Extending Refactoring Detection to Kotlin: A Dataset and Comparative Study

Iman Hemati Moghadam
Formal Methods and Tools,
University of Twente, The Netherlands
Email: iman.hematimoghadam@utwente.nl

Mohammad Mehdi Afkhami
Parsa Kamalipour
Vali-e-Asr University of Rafsanjan, Iran

Vadim Zaytsev
Formal Methods and Tools,
University of Twente, The Netherlands
Email: vadim@grammarware.net

NB: This PDF is the authors' version of the paper. It contains a fully hyperlinked bibliography and might have other subtle differences, but the core content is guaranteed to be the same as published by IEEE in the Proceedings of SANER 2024 at https://doi.org/10.1109/SANER60148.2024.00034. You have been warned.

***Abstract*—Refactoring, as one of the best practices in software development, has been also the center of attention of much research. Particularly, a plethora of studies has been performed to understand the impact of refactorings on different dimensions of software development including software quality, program comprehension, fault-proneness, and non-functional requirements, among others. Among the employed approaches, analysing refactorings applied previously in real-world scenarios has been used by many researchers and proves to be a valuable way to delve deeper into the subject. The results of these research studies not only enhance our understanding of the advantages and potential drawbacks of refactorings but also guide us in developing more efficient automated refactoring tools based on how developers actually use refactorings in practice. However, the majority of studies in this regard have focused on refactorings applied in Java programs, and the other programming languages have received significantly less attention. The primary obstacle can be the lack of automated tool support for identifying refactorings applied in programs implemented in other languages. In reality, the lack of a comprehensive dataset of real-world applied refactorings makes it challenging for researchers to conduct comprehensive studies in programming languages other than Java.**

To mitigate this limitation, we extended a previously available refactoring detection tool, RefDetect, to be able to identify refactorings applied in Kotlin programs. We conducted an experiment on 180 commits of 9 Kotlin repositories sourced on GitHub and compared the performance of our tool with an existing Kotlin refactoring detection tool called KotlinRMiner. We found that our tool has a precision of 92% and a recall of 83%, achieving an *average* F-score of 86% which is 13% better than the one achieved by KotlinRMiner. We also provide the resulting dataset containing 1,485 true refactoring instances validated by one up to three refactoring experts publicly accessible. By releasing this initial dataset, we aim to address the existing gap in the availability of Kotlin refactoring datasets.

***Index Terms*—Refactoring, Automated refactoring mining, Kotlin, dataset**

I. Introduction

Refactoring, the process of altering a program's code without affecting its behaviour [10], is considered one of the best practices in software development. However, refactoring is not without its challenges [15]. For example, while it is demonstrated that refactoring has a positive impact on software quality for instance by minimising technical debt effects [19], it may negatively affect the quality of the software if not applied carefully. This includes, for instance, introducing new faults [8], violating security best practices [13] and even introducing new bad smells [26] which contradict the primary objective of refactoring to improve program quality [10]. The reluctance of developers to use refactoring tools and their preference to manually apply refactorings may be a probable cause for these issues [23]. In any case, a comprehensive understanding of refactorings, applied in real-world scenarios, will help to emphasise the benefits of refactorings as well as better know the issues with the current practice of refactoring and grasp the underlying reasons behind them. This knowledge may also help to effectively address the issues, for instance, by developing enhanced refactoring recommendation tools [4].

To acquire this knowledge, it is crucial to have access to comprehensive datasets of refactorings applied in real-world scenarios and across various programming paradigms and frameworks. The existing version-controlled repositories such as GitHub provide convenient methods to know about changes applied in projects [21]. However, it is still challenging to create a dataset of applied refactorings even with this valuable information [3]. To highlight some challenges in designing tools capable of mining applied refactorings, some key points are reviewed. Firstly, refactorings are usually interleaved with other non-refactoring changes [20], and this overlapping makes the refactoring mining process significantly challenging even for the state-of-the-art refactoring mining tools [12], [22], [27]. In addition, while analysing the commit messages in version-controlled repositories may facilitate this process [17], recent studies show that developers rarely document their refactoring operations [2], [16], [20]. Furthermore, the developer may use different patterns to document the applied refactorings [1], [16], which increases the complexity of the process. In addition to that, each programming paradigm has its own characteristics, and this makes it significantly difficult, if not impossible, to find a one-size-fits-all solution for identifying applied refactorings in different programming paradigms.

However, with the existence of these difficulties, impressive performance has been achieved to date in identifying refactorings applied in programs implemented in Java [12], [22], [24], [27]. Evidence of this success is the existence of a dataset containing more than two million refactorings from 11,149 real-world Java programs [4] mined with the help of the current state-of-the-art refactoring detection tool, RefactoringMiner [24]. However, Java is not the only programming language employed in the current software development practice. Therefore, it is still essential to facilitate the creation of such datasets for other programming languages.

Recently, several studies have focused on bridging this gap by developing tools capable of identifying refactorings in programs implemented by other programming languages. This includes programming languages such as Python [5], [9], Kotlin [18], Go [7], C++ [12] and JavaScript [22]. There are two particular similarities among these recent studies. Initially, the introduced approaches are all constructed on the basis of the current state-of-the-art refactoring detection tools in Java, namely RefactoringMiner [24], RefDetect [12] and RefDiff [22]. In addition to that, the introduced tools lack a comprehensive evaluation; for example, in languages like Kotlin [18], C++ [12] and JavaScript [22] and even in cases where comprehensive evaluations are conducted, as seen in Python [5], [9] and Go [7], only a minor portion of identified refactorings are manually validated by experts. It is worth noting that while manual validation of identified refactorings is necessary, especially in the absence of comparative tools, it is a time-consuming and error-prone task, as reported by the prior researchers [12], [22], [25], [27]. For instance, it took Tsantalis et al. [25] a total of 9 person-months to manually validate approximately 4,000 refactorings. However, despite its difficulties, it is a necessary step in assessing the accuracy of the proposed tool and providing an accurate dataset serves as a foundational resource for other related research activities.

Consequently, while significant progress in the field of refactoring mining has been made to date, there still exists a considerable untapped potential, especially in providing datasets including refactoring instances in diverse programming languages. To advance in this direction, we provide an extension to REFDETECT [12], a state-of-the-art refactoring detection tool in Java, in order to identify refactorings applied in programs implemented by *Kotlin*. We chose Kotlin as it is one of the most widely used programming languages, especially in the Android development community.[1] Furthermore, while Java and Kotlin share similarities, they also have differences, and in order to better understand the landscape of refactorings in these languages, a comprehensive understanding of refactorings applied in programs written in these programming languages is necessary.

To assess the effectiveness of our tool extension, we conducted a performance comparison with KOTLINRMINER [18], which was the only tool capable of detecting refactorings in Kotlin programs. It is worth mentioning that the original paper on KOTLINRMINER only evaluated the tool based on

[1]Kotlin is recommended by Google as an official language for building Android applications, and is officially supported by JetBrains. Kotlin is well-supported in major IDEs such as IntelliJ IDEA, Eclipse and Android Studio.

its execution time and its accuracy in terms of precision and recall was not evaluated [18]. In experiments, we applied both REFDETECT and KOTLINRMINER to 180 commits of 9 Kotlin repositories containing nearly 1,500 true refactoring instances, where REFDETECT demonstrated an F-score approximately 13% better than that achieved by KOTLINRMINER (86% vs. 73%). In summary, our contributions are as follows:

1) An extension to REFDETECT for the identification of refactorings applied in Kotlin programs. The implemented tool is evaluated and compared for its accuracy with KOTLINRMINER.
2) The introduction of an initial publicly available dataset [11], comprising 1,485 manually validated refactorings found in 180 commits across 9 open-source Kotlin repositories.

The remainder of this paper is organised as follows: we first provide a review of related work in Section II, and discuss our approach to automatic detection of the refactorings in Kotlin programs in Section III. We present experiment results and threats to the validity of our study in Section IV, and finally conclude with future work in Section V.

II. Related Work

Among the existing tools, RefactoringMiner, proposed by Tsantalis et al. [24], exhibited the highest precision in identifying refactorings in Java programs. RefactoringMiner's superiority arises from its unique characteristic of not relying on any similarity thresholds for matching program entities (e.g., classes, methods, etc.). Indeed, Tsantalis et al. employed a replacement technique and used some predefined heuristics to match entities without requiring user-specified thresholds [24].

Recently, researchers have adapted RefactoringMiner to identify refactorings applied in programs implemented in Python and Kotlin. For instance, Dilhara et al. [9] developed Python-adapter RefactoringMiner, which converts Python code to Java and employs RefactoringMiner to identify refactorings in the resulting Java codes. The other introduced tools, such as PyRef [5] and KotlinRMiner [18], adapted replacement and heuristics rules defined by Tsantalis et al. [24] to suit their respective languages. However, a significant challenge arises due to the unique features inherent in each language, which makes code translation challenging. In addition, an incomplete set of replacement and heuristics rules may result in a reduction in the accuracy of the tool as confirmed by Tsantalis et al. [24], and demonstrated by other researchers including Silva et al. [22] and Hemati Moghadamd et al. [12].

To overcome such limitations, researchers such as Xing and Stroulia [27], Hemati Moghadam et al. [12], and Silva et al. [22] proposed to employ a unified view from the source code applicable across various programming languages. The idea, while implemented differently in each of these approaches, is to abstract away the language-dependent programming instructions (e.g., while, if statements, etc.) and include only essential program features such as entities' signatures and their relationships with other entities (which are essential in

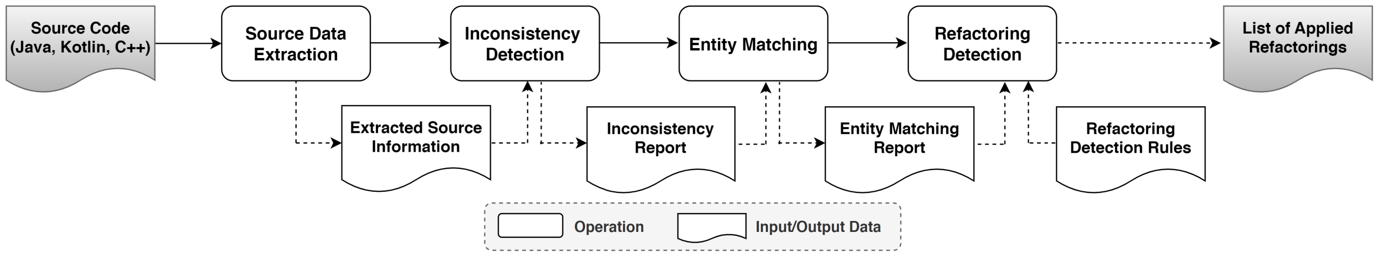

Fig. 1: RefDetect Workflow: Steps in Refactoring Detection

the process of refactoring identification) in the view. While the proposed approaches demonstrated success in identifying refactorings across different programming languages such as Java [12], [22], [27], C++ [12], Go [7], JavaScript and C [22], the employed technique has its own challenges and drawbacks. Initially, as the programming instructions (such as while, if, return, etc. included in the method body) are not taken into account, the introduced tools are not capable of detecting low-level refactorings such as *Replace Conditional with Polymorphism*, or *Replace Loop with Pipeline*, etc. In addition, the proposed approaches rely on similarity thresholds to identify refactorings, which is proven to be difficult, if not impossible, to select a universe threshold to work perfectly in programs in different domains and implemented by different programming languages as confirmed by various tools [7], [12], [22], [27].

Hopefully, with the help of existing refactoring identification tools, there exists a set of datasets containing refactorings applied in real-world scenarios across programs implemented by different programming languages. Among the existing datasets, the most noteworthy one is provided by Aniche et al. [4], who employed RefactoringMiner and identified more than two million refactorings applied in more than 11,000 open-source Java programs. It is worth mentioning that while refactorings included in this dataset lack manual validation, their reliability is ensured as the accuracy of RefactoringMiner is confirmed by numerous research studies. Recently, Dilhara et al. [9] used Python-adapter RefactoringMiner and extracted more than 8,000 code change patterns from 4 million commits in Python programs. In another study, Brito and Valente [7] extracted over 68,000 refactorings applied across six open-source Go projects. However, in this case, only 258 instances, approximately 0.4% of the identified refactorings, were manually validated [7]. Worth mentioning that the automatic creation of large refactoring datasets by the existing refactoring detection tools requires significantly less effort, but it may result in many false-positives samples where human experts are needed to validate the results. In our project, the goal is to extend refactoring detection to Kotlin and also create an initial validated dataset of refactorings applied in Kotlin programs.

III. RefDetect: Architectural Insights and Kotlin Integration

Recently, Hemati Moghadam et al. [12] introduced a language-neutral technique for identifying refactorings. The effectiveness of the introduced tool called RefDetect, was comprehensively assessed on Java programs and compared with RefactoringMiner. Additionally, the authors evaluated the tool's performance in detecting refactorings applied to C++ programs through a small dataset containing 300 refactorings.

The RefDetect's language neutrality is achieved through two key strategies. Firstly, the specific details of each programming language especially statements defined in the method's body (e.g., while, if, etc.) are abstracted away and only program entity declaration (e.g., method name and its parameters), and their relationships (e.g., method invocation, field access) are extracted and saved in an intermediate representation. Subsequently, RefDetect uses this information - the entities declaration and their relationships - to match entities in the initial and refactored versions of input programs and identifies applied refactorings according to matched entities [12].

Fig. 1 illustrates the workflow of steps taken by RefDetect to identify applied refactorings. As depicted in the figure, the workflow comprises four distinct steps, each accompanied by a concise description as follows: The initial step, *Source Data Extraction*, is the only language-dependent step and different code analysis tools must be used for different programming languages [12]. In this paper, we used the Program Structure Interface (PSI) for parsing Kotlin files and extracting the necessary data information, as required by RefDetect. It is worth mentioning that, while Java and Kotlin exhibit similarities, their source code extraction processes differ due to distinct characteristics inherent to each language. For instance, extension functions and properties [2], smart casts, type inference, and null safety are common practices in Kotlin programming [14] while these features are either non-existent or less practical for Java programmers [6].

In the second step, *Inconsistency Detection*, the two input program versions are compared with each other, and their inconsistency in terms of entities (e.g., class, field, etc.) deleted from the previous version and those introduced in the new version are determined. Entities of the same type are compared based on their signatures. For instance, methods are compared based on their names, input parameters, and return types.

The goal of the third step, *Entity Matching*, is to establish associations between inconsistent entities identified in the previous step. The entities that differ in each version are compared

[2]Using extension functions and properties, programmers can write new functions and properties for a class from a third-party library and call them in the standard manner as if they were methods or properties of the class [14].

with the entities differing in the corresponding version, and those with a similarity value higher than a *similarity threshold* are considered to be the same. The similarity threshold can be calibrated using a dataset of refactorings from programs similar to the input program or can be determined by developers based on their understanding of the input program [12].

Finally, as the last step, named *Refactoring Identification*, candidate refactorings are identified by applying some predefined rules to the entities that have been matched in the previous step. For example, PushDown/PullUp Filed refactorings are detected when a field in the initial version is matched with a field in the refactored version, with the condition that there exists an inheritance relationship between their classes. Conversely, a Move Field is detected if there is no inheritance relationship between the classes of the respective fields [12].

IV. Evaluation

To evaluate the efficacy of our extension to RefDetect, we conducted a set of experiments on various Kotlin repositories and compared the accuracy of our approach with KotlinRMiner [18]. As mentioned earlier, KotlinRMiner is the only introduced tool capable of detecting refactorings applied in Kotlin programs. Essentially, it is an extension to RefactoringMiner, with adaptations made to its replacement and heuristics rules for compatibility with Kotlin language [18]. Worth mentioning that KotlinRMiner is only evaluated for its execution time in its original paper, and this paper is the first study that evaluates its accuracy.

In the experiments, we ran both KotlinRMiner and RefDetect on 180 commits extracted from 9 popular Kotlin repositories hosted on GitHub.[3] Table I shows the list of repositories employed in this study. We selected 20 commits from each repository, and to determine the chosen commits, we initially ran KotlinRMiner on each repository and sorted their commits based on both the *quantity* and *diversity* of refactorings identified by KotlinRMiner. We then chose the first 20 commits and ran RefDetect on this subset. Indeed, to prevent any bias in favour of our tool, the commits are selected based on the results of KotlinRMiner.

RefDetect relies on similarity thresholds to match entities in two versions of the input program. To determine the similarity threshold values, we calibrated thresholds based on a dataset containing more than 3,000 refactorings applied in Java programs. These refactorings are validated by Tsantalis et al. [24] and Hemati Moghadam et al. [12]. We adopted this approach as we anticipate similarities in the refactorings applied in Java and Kotlin programs and expect this similarity to help identify more appropriate threshold values.

Our experiment aims to answer this research question: Is RefDetect effective than KotlinRMiner in accurately identifying refactorings applied in Kotlin programs, considering both correctness (precision) and completeness (recall)?

[3]The repositories were sorted by the GitHub API based on their star counts.

A. *Comparison of precision/recall with* KotlinRMiner

Table II shows a comparison of the performance of RefDetect and KotlinRMiner across 180 commits used in the experiment. In total, the employed tools identified a total of 1,485 instances of true refactorings across 21 different refactoring types. The breakdown of refactorings detected in each repository is shown in Table I. In terms of the number of identified refactorings, method-level refactorings were nearly twice and four times as numerous as those detected at the class and field levels respectively. A nearly similar pattern was observed in the study done with Java applications [12].

As depicted in Table II, RefDetect outperforms KotlinRMiner with an average overall F-score of 86% which is 13% higher than that of KotlinRMiner. While both tools show comparable accuracy in detecting class-level refactorings (87% vs. 84%), RefDetect worked better in identifying method-level refactorings (81% vs. 73%) and, notably, field-level (93% vs. 53%) refactorings. Worth mentioning that RefDetect in its original version [12] also demonstrated a better accuracy compared with RefactoringMiner in detecting class and method-level refactorings applied in Java programs. However, RefactoringMiner outperformed RefDetect in detecting field-level refactoring. Hence, this reduction in the accuracy of field-level refactoring in KotlinRMiner, especially in Rename Field, might be attributed to a bug in the implementation during the conversion of rules from RefactoringMiner to KotlinRMiner. Worth mentioning that KotlinRMiner shows a slightly better precision in field-level refactoring compared to RefDetect.

Referring to Table II, RefDetect demonstrated almost a balance between precision and recall in the majority of refactoring types. However, in specific refactoring types specifically Move and Inline Method refactoring, recall is comparatively weaker than precision. Our close inspection revealed that when the refactored entity and its relationships with other entities change significantly, especially through non-refactoring changes, RefDetect is not capable of detecting the applied refactoring. As an example, in the Inline Method refactoring when changes in the calling method exceed those associated with the inlined method, RefDetect may not detect the refactorings as the similarity between the inserted statements and those in the inlined method falls below the established threshold. Worth mentioning that while we calibrate similarity thresholds on a dataset of refactorings applied in Java programs, selecting an appropriate threshold is still a limitation in RefDetect. Worth mentioning that, all limitations mentioned by Hemati Moghadam et al. [12] for RefDetect had negative impacts on our results as well.

In addition to the inherent limitations of RefDetect, our experiments also face certain threats to validity. The first threat is the experiment scale. While our resulting Kotlin dataset contains 1,485 *manually* validated refactorings, it is roughly half of the Java refactoring datasets [12], [22], [25]. In addition, as part of future work, we aim to enhance the comprehensiveness of the dataset by including additional

TABLE I: Kotlin repositories used in the evaluation

Repository	Commits	Identified Refs.	Repository	Commits	Identified Refs.	Repository	Commits	Identified Refs.
iosched	3,139	283	BaseRecyclerView	1,481	164	okhttp	5,473	148
leakcanary	2,072	408	plaid	1,277	163	shadowsocks-android	3,694	124
picasso	1,651	82	architecture-samples	807	78	sunflower	551	35

TABLE II: Precision (P), Recall (R), and F-Score (F) results

		RefDetect			KotlinRminer		
Refactoring Type	**#TP**	**P**	**R**	**F**	**P**	**R**	**F**
1. Rename Class	87	0.99	0.8	0.89	0.92	0.93	0.93
2. Move Class	215	1	1	1	1	0.84	0.91
3. Move & Rename Class	17	0.92	0.71	0.8	0.79	0.88	0.83
4. Extract Superclass	7	0.86	0.86	0.86	1	0.71	0.83
5 Extract Subclass	4	1	1	1	1	1	1
6. Extract Class	28	0.62	0.93	0.74	0.85	0.39	0.54
7. Extract Interface	15	0.8	0.8	0.8	0.92	0.8	0.86
Class-Level Refactorings	373	0.88	0.87	0.87	0.93	0.79	0.84
8. Rename Field	40	0.95	1	0.98	1	0	0
9. Move Field	75	0.92	0.93	0.93	1	0.4	0.57
10. Push Down Field	18	1	0.89	0.94	0.91	0.56	0.69
11. Pull Up Field	23	1	0.78	0.88	1	0.74	0.85
Field-Level Refactorings	156	0.97	0.9	0.93	0.98	0.43	0.53
12. Rename Method	168	0.99	0.8	0.88	0.96	0.72	0.82
13. Move Method	124	0.86	0.88	0.87	0.77	0.85	0.8
14. Push Down Method	42	1	0.88	0.94	1	0.98	0.99
15. Pull Up Method	74	1	0.91	0.95	1	0.49	0.65
16. Extract & Move Method	8	0.83	0.62	0.71	0.75	0.38	0.5
17. Extract Method	22	0.94	0.73	0.82	0.94	0.68	0.79
28. Inline Method	11	0.8	0.73	0.76	0.53	0.91	0.67
19. Move & Inline Method	8	1	0.25	0.4	0.6	0.75	0.67
20. Change Method Parameters	491	0.9	0.85	0.87	0.9	0.57	0.7
21. Move&Change Method Parameters	8	0.8	1	0.89	1	0.5	0.67
Method-Level Refactorings	956	0.91	0.77	0.81	0.85	0.68	0.73
All Refactoring Types	1485	0.92	0.83	0.86	0.9	0.68	0.73

repositories in the experiments. Experimenter bias in the manually validating refactorings may be considered another threat to validity. To mitigate this threat, three of the authors of the paper validate refactorings, and in cases of uncertainty, discussions were held until a consensus was reached. Another factor that may affect the results is the chosen threshold values. Specifically, we might miss some refactorings due to a high threshold. We used two tools to detect applied refactorings, but we acknowledge that in our manual validation, we observed refactorings which were not detected by the employed tools.

V. Conclusion and Future Work

In this paper, we introduced an enhancement to RefDetect to identify refactorings applied in Kotlin programs. In experiments, we observed an F-score of 86%, which is 13% better than KotlinRMiner. This study is the first that presents the initial publicly available dataset [11] of nearly 1,500 manually validated refactorings applied in 180 commits of 9 Kotlin repositories. However, we are actively expanding the dataset to offer a more comprehensive resource for the community, hoping it helps to improve understanding of refactoring practices, and designing better refactoring tools.

References

[1] E. AlOmar, M. W. Mkaouer, and A. Ouni, “Can Refactoring be Self-affirmed? An Exploratory Study on How Developers Document Their Refactoring Activities in Commit Messages,” in *Proceedings of the 3rd IEEE/ACM International Workshop on Refactoring (IWoR)*. IEEE, 2019, DOI:10.1109/IWoR.2019.00017.

[2] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, “Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox,” in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE/SEIP)*. IEEE, 2021, DOI:10.1109/ICSE-SEIP52600.2021.00044.

[3] E. A. AlOmar, M. W. Mkaouer, and A. Ouni, “Mining and Managing Big Data Refactoring for Design Improvement: Are We There Yet?” *Knowledge Management in the Development of Data-Intensive Systems*, 2021, DOI:10.1201/9781003001188.

[4] M. Aniche, E. Maziero, R. Durelli, and V. H. Durelli, “The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring,” *IEEE Transactions on Software Engineering*, vol. 48, no. 4, 2020, DOI:10.1109/TSE.2020.3021736.

[5] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza, “PyRef: Refactoring Detection in Python Projects,” in *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2021, DOI:10.1109/SCAM52516.2021.00025.

[6] J. Bloch, *Effective Java*. Addison-Wesley Professional, 2017.

[7] R. Brito and M. T. Valente, “RefDiff4Go: Detecting Refactorings in Go,” in *Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, 2020, DOI:10.1145/3425269.3425274.

[8] M. Di Penta, G. Bavota, and F. Zampetti, “On the Relationship between Refactoring Actions and Bugs: A Differentiated Replication,” in *Proceedings of the 28th ACM Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, DOI:10.1145/3368089.3409695.

[9] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig, “Discovering Repetitive Code Changes in Python ML Systems,” in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2022, DOI:10.1145/3510003.3510225.

[10] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] I. Hemati Moghadam, M. M. Afkhami, P. Kamalipour, and V. Zaytsev, “Extending Refactoring Detection to Kotlin: A Dataset and Comparative Study: Supplementary Material,” DOI:10.5281/zenodo.10465264, Jan. 2024.

[12] I. Hemati Moghadam, M. Ó. Cinnéide, F. Zarepour, and M. A. Jahanmir, “RefDetect: A multi-language refactoring detection tool based on string alignment,” *IEEE Access*, vol. 9, 2021, DOI:10.1109/ACCESS.2021.3086689.

[13] E. Iannone, Z. Codabux, V. Lenarduzzi, A. De Lucia, and F. Palomba, “Rubbing Salt in the Wound? A Large-Scale Investigation into the Effects of Refactoring on Security,” *Empirical Software Engineering*, vol. 28, no. 4, 2023, DOI:10.1007/s10664-023-10287-x.

[14] D. Jemerov and S. Isakova, *Kotlin in Action*. Simon&Schuster, 2017.

[15] M. Kim, T. Zimmermann, and N. Nagappan, “A Field Study of Refactoring Challenges and Benefits,” in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, 2012, DOI:10.1145/2393596.2393655.

[16] ——, “An empirical study of refactoring challenges and benefits at microsoft,” *IEEE Transactions on Software Engineering*, vol. 40, no. 7, 2014, DOI:10.1109/TSE.2014.2318734.

[17] R. Krasniqi and J. Cleland-Huang, “Enhancing Source Code Refactoring Detection with Explanations from Commit Messages,” in *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, DOI:10.1109/SANER48275.2020.9054816.

[18] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. Andreescu, M. Anton, R. Venediktov, E. Tikhomirova, and T. Bryksin, “RefactorInsight: Enhancing IDE Representation of Changes in `git` with Refactorings Information,” in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, DOI:10.1109/ASE51524.2021.9678646.

[19] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, “Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations,” *Journal of Systems and Software*, vol. 167, p. 110610, 2020, DOI:10.1016/j.jss.2020.110610.

[20] E. Murphy-Hill, C. Parnin, and A. P. Black, “How We Refactor, and How We Know It,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, 2012, DOI:10.1109/TSE.2011.41.

[21] A. S. Nyamawe, “Research on Mining Software Repositories to Facilitate Refactoring,” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 13, no. 5, 2023, DOI:10.1002/widm.1508.

[22] D. Silva, J. Silva, G. Santos, R. Terra, and M. T. Valente, “RefDiff 2.0: A Multi-language Refactoring Detection Tool,” *IEEE Transactions on Software Engineering*, vol. 47, no. 12, 2020, DOI:10.1109/TSE.2020.2968072.

[23] D. Silva, N. Tsantalis, and M. T. Valente, “Why We Refactor? Confessions of GitHub Contributors,” in *Proceedings of the 24th ACM International Symposium on Foundations of Software Engineering*. ACM, 2016, DOI:10.1145/2950290.2950305.

[24] N. Tsantalis, A. Ketkar, and D. Dig, “RefactoringMiner 2.0,” *IEEE Transactions on Software Engineering*, vol. 48, no. 3, 2020, DOI:10.1109/TSE.2020.3007722.

[25] 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)*. ACM, 2018, DOI:10.1145/3180155.3180206.

[26] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, “When and Why Your Code Starts to Smell Bad,” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, 2017, DOI:10.1109/TSE.2017.2653105.

[27] Z. Xing and E. Stroulia, “UMLDiff: An Algorithm for Object-Oriented Design Differencing,” in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE)*. ACM, 2005, DOI:10.1145/1101908.1101919.