactoring is a widespread practice that aims to help improve the quality of a software system without altering its external behaviour. In practice, developers can perform refactoring operations on test and source code. However, while prior work showed that refactoring source code brings many benefits, few studies investigated test code refactoring and whether it co-occurred with source code. To examine the co-occurring refactorings, we conducted an empirical study of 60,465 commits spanning 77 open-source Java projects. First, we quantitatively analyzed the commits from those projects to identify co-occurring refactoring commits (i.e., commits contain refactorings performed on test and source code). Our results showed that on average 17.9% of refactoring commits are co-occurring refactoring commits, which is twice as much as test code-only refactoring commits. Also, we investigated the type of refactorings applied to test code in those co-occurring commits. We found *Change Variable Type and Move Class* are the most common applied refactorings. Second, we trained random forest classifiers to predict when refactoring test code should co-occur with refactoring source code using features extracted from the refactoring source code in ten selected projects. Our results showed that the classifier can accurately predict when test and source code refactoring co-occurs with AUC values between 0.67-0.92. Our analysis also showed that the most important features in our classifiers are related the refactoring size and developer refactoring experience.

KEYWORDS

Source and Test Code Refactoring, Empirical Study

ACM Reference Format:
Nicholas Alexandre Nagy and Rabe Abdalkareem. 2022. On the Co-Occurrence of Refactoring of Test and Source Code . In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528529

1 INTRODUCTION

Refactoring is recognized as a fundamental practice to maintain a healthy codebase. In practice, developers mainly refactor their code to handle changes (i.e. new features and bug fixes [15]). In addition, test code design and smells play an important role in test code quality [7]. In particular, test smells are known to have a high survivability rate and are difficult to identify by developers [20]. Therefore, prior works have focused on detecting smells and design issues in test code and proposed tools to help developers identify them [1] and approaches to refactor them [8, 22].

Since test code is written to ensure that the functionality of source code is working correctly, it is fair to assume that source and test code are somehow related. Therefore, when developers address source code quality issues by refactoring the source code, they may need to refactor the test code as well. However, little knowledge is available about this relationship and what factors of source code refactoring influence test code refactoring, nor how it can be used to suggest test code refactoring opportunities.

To that end, we began our study by identifying the refactoring commits in 77 projects from the SmartSHARK dataset [16]. Then, we determined whether they applied source code refactorings, test code refactorings, or both (i.e., co-occurring refactorings commits) using RefactoringMiner [19]. In the cases of co-occurring refactoring commits, we examined what types of refactorings are applied to test code. Next, we extracted features from source code refactorings from both the commits that only refactor source code and those that co-occur from ten selected projects. Then, we trained Random Forest classifiers on these features to predict whether the commits should co-occur and refactor test code. Also, we extracted the feature importance from our classifiers to determine which features are most valuable in predicting when commits should co-occur. We formulated our study into the following two RQs:

RQ1: How often do source and test code refactoring co-occur, and what are the refactoring types applied to test code? We found that most refactoring commits only refactor source code, representing 73.9% of refactoring commits for the average project. Co-occurring refactoring commits also represent 17.9%, which surprisingly is more than double the 8.2% of commits that only refactor test code. Additionally, we found that *Change Variable Type, Move Class, and Rename Method* are the most common refactorings applied to test code in those co-occurring refactoring commits.

RQ2: Can we predict when refactoring test code should be co-occurring with source code refactoring? Through our built classifiers, we could accurately predict when test and source code refactoring co-occurs with AUC values between 0.67-0.92 and median Precision, Recall, and F1-score equal to 0.70, 0.63, 0.66, respectively in the ten studied projects. Also, our analysis showed that the most important features for our predictive classifier are related to refactoring size, developer refactoring experience, and file refactoring history.

Finally, we make our scripts and dataset available online [12].

2 CASE STUDY DESIGN

Our main goal is to investigate the co-occurrence of source code and test code refactoring changes based on commit level granularity. Thus, we first wanted to differentiate between commits that perform

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528529

refactorings to test code, source code, or both. Then, we investigated the occurrences of each type of refactoring commit and the types of refactoring operations applied to test code in co-occurring commits. Second, we extracted commit related features to build classifiers to predict when refactoring test code should accompany source code refactoring and examined the most important features.

2.1 Studied Projects

Since our analysis aimed to investigate the co-occurrence of test and source code refactoring opportunities, we needed to study projects with sufficient development history. Thus, we resorted to using the SmartSHARK dataset [16]. SmartSHARK is a publicly available dataset that provides detailed information about the development history of 77 Java projects. We obtained the SmartSHARK MongoDB data dump published on 2021-09-08 [17]. The dataset combines meta-data from several sources such as Git and issue tracking systems. Also, the dataset provides information about code changes, particularly the performed refactorings, which are essential for our study. We cloned and analyzed the 77 projects.

2.2 Identifying Refactoring Related Commits

Once we obtained the 77 projects, we next wanted to determine which commits in those projects performed refactorings. Since the SmartSHARK dataset provides this information, we began by examining the collected refactoring information. The SmartSHARK dataset detected refactorings on commits level using two different tools, RefactoringMiner [19] and RefDiff [14]. In this study, we decided to focus on studying the refactorings identified by RefactoringMiner, since 1) prior work showed that it has better recall and precision and 2) the newest version (i.e., version 2.2 [18]) supports the identification of 85 different refactoring types, which is not supported in the newest version of RefDiff [19].

Since we wanted to identify whether a refactoring is applied to test or source code, we needed the associated commit and file information for which it was identified. In the SmartSHARK dataset, each refactor has a commit-id, which afforded us the ability to find which commit the refactor was identified at. However, finding the files touched by a refactoring from those that RefactoringMiner identified was not a straightforward step. Thus, we decided to use the hunks field information provided in the dataset for each refactoring operation, which RefactoringMiner provided.

Yet, upon examining the hunk information in the SmartSHARK dataset, we noted that some hunk information is missing (i.e., some refactoring documents in the collection were missing hunk fields). This is perhaps due to the complexity of linking the refactoring changes identified by RefactoringMiner to the hunks identified in each commit by the vcsSHARK tool [16].

Therefore, we could not reliably use the hunks from the dataset's refactoring collection, so we resorted to running RefactoringMiner ourselves on the identified refactoring commits from the 77 studied projects. By the end of this step, we were able to extract the file information for each refactoring operation, which we used in the next step to determine whether a refactoring in a commit was performed on the test, source code, or both. Table 1 shows the summary statistics of all commits and the identified refactoring related commits and their percentage in the 77 projects. We observed that on median 16.7% (mean = 16.9%) of the commits in studied projects are performing refactorings.

Table 1: Summary of all commits and refactoring commits.

Commit	Min.	Median	Mean	Max.
Total	1,066	3,380	4,856	21,365
Refactor (%)	1 (0.0)	579 (16.7)	758.2 (16.9)	3,239 (36.2)

2.3 Categorizing Refactoring Related Commits

Once we identified the commits that perform refactorings, now we wanted to determine whether the performed refactorings are done on test code, source code, or both, and identify the applied refactoring types. As described in Section 2.2, we first run RefactoringMiner on each refactoring commit identified by the dataset. This step resulted in a list of refactorings detected for each commit and their associated information, such as the type (e.g. , move class, etc.), left side locations (i.e. parent commit locations), right side locations (i.e. child commit locations), and the refactoring description. In addition and based on the refactoring location information, we were able to determine which files were touched by the refactoring, and we used the path to these files to identify whether the refactoring is being applied to test code or source code.

To determine whether a refactoring was performed on test code or source code, we applied the same detection approach that was used in prior [3, 13] on each file touched by the refactoring operations. The approach is mainly composed of two steps. The first step is to examine whether the file name either starts or ends with the word "test". Second, if this is the case, then we used JavaParser [21] to parse the Java file (i.e., files ending in ".java"). Then, with the parsed file, we were able to eliminate test files with syntax errors and detected JUnit-based test methods, which significantly cuts down on false-positive test files [3]. Following these steps, if any of the files touched by the refactoring are identified as a test file, then we classified the refactoring as a test refactoring.

Once we had the refactoring data for each commit, we began categorizing all the commits based on the refactoring activity applied. In our study, there were three categories that we identified for the examined commits: test refactoring commits, source refactoring commits, and co-occurring refactoring commits. Each commit in our dataset will be categorized as one of those three types. Below, we provide more details about each commit type:

- *Test Refactoring Commit:* For this type of commit, all applied refactoring operations are applied on test code.
- *Source Refactoring Commit:* For this type of commit, all applied refactoring operations are applied on source code.
- *Co-Occurring Refactoring Commit:* For this type of commit, there are different refactoring operations that are applied on both source and test code.

3 CASE STUDY RESULTS

In this section, we present the results of our case study.

RQ1: How often do source and test code refactoring co-occur, and what are the refactoring types applied to test code?

Motivation: Refactoring is a heavily studied subject among researchers, and most prior works have examined refactoring when applied to source code (e.g., [2, 11]). However, there is little work done on the different refactorings applied to test code, particularly on test code that co-occurs with source refactoring (e.g., [3]). Also, since test code quality is important [4, 10, 20], we set out to

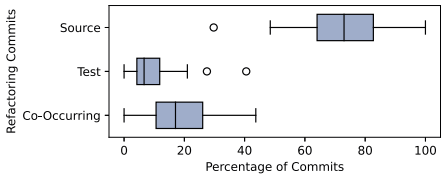

Figure 1: Boxplots representing the % commits with that classification over the projects with refactoring commits.

investigate test code refactorings that co-occur with source code refactorings.

Approach: As explained early, we applied a similar approach of prior studies that used the JUnit standards for defining what a test file is [3, 13]. In this approach, we used RefactoringMiner to first determine which files each refactoring touched and flag those that touched test files. Then, we were able to identify which refactorings are applied to tests and which are only applied to source code.

With this method, while analyzing the commits with RefactoringMiner, we categorized every commit in our dataset into; test refactoring commits, source refactoring commits, or co-occurring refactoring commits. Then, we calculated the percentage of each commit type in each project in our dataset. Also, to determine the types of refactorings applied to tests that co-occur with source code refactorings, we used the test refactorings that occur in commits that also apply refactorings to source code. Thus, we extracted the refactoring type report by RefactoringMiner, and for each type, we calculated its frequency per project in our dataset.

Result: Figure 1 shows the distributions of the percentage of the test, source, and co-occurring refactoring commits in all the studied projects. Although most refactorings are applied on source code, instances of all refactoring types detectable by RefactoringMiner can be found in the test code that co-occurs with source code refactoring. As shown in Figure 1, aside from a few outliers, the majority of projects refactor source code more often than test code. Despite this, we observed a wide variety of refactoring types being applied. In Figure 1, source code refactoring commits make up 73.9% on average, compared to the 8.2% and 17.9% on average for test refactoring commits and co-occurring commits, respectively. Despite this, there is still a small portion of projects that take test code refactoring seriously. Even when refactoring source code simultaneously, every type of refactoring is observed in test code refactoring. With this in mind, there is still a limited amount of refactoring types that co-occur on test code across most projects. In Figure 2, we showed the top ten most highly co-occurring refactoring types per project based on the median. Please note we showed only the top ten due to the limited space of the paper (a full list is in our replication package). Figure 2 shows that the most frequent refactoring type applied to test code in co-occurring refactoring commits are the change variable type, move class, and rename method.

> For the majority of projects, co-occurring refactoring commits are infrequent compared to source code refactorings, and the variety of refactorings applied to these commits is limited.

RQ2: Can we predict when refactoring test code should be co-occurring with source code refactoring?

Motivation: As can be seen in RQ1, source code refactoring is a more common activity for developers than test code refactoring.

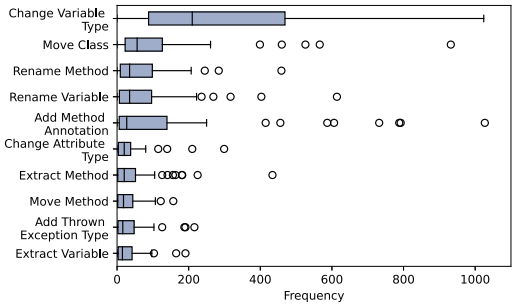

Figure 2: Boxplots of refactorings types applied to tests in co-occurring refactoring commits.

Despite this, test code is still vulnerable to various smells that deter the overall quality of the tests (e.g., [20]). To that end, we attempt to use the information from the source code refactorings commits to see if we can predict whether refactoring source code should be accompanied with test refactorings.

Approach: We trained classifiers that can automatically identify co-occurring refactoring commits. First, our classifiers take as input a list of commits that are labelled as source code refactoring commits or co-occurring refactoring commits (see Figure 1). Then, our classifiers are trained based on features extracted from the applied refactorings on the source code alone. In this study, we proposed using different features that we believe can give a good indication of whether test code should be refactored when source code is. We first used all the information provided by the RefactoringMiner tool [19]. Second, we extracted other features (e.g., the developers refactoring experiences). It is important to note that all the studied features are extracted from the source code refactorings in source code refactoring commits or co-occurring refactoring commits. For the sake of conciseness, we presented the features used in Table 2.

Since our dataset contains several projects and based on our results for the first RQ, some of these projects have a very small

Table 2: Features extracted from the applied changes to source code only in source code refactoring commits and co-occurring refactoring commit. *These features describe a set of commit features as opposed to a single feature.

Features	Definition
# of refactorings	# of source code refactorings observed in a commit.
# of left side locations	# of locations touched by refactorings from the parent commit.
# of right side locations	# of locations touched by refactorings from the child commit.
LOC left side	Lines of code touched in parent commit by refactorings.
LOC right side	Lines of code touched in child commit by refactorings.
# of files	# of files touched by refactorings.
Average number of files	Average number of files touched across refactorings in a commit.
# of unique refactoring types	# of unique refactoring types applied in a commit.
# of unique code elements	# of unique code elements identified by each refactor.
# of previous refactorings	# of refactorings each file has had previously with the average taken of all files.
Refactoring age	# of days since the last refactoring for each file, with the average taken as its value.
Developer refactoring experience	# of refactorings applied by the developer previously to this commit.
Developer refactoring commit experience	# of refactoring commits made by the developer previously to this commit.
Refactoring type count*	A feature for each refactoring type in the commit, along with its number of occurrences as its value.
Code element type count*	A feature for each code element type in the commit, along with its number of occurrences as its value.

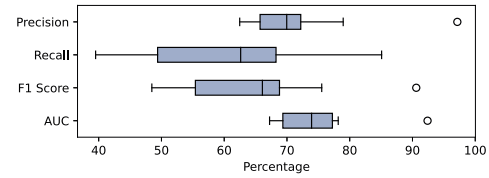

Figure 3: Boxplots of the resulting values of the metrics used to evaluate the built Random Forest classifiers.

number of refactoring commits (see Table 1). In addition, since we wanted to study the possibility of determining co-occurring refactoring commits, we needed to study projects with a sufficient number of co-occurring refactoring commits. Thus, to build our classifiers, we selected ten projects from our dataset that maintain a good quality dataset in terms of the number of commits (i.e., source code and co-occurring refactoring commits). The list of projects can be found in our replication package [12]. Furthermore, to deal with the imbalance problem in our study, we applied the synthetic minority oversampling technique (SMOTE), which is a strategy for oversampling and can effectively boost a classifier's performance in our case [5]. However, it is worth noting that we only applied the sampling technique to the training dataset, and not applied it to the testing dataset.

Once our dataset was chosen, we began applying various machine learning classifiers. To compare the classifiers against one another and to determine the performance of each classifier, we compute the well-known evaluation metrics, which are Recall, Precision, F1-score, and Area Under the ROC Curve (AUC) value. In this study, we applied various classifiers that include Logistic Regression, Support Vector Machine, and Random Forest, at their default settings on a per-project basis using 10-fold cross validation [6]. We found the Random Forest classifier achieved the best performance.

In addition, in this RQ, we examined the importance of each of our features. For this, we used sci-kit's feature importance based on the mean decrease in impurity since our dataset does not have any categorical features and therefore does not have any high cardinality features, but may have correlated features, which has a negative impact on the alternative feature permutation-based importance [9]. Then, using this metric, we evaluated the importance of each feature for all of the Random Forest classifiers created for each project dataset. Finally, we aggregated the feature importance to demonstrate which features play the most important roles in predicting co-occurring refactoring commits.

Result: Figure 3 shows the results of the built Random Forest classifiers. The Figure presents the distribution of the precision, recall, F1-score, and AUC values for all the ten examined projects. As we can see, the built Random Forest classifiers achieve a mean F1-score of 0.65 (median = 0.66) and a mean AUC of 0.75 (median = 0.74). Overall, although these may seem like modest performance numbers, they are quite significant given the unbalanced nature of the studied dataset (i.e., only a small portion of the commits are co-occurring refactoring commits). Interesting, there are two projects that achieved an extremely high performance namely `Kafka` and `Phoenix` that achieve AUC of 0.78 (F1-score = 0.76) and AUC of 0.92 (F1-score = 0.91), respectively.

In addition, we examined the most important feature in determining when test code should be refactored with source code refactorings. Figure 4 shows the distribution of top ten most important

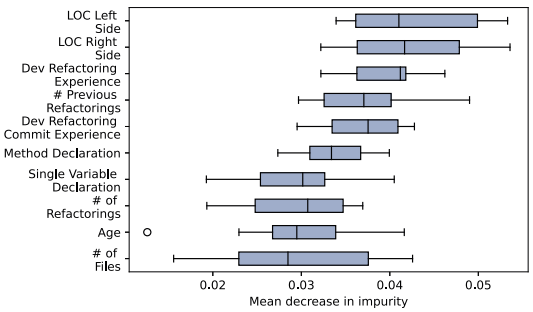

Figure 4: Boxplot of top features, chosen by importance in built Random Forest classifiers.

features used in the built Random Forest classifiers. Out of the top 10 features, they can be grouped into commit size features (LOC left side, LOC right side, # of files, # of refactorings), experience features (Dev refactoring experience, dev refactoring commit experience), history features (# Previous Refactorings, age), and code elements (Method and Single Variable declarations). Each group has its own significance, but this may demonstrate that each of these groups of features has an influence on whether test code should be refactored when source code is refactored.

> Our classifiers can effectively determine the co-occurrence of test and source code refactoring commits with a median AUC of 0.74. Also, our results showed that the refactoring size and developer refactoring experience are the most important features.

4 THREATS TO VALIDITY

There are a few significant limitations to our work that need to be considered when interpreting our findings. The first limitation is that we relied on the accuracy of the RefactoringMiner tool to identify, classify, and provide meaningful features for our classifiers. However, with an expected precision of 99.6%, and a recall of 94%, we expect the impact to be pretty negligible [19]. Another limitation of our study can be our dataset. Our work focuses only on 77 projects, all written in the same programming language, Java, and from the Apache organization. This means that our findings may not extrapolate well to other programming languages or other organizations.

5 CONCLUSIONS

In this work, we studied the co-occurrence of source code and test code refactorings opportunities. We started by analyzing 77 projects to identify the prevalence of co-occurring refactorings between source and test code. Our findings showed that on average 17.9% of refactoring commits are co-occurring refactoring commits. We also found that *Change Variable Type and Move Class* are the most common refactorings applied to test code in co-occurring refactoring commits. Additionally, we built predictive classifiers to determine when refactoring test code can co-occur with source code refactoring for ten projects. Our findings showed that the built classifiers could accurately predict when test and source code refactoring co-occur with AUC values between 0.67-0.92. Finally, our analysis also showed that the most important features for our predictive classifiers are related to the refactoring size and developer refactoring experience.

REFERENCES

[1] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. 2021. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering* (2021), 170–180.

[2] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2021. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software* 171 (2021), 110821.

[3] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Marouane Kessentini. 2021. How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation. *Expert Systems with Applications* 167 (2021), 114176. https://doi.org/10.1016/j.eswa.2020.114176

[4] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea Lucia, and Dave Binkley. 2015. Are Test Smells Really Harmful? An Empirical Study. *Empirical Softw. Engg.* 20, 4 (aug 2015), 1052–1094. https://doi.org/10.1007/s10664-014-9313-0

[5] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[6] Bradley Efron. 1983. Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American statistical association* 78, 382 (1983), 316–331.

[7] Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C. Gall. 2020. Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 336–347. https://doi.org/10.1109/ICSME46990.2020.00040

[8] Eduardo Martins Guerra and Clovis Torres Fernandes. 2007. Refactoring test code safely. In *International Conference on Software Engineering Advances (ICSEA 2007)*. IEEE, 44–44.

[9] Giles Hooker, Lucas Mentch, and Siyu Zhou. 2021. Unrestricted permutation forces extrapolation: variable importance requires at least one more model, or there is no free variable importance. *Stat. Comput.* 31, 6 (2021), 82. https://doi.org/10.1007/s11222-021-10057-z

[10] Dong Jae Kim, Tse-Hsun Peter Chen, and Jinqiu Yang. 2021. The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering* 26, 5 (2021), 1–47.

[11] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.

[12] Nicholas Alexandre Nagy and Rabe Abdalkareem. 2022. Refactoring Co-occurrence Replication Scripts + Data. https://doi.org/10.5281/zenodo.5979790

[13] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '19)*. IBM Corp., USA, 193–202.

[14] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: A Multi-Language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2786–2802. https://doi.org/10.1109/TSE.2020.2968072

[15] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[16] Alexander Trautsch, Fabian Trautsch, and Steffen Herbold. 2021. MSR Mining Challenge: The SmartSHARK Repository Data. In *Proceedings of the International Conference on Mining Software Repositories (MSR 2022)*.

[17] Alexander Trautsch, Fabian Trautsch, and Steffen Herbold. 2021. SmartSHARK 2.1 Full. https://doi.org/10.25625/7OZ1SP

[18] Nikolaos Tsantalis. 2021. tsantalis/RefactoringMiner at 2.2.0. https://github.com/tsantalis/RefactoringMiner/tree/2.2.0. (accessed on 02/05/2022).

[19] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2020), 21 pages. https://doi.org/10.1109/TSE.2020.3007722

[20] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An Empirical Investigation into the Nature of Test Smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 4–15. https://doi.org/10.1145/2970276.2970340

[21] Danny van Bruggen. 2008. JavaParser - Home. https://javaparser.org/. (accessed on 02/05/2022).

[22] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 92–95.