Exploring refactorings as indicators of design tradeoffs

Thomas Schweizer
University of Montreal
Canada
thomas.schweizer@umontreal.ca

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

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

Michalis Famelis
University of Montreal
Canada
famelis@iro.umontreal.ca

ABSTRACT

Refactoring is used to change deliberately non-functional aspects of software to facilitate future extensions and reverse design erosion. However, these changes do not always improve internal quality monotonically and thus represents tradeoffs in a design. Past studies about the relationship between refactoring and design quality have focused primarily on the identification of refactorings across releases and on analyzing their impact on the design, in terms of the presence of code smells or the fluctuation of code metrics. We are interested in the opposite direction: whether knowledge regarding refactoring activities can be used as an indicator to find design tradeoffs. In this exploratory study, we analyzed revisions containing refactorings of the open source project JFreeChart to determine whether fluctuations in internal quality metrics, can be used as indicators of 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, and an agenda for future research.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; *Software design tradeoffs*;

KEYWORDS

Software maintenance, Refactoring, Software design, Internal quality metrics, Revision history

ACM Reference Format:
Thomas Schweizer, Vassilis Zafeiris, Marios Fokaefs, and Michalis Famelis. 2019. Exploring refactorings as indicators of design tradeoffs. In *Proceedings of CASCON*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

CASCON, November 04-06, 2019, Markham, Ontario, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Refactoring is used to change non-functional aspects of software to facilitate future extensions and reverse design erosion. In development settings that focus on implementation with minimal up-front design, refactoring is interleaved with the implementation of functionality and enables continuous adaptation of design to new requirements.

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 [14], 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 [30] and design evaporation [22].

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* [7]. During this phase, *refactoring* [21] is a key activity. Refactoring 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 [3, 26], as well as with respect to developer habits concerning its application on software systems [23, 28]. Such studies have focused primarily on the identification of refactorings across releases and on analyzing their impact on the design, in terms of the presence of code smells or the fluctuation of code metrics. 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 [23] and use recommender systems to identify the most suitable refactorings to best suit their intents [5].

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. It is generally unsurprising when refactoring improves the quality of software monotonically. This is after all the intended

purpose of refactoring. Things are more interesting when refactoring coincides with *design tradeoffs*, i.e., decisions about design that are non-monotonic with respect to quality, i.e., improve some quality characteristic at the expense of others. 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. We studied the development history and refactoring activity of JFreeChart [1], a well-studied [3, 32] software project in order to answer this question and develop an empirical, tool-supported methodology. 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.
(4) A set of refined research questions for conducting future research in this domain.

The rest of the paper is organized as follows: Section 2 describes the design of the field study on JFreeChart. Section 3 presents the results of our exploratory study, while Section 4 discusses threats to validity. Section 5 introduces earlier works related to our study. Section 6 presents the lessons learned from the experience, refining them to specific research questions for future investigation. Moreover, it discusses the limitations of the study design and the steps towards a systematization of the empirical methodology. Finally, we conclude our study in Section 7.

2 EXPLORATION

2.1 Project selection

We selected the open source project JFreeChart, a Java project that has been studied extensively by the refactoring community [3, 32]. Its medium size (~600 classes) and history (over 10 years old) is ideal. It is big enough to be relevant in quantitative analysis, while being small enough to allow manual and qualitative analysis. Its size may also support relatively strong conclusions and help to guide our future studies. The project has been used by a variety of applications from different domains over the years and is still actively developed[2].

2.2 Objects

The JFreeChart project is composed of two source code repositories, two bug trackers, mailing lists, a forum, and a website:

Repositories

- https://sourceforge.net/p/jfreechart
- https://github.com/jfree/jfreechart

Bug trackers

- https://sourceforge.net/p/jfreechart/bugs
- https://github.com/jfree/jfreechart/issues

Mailing lists : https://sourceforge.net/p/jfreechart/mailman
Forum : http://www.jfree.org/forum/index.phps
Website : http://www.jfree.org/jfreechart/

This project has a particularity: The development started on SourceForge and was then imported to GitHub. However, the content of the bug tracker was not imported to GitHub at once; they gradually stopped using the one provided by SourceForge and moved gradually to the one provided by GitHub. As a result, the source code and the issue repositories are split between the two platforms.

Thus, we selected all the revisions available on the GitHub repository before 2018-05-01. This selection contains 3646 revisions covering over 10 years of development in a mature project. Each revision is characterized by its source code, comments, updates to the changelog (this artifact is edited by the developers to detail the modifications to the source code for every revision. It is stored in the same repository as the source code), and the commit message.

2.3 Refactoring detection

To isolate the revisions containing refactorings, we used RMiner [29], a specialized tool that can detect refactorings automatically in the history of a project with high recall and precision compared to other similar tools, such as RefDiff[24]. We ignore refactorings related to tests because we are not studying the role of tests in a software's design. The refactoring detection yields a list of revisions containing at least one refactoring. Henceforth, we refer to these revisions as *refactoring revisions* (RRs). Each of these RRs is also accompanied by a detailed list of the refactorings it contains.

2.4 Computing the dataset

We computed and characterized changes in internal quality metrics across all refactoring revisions of JFreeChart. The data processing pipeline is summarized in Figure 1 and has been executed on a Virtual Machine running Ubuntu 16.04, with 16 GB memory and 8 virtual cores. At first, we appropriately selected metrics with aim to cover different internal quality characteristics. For this exploratory study, 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

[1] http:/www.jfree.org/jfreechart/

[2] http://www.jfree.org/jfreechart/users.html

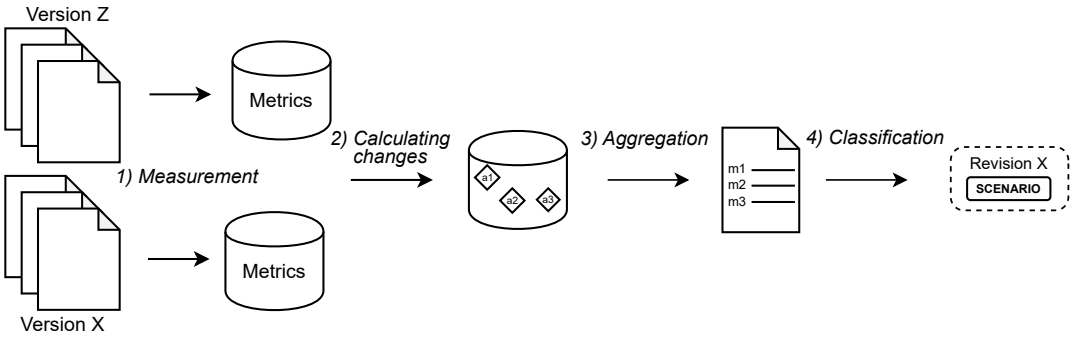

Figure 1: Illustration of the pipeline.

representative for the particular properties and good indicators of design quality [17, 26].

We used SourceMeter [1] to compute metric values for all classes and across all refactoring revisions and their parents (Stage 1, Figure 1). SourceMeter supports a wide variety of metrics and granularities (e.g., method, class, package). Since running it manually for each revision is labor intensive, we integrated it to Metric History [2], our custom pipeline for mining repositories.

Metric History reduces SourceMeter output to a dataset of metric differences per class and refactoring revision. Specifically, once all metrics are collected, Metric History computes for each refactoring revision the difference in metrics of each changed class with respect to their values in the parent revision (Stage 2, Figure 1). Notice that our analysis focuses on the changed classes of a revision, while added or deleted classes are ignored.

In a next stage, we aggregated, for each revision and each individual metric, the metric differences across changed classes (Stage 3, Figure 1). The output of this stage is a quadruple of aggregate metric differences (one value per metric) for each refactoring revision. In order 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.

Metric changes at revision level, produced at Stage 3, are further used to proxy the direction of change in internal design quality. The focus is on the direction of change that constitutes a trend, rather than on change magnitude. To better understand such trends, we defined four intuitive scenarios to classify the patterns of activity for a refactoring revision, given the metric changes (Stage 4, Figure 1). These scenarios are described in the next paragraphs and summarized in Table 1.

Scenario 1: Refactoring revisions (RRs) with no change in metrics. An example of this scenario 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: RRs with a change in a single metric. In this scenario, 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: the RRs where all metrics change monotonically towards improving or declining direction. This scenario 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 represents the RRs where multiple metrics change in different directions. Scenario 4 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 [26]. In our view, these instances are the most interesting ones, as they indicate conflicting goals.

In the context of our work, 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. This is why we consider instances in Scenario 4 to be closely related to design decisions.

2.5 Manual annotation

During this step, 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:

Scenario	Definition
1	No metric changes
2	One metric changes
3	At least two metrics change, and the changes are in the same direction (all improve or all worsen)
4	At least two metrics change, and the changes are in mixed directions (some improve and some worsen)

Table 1: Summary of the scenarios

(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.

The detection of design decisions in a refactoring revision is a rather challenging task since it requires understanding not only the changed code parts, but the overall design of affected classes. Moreover, determining whether a set of refactorings enforce a past design decision, requires tracing back to previous revisions of refactored code. A successful strategy to improve this process was to begin the analysis with the oldest refactoring and then go forward in time: This helps the reviewer to understand the evolution of the design. In order to reduce the subjectivity of this process, the evaluation was performed independently by two of the authors and it was followed by a strict conflict resolution procedure. The inter-rater agreement between their assessments was initially moderate, indicated by a value of 0.49 for Cohen's Kappa [9, 15].

3 RESULTS

3.1 Overview

We have automatically analyzed 3646 commits in the version history of JFreeChart with an extended version of RMiner [29]. 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 *refactoring revisions* containing them did not include any true positives and were also rejected from further analysis (4 revisions). Table 2 presents the distribution of true positives to different refactoring types in the 64 remaining refactoring revisions. The 64 refactoring revisions were further processed in order to measure the differences of internal metrics for all changed classes, as explained in Section 2.4. This process took a little less than 3 hours to complete.

We then automatically classified each refactoring revision to one of the four scenarios introduced in Section 2.4. We show the classification in Table 3. Noticeably, a large part of refactoring revisions (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. Refactoring revisions with a single changed metric (Scenario 2), amount for 35.9% of total revisions.

Table 2: 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%)
Total	239 (100%)

Table 3: Refactoring revisions for each Scenario

Scenario	Revisions (%)
1	19 (29.7%)
2	23 (35.9%)
3	16 (25.0%)
4	6 (9.4%)

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. Finally, we found that in 9.4% of refactoring revisions 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.

We summarize the types of implementation tasks that developers were involved in refactoring revisions in Table 4. 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 Murphy-Hill et al. [20]) 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.

Table 4: Implementation tasks and refactoring revisions

Task type	Revisions (%)
Refactoring	30 (46.9%)
Feature Implementation	29 (45.3%)
Bug Fix	5 (7.8%)

Table 5: 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

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 Murphy-Hill et al. [20]). 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 Section 3.2.

We provide a replication package at https://tinyurl.com/yyhy5mrw

3.2 Interesting Cases

Our manual evaluation of revisions revealed several design decisions related to the refactorings that we detected. In this section, we select and explain interesting design decisions identified in refactoring revisions 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 5. Each revision is given a number, which we use in the rest of the text for identification. Further, for each one, in Columns 2–7, we list under what scenario it was classified, its Git Commit ID and the 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. In this revision, 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[3]. 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`).

> Revision 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`.

> Revision 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.

> Revision R3 shows a design decision spanning over multiple class where there is a trade-off in metrics. Moreover, it shows that examining metrics at revision level can masks important details happening in smaller levels. Additionally, this is an

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

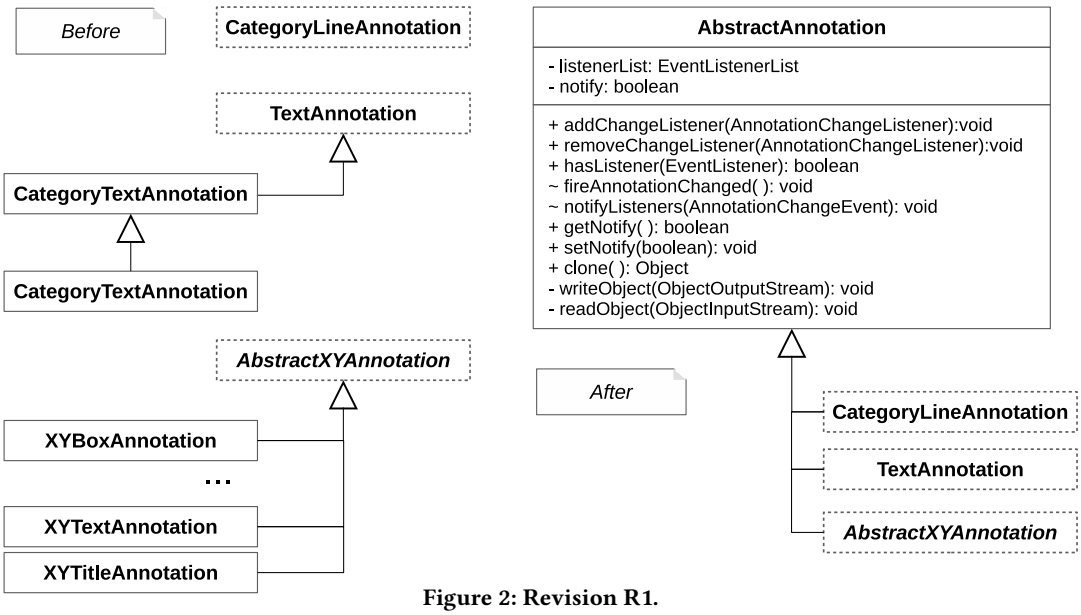

Figure 2: Revision R1.

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.

Revision 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.

Revision 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.

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

In-depth inspection of revisions R1–R6 lead us to noteworthy observations on the presence of design decisions and the hints that refactorings and metric fluctuations provide for their identification. First of all, combined fluctuations of DIT and CBO within revisions, as is the case in {R1, R3} provide 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 metrics, usually indicating changes to class responsibilities, provide a strong indication of design decisions when they cause tradeoffs with other metrics (e.g, R3–R5). However, the impact of refactorings to fluctuations of WMC and LCOM5 is often obscured by code additions that implement new features. The problem is exaggerated in refactoring revisions with tangled changes – revisions containing unrelated changes for the same commit [12], as is the case in {R3, R5}.

4 THREATS TO VALIDITY

Construct validity is threatened by multiple sources. First, by using only four metrics, we are bound to miss some aspects of the changes in a refactoring. To mitigate this, we selected metrics that have been tied to well known internal quality attributes and that are general enough to capture a maximum of their respective code aspect. Second, as mentioned in Section 2.4, we ignore the metric fluctuations generated from classes that are added or deleted. This means our pipeline does not capture all the tradeoffs between classes if a class is added or deleted in the refactoring. Third, we concentrated on tradeoffs at the class level. This granularity does not necessarily allow drawing general conclusions at the project level. Regardless, this does not impact the main contributions of this exploratory, i.e., the systematized methodology, classification scheme and research questions.

Internal validity is threatened by our refactoring detection technique. First, RMiner can produce false positives and miss refactorings. To mitigate the threat, we combed through the refactorings manually to remove false positives. Second, we depend on SourceMeter for the calculation of metrics and are therefore tied to its quality. To mitigate this in future reproductions of the study, we architected our toolchain such that new versions of the tool or entirely different tools can be easily integrated. Third, the process of design intent detection is subject to researcher bias. To mitigate this, the detection was done independently by two reviewers who used the same detection guidelines. Once the reviewers were done, a conflict resolution took place to reach a consensus, resulting in the approach described in Section 2.5.

External validity is threatened by the generalizability of the study. We analyzed a single project: JFreeChart. It is a library developed in Java maintained mostly by one person. We also focused on a subset of the project's history. However, our study is exploratory in nature, with a clearly defined scope; we, therefore, do not claim that our results are generalizable, but rather make an existential argument that it is possible to detect design tradeoffs using refactorings as an indicator.

Empirical reliability concerns the reproducibility of our study. To ensure that our findings are reproducible, we explicitly documented each step of our study, using existing, publicly available data and tools and published our custom-developed tool Metric History as open source.

5 RELATED WORK

Metrics have played an important role in evaluating the quality of a software system and in guiding its design and evolution. This role is summarized very well by Stroggylos and Spinellis [26], who present a set of research works that have used metrics for these tasks especially in the context of refactoring. The authors also present a study where, similarly to our work, they measure software quality metrics before and after refactorings for a set of object-oriented systems. According to their findings, the impact on metrics depends on the subject system, the type of refactoring and in some cases on the tool used to measure the metrics. Nevertheless, the impact is not always positive, as one would expect. This has motivated our study and definition of tradeoffs, in order to correlate metrics and refactoring activity with design intent that possibly justifies any potential deterioration in quality metrics.

Each type of refactoring may affect multiple metrics and not always in the same direction. Researchers have explored this complex impact to detect design problems known as code smells. Marinescu [17] defined thresholds on a number of metrics and then combined those using AND/OR operators in rules called detection strategies. A detection strategy could identify an instance of a design anomaly, which could orthogonally be fixed by a corresponding refactoring. A very similar approach, using metrics and thresholds, was followed by Munro [19]. Although, in principle, metric tradeoffs could be captured in detection rules, Tsantalis and Chatzigeorgiou [27] went one step further and defined a new metric to capture a tradeoff. Since, coupling and cohesion metrics can often be impacted in opposite directions during refactoring [26], Tsantalis and Chatzigeorgiou defined a new metric, *Entity Placement*, that combines coupling and cohesion. Quality assessment using Entity Placement is supposed to give more global results with respect to detection and improvement after refactoring. Kádár et al. [13] and Hegedüs et al. [11] focused exclusively on the relation between metrics and maintainability in-between releases. A cyclic relation was found, where low maintainability leads to extended refactoring activity, which in turn increases the quality of the system.

The activity of refactoring and its relation to design has been extensively studied. Chávez et al. [8] performed a large-scale study to understand how refactoring affects internal quality attributes at a metric level. In contrast, our study aims at exploring the specific role of refactoring on internal qualities when metrics embody a tradeoff and how it relates to design. Cedrim et al. [6] investigated the extent to which developers are successful at removing code smells while refactoring. Soetens and Demeyer [25] analyzed the effects of refactorings on the code's complexity. Tsantalis et al. [28] investigated refactorings across 3 projects and examined the relationship between refactoring activity, test code and release time. They found that refactoring activity is increased before release and it's mostly targeted at resolving code smells. Compared to the study presented in this paper, their study was not guided by metrics, nor did it involve extensive qualitative analysis and, finally, it did not discuss more major design decisions as part of the intent.

6 PERSPECTIVES

In Section 3 we provide a breakdown of the refactorings; we describe the repartition of refactoring operations and types. We found that the most interesting revisions belong to Scenario 3 or 4 as they involve more refactoring operations which in turn affects multiple metrics. Conjointly, we found that pure refactorings composed 46.9% of the total set with a majority of straightforward Rename

Method and a minority of Move/Extract Methods. This minority of refactoring is commonly applied in 7.8% of the bug-fix refactoring. However, the most interesting cases of refactorings, design decision wise, were in the remaining 45.3% feature implementation refactoring. Interestingly, Murphy-Hill et al. [20] found that pure refactorings represent a much less portion of the refactoring population. This difference can be explained, at least partially, by the fact that we analyzed the mature part of only one project and that our refactoring detection methods are different. To be able to address this difference of result properly, we need to analyze more projects, ideally their whole history.

In Section 3.2 we detail interesting refactoring cases and put them in relation with their scenario; we were successful at detecting some revisions containing design tradeoffs even with our basic pipeline (only four metrics and restriction at the class level). Moreover, we confirmed the presence of multiple intents in a refactoring revision [12] as well as multiple refactorings. In order to propose a systematic approach, it will be crucial to move the unit of analysis from the revision to each individual refactoring. Among other advantages, this allows us to compare and quantify them against multiple scopes (e.g. revision, package, class, method) which is unwieldy in the current framework.

Another question raised by this exploration is the following: what is a design decision? In the results, we described mostly cases where architectural changes were made. One could argue that pure rename refactorings (who constitutes the majority of Scenario 1) also defines design decisions. After all, we use natural language to communicate intent. If the name of an entity changes, so could also the developer's intent about the entity's purpose.

Concerning our pipeline, the virtual machine sped up notably the computation of the metrics fluctuations. We are confident that we can scale up to bigger projects. The task consuming the most time is the calculation of the metrics for each refactoring revision and its parent. Fortunately, this step is required only once if all the metrics are calculated at once (this is the case for SourceMeter).

The resources we selected were relevant and allowed us to detect intents with confidence, thanks, in part, to the exemplary rigour of the maintainer of the project; the global changelog and individual changelogs of each class were a great asset. Despite these qualities, as mentioned before, the biggest challenge was to separate tangled concerns in a revision because developers tend to commit many changes at once, making it difficult to grasp the individual intents. As of now, we are not aware of work successfully performing intent detection in an automated way that uses natural language processing on source code and commit messages simultaneously.

6.1 Agenda

In the light of our preliminary findings, we define three research directions in order to understand the mecanism of tradeoffs in refactoring activities. Specifically we want to investigate how design tradeoffs happen and their relation with developer. We also want to look at the effects of granularity on the detection of tradeoffs and design intents.

- **RQ1:** *How do design tradeoffs manifest themselves during refactoring?* Refactorings are intended to improve particular design quality metrics. However, this is not monotonic for all metrics; some metrics may be improved, while others deteriorate. In this part, we want to explore the existence of opposite values of some metrics for a set of refactorings and the importance of these opposite fluctuations (henceforth referred to as *tradeoffs*) in design decisions.
- **RQ2:** *Are developers aware of these tradeoffs and do they make conscious decisions based on them in design decisions?* In this part, we want to examine the intent of developers with respect to refactoring. We want to know if refactorings are performed explicitly to improve specific metrics, if the developers are aware that they may deteriorate others and if they rank alternatives for larger design decisions based on these tradeoffs.
- **RQ3:** *Do tradeoffs and design intent differ with respect to the level of granularity?* In this part of the study, we distinguish between class-level and package-level refactorings and study whether tradeoffs are significantly different and whether developer intent and decisions change for different levels.

6.2 Towards a Methodology

Our experience wit JFreeChart led us to a number of conclusions besides the correlation between metrics, refactoring and design decisions. Possibly, the most important conclusion is the need for a rigorous and systematic process for conducting similar studies, but also to support software development activities for particular projects with respect to design and maintenance decisions. In this section, we discuss a future iteration of the proposed methodology, whose steps were outlined in Section 2. In this iteration we plan to increase the degree of automation for several of the steps as described next.

Project selection. For a study about design and refactoring, several factors can be important in selecting a project. Depending on the research questions and what one is looking for, important factors may include programming language, architecture style (object-oriented or other), history length (i.e., number of revisions or commits), number of developers and contributors and others. Thankfully, most repository systems, like GitHub[4], Bitbucket[5], SourceForge[6], which contain a large number of open source software systems of great variety, already contain this metadata for each project. Therefore, we can use a repository crawler like Sourcerer [16] or develop a customizable one, where we can specify the specific criteria and fetch the repositories to be analysed.

Input Data. Another important parameter about such studies is what kind of input data we want to use and what data is available in the projects. For example, in JFreeChart, we extensively used the source code itself, code comments and commit comments. A peculiarity of the project was that especially with respect to changes and refactorings more information was conveyed from the code comments, and more particularly from the changelog, rather than from the commit comments. Additional sources could include mailing lists, instant messaging between developers, documentation, bug reports and issue trackers. It has been shown in previous work

[4]https://github.com/
[5]https://bitbucket.org/
[6]https://sourceforge.net/

that diverse resources for the same project can be automatically integrated, correlated and analyzed together to get better insights for the project [4].

Refactoring Detection. In the context of the JFreeChart study, this was possibly the most automated step of the process thanks to the RMiner tool [29]. Nevertheless, the accuracy and the completeness of the detection step is as good as the tool used. For example, RMiner can identify a comprehensive but not exhaustive list of refactorings. Another detail is what information these tools use to detect refactorings. For example, RMiner detects refactorings from the source code, while UMLDiff [31] employs design documents, namely UML diagrams, on which it can detect a set of refactorings. Such detail can dictate the data that will be acquired from the previous step. Eventually, it is possible to combine different tools by cross-checking common findings and acquiring a more comprehensive list of applied refactorings.

Metric Selection and Fluctuation Calculation. Metrics are another possible source of limitation for the study. On the one hand, the number of proposed software metrics, including design metrics, size metrics, documentation metrics, is rather extensive, and thus makes it difficult to consider all of them in a way that minimizes hidden correlations caused by high dimensionality [18]. For example, SourceMeter [1], which we used in our JFreeChart study, reports more than 50 metrics. Eventually, our selection focused on metrics relevant to design features and it was based on related literature. It is more than possible the inclusion of more metrics could reveal more cases of design tradeoffs, but it is not certain that these would be as relevant. For example, size metrics can fluctuate dramatically during the evolution of software, however, this is expected and it may not indicate anything about design decisions. Other concerns include the granularity of the metric calculation (method, class, package) and the particular implementation of each metric. Once again, given the nature of design as an activity, these decisions can affect the results, however, we do not believe there is one correct answer, but the decisions for each studied should be clearly stated and justified. Several of these issues have already been discussed by researchers [10, 17].

Revision Classification. The purpose behind the classification of revisions in scenarios was to formalize the significance of metrics as an indicator for design decisions and quantify the impact of refactoring as a design tool. In the JFreeChart study, the classification method considers a set of simple metric fluctuation patterns on the basis of two dimensions: a) number of affected metrics (zero, one, many) and the change pattern (improving, deteriorating, neutral, trade-off), as presented in Section 2. While these patterns gave us a good enough picture about the metric changes and their significance between revisions, we believe that a more rigorous and systematic approach can be achieved. More specifically, we plan to conduct a large-scale study with more and more diverse projects (inspired by Marinescu [17]) and eventually train a classifier or an artificial neural network to classify revisions based on metric fluctuations. By following this automated method, we can actually include more metrics and have the classifier assess, which ones are statistically significant for the classification process. Having a classifier and an automated method to train it implies that we can update it as new projects are being developed or additional metrics become available.

Design Intents and Cross-reference. The identification of design intent was possibly the most complicated step of our process and the one that required extensive manual effort. The reason for this is because refactorings, design decisions and generally the intent of developers is not always explicitly expressed in comments or documentation. For JFreeChart, the artifact that conveyed most information was the changelog. The changelog was javadoc comments that preceded one or more classes within a revision (usually classes that had the most significant changes) and described in natural language how the classes were changed in this revision. Conversely, commit comments did not contain much information about design intent or something more high-level other than the change itself. In order to understand the intent, we studied the commit comments, the changelog and the source code itself, as well as we went back in the project's history to understand more about the changes and the evolution of the metrics. The automation of this step would require significant effort. The first idea is to employ Natural Language Processing (NLP) to mine all textual data of the project (mails, commit comments, source code comments, changelogs etc.) and look for specific textual patterns pertaining to design and design maintenance. Next, we will use time-series clustering to find relationships between current and previous changes to track design decisions through the project's history. Finally, we will use association rules to complete the cross-reference step between the classified revisions according to metrics and the design decisions as identified before. It is not certain that manual effort will be completely eradicated even after the use of these machine learning techniques, but the goal is to minimize it as much as possible and increase the accuracy of the analysis.

7 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. This is a potentially crucial indicator for the recovery of developer intent in cases where knowledge about the intended design of a system has been lost. Based on the assumption that changes in design are reflected in quality metrics, in this paper, we have studied the conjecture that refactorings that cause non-monotonic fluctuations in metrics (improving some while deteriorating others) are evidence of developers intentionally resolving a design dilemma by making specific tradeoffs.

To study this conjecture, we extracted 64 refactoring revisions out of a pool of 3 646 commits from the JFreeChart open source project. We classified these revisions into four scenarios using software metrics and manually analyzed the revisions to understand the changes and their relation to design.

We found that refactoring revisions classified in Scenario 4 are likely to be relevant indicators for the presence of design tradeoffs. Our qualitative analysis also revealed interesting changes among revisions in Scenarios 3 and 4. Overall, we found that 29.7% of refactoring revisions do not contain changes in metrics suggesting pure functionality additions or understandability enhancements.

In 60.9% of refactoring revisions we observed monotonic improvement or deterioration in a single (35.9%) or multiple (25%) metrics confirming results from previous research that refactoring doesn't always improve the quality of software. The last 9.4% of refactoring revisions contained changes in metrics containing tradeoffs where one or multiple metrics would improve at the expense of deterioration in others. We found that this category was likely to be a good indicator of design tradeoffs.

The results and the data we collected point to multiple directions for future work. First, we aim to move towards the realization of the approaches outlined in Section 6.2, such as the automated detection of design intent in refactoring revision histories. Second, we want to study more kinds of tradeoffs, notably between design quality and other concerns, such as security, privacy and usability. We intend to combine internal quality metrics with other kinds of indicators, such as static and dynamic code analysis, performance, and simulation. This will further allow us to create an understanding of high level tradeoffs from low level artifacts. Third, we want to understand the tradeoffs happening at multiple levels of granularity. In this study, we have focused at measurement at the class level, aggregating metrics to the level of revisions. However, tradeoffs can happen at lower levels of granularity, such as individual methods, or higher levels, such as at the package or component level. Tradeoffs can also be effected longer stretches of time, requiring the study of chains of refactorings that are potentially interleaved with other activities. Finally, we want to expand our investigation to the relationship between refactoring and other activities related to design, such as the removal of code smells, self-admitted technical debt, discovery of inadvertent technical debt, and documentation decay.

REFERENCES

[1] 2014-2018. SourceMeter. Retrieved July 09, 2018 from https://www.sourcemeter.com/

[2] 2019. Metric History. Retrieved July 01, 2019 from https:github.com/thomsch/metric-history

[3] Jehad Al Dallal and Anas Abdin. 2018. Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review. *IEEE Trans. Softw. Eng.* 44, 1 (Jan. 2018), 44–69. https://doi.org/10.1109/TSE.2017.2658573

[4] Ken Bauer, Marios Fokaefs, Brendan Tansey, and Eleni Stroulia. 2009. WikiDev 2.0: discovering clusters of related team artifacts. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 174–187.

[5] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. *Recommending Refactoring Operations in Large Software Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 387–419. https://doi.org/10.1007/978-3-642-45135-5_15

[6] Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. 2016. Does Refactoring Improve Software Structural Quality? A Longitudinal Study of 25 Projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering (SBES '16)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/2973839.2973848

[7] Ned Chapin, Joanne E Hale, Khaled Md Khan, Juan F Ramil, and Wui-Gee Tan. 2001. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice* 13, 1 (2001), 3–30.

[8] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How Does Refactoring Affect Internal Quality Attributes?: A Multi-project Study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES'17)*. ACM, New York, NY, USA, 74–83. https://doi.org/10.1145/3131151.3131171

[9] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[10] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. 2000. Finding refactorings via change metrics. In *ACM SIGPLAN Notices*, Vol. 35. ACM, 166–177.

[11] Péter Hegedüs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information & Software Technology* 95 (2018), 313–327.

[12] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 121–130. http://dl.acm.org/citation.cfm?id=2487085.2487113

[13] I. Kádár, P. Hegedus, R. Ferenc, and T. Gyimóthy. 2016. 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. 599–603. https://doi.org/10.1109/SANER.2016.42

[14] P. Kruchten, R. L. Nord, and I. Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software* 29, 6 (Nov 2012), 18–21. https://doi.org/10.1109/MS.2012.167

[15] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* 33 (1977).

[16] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery* 18, 2 (2009), 300–336.

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

[18] Tim Menzies. 2019. Take Control: On the Unreasonable Effectiveness of Software Analytics. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '10)*. IEEE Press, Piscataway, NJ, USA, 265–266. https://doi.org/10.1109/ICSE-SEIP.2019.00037

[19] Matthew James Munro. 2005. Product metrics for automatic identification of" bad smell" design problems in java source-code. In *Software Metrics, 2005. 11th IEEE International Symposium*. IEEE, 15–15.

[20] E. Murphy-Hill, C. Parnin, and A. P. Black. 2012. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 5–18. https://doi.org/10.1109/TSE.2011.41

[21] William F Opdyke. 1992. Refactoring object-oriented frameworks. (1992).

[22] Martin P. Robillard. 2016. Sustainable Software Design. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 920–923. https://doi.org/10.1145/2950290.2983983

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

[24] D. Silva and M. T. Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 269–279. https://doi.org/10.1109/MSR.2017.14

[25] Quinten David Soetens and Serge Demeyer. 2010. Studying the effect of refactorings: a complexity metrics perspective. In *International Conference on the Quality of Information and Communications Technology*. IEEE, 313–318.

[26] K. Stroggylos and D. Spinellis. 2007. Refactoring–Does It Improve Software Quality?. In *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*. 10–10. https://doi.org/10.1109/WOSQ.2007.11

[27] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

[28] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A Multidimensional Empirical Study on Refactoring Activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '13)*. IBM Corp., Riverton, NJ, USA, 132–146. http://dl.acm.org/citation.cfm?id=2555523.2555539

[29] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *40th International Conference on Software Engineering (ICSE 2018)*. IEEE, Gothenburg, Sweden.

[30] Jilles van Gurp and Jan Bosch. 2002. Design erosion: problems and causes. *Journal of Systems and Software* 61, 2 (2002), 105 – 119. https://doi.org/10.1016/S0164-1212(01)00152-2

[31] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 54–65.

[32] Zhenchang Xing and Eleni Stroulia. 2006. Refactoring Detection based on UMLDiff Change-Facts Queries. In *Proceedings of the 13th Working Conference on Reverse Engineering*. IEEE Computer Society, 263–274.