Can Refactorings Indicate Design Tradeoffs?

Thomas Schweizer
Université de Montréal
Montreal, Canada
thomas.schweizer@umontreal.ca

Vassilis Zafeiris
Athens University of Economics and Business
Athens, Greece
bzafiris@aueb.gr

Marios Fokaefs
Polytechnique Montréal
Montreal, Canada
marios.fokaefs@polymtl.ca

Michalis Famelis
Université de Montréal
Montreal, Canada
famelis@iro.umontreal.ca

***Abstract*—Refactoring does not always improve monotonically the quality of software. In this exploratory study, we analyze the revision history of JFreechart to see if fluctuations in internal quality metrics in commits containing refactoring can be used as indicators for the presence of design tradeoffs. We present qualitative and quantitative results suggesting that, in the context of refactoring, tradeoffs in internal quality metrics can be used to find design tradeoffs.**

***Index Terms*—Refactoring, Version History, Design Tradeoffs**

I. Introduction

Contemporary software development practices, such as Agile, place emphasis on the creation of working implementation increments, often at the expense of detailed up-front design. Non-functional requirements, including design quality, are assumed to be taken care of at a later maintenance stage. On the one hand, this allows rapid release cycles, where patches, corrections, and enhancements are applied after the release. On the other hand, this practice tends to accumulate technical debt [1], thus requiring a lot of maintenance effort in order to continue development. More generally, technical debt affects all software systems due to the common problems of design erosion [2] and design evaporation [3].

Post-release changes to non-functional aspects of a software system, namely structure and design quality, aiming to prepare it for future extensions and functional enhancements, are known in the literature as *preventive maintenance* [4]. During this phase, refactoring is a key activity [5]. It is used to introduce typically small, local changes to the code to improve non-functional requirements without affecting the application's observable behaviour. Refactoring has also been extensively studied for its impact on design quality [6], [7], as well as with respect to developer habits concerning its application on software systems [8], [9]. The resulting consensus is that refactoring definitely impacts the design of a system in a significant way. This is not an incidental relationship: developers purposefully use refactoring to express specific design intentions [9] and use recommender systems to identify the most suitable refactorings to best suit their intents [10].

However, despite this well proven relationship between refactoring and design, to the best of our knowledge there is no work that uses refactoring as an indicator for detecting the presence of design decisions. While, cases where refactoring improves the quality of software monotonically are straightforward, cases where refactoring coincides with *design tradeoffs* – decisions about design that are non-monotonic with respect to quality, i.e., improve some quality characteristic at the expense of others – are less so. Since refactoring is an activity in which developers embark intentionally, it is reasonable to conjecture that the co-occurrence of refactoring and design tradeoffs indicates a potentially pivotal point with respect to the design of a software artifact. The ability to recover such pivotal points in time is very useful. For example, developers can use this knowledge to guide their decisions during software evolution. Further, recovering such design tradeoffs can help mitigate design erosion and evaporation as source code is the most reliable and immutable information about a system.

In practical terms, we need a better understanding of how developers use refactorings not just as quick fixes, but as a tool to introduce larger scale design and architectural decisions. We conjecture that a first indicator can be the design tradeoffs that are made during refactoring. Refactorings are intended to improve software quality and should thus improve particular design quality metrics. However, this is not monotonic for all metrics; a refactoring may cause some metrics to improve, while others to deteriorate. Can such fluctuations be used to detect design tradeoffs?

In this paper, we present an exploratory study to investigate whether refactorings are an indicator of design tradeoffs. Its purpose is to estimate a refactoring revision's contribution to design quality through the use of internal quality metrics. We make the following contributions: (1) A deep analysis of revisions containing refactorings in JFreechart. (2) A classification scheme to qualify fluctuations in internal quality metrics between two revisions. (3) An automated methodology to process additional projects. The paper is organized as follows: We discuss related work in Sec. II. We present our study design in Sec.III and show quantitative results in Sec. IV and a qualitative analysis in Sec. V. We conclude in Sec. VI.

II. Related Work

Our study and our definition of tradeoffs are inspired by the work of Stroggylos et al. [7]. They presented a set of studies that have used metrics for quality evaluation and to guide design and evolution in the context of refactoring. By comparing quality metrics before and after refactorings they found that the impact of refactoring is not always positive.

The activity of refactoring and its relation to design has been extensively studied [6]. Soetens et al. analyzed the effects of refactoring on the code's complexity [11]. Bavota et al. [12]

investigated the relationship between refactoring and low maintainability, evidenced by internal metrics' values and the presence of code smells. The study revealed that refactorings are not applied to classes indicated by quality metrics, but, often, target classes affected by code smells. Cedrim et al. used RMiner [13] for the detection of refactorings in the commit history of 25 projects and showed that a significant part of refactorings do not remove code smells [14]. Kadar et al. [15] and Hegedüs et al. [16] studied the relation between metrics and maintainability in-between releases, finding a cyclic relation, where low maintainability leads to refactoring activity. Chávez et al. studied how refactoring affects quality attributes at the metric level [17]. In contrast, our study focuses specifically on the role of refactoring on internal qualities when metrics reflect a tradeoff and how it relates to design.

III. Study Design

Setup. JFreeChart is a Java project that has been studied extensively by the refactoring community [6], [18] due to its medium size (~600 classes) and history (over 10 years old). It is big enough to be relevant in quantitative analysis, while being small enough to allow manual and qualitative analysis to help guide our future studies. We selected all the revisions available on the GitHub repository of JFreeChart before 2018-05-01: 3646 revisions over 10 years of development.

To isolate the revisions containing refactorings, we used RMiner [13], a specialized tool for refactoring detection in revision histories. We ignore refactorings related to test code. RMiner yields a list of revisions with at least one refactoring. We call these revisions *refactoring revisions* (RRs).

We computed changes in internal quality metrics across all RRs using SourceMeter [19]. First, we selected an set of metrics that cover different internal quality characteristics. We selected LCOM5 to measure cohesion, WMC for method complexity, CBO for coupling, and DIT for inheritance complexity. The metrics were selected as they are considered some of the most representative for the particular properties and good indicators of design quality [20], [7]. We created a custom pipeline to generate a dataset of metric differences per class and refactoring revision by comparing the values of metrics between each RR and its parent. Our analysis focuses on the changed classes of a revision and added or deleted classes are ignored. Next we aggregated for each revision and each individual metric the metric differences across changed classes. This results in a quadruple of aggregate metric differences for each RR. To handle cases where metric changes across multiple classes cancel out each other, we count for each revision the number of times each metric has changed. We use this count in the next processing stage to remove ambiguities.

Classification. We focus on the direction of change as a trend, rather than on magnitude. To better understand such trends, we define four classification scenarios:

Scenario 1: no change in metrics. An example of this is a revision where a refactoring was found to have been applied, but no change in any of the selected metrics was found. This is the case for refactorings like renames. Based on the metrics we have selected, Scenario 1 instances are not normally expected to represent important design decisions, but rather pure functionality addition or understandability enhancements.

Scenario 2: a change in a single metric. Here we include RRs that affect a single metric, positively or negatively. Especially, in the case of positive impact, these instances could correspond to targeted changes to specifically improve the particular metric. While this may show clear intent, the intent is not necessarily related to design decisions.

Scenario 3: all metrics monotonically improving or decline. This includes RRs where more than one metric was impacted. A special inclusion condition is that all the affected metrics should have changed towards the same direction, either all positively or all negatively. Similar to Scenario 2, RRs in this scenario show clear intent. However, due to the scale of change and the impact on metrics, the intent is more inclined to be closer to a design decision.

Scenario 4: multiple metrics change in different directions. This is the same as Scenario 3 in terms of multiple metrics being affected, with the important difference that not all metrics change towards the same direction. One popular example is the metrics for cohesion and coupling, which in many cases change at the same time, but in opposite directions, especially during remodularization tasks [7]. In our view, these instances are the most interesting ones, as they indicate conflicting goals.

We call instances of Scenario 4, *design tradeoffs*. In practice, a design tradeoff is a situation where a change, i.e., a refactoring action, would result in a controversial impact to design quality; while some dimensions are improved, others may deteriorate. In this situation, the developer will have to make a decision as to which metrics and quality aspects are more important than others (given the current requirements) and eventually settle for specific tradeoffs. We thus consider Scenario 4 RRs to be closely related to design decisions.

Manual analysis. Finally, we manually analyzed each refactoring revision to identify the design intent behind applied refactorings. We based our analysis on code and comment inspection, commit messages, and the changelog of refactored classes. Specifically, we studied the developers' design intent from two perspectives: (1) The involvement of design decisions in the refactoring process, i.e., whether the developer applied the identified refactorings as part of introducing new design decisions or enforcing design decisions that were established in previous revisions. (2) The type of implementation task the developer was engaged in, while changing code structure through refactoring, i.e., whether any design decisions were enforced as part of (a) refactoring low quality code, (b) implementing new features, or (c) fixing bugs.

Threats to validity. *Construct validity:* By using only four metrics we may miss some aspects of the changes in an RR. To mitigate this, we selected metrics that are tied to well known internal quality attributes. We ignore metric fluctuations due added/deleted classes. We concentrated on tradeoffs at the class level, which might hinder drawing conclusion at the project level. Our study is exploratory; we intend to investigate these effects in the future. *Internal validity:* We

TABLE I
REFACTORING OPERATIONS IN JFREECHART

Refactoring Type	Count
Extract And Move Method	6 (2.5%)
Extract Method	80 (33.5%)
Extract Superclass	2 (0.8%)
Inline Method	4 (1.7%)
Move Class	20 (8.4%)
Move Method	6 (2.5%)
Move Source Folder	1 (0.4%)
Pull Up Attribute	12 (5.0%)
Pull Up Method	14 (5.9%)
Rename Class	15 (6.3%)
Rename Method	79 (33.0%)

depend on RMiner, which is not a perfect detector of RRs. To mitigate this, we manually removed false positives. We depend on SourceMeter and are therefore tied to its quality. We engineered our pipeline to easily incorporate newer versions of these two external tools. Manual design intent detection is subject to researcher bias. To mitigate this, the detection was done independently by two co-authors, followed by a consensus establishing step. *External validity:* We analyzed a single project, JFreeChart, and so our conclusions may not generalize. We also focused on a subset of the project's history. Our study is exploratory, with a clearly defined scope; we therefore do not claim generalizability, but rather make an existential argument that it is possible to detect design tradeoffs using refactorings as an indicator. *Empirical reliability:* We provide a replication package at https://zenodo.org/record/3995396.

IV. Quantitative Results

We automatically analyzed 3646 commits in the version history of JFreeChart with an extended version of RMiner [13]. The tool identified 247 refactoring operations in the production code that were distributed across 68 revisions. The automatically identified refactorings were manually validated and eight of them (7 cases of EXTRACT METHOD, 1 case of RENAME METHOD) were rejected as false positives. The RRs containing them did not include any true positives and were also rejected from further analysis (4 revisions). Table I presents the distribution of true positives to different refactoring types in the 64 remaining RRs. The 64 RRs were further processed in order to measure the differences of internal metrics for all changed classes.

We then automatically classified each refactoring revision to one of the four scenarios introduced in Sec. III. We show the classification in Table II. Noticeably, a large part of RRs (29.7%) do not involve changes to internal metrics (Scenario 1). Source code changes in these revisions are due to rename and move class refactoring operations. RRs with a single changed metric (Scenario 2), amount for 35.9% of total revisions. These revisions involve mainly extract method refactorings that affect the WMC metric. Revisions classified to Scenario 3 make up 25% of the total. In them, developers applied a more extensive set of refactoring operations, such as MOVE ATTRIBUTE/METHOD, EXTRACT SUPERCLASS, and MOVE CLASS. Such refactorings have a combined effect on internal metrics, either improving or deteriorating all of them.

TABLE II
REFACTORING REVISIONS FOR EACH SCENARIO

	Scenario 1	Scenario 2	Scenario 3	Scenario 4
#Revisions	19	23	16	6
Percentage	29.7%	35.9%	25.0%	9.4%

Finally, we found that in 9.4% of RRs multiple metrics are changed towards different directions. Such revisions usually involve *design tradeoffs*, i.e., improvement of a design property of one or more classes at the expense of deteriorating another. For instance, a MOVE METHOD refactoring may improve the cohesion of the origin class at the expense of increasing the coupling of the destination class.

In terms of the tasks that developers performed in RRs, we found that in 30 (46.9%) RRs, developers were engaged purely in refactoring; in 29 (45.3%) RRs they were implementing new features; and in 5 (7.8%) RRs they were fixing bugs. We determined the type of implementation task through inspection of code differences combined with analysis of commit logs, and embedded change logs of refactored classes. In several cases, commit and change logs included references to issue tracking identifiers. Revisions with a pure refactoring purpose (termed "root canal" by [21]) correspond to 46.9% of total revisions. Most of these revisions (20 out of 30) involved only renaming operations, while the rest applied EXTRACT/INLINE/MOVE METHOD refactorings. Simple refactorings (EXTRACT/MOVE METHOD) are also applied within revisions that focus on fixing bugs. The most complex and, also, interesting cases of refactorings are part of revisions that focus on new feature implementation tasks (termed "flossing" by [21]). These revisions correspond to 45.3% of the total and involve moving state and behaviour among classes, as well as, superclass extraction in class hierarchies. We discuss the most interesting of these cases that are also characterized by design tradeoffs in Sec. V.

V. Qualitative Analysis

Our manual evaluation of revisions revealed several design decisions related to the refactorings that we detected. Here, we select and explain interesting design decisions identified in RRs from Scenarios 3-4. Moreover, we discuss the effect on internal metrics of the refactorings applied in each revision. We summarize these revisions in Table III. Each revision is given a number, used below for identification; we also list under what scenario it was classified, its Git commit ID and aggregate metric differences. The first two revisions were classified in Scenario 3 and include some interesting design decisions. The remaining four revisions were classified in Scenario 4. One of these revisions, R3, involves one of the most complex refactorings in the revision history of JFreeChart.

Revision R1. Here, an EXTRACT SUPERCLASS refactoring unifies under a common parent, the `TextAnnotation` and `AbstractXYAnnotation` class hierarchies, as well as the individual class `CategoryLineAnnotation`. This way, a larger class hierarchy is formed having the extracted superclass `AbstractAnnotation` as root. The refactoring was motivated by the need to add an event notification mechanism to plot annotation classes[1]. The developers decided to add this feature to all plot annotation classes through its implementation in a common superclass (`AbstractAnnotation`). The implementation comprises appropriate state variables and methods for adding/removing listeners and firing change events. The new feature increased the DIT value of all `AbstractAnnotation` subclasses, as well as their coupling (CBO) due to invocations of inherited methods. The negative impact on WMC and LCOM5 metrics is due to extra functionality added to client classes of the new feature (e.g. `Plot`, `CategoryPlot`).

R1 shows an occurrence of a design decision that spans over multiple classes where there is no tradeoff with respect to metrics.

Revision R2. This revision involves two PULL UP METHOD refactorings from `AbstractCategoryItemRenderer` to the parent class `AbstractRenderer`. The refactorings enable reuse of functionality related to adding rendering hints to a graphics object. The functionality was introduced in a previous revision to `AbstractCategoryItemRenderer` and is reused in order to provide hinting support to all renderers. In revision R2 the methods are invoked from `AbstractXYItemRenderer` and its subclass `XYBarRenderer`. The refactorings added extra methods to `AbstractRenderer` and, thus, increased the values of WMC, LCOM5 and CBO metrics. Although metric values were improved (negative change) for `AbstractCategoryItemRenderer`, the aggregate change values for the revision are still positive due to method declarations and invocations in `AbstractXYItemRenderer` and `XYBarRenderer`.

R2, while very similar to R1, shows a that the direction of changes happening at the class granularity can be masked by the revision granularity in the same design decision.

Revision R3. This revision includes 20 refactoring operations comprising 1 EXTRACT SUPERCLASS, 1 EXTRACT METHOD, 1 RENAME METHOD, 10 PULL UP ATTRIBUTE and 8 PULL UP METHOD. The refactoring inserts an intermediate subclass (`DefaultValueAxisEditor`) between `DefaultAxisEditor`, the hierarchy root, and `DefaultNumberAxisEditor`, its direct child. The new parent of `DefaultNumberAxisEditor` absorbs a large part of its state and behaviour. The refactoring was motivated by the need to introduce a properties editing panel for the logarithmic scale numeric axis. The new panel (`DefaultLogAxisEditor`) has overlapping functionality with `DefaultNumberAxisEditor`. This functionality is reused through inheritance and `DefaultLogAxisEditor` is implemented as a subclass of `DefaultValueAxisEditor`. Moreover, the developers decided to reuse `DefaultNumberAxisEditor` functionality through a new parent class, in order to maintain the abstraction level of the hierarchy root. However, the CBO and WMC of `DefaultAxisEditor` have increased, since it, also, serves as a factory for creating instances of its subclasses. The positive impact on metrics in revision R3 (reduction of WMC, LCOM5) is dominated by the simplification of the `DefaultNumberAxisEditor` implementation due to pull up refactorings.

[1]https://sourceforge.net/p/jfreechart/patches/253/

TABLE III
METRIC FLUCTUATIONS IN INTERESTING CASES

Id	Scenario	Commit	WMC	LCOM5	CBO	DIT
R1	3	4c2a050	10	3	25	18
R2	3	74a5c5d	4	2	2	0
R3	4	1707a94	-9	-1	5	2
R4	4	202f00e	1	0	-1	0
R5	4	528da74	-2	-2	1	-1
R6	4	efd8856	12	-3	0	0

R3 shows a design decision spanning over multiple class where there is a trade-off in metrics. Also, it shows that examining metrics at revision level can masks important details happening in smaller levels. Additionally, this is an example of tangled commit where an implementation is also added for PolarPlot editor.

Revision R4. The focus of code changes in this revision is the simplification of the API that `Plot` class provides to its subclasses. The applied refactorings extract the notify listeners functionality to a new method, `fireChangeEvent()` with protected visibility. Although the implementation of the extracted method is rather simple, it replaces the notification logic in fifteen locations in the `Plot` class and in several locations in its subclasses `CategoryPlot`, `FastScatterPlot` and `XYPlot`. Moreover, it decouples `Plot` subclasses from the implementation of the change event. The refactoring increases the WMC of `Plot` due to the new method declaration and decreases the CBO of its subclasses due to the removal of references to the change event implementation (`PlotChangeEvent`). We note that due to unused imports of the `PlotChangeEvent` class in `Plot` subclasses, the SourceMeter tool does not recognize the reduction of CBO in all cases.

R4 shows a design decision affecting multiple classes where the trade-off is between two metrics only. Additionally, this is a revision where there is no granularity conflict between revision and classes. This represents a ”happy case”: the revision contains only the refactoring implementation which corresponds to a single design decision that is represented by a metric trade-off.

Revision R5. In this revision, the identified refactorings involve moving an attribute and two methods, relevant to rendering a zoom rectangle, from `ChartViewerSkin` to `ChartViewer` class. The `ChartViewerSkin` is removed from project and `ChartViewer` is turned from a UI control to a container for the layout of chart canvas and zoom rectangle components. The simplification of `ChartViewer` is responsible for the improvement of WMC, LCOM5 and CBO in revision R5. However, the CBO improvement has been counterbalanced due to another refactoring, not detected by RMiner, that implements a second design decision within the same revision. The refactoring involves move and inlining of two `ChartCanvas` methods in the `DispatchHandlerFX` class. The methods are related to dispatching of mouse events and their relocation introduces a *Feature Envy* code smell in `DispatchHandlerFX` and respective increase in the CBO metric. Nevertheless, this solution is preferred since it enforces a basic decision in the design of ChartCanvas: its behaviour related to user interaction should be dynamically extensible through registration of `AbstractMouseHandlerFX` instances.

R5 shows two design decisions affecting multiple classes resulting in a classification into Scenario 4. If only one design decision where to have been implemented, it would have been categorized as Scenario 3.

Revision R6. Finally, this revision includes a MOVE METHOD refactoring from `SWTGraphics2D` to `SWTUtils`. The refactoring enforces the decision that reusable functionality related to conversions between AWT and SWT frameworks should be located in `SWTUtils` class. The move method lowers the complexity and improves the cohesion of `SWTGraphics2D`, although its WMC value is not changed due to extra functionality added in the same revision. On the other hand, the cohesion of `SWTUtils` is slightly changed contributing, thus, to the tradeoff between WMC and LCOM5 at revision level.

R6 shows a design decision paired with a feature implementation creating an opposite change for one metric at the class granularity.

Overall, we can make some interesting observations. On one hand, the combined fluctuations of DIT and CBO within revisions (R1, R3) are evidence of structural changes potentially related to design decisions. The type and target of refactoring operations can contribute to tracing the classes affected by these decisions. On the other hand, fluctuations of WMC and LCOM5, usually indicating changes to class responsibilities, indicate design decisions when they cause tradeoffs with other metrics (R3, R4, R5). However, the impact of refactorings to fluctuations of WMC and LCOM5 can be obscured by the implementation of new features. The problem is exaggerated in tangled RRs, i.e., one that containing unrelated changes for the same commit [22] (R3, R5).

VI. CONCLUSION

The relationship between refactoring and design has been extensively studied and theorized in the literature. However, to the best of our knowledge, there has been no study that uses refactoring as an *indicator* for the presence of design tradeoffs. Assuming that changes in design are reflected in quality metrics, we investigated the conjecture that refactorings that cause non-monotonic fluctuations in metrics are evidence of developers intentionally resolving a design dilemma by making specific tradeoffs. We analyzed the revision history of JFreeChart and found that a small minority of refactoring decisions contain metric fluctuations. Qualitative analysis of revisions in this minority uncovered interesting design choices. This gives us evidence that our conjecture is in the right direction and encourages us too study the phenomenon at a larger scale. We are currently performing a larger study with more projects and taking more metrics and fluctuation patterns into account. We also intend to compare metric fluctuations to other contexts than refactoring activities (e.g., code smells, self-admitted technical debt) and to study how fluctuations evolve over time.

REFERENCES

[1] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, Nov 2012.

[2] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, no. 2, pp. 105 – 119, 2002.

[3] M. P. Robillard, "Sustainable software design," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. ACM, 2016, pp. 920–923.

[4] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of software maintenance and evolution: Research and Practice*, vol. 13, no. 1, 2001.

[5] W. F. Opdyke, "Refactoring object-oriented frameworks," 1992.

[6] J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 44–69, Jan. 2018.

[7] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*, May 2007, pp. 10–10.

[8] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '13. Riverton, NJ, USA: IBM Corp., 2013, pp. 132–146.

[9] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 858–870.

[10] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, *Recommending Refactoring Operations in Large Software Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 387–419.

[11] Q. D. Soetens and S. Demeyer, "Studying the effect of refactorings: a complexity metrics perspective," in *International Conference on the Quality of Information and Communications Technology*. IEEE, 2010.

[12] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, 2015.

[13] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *40th International Conference on Software Engineering (ICSE 2018)*. Gothenburg, Sweden: IEEE, May 27 - June 3 2018.

[14] D. Cedrim, L. Sousa, A. Garcia, and R. Gheyi, "Does refactoring improve software structural quality? a longitudinal study of 25 projects," in *Proceedings of the 30th Brazilian Symposium on Software Engineering*, ser. SBES '16. New York, NY, USA: ACM, 2016, pp. 73–82.

[15] I. Kádár, P. Hegedus, R. Ferenc, and T. Gyimóthy, "A code refactoring dataset and its assessment regarding software maintainability," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 599–603.

[16] P. Hegedüs, I. Kádár, R. Ferenc, and T. Gyimóthy, "Empirical evaluation of software maintainability based on a manually validated refactoring dataset," *Information & Software Technology*, vol. 95, 2018.

[17] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes?: A multi-project study," in *Proceedings of the 31st Brazilian Symposium on Software Engineering*, ser. SBES'17. ACM, 2017, pp. 74–83.

[18] Z. Xing and E. Stroulia, "Refactoring detection based on umldiff change-facts queries," in *Proceedings of the 13th Working Conference on Reverse Engineering*. IEEE Computer Society, 2006, pp. 263–274.

[19] Sourcemeter. [Online]. Available: https://www.sourcemeter.com/

[20] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 2004, pp. 350–359.

[21] 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, pp. 5–18, Jan 2012.

[22] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 121–130.