A Survey of Refactoring Detection Tools

Liang Tan
Philipps-Universität Marburg
Programming Languages and Tools
Marburg, Germany
tan@staff.uni-marburg.de

Christoph Bockisch
Philipps-Universität Marburg
Programming Languages and Tools
Marburg, Germany
bockisch@acm.org

***Abstract*—Several tools for detecting refactorings in the code exist and have been evaluated in the literature. However, we found that the benchmarks used for the evaluation so far are incomplete and therefore, the validity of the previous evaluations is at stake. While our completed benchmark largely confirmed the previous results, in particular confirming that RefactoringMiner generally outperforms its competitors, we also identified a weak spot of RefactoringMiner that was not noted before: Refactorings of the type *Move Class* and *Rename Package* are frequently classified falsely. In this paper we discuss the reasons for this wrong classification and outline a possible fix, which potentially boosts the overall precision and recall of RefactoringMiner to over 95%.**

***Index Terms*—Refactoring Detection, Reproduction Study, Move Method, Rename Package, RefactoringMiner**

I. Introduction

During software upgrades and maintenance, programmers frequently need to review code, e.g., to fix bugs or to identify how to implement new features. A primary task during a code review is understanding the code structure and logic. If code changed since the last review, developers often also need to understand the change to update their understanding. Such code changes, in particular to the code structure—i.e., to improve the internal code quality or prepare future extensions—, are often realized by means of *refactorings* [1].

The technique of refactoring software system has been around for a long time already, however, the term was probably coined in 1989 by Bill Opdyke and Ralph Johnson [2]. It has gained wide adoption especially in the agile software development community, marked by the popular book by Martin Fowler et al. [1]. A refactoring has the special property that it only changes the code's structure without changing its external behavior. Therefore, knowing a refactoring has been applied is important such that reviewers recognize that a change was not to the software's functionality. Also knowing which particular refactoring has been applied is useful, since refactorings typically are associated with a specific intention hinting, e.g., at the kind of extension being prepared.

Unfortunately, developers rarely document it when refactorings are applied. While it would be easy to automatically document refactorings when they are applied using a tool, manual refactorings are still very common. As a consequence this information is not immediately available to reviewers [3]. To counter this, several tools have been developed for automatically identifying refactorings that have been applied.

We have performed a common evaluation of these tools to make them more broadly comparable and have confirmed that the tool *RefactoringMiner* [4] shows the best results and identifies refactorings rather reliably in most cases. However, in our study applying refactoring detection tools to 55 code versions, we found some strange phenomena. Particularly, RefactoringMiner has problems distinguishing between the refactorings *Move Class* and *Rename Package*. As we will discuss, the evaluation by the original tool developers was incomplete and therefore this problem did not surface before and has not been investigated yet.

In this paper we first present a full evaluation using a completed benchmark, and then focus evaluating the tool RefactoringMiner, which performed best in this evaluation. This second evaluation reveals that RefactoringMiner performs uncharacteristically low for *Move Class* and *Rename Package*. As we will discuss, this is founded jointly in the nature of these refactorings and the mode of operation of RefactoringMiner, which is to analyze only the *diff* between versions of the code in a *software configuration management* repository. The eventual goal of our work is to understand the workings of tools contributing to the evolution and maintenance of long-living software systems.

II. Refactoring Detection in a Nutshell

When reviewers inspect code, they will typically look at the current version and focus on code that changed during the previous review. Since refactorings often result in moving code, some code may now be in a different place and therefore appear to be new. Thus, reviewers will focus on understanding this code and possibly attempt to ensure its internal quality, without recognizing that this code has already been reviewed—just in another context.

Today, it is a common standard to use software configuration management like Git to keep track of code changes, including refactorings. Reviewers, therefore, can easily access the changes by looking at the difference between two versions in Git. But, unless documentation of the applied refactorings is explicitly added, they will only see, which lines have been added, removed or changed. In this case, reviewers will either not be able to properly understand the reason of the code change, which can lead to a loss of efficiency and misjudgment during code reviews. Or they need to analyze the diff more

closely to recognize that several of the changes considered together form a specific refactoring.

For example, consider that the Extract Method refactoring has been applied: In this refactoring, a sequence of statements is cut out of the middle of a method and placed into a new method; a call to this method is then placed in the original location of the statement sequence. The new method would appear to be added and therefore it would be assumed to require a thorough review. Looking at the diff for this change, it would be possible to see that the body of the new method matches the lines that have been removed in another place and that a call to the new method has been inserted at the same place.

Recognizing the Extract Method refactoring in this way may sound trivial. However, often the commit discipline of developers is such that multiple refactorings are applied at once and committed jointly, possibly even mixed with functional changes. If single refactorings are committed separately, a large number of commits would have to be investigated, which is tedious when done manually. Also, e.g., during an Extract Method refactoring, the new method's body will not always be identical to the original code: Accesses to formerly local variables, fields or methods may need to change, or the implementation is generalized along with the method extraction. Therefore, recognizing a refactoring is, in general, a difficult task.

Figure 1 shows another example, namely applying the *Move Class* refactoring, which we will consider in more depth in the remainder of this paper. This also shows the difference as well as connection between *refactoring* and *refactoring detection*. Refactoring is a one-way technique to improve code, taking as input one code version and emitting as output the new version. Refactoring detection is bidirectional in the sense that it requires two code versions as input, the new and the old version (parent and child commits in git), and outputs the applied refactoring.

There are two major aspects in the process of refactoring detection: code matching and refactoring analysis. Code matching is the premise for finding refactorings. The difference between the code in the two versions is the clue of a refactoring. Finding these clues is the basis of efficient detection. These clues are further processed by the refactoring analysis. A quick and accurate determination of the refactoring type is the ultimate goal of detection.

Many experts and scholars have done research in this direction and some tools have been developed to detect the refactoring information (refactoring location and refactoring type) from the code. Those tools, which can recognize multiple refactoring types, are summarized in the following subsection.

III. Refactoring Detection Tools

In 2000, Demeyer et al. [5] proposed reverse engineering of reconstructed code and proposed a detection heuristics. Another relevant development is the increasing availability of technologies, which enable the detection of refactorings. For example, Toshihiro Kamiya et al. [6] developed CCfinder, a

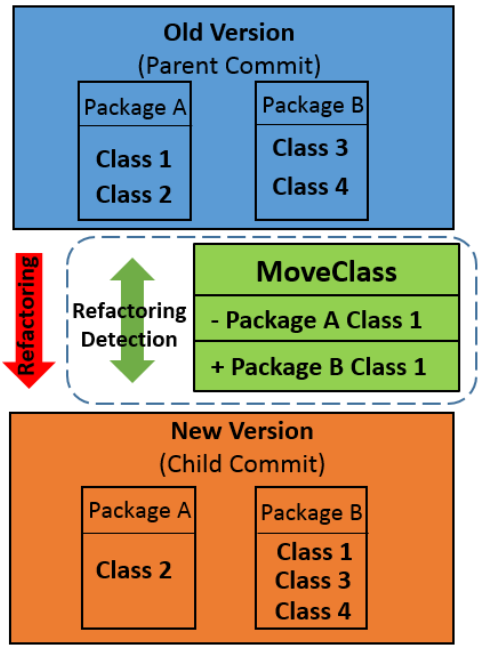

Fig. 1. An example of the information involved in refactoring detection.

clone detection technique that consists of the transformation of input source text and a token-by-token comparison. Since then, methods for the detection of refactorings and related technologies have entered a stage of rapid development.

The first refactoring detection approaches focused on specific refactorings that are relatively simple. Van Ryssel-berghe and Demeyer [7]—inspired by palaeontological fossil research—proposed a "software palaeontology" heuristics in 2003. This method paid special attention to the evaluation of the "Move Method" refactoring. Their work consisted of comparing different releases of existing source code, analyzing differences and reconstructing past evolution processes. In the same year, Malpohl et al. [8] proposed the algorithm "Renaming Detection", which is equally applicable to data description languages such as XML. The detector works with multiple file pairs, also finding renamings that span several files. It is part of a suite of intelligent tools for merging programs exploiting the semantics of programming languages.

After the first concrete approaches to detecting specific refactorings, theories for more holistic methods have been developed. Antoniol et al. [9] presented an approach, based on Vector Space Cosine Similarity (used in machine learning) of class identifiers, to automatically identify class-level refactorings between two subsequent releases. The approach was useful to identify some replacement, merge and split during the evolution of "dnsjava" [1]. It uses a calibration threshold to influence the precision and recall of the detection method. An object-oriented design structure difference algorithm ("UMLDiff") was proposed by Xing and Stroulia [10] in 2005. UMLDiff can detect additions, removals, moves,

[1]See the "dnsjava" homepage: http://www.dnsjava.org

renamings of packages, classes, interfaces, fields and methods, changes of attributes and changes of the dependencies among these entities. Based on the UMLDiff algorithm, JDEvAn (Java Design Evolution Analysis) [11] can automatically detect the design changes between two models corresponding to two versions of a system.

The detection approaches and algorithms proposed by some early researchers laid a theoretical foundation for later refactoring detection tools. Some researchers used the above algorithms and ideas to develop complete refactoring detection systems, i.e., tools that automatically detect the application of different types of refactorings in the code. These systems will be the subject to the study presented later in this paper.

The earliest comprehensive detection tool developed is RefactoringCrawler by Danny Dig et al. [12] in 2006. This tool was implemented as a plugin for Eclipse and can detect seven types of refactorings in Java code, focusing on rename and move refactoring. In 2010, the tool Ref-Finder, developed by Kyle Prete and his team, was proposed [13]. This tool is based on the tool LSdiff (Logical Structural Diff) released by Kim et al. [14], [15], to compute the delta between two versions of the source code. Prete and his team used a logic meta-programming approach to identify complex refactorings from two program versions and a tool, which they say supports 63 refactoring types. RefactoringMiner was proposed in 2013 by Tsantalis [4], [16] and his research group. It implements a lightweight version of the UMLDiff algorithm for computing the differences between object-oriented models independent of the IDE. It can detect ten kinds of refactorings. In 2017, RefDiff was created by Danilo Silva and Marco Tulio Valente [17], an automated approach that identifies 13 different refactoring types by inspecting two code revisions in a git repository, using a similarity index.

IV. An Evaluation of Refactoring Detection

To compare the four most complete refactoring detection tools discussed above, we created a common benchmark[2] to which we applied all tools. In this way, their ability to detect refactorings is determined using the same code bases, change sets, and expected results. The benchmark combines code bases and corresponding changes used in previous studies to evaluate refactoring detection tools. We improved it to be applicable to all tools by making it accommodate for the different forms of input required by the different tools: either by means of a Git repository or by means of two Eclipse projects reflecting the code before and after the change.

In Table I, we show the results for this combined benchmark and for all tools, but limited to only the refactoring types commonly detected by all four tools (which is only four types). This confirms the previous results from the literature, namely that RefactoringMiner performs best among the available tools. While RefactoringCrawler has a slightly higher precision (96% rather than 93%) than RefactoringMiner, the latter has a higher recall (73% compared to 60%). The F1-score computes a combined measure for precision and recall [18], [19], and (as can be seen in the table) RefactoringMiner has the highest ranking in the F1-score.

As said before, this comparison of all tools only considered the common subset of refactoring types. Because RefactoringMiner produced the best results in this comparison (which is consistent with the literature), we further focused on this tool and also determined the precision and recall for all refactoring types supported by RefactoringMiner (using the same benchmark applications as before). In this more detailed benchmark—results are shown in Figure II—the results remain very good, but diverge slightly from the benchmark presented above, which was limited to only four refactoring types. Considering all code bases in the benchmark and all supported refactorings, RefactoringMiner has a total precision of 94% and a total recall of 75% (Figure II), which is in line with the results reported by the original authors.

Although still high, the precision in our detailed benchmark is lower than in the limited benchmark, while the recall is higher. Taking a closer look, we identified that RefactoringMiner has problems with the two refactoring types *Move Class* and *Rename Package*, as shown in Table II. For these refactoring types, the recall drops to 34% for *Move Class*, respectively to 11% for Rename Package. This is inconsistent with the results previously presented in the literature. The reason is that the benchmarks previously used were less complete and even missed out several actually applied refactorings in the set of expected results.

A. *Detection of* Move Class *and* Rename Package

After discovering this behavior of RefactoringMiner, we wanted to understand it further and conducted a more comprehensive and in-depth test for the refactorings *Move Class* and *Rename Package* respectively. Again, we used the original benchmarks with our extended set of expected results. Since we specifically focused on the two refactorings, we only considered samples that contain at least two instances of the refactoring we are investigating. The test results are shown in the tables III and IV. The "Number" column in the table refers to the test samples in the code base to make the experiment repeatable and the results traceable. A benchmark sample consists of two versions of a code base, where the second version is based on the first one but with the refactorings applied. In the tables, we refer to the two versions by giving the Git commit number, thus, the base and commit form the two versions. We also provide the expected number of *Move Class* or *Rename Package* refactorings to be found, as well as the true and false positives and the false negatives in the results of RefactoringMiner. Lastly, we present the precision and recall calculated from these figures.

The data in Table III, presenting the results for *Move Class*, show that of four data points the results are significantly different from the results of the other items in terms of recall. The excellent results in precision, cannot cover the instability of recall. The number of undetected *Move Class* refactorings in

[2]The benchmark can be accessed online:
https://bitbucket.org/tanliang11/struts/src
https://bitbucket.org/bockisch/jhotdraw/src and https://umrplt.bitbucket.io/

TABLE I
RESULTS FOR THE ACCURACY OF ALL REFACTORING DETECTION TOOLS, USING ONLY THEIR COMMONLY SUPPORTED REFACTORING TYPES.

Type	True positive	False positive	False negative	Precision	Recall	F1 score
RefactoringCrawler	47	2	32	0.959	0.595	0.734
Ref-Finder	91	135	79	0.403	0.535	0.460
RefactoringMiner	124	9	46	0.932	0.729	0.818
RefDiff	113	72	57	0.611	0.665	0.637

TABLE II
RESULTS FOR THE ACCURACY OF REFACTORINGMINER USING ALL ITS SUPPORTED REFACTORING TYPES.

Type	True Positive	False Positive	False Negative	Precision	Recall
Extract Method	28	1	1	0.966	0.966
Inline Method	26	2	0	0.929	1.000
Pull Up Method	19	0	0	1.000	1.000
Push Down Method	10	1	0	0.909	1.000
Move Method	44	4	4	0.917	0.917
Move Class	21	0	44	1.000	0.338
Extract Superclass	9	0	0	1.000	1.000
Extract Interface	5	0	0	1.000	1.000
Move Attribute	85	8	0	0.914	1.000
Rename Package	5	0	39	1.000	0.114
Push Down Attribute	10	0	0	1.000	1.000
Total	262	16	88	0.942	0.749

TABLE III
DETAILED ACCURACY RESULTS FOR REFACTORINGMINER AND THE *Move Class* REFACTORING TYPE.

Number	commit	Expect Result	True positive	False positive	False negative	Precision	Recall
1102923	eclipse-themes 72f61ec	3	3	0	0	1.000	1.000
1107905	elasticsearch f77804d	4	0	0	4	N/A	0.000
1116663	buck 1c7c03d	10	7	2	3	0.778	0.700
1118645	okhttp c753d2e	29	0	0	29	N/A	0.000
1130125	WordPress-Android 9dc3cbd	3	3	0	0	1.000	1.000
1132674	orientdb f50f234	12	4	1	8	0.800	0.333
1134151	gradle 36ccb0f	7	7	0	0	1.000	1.000
1139721	liferay-plugins 78b5475	5	5	0	0	1.000	1.000
1140071	docx4j e29924b	184	184	0	0	1.000	1.000
1147092	neo4j 4beba7b	6	6	0	0	1.000	1.000
1147835	jersey ee5aa50	8	8	0	0	1.000	1.000
1150594	hazelcast f1e26fa	13	13	0	0	1.000	1.000
1152530	hydra 7fea4c9	7	4	0	3	1.000	0.571
1159198	jedis 6c3dde4	26	26	0	0	1.000	1.000
Total		317	270	3	47	0.989	0.852

these four abnormal data points, with a recall of less than 60%, is 44, accounting for 14% of the total number of refactorings. (Note that the original authors reported 100% precision and 96.24% recall for *Move Class* [4].) In two out of the 14 commits, RefactoringMiner did not report any results for *Move Class*, which means that we cannot compute the individual precision for these cases.

This is similar for the results for *Rename Package*, found in Table IV. In four out of nine cases, no results were reported at all and we could not compute the precision. For the remaining commits, the precision was at 100%. In terms of recall, from the total 9 samples, RefactoringMiner has a recall below 50% in 7 cases and even could not find any true positives in 4 cases. The number of false negatives is in total at 91%. In the original evaluation of the RefactoringMiner authors, a precision of 85% and a recall of 100% had been reported for *Rename Package* [4].

When investigating the results further, we found that false negatives for *Move Class* were often falsely classified as *Rename Package* and vice versa. Therefore, we believe that RefactoringMiner easily confuses these two refactorings.

TABLE IV
DETAILED ACCURACY RESULTS FOR REFACTORINGMINER AND THE *Rename Package* REFACTORING TYPE.

Number	commit	Expect Result	True positive	False positive	False negative	Precision	Recall
1101310	sonarqube abbf325	2	1	0	1	1.000	0.500
1101296	sonarqube 4a2247c	2	0	0	2	N/A	0.000
1123966	spring-data-neo4j 071588a	27	2	0	25	1.000	0.074
1125333	facebook-android-sdke 813a0b	8	0	0	8	N/A	0.000
1134096	hibernate-orm 44a02e5	3	2	0	1	1.000	0.667
1136729	reactor 669b96c	2	0	0	2	N/A	0.000
1140316	aws-sdk-java 14593c6	3	0	0	3	N/A	0.000
1142116	infinispan 8f446b6	7	2	0	5	1.000	0.286
1157300	android c976598	35	1	0	34	1.000	0.029
Total		89	8	0	81	1.000	0.090

B. *Distinguishing* Move Class *and* Rename Package

To understand this, let us start by looking at the definition of the two types of refactoring.

Move Class: Move a class to another package. The class's simple name is not changed, but the file is moved to a different path and the package statement is changed. The contents of the class are unchanged.

Rename Package: Rename a package. All Java-files contained in the path of the original package are moved to the path corresponding to the new package name. The simple class names stay the same, i.e., the contents of the package does not change, except for the package statement which now uses the changed package name.

The descriptions of the two refactoring types are very similar, namely files are moved to a different path, package statements change and classes stay otherwise the same. An essential difference is that in one case only one Java-file is moved and the rest of the classes in a package stay untouched, and in the other case, all Java-files and resources are moved to a new path. Apparently, RefactoringMiner has difficulties recognizing this difference.

We examined the inner workings of the RefactoringMiner to further understand this. It simply analyzes the difference between two code versions using the diff-feature of Git. Thus, it does not need to perform a comprehensive matching screening for all the code of the two project versions.

Figures 2 and 3 show the diff output for two samples in our benchmark in the side-by-side view of Bitbucket. The refactoring in Figure 2 is reported as *Move Class* by RefactoringMiner and actually really shows a *Move Class*. The refactoring in Figure 3 is also reported to be *Move Class*, however, this time actually a *Rename Package* refactoring had been performed. We can see that the diff only shows the changed file names and package statements. In both cases, the structure of the diff is identical and it is impossible to judge which of the refactorings has been applied by using only this data.

This is not surprising if we recall the definition of the two refactorings, which both include the step of moving a Java-file and changing its package statement while leaving the file otherwise untouched. The difference lies within the context in which the moved file appears: For *Move Class*, the class is moved to a package (and thus, the Java-file is moved to a path), which already existed before. Except for this one class, the contents of the original package is left unchanged, and the package will not disappear after the refactoring. For *Rename Package*, the target package (and this path) did not exist before and the old package disappears.

However, this required context information is not visible in the diff, which is focused on showing differences in file contents. Whether a directory was newly created or disappeared in a commit, is not reflected. Likewise, unchanged contents are not reported, i.e., it cannot be seen if a directory contains other files than the ones that changed.

V. CONCLUSION

In this article, we have presented an overview of the history of refactoring detection tools, a description of the four refactoring detection tools RefactoringCrawler, Ref-Finder, RefactoringMiner and RefDiff. These are, to our knowledge, the only tools available today for recognizing a number of different refactoring types at once. For these four tools, we have presented a common evaluation for a direct comparison of their performance. As the tool performing best in our evaluation, we have further-on presented a more detailed evaluation of RefactoringMiner. Our evaluation largely reflected the results reported in the literature before, namely that RefactoringMiner outperforms the other tools with an F1 score of nearly 82%. For the two refactorings *Move Class* and *Rename Package*, we measured values for precision and recall which significantly diverge from all other supported refactorings, as well as from the evaluations presented in the literature so far.

Therefore, we further investigated the approach of RefactoringMiner for these two cases and found that the implementation approach of using only the diff of two code versions on Git hinders the proper detection of *Move Class* and *Rename Package*. The reason is that the difference of these two refactorings lies within the content which did *not* change, and this is not shown in the diff. For this reason, RefactoringMiner frequently confuses these two refactorings. We, thus, conclude that analyzing the diff provided by a version control system

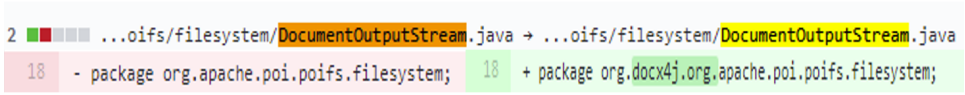

Fig. 2. Example diff for a *Move Class* refactoring.

Fig. 3. Example diff for a *Rename Package* refactoring.

such as Git is powerful for detecting refactorings that do not depend on the context in which they appear.

If the refactorings *Move Class* and *Rename Package* would be disregarded, RefactoringMiner would even have a precision and recall of 94% and 98%, respectively, in the evaluation shown in this paper. Since the information required to distinguish between *Move Class* and *Rename Package* (namely which directories have been created or deleted during a commit) could be easily obtained, we conclude that RefactoringMiner—with a simple extension—could in principle reach this high level of accuracy.

Since RefactoringMiner does not support detection for all refactoring types described in the literature, but only for a select subset thereof, we need to further investigate if this conclusion holds in general. In the future, diff-based detection for additional refactoring types should be researched, as well as additional contexts that are relevant for the distinction of refactorings.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code (Object Technology Series)*. Addison-Wesley, 1999.

[2] O. William F. and J. Ralph E., "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems," in *SOOPPA*. Pough-keepsie: ACM, 1990, pp. 145—-161.

[3] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5082 LNCS. Springer-Verlag, 2008, pp. 252–266.

[4] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and Efficient Refactoring Detection in Commit History," in *40th International Conference on Software Engineering (ICSE'18)*. New York, New York, USA: ACM Press, 2018, pp. 483–494.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 166–177, 2000.

[8] G. Malpohl, J. J. Hunt, and W. F. Tichy, "Renaming detection," in *Proceedings ASE 2000: 15th IEEE International Conference on Automated Software Engineering*. IEEE, 2000, pp. 73–80.

[6] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, jul 2002.

[7] F. Van Rysselberghe and S. Demeyer, "Reconstruction of successful software evolution using clone detection," in *International Workshop on Principles of Software Evolution (IWPSE)*, vol. 2003-Janua. IEEE, 2003, pp. 126–130.

[9] G. Antoniol, M. Di Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *Proceedings. 7th International Workshop on Principles of Software Evolution*. IEEE, 2004, pp. 31–40.

[10] Z. Xing and E. Stroulia, "UMLDiff," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05*. New York, New York, USA: ACM Press, 2005, p. 54.

[11] ——, "The JDEvAn tool suite in support of object-oriented evolutionary development," in *Companion of the 13th international conference on Software engineering - ICSE Companion '08*. New York, New York, USA: ACM Press, 2008, p. 951.

[12] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated Detection of Refactorings in Evolving Components," in *Proceedings of the 20th European Conference on Object-Oriented Programming*. Springer-Verlag, 2006, pp. 404–428.

[13] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *IEEE International Conference on Software Maintenance, ICSM*. IEEE, sep 2010, pp. 1–10.

[14] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-Finder," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*. New York, New York, USA: ACM Press, 2010, p. 371.

[15] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings - International Conference on Software Engineering*. IEEE, 2009, pp. 309–319.

[16] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A Multidimensional Empirical Study on Refactoring Activity," *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, no. November, pp. 132–146, 2013.

[17] D. Silva and M. T. Valente, "RefDiff: Detecting Refactorings in Version Histories," in *IEEE International Working Conference on Mining Software Repositories*. IEEE, may 2017, pp. 269–279.

[18] D. Silva, N. Tsantalis, and M. T. Valente, "Why We Refactor? Confessions of GitHub Contributors," *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pp. 858–870, 2016.

[19] "Why We Refactor? Confessions of GitHub Contributors," visited: 2019-01-09. [Online]. Available: https://aserg-ufmg.github.io/why-we-refactor/#/