Using Refactoring History To Find Design Intent

Anonymous authors

***Abstract*—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. Past studies of 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 about refactoring can be used to find design intent. To answer this question, we analyzed commits with refactorings in the revision history of 11 projects to determine whether tradeoffs, defined as patterns of fluctuation in the values of quality metrics, can be used as indicators for the presence of design intent. Specifically, we developed an empirical, tool-supported methodology, consisting of mining refactoring commits, estimating the internal quality metrics' fluctuations at revision granularity, and confirming the presence of design tradeoffs associated with refactorings, using four prototypical scenarios and manually analyzing revisions to annotate design decisions and design principles. Our findings suggest that metric tradeoffs and the application of design principles in revisions containing refactoring are promising indicators to find design intent in revision histories.**

***Index Terms*—Refactoring, Design Quality, Metrics, Mining Software Repositories**

I. Introduction

Contemporary software development practices, such as Agile, place a strong 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 addressed later, i.e., during maintenance. On the one hand, this allows flexibility and rapid development cycles, leaving patches, corrections, and enhancements to be applied post-release. On the other hand, this practice tends to accumulate technical debt [11] and leading to a deterioration in non-functional properties of the system, such as maintainability, extendability, understandability, and so on. To counter these problems, developers use *refactoring* [17], i.e., typically small, local changes that improve design quality without affecting the system's observable behavior.

Refactoring has been extensively studied for its impact on design quality [1], [22] as well as with respect to developer habits concerning its application on software systems [20], [24]. Such studies have focused primarily on the identification of refactorings and refactoring opportunities and on analyzing their impact on design, in terms of the presence of code smells [7], [23] or the fluctuation of code metrics [13]. The resulting consensus is that refactoring impacts the design of a system in a significant way. This relationship is not an incidental : developers purposefully use refactoring to express specific design intentions [20] and use recommender systems to identify the most suitable refactorings to best suit their intents [2]. Refactoring activity can carry various kinds of design intent, including but not limited to: removal of code smells, resolution of technical debt, application of design principles, and introduction of design patterns.

However, even though refactoring is used as a tool to do design post-release, the traceability between refactoring activities and design intent is rarely explicitly maintained. Given the rapidity of development cycles, this inevitably leads to design erosion [26], where assumptions held originally by designers are no longer valid, and design evaporation [18], where knowledge about design is lost. It is thus necessary to recover the design intent behind refactoring activities from the project history.

This is challenging because refactoring is also used to apply small-scale code changes that do not necessarily concern system design. In practical terms, given the version history of a project, we need a way to differentiate refactoring commits that carry design intent from the rest. We know that changes in design, positive or negative, are reflected in software metrics [12]. In this paper, we propose to use fluctuations in metrics in versions before and after refactoring commits to classify different kinds of refactoring activity. *We conjecture that when a refactoring commit affects multiple metrics, improving some while deteriorating others, we have evidence of a developer intentionally resolving a design dilemma by making specific tradeoffs.* If this conjecture holds, the search for design intent can be focused on refactoring commits that involve tradeoffs in metrics.

We present an empirical study to investigate the above-stated conjecture. The study answers the following research questions:

- RQ1: How does refactoring impact internal quality metrics?
- RQ2: Do fluctuations of design metrics in refactoring commits correlate with design intent?
- RQ3: Are fluctuations of design metrics in refactoring commits caused by design intent?

To answer them, we studied the development history and refactoring activity of 11 software projects. First, we used RMiner [25] to retrieve the commits in the revision history of each project that contained refactoring operations. Then, we calculated internal quality metrics for the versions before and after such refactoring commits using SourceMeter [5].

To answer RQ1, we studied the fluctuations in metrics for these revisions, finding that refactoring activities often lead to *tradeoffs* in metrics. To answer RQ2, we manually annotated a sample of all refactoring commits, finding that, on average, 67% of refactoring commits involving tradeoffs also carry design intent. To answer RQ3, we studied in depth all refactoring commits involving tradeoffs for two projects, finding that the design rationale of 79% of them can be attributed to conscious applications of known design principles. These observations allowed us to draw conclusions about the relationship between refactoring, tradeoffs and design intent. Notably, we find that fluctuations to a handful of internal quality metrics can be reliably used to find commits in a project's version history that carry design intent.

In short, we make the following contributions:

1) A study of the tradeoffs in internal quality metrics in the refactoring history of 11 software projects.
2) A scheme for classifying refactoring commits based on how the cause the values of quality metrics to fluctuate.
3) A deep qualitative study of the refactoring history of 2 software projects.
4) The outline of an approach for filtering a project's revision history to a set of revisions that have a high likelihood to carry design intent.

The rest of the paper is organized as follows: Section II introduces close work related to our study. Section III describes the overall design of our empirical study. In Sections IV–VI, we describe the steps taken to answer RQs 1–3 and the observed results. We discuss threats to validity in Section VII. We discuss lessons learned from some of the more interesting findings in Section VIII to refine our set of research questions for future investigation and conclude the paper in Section IX.

II. Related Work

Metrics have played an essential 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 [22], 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 the definition of tradeoffs, in order to correlate metrics and refactoring activity with design intent, which can justify a 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 code smells (design problems). Marinescu et al. [13] 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 et al. [15]. Although, in principle, tradeoffs could be captured in detection rules, Tsantalis et al. [23] 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 [22], 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. Kadar et al. [10] and Hegedus et al. [8] focused exclusively on the relationship 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. Chavez et al. [4] performed a large-scale study to understand how refactoring affects internal quality attributes on a microscopic level, or in other words on a metric basis. In contrast, our study aims at exploring the role of refactoring on a macroscopic level and how it relates to greater design decisions. Cedrim et al. [3] investigated the extent to which developers are successful at removing code smells while refactoring. Soetens et al. [21] analyzed the effects of refactorings on the code's complexity. Tsantalis et al. [24] investigated refactorings across three projects and examined the relationship between refactoring activity, test code and release time. They found that refactoring activity is increased before release and is mostly targeted at resolving code smells. Compared to the study presented in this paper, that study was not guided by metrics, it did not involve extensive qualitative analysis and did not discuss more major design decisions as part of the intent.

Recovery of design decisions has been studied from an architectural perspective. Jansen et al. proposed a methodology for recovering architectural design decisions across releases of a software system [9]. The methodology provides a systematic procedure for keeping up-to-date the architecture documentation and prescribes the steps that the software architect must follow in order to identify and document architectural design decisions across releases. A fully automated technique for the recovery of architectural design decisions has been, recently, proposed by Shahbazian et al. [19]. The technique extracts architectural changes from the version history of a project and maps them to relevant issues from the project's issue tracker. Each disconnected subgraph of the resulting graph corresponds to an architectural decision. The recovered decisions are relevant to structural changes in system components, applied across successive releases. Our method focuses on decisions affecting detailed design and concern the structure of classes and the distribution of state and behaviour among them.

Moreover, decision recovery takes place at the revision level and is guided by metrics' fluctuations and fine-grained changes due to refactorings. Besides, we employ issue tracker information for manual cross-checking of design decisions as well as commit messages and source code comments.

TABLE I: Projects analyzed in the study.

Project	Code source	Branch	Earliest commit	Commits
JFreeChart	https://github.com/jfree/jfreechart	master	2007-06-29	3 646
Retrofit	https://github.com/square/retrofit	master	2010-09-06	1 591
JUnit4	https://github.com/junit-team/junit4	master	2000-12-03	2 216
OkHttp	https://github.com/square/okhttp	master	2011-05-25	3 157
Dagger2	https://github.com/google/dagger	master	2012-06-25	2 074
RxJava	https://github.com/ReactiveX/RxJava	2.x	2012-03-18	5 314
Jena	https://github.com/apache/jena	master	2012-05-08	7 204
Ant	https://github.com/apache/ant	master	2000-01-13	14 054
jEdit	https://sourceforge.net/p/jedit/svn/HEAD/tree/jEdit/trunk/	trunk	2001-09-02	7 985
JMeter	https://github.com/apache/jmeter	trunk	1998-09-03	15 472
ArgoUML	http://argouml.tigris.org/source/browse/argouml/trunk/src/	trunk	1998-01-26	17 795

III. Overview

We define the goal, purpose, quality focus, perspective, and context of our study according to the guidelines defined by Wohlin et al. [27]. The *goal* of our study is to analyze software revisions that involve refactoring operations for the *purpose* of evaluating the relationship between refactoring, fluctuation of internal quality metrics, and design intent (cf. RQs 1–3). The *quality focus* is on the effectiveness of employing refactoring and metric fluctuation data for the identification of revisions that involve design intent and, thus, are crucial for shaping system design. Study results are interpreted from the *perspective* of researchers with interest in the area of design sustainability. The results can also be of interest to developers that need to understand the design of a project through studying important milestones in its evolution and by software architects seeking to review design decisions in committed code in order to confirm their architectural conformance. The *context* of this study comprises change and issue management repositories of a set of open source projects.

a) Project selection: Overall, we studied data from 11 popular open-source Java projects. We list them in Table I, along with the slice of their version history that we used, and links to their version management repositories. Our goal was to study a diverse set of projects. We selected JFreeChart, JUnit4, Ant, jEdit, JMeter, and ArgoUML because they have been the subject of past academic studies in the field. Additionally, we picked Retrofit, OkHttp, Dagger2, RxJava, and Jena to further diversify our analysis set. Overall, we based our selection on the projects' popularity, past use in academia as research subjects, size, number of contributors, platform, and type (e.g., library, desktop application). We also picked projects with different developers aiming to observe a variety of development styles. Each project has a website and a public repository of code under version control. Some projects also had issue trackers, mailing lists and forums. In addition, some also provide changelogs, i.e., files maintained by developers to keep track of changes happening during important revisions.

b) Data input: For each project, we used the entire commit history that was publicly available from the project's start until April 2018. The oldest project is ArgoUML starting on 1991-01-26; the youngest is RxJava starting on 2012-06-25. In total, we based our study on a set of 80 508 commits spanning 20 work-years of development history. We give more information about each project, including which branches we used, in Table I. Due to the large size of the data collected by our study, we intend to make it available upon publication via BitTorrent[1].

c) Refactoring detection: To extracted refactorings from the revision histories, we used *RMiner* [25], a tool that can detect refactorings automatically in the history of a project with high recall and precision. As RMiner only works for git we converted any SVN repositories to Git[2]. The output of RMiner is a list of revisions, each one containing at least one refactoring. We refer to these revisions as *refactoring revisions* (RRs). In our study, we ignored test code.

Using RMiner, we thus filtered the 80 508 commits across all the projects, down to 5 921 RRs, i.e., revisions containing at least one refactoring. The median percentage of refactoring revisions compared to the entire revision history is 7.35%. The project with the lowest percentage is JFreeChart with 2.19% and highest is Dagger2 with 17.36%. This gap can have many causes including the accuracy of RMiner, the complexity of refactorings used by the developers, and their development habits and guidelines.

d) Metric measurement: We used *SourceMeter* [5] to calculate the quality metrics for individual revisions. SourceMeter supports a wide variety of metrics and granularities (e.g., method, class, package). We automated the execution of SourceMeter to run in batch for all refactoring revisions of a project. Calculating the metrics for multiple revisions is a computationally intensive task, especially for large projects. We thus used a virtual machine with 16 GB memory, 8 virtual cores and 160GB hard drive running Ubuntu 16.04.

e) Design of studies: To answer RQs 1–3, we used three related study setups, based on the same set of data and tools described earlier. Specifically, to answer RQ1 (Section IV), we used all the RRs identified by RMiner for all 11 projects, ultimately classifying 5 921 RRs in different groups in function of their metrics fluctuations. To answer RQ2 (Section V), we did purposeful random sampling from each of the 11 projects, resulting in a sample of 106 RRs. To answer RQ3 (Section VI), we used all RRs of two specific projects (JFreeChart and Retrofit). To answer RQ2 and RQ3, in addition to the source

[1]http://academictorrents.com/

[2]https://www.atlassian.com/git/tutorials/migrating-overview

TABLE II: Distribution of refactoring revisions in each scenario for every project.

Project	Scenario 1	Scenario 2	Scenario 3	Scenario 4
JFreeChart	27 (33.75%)	29 (36.25%)	18 (22.50%)	6 (7.50%)
Retrofit	30 (28.30%)	20 (18.87%)	38 (35.85%)	18 (16.98%)
JUnit4	62 (25.20%)	67 (27.24%)	66 (26.83%)	51 (20.73%)
OkHttp	46 (14.74%)	81 (25.96%)	107 (34.29%)	78 (25.00%)
Dagger2	40 (11.11%)	53 (14.72%)	159 (44.17%)	108 (30.00%)
RxJava	98 (24.02%)	59 (14.46%)	132 (32.35%)	119 (29.17%)
Jena	133 (17.69%)	172 (22.87%)	262 (34.84%)	185 (24.60%)
Ant	198 (18.91%)	324 (30.95%)	316 (30.18%)	209 (19.96%)
jEdit	46 (5.31%)	222 (25.64%)	356 (41.11%)	242 (27.94%)
JMeter	169 (23.21%)	242 (33.24%)	207 (28.43%)	110 (15.11%)
ArgoUML	134 (13.19%)	239 (23.52%)	404 (39.76%)	239 (23.52%)
Total	983 (16.6%)	1508 (25.5%)	2065 (34.9%)	1365 (23.0%)

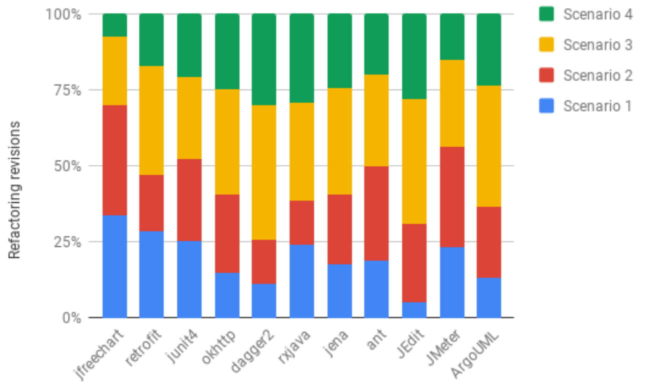

Fig. 1: Distribution of the refactoring revisions in each scenario for every project.

code and revision history, we also used the issue repository and changelogs of the projects.

IV. Refactoring and Fluctuations in Metrics

A. Setup

We aim to answer RQ1: "How does refactoring impact internal quality metrics?" We therefore built *Design Archaeologist*, a differential software measurement tool. We used SourceMeter for measurement of individual revisions but Design Archaeologist has been designed to use any third party measurement tool easily. Given a revision and its parent, Design Archaeologist identifies changes in code metrics and computes the fluctuation as the difference between the value of the metric in the two revisions. Specifically, Design Archaeologist reduces the output produced by SourceMeter to a dataset of metric differences per class and refactoring revision. We counted class-level metric differences and aggregated them per revision to produce revision-level data. For our study, we fixed the level of granularity at the class level. We only take into account the classes whose content changed between revisions or that have been renamed, ignoring classes that were deleted or added from one revision to the next. In other words, we used internal metrics to estimate a refactoring revision's contribution to design quality. To detect and characterize changes in the software's design, we measure its internal quality properties, using multiple metrics that cover different internal qualities. For this study, we selected LCOM5 to measure cohesion, WMC for method complexity, CBO for coupling, and DIT for inheritance complexity. We focus on this small set of metrics as they are considered some of the most representative for the particular properties and good indicators of design quality [13], [22].

B. Classification in Fluctuation Scenarios

The focus of our study is not to provide a quantification of the impact of refactoring to design quality, as quantified by metrics. Instead, we want to assess the overall trend across different metrics. To better understand such trends, we define four intuitive scenarios to classify the patterns of activity for a refactoring revision given the metric changes.

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 generally not expected to represent design decisions but are more likely to suggest 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 design intent, the intent is not necessarily related to design decisions.

Scenario 3: RRs where all metrics change monotonically towards improving or declining direction: This scenario includes RRs where more than one metric was impacted, and all the affected metrics changed towards the same direction, either all positively or all negatively. Similar to Scenario 2, RRs in this scenario is susceptible to contain design intents. However, due to the conjoint changes in more than one metric in a common direction, this intent is more inclined to contain design decisions than Scenario 2.

Scenario 4: RRs where multiple metrics change in different directions: This scenario also includes RRs affecting multiple metrics (like Scenario 3) but with the pivotal difference that not all metrics change towards the same direction. This case represents a *tradeoff*, i.e., a refactoring action that results in a controversial impact to design quality: while some dimensions are improved, others may deteriorate. If this effect is intentional, the developer is thus making a decision as to which metrics and quality aspects are more important and settling for specific tradeoffs. This scenario reflects our initial conjecture that developers use refactoring to implement their intent consciously in the face of quality tradeoffs. One well-understood 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 [22].

To automatically classify instances in scenarios, we associate each revision with a tuple:

$$s = (WMC, LCOM5, CBO, DIT, hit\_count)$$

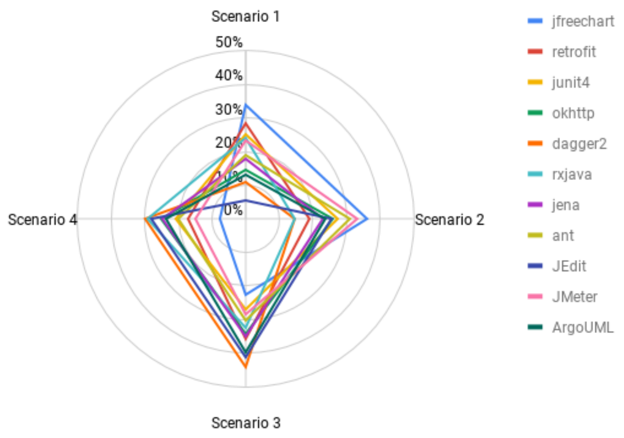

Fig. 2: Distribution of the refactoring revisions in each scenario for every project.

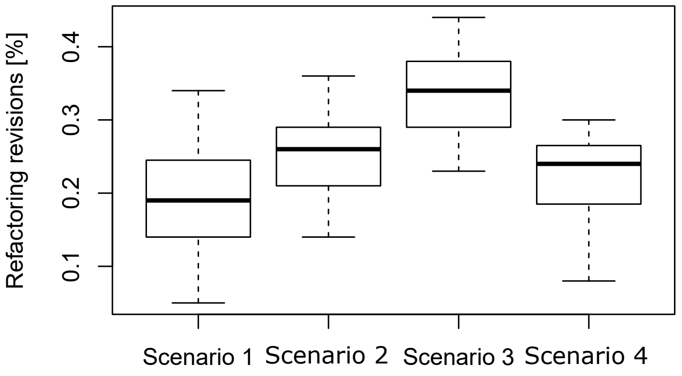

Fig. 3: Distribution of the refactoring revisions in each scenario across projects

where `WMC`, `LCOM5`, `CBO`, `DIT` are the respective sums of the metric's difference for each class that changed in the revision. The variable *hit_count* takes an integer value between 0 and 4 that counts the number of metrics affected. Given this definition, then:

- Scenario 1 contains the revisions with $s_1 = (0, 0, 0, 0, 0)$.
- Scenario 2 contains the revisions with $s_2 = (\_, \_, \_, \_, 1)$ where _ can by any value.
- Scenario 3 contains the union of revisions where changed metrics have changed in the same direction, either improving, i.e., $s_{3i} = (\leq 0, \leq 0, \leq 0, \leq 0, > 1)$ or declining, i.e., $s_{3d} = (\geq 0, \geq 0, \geq 0, \geq 0, > 1)$.
- Scenario 4 contains the revisions, where in s_4 at least one metric has changed positively (> 0), and at least another metric has changed negatively (< 0), thus, in $hit\_count > 1$.

C. Results

We show the distribution of the classification of all the RRs across all projects into scenarios in Table II and illustrate it in Figure 1. We also visualize the distribution as a radar graph in Figure 2 and show the distribution of RRs in scenarios across all projects in Figure 3 as a box plot.

First we notice that in Figure 2 the graph for all projects follows a similar "diamond" pattern with a few RRs in Scenario 1, about the same number of RRs in Scenarios 2 and 4 with the most revisions in Scenario 3. We note that JFreeChart is an outlier that does not follow the pattern. We hypothesize that the low number of Scenario 4 RRs in JFreeChart is because of the stage in which the project is in its lifecycle. Specifically, the slice of its history that is publicly available represents a mature phase in its development.

Overall, we observe that in the majority of RRs (60.4%), refactoring tends to change some quality metrics (Scenario 2 and 3) either by increasing them all or decreasing them. In about half of that, there is strong indication that the change in the metrics was on purpose, since multiple metrics changed at once. In a small percentage of RRs (16.6%), refactoring has no effect (Scenario 1). Finally, there exists a significant minority of RRs (23.0%) where refactoring causes a tradeoff in quality metrics (Scenario 4).

Based on the above we formulate the following answer to RQ1:

AQ1: Refactoring does not always monotonically improve design quality: in some cases it causes some quality attributes to improve at the expense of others.

V. Design Intent in Refactoring Revisions

A. Setup

We aim to answer RQ2: "Do fluctuations of design metrics in refactoring commits correlate with design intent?" We therefore manually analyzed a sample of 106 out of 1365 revisions consisting of 10 random RRs from scenario 4 of each of the 11 projects (for JFreeChart we analyzed the only 6 found in Section IV) to identify the design intent behind applied refactorings. In other words, we wanted to determine whether in each RR in the sample the developer applied the identified refactorings as part of introducing new design decisions or enforcing design decisions that were established in previous revisions. Figure 6 presents the distribution of refactoring types in the sampled revisions. To better understand the context in which these activities happen, we also wanted to find out the type of implementation task the developer was engaged in while 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.

We based our analysis on code and comment inspection, commit messages, and the changelog of refactored classes. Using this information, we recorded for every RR in the sample (a) whether a proposition "the revision carries design intent" holds, and (b) the task type (refactoring, feature implementation, or bug fix). To calibrate the manual analysis, two of the coauthors independently analyzed all RRs found by RMiner in JFreeChart. The inter-rater agreement between their assessments was moderate, indicated by a value of 0.490

TABLE III: Distribution of the implementation tasks in the sampled set of refactoring revisions.

Task type	Revisions	Design intent present
"Root canal" refactoring	56 (52.83%)	40 (37.74%)
"Flossing" (feat. implem.)	33 (31.13%)	24 (22.64%)
Bug fix	17 (16.04%)	7 (6.60%)
Total	106 (100%)	71 (66.98%)

for Cohen's Kappa and the percentage of observed agreements was 73.53%. Then, they did a consolidation phase, where they defined a common annotation protocol (described below). The protocol was then used by the coauthors to analyze the rest of the RRs in the sample and every RR of Retrofit.

Specifically, we defined and followed the following annotation protocol for each RR:

a) Find relevant entities: First, we identify all refactorings in the RR and the entities they affect. This information is generated by RMiner. Unless they are related to other structural changes, we ignore Rename refactorings because we are interested in the intent behind decisions affecting the structural design of the software.

b) Decide if the refactoring itself implements a design decision: Then, we decide if at least one of the refactorings in the RR is an expression of a design decision. For example, the developer may decide to delegate some functionality to a utility class using a Move Method refactoring. To determine whether a refactoring (or a set of refactorings) enforce a past design decision, we also trace back to previous revisions of the refactored code to understand the evolution of the design.

c) Decide if the refactoring is part of a design decision: In this step, we decide if at least one of the refactorings in the RR is used to implement a wider-range design decision that primarily affects another code entity (i.e., an entity not affected by a refactoring) in the same RR. For example, the developer may decide to introduce a new design pattern to regulate the communication between two classes and use Move Method as part of the implementation of this decision. To better understand the architectural role of each entity, we also first study the overall context of the project, its organization, structure, and business logic.

If in steps (b) or (c) we identify a design decision, we annotate the RR in the sample as containing a design decision.

d) Determine the task type: Finally, we record the type of implementation task that the developer was involved in when they did the refactoring. We use the terms defined by Murphy-Hill et al. [16]: "root canal" describes revisions with a pure refactoring purpose, whereas refactorings that are part of the implementation of new features are termed "flossing". A refactoring can also be part of a bug fix. We detected the task type by inspecting code differences combined, and analyzing commit logs and embedded change logs of refactored classes. In several cases, we took advantage of references to issue tracking identifiers in commit and change logs.

B. Results

We found that 71 out of 106 total RRs in our sample carry design intent. That corresponds to an average 67% RRs out of all scenario 4 sampled RRs in the projects that we studied. With a confidence of interval of 95%, the results should be taken with a margin of error of 8.6. We summarize our findings per project in Figure 4. We observe a degree of variation in the percentages for each project, but for 10 out of 11 projects, the coincidence of metric tradeoffs and refactoring indicates the presence of design intent in more than half of the cases. For RxJava the percentage is 50% and for Jena 40%, making it the only exception. Jena is so low because the sampling gave RRs with a lot of Rename refactorings that are not directly tied to design decisions.

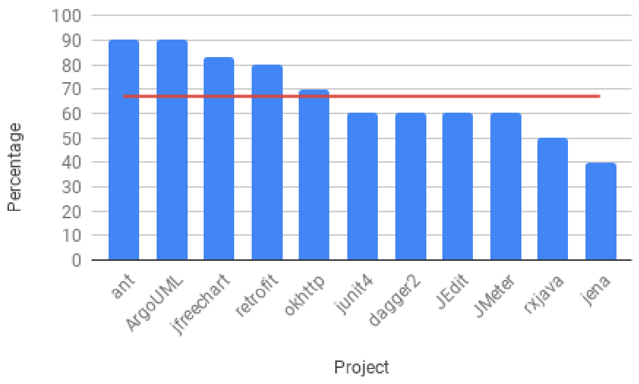

Fig. 4: Percentage of design decisions in each sample per project. The red line is the average number of detected design decisions across all samples.

We summarize the types of implementation tasks that developers were involved in refactoring revisions in Table III. "Root canal" refactoring corresponds to 52.83% of total revisions. "Flossing" refactoring corresponds to 31.13% of total revisions and involve moving state and behavior among classes, as well as superclass extraction in class hierarchies. If we only take into account the RRs containing decisions we can see that pure refactoring and feature addition contains the most design decisions while bug fixes has the lowest. Indeed, the Refactoring category has a percentage decrease of 28.57%, Feature Implementation has 27.27%, and Bug Fix has 58.82%.

We can see a concrete example in action in a commit submitted to JFreeChart on 2011-11-12 by the user `matinh`[3] and was classified as Scenario 4 by Design Archaeologist, computing the tuple s=(WMC=-9, $LCOM5$=-1, CBO=5, DIT=2, hit_count=4). This revision includes 20 refactoring operations: 1 instance of Extract Superclass, 1 of Extract Method, 1 of Rename Method, 10 of Pull Up Attribute, and 8 of Pull Up Method. The original version of the code contained the class `DefaultAxisEditor` and a subclass `DefaultNumberAxisEditor`. These classes implement panels in JFreeChart. In this revision, the developer wants to add a new panel class, `DefaultLogAxisEditor` that allows editing logarithmic axes. However, the functionality of this new class overlaps with the existing class `DefaultNumberAxisEditor`.

[3]https://goo.gl/x3iDNN

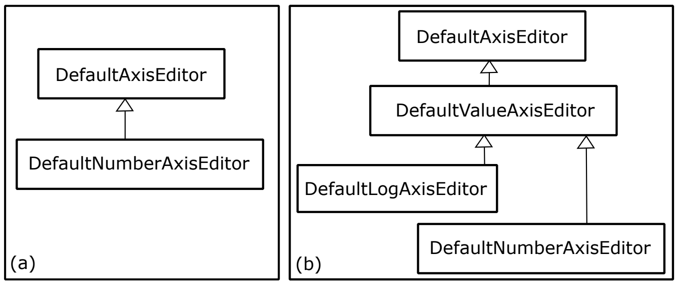

Fig. 5: (a) Slice of the JFreeChart design. (b) The same slice after the introduction of `DefaultLogAxisEditor` and the compound refactoring that created `DefaultValueAxisEditor`.

To avoid duplication, the developer refactors the class inheritance hierarchy. He inserts a new intermediate subclass, `DefaultValueAxisEditor` between `DefaultAxisEditor` and `DefaultNumberAxisEditor`. Large parts of the state and behaviour of `DefaultNumberAxisEditor` are then pulled up to the new class so it can be reused by . `DefaultLogAxisEditor` through inheritance. We illustrate the two versions in the UML Class Diagram shown in Figure 5.

At the revision level, the WMC (complexity) and LCOM5 (cohesion) metrics are improved (i.e., reduced) due to the simplification of the `DefaultNumberAxisEditor` class implementation caused by the pull up refactorings. On the other hand, the design becomes more complex, evidenced by the deterioration (i.e., increase) in the DIT (inheritance depth) and CBO (coupling) metrics. This is clearly the result of the developer's intent to incorporate the new class in the existing design.

Based on the above we formulate the following answer to RQ2:

AQ2: Fluctuations of design metrics in refactoring commits, both in the case of "root canal" and "flossing" revisions, tend to correlate with the presence of design intent.

VI. Refactoring to Apply Design Principles

A. Setup

We aim to answer RQ3: "Are fluctuations of design metrics in refactoring commits caused by design intent?" To address it, we investigate whether it is possible to attribute a refactoring to the introduction of known design principles. To study the relationship between metric fluctuations in RRs and the introduction of design principles, we manually analyzed refactoring revisions by reviewing code changes and their documentation in comments, commit messages and published issues. We narrowed our analysis to two projects: JFreeChart and Retrofit but considered all RRs in their revision histories. Specifically, we manually analyzed a total of 176 RRs: 68 from JFreeChart and 108 from Retrofit.

We manually validated the refactorings involved in these RRs to exclude false positives generated by RMiner. Specifically, we excluded from our analysis 8 refactorings from JFreeChart (7 cases of Extract Method, 1 case of Rename Method) and 2 refactorings from Retrofit (2 cases of Rename Method). The refactoring revisions containing them did not include any true positives and were also rejected from further analysis (3 and 2 revisions respectively). We show the distribution of refactoring types for both projects in Table IV only including true positives.

TABLE IV: Refactoring operations in JFreeChart and Retrofit

Refactoring Type	JFreeChart	Retrofit
Extract And Move Method	6 (2.5%)	3 (0.8%)
Extract Method	80 (33.4%)	27 (6.7%)
Extract Interface	0 (0.0%)	1 (0.3%)
Extract Superclass	2 (0.8%)	1 (0.3%)
Inline Method	4 (1.7%)	7 (1.8%)
Move Attribute	1 (0.4%)	59 (14.7%)
Move Class	20 (8.3%)	94 (23.4%)
Move Method	6 (2.5%)	39 (9.7%)
Move Source Folder	1 (0.4%)	17 (4.2%)
Pull Up Attribute	12 (5.0%)	13 (3.2%)
Pull Up Method	14 (5.8%)	15 (3.7%)
Push Down Attribute	0 (0.0%)	4 (1.0%)
Push Down Method	0 (0.0%)	2 (0.5%)
Rename Class	15 (6.3%)	45 (11.2%)
Rename Method	79 (32.9%)	71 (17.7%)
Rename Package	0 (0.0%)	3 (0.8%)
Total	240 (100%)	401 (100%)

B. Design Principles

We tagged each refactoring revision to indicate which design principles we detected. We used SOLID [14] as the reference framework for design principles. SOLID principles are meant to guide the practice of agile design by helping to identify and remove design smells that impact the flexibility, reusability and maintainability of system design. In theory, the implementation of decisions based on SOLID principles is performed with manual or automated refactorings. SOLID defines a set of well established Object Oriented Design Principles, namely: (a) Single Responsibility Principle (SRP), (b) Open/Closed Principle (OCP), (c) Liskov Substitution Principle (LSP), (d) Interface Segregation Principle (ISP) and (e) Dependency-Inversion Principle (DIP). We used one tag for each design principle from the SOLID framework.

The tagging of refactoring revisions (RRs) with SOLID principles was based on the annotation protocol specified in Section V and was performed by the same two coauthors. The consolidation phase, in this case, involved the establishment of heuristics for deciding which SOLID principle(s) influenced design changes in each refactoring revision. For instance, the redistribution of class members through Move Method, Move Attribute, Pull Up Method refactorings that improve class cohesion are typical applications of SRP. A typical indication for the application of OCP is the replacement of class dependencies on concrete implementations with abstract classes or interfaces. The introduction of Template Method and Strategy design patterns is another hint for the application of OCP. Moreover, splitting a "fat" interface to simpler ones denotes the application of ISP. Finally, since both OCP and DIP require classes to *depend on abstractions*, we

TABLE V: Application of SOLID principles in RRs of JFreeChart and Retrofit.

Project	Scenario	Revisions	SRP	OCP	LSP	ISP	DIP	None
JFreeChart	1	19	-	-	-	-	-	19 (100%)
	2	23	2 (8.7%)	-	-	-	-	21 (91.3%)
	3	17	7 (41.2%)	1 (5.9%)	-	-	-	9 (52.9%)
	4	6	4 (66.7%)	-	-	-	-	2 (33.3%)
Retrofit	1	30	-	-	-	-	-	30 (100%)
	2	19	1 (5.3%)	-	-	-	-	18 (94.7%)
	3	37	10 (27.0%)	5 (13.5%)	-	-	1 (2.7%)	22 (59.5%)
	4	18	10 (55.6%)	1 (5.6%)	-	1 (5.6%)	3 (16.7%)	3 (16.7%)

distinguished the application of DIP on the basis of "ownership inversion" of interfaces. Ownership inversion requires that an interface is packaged in the same module with the client component that uses it. On the other hand, interface implementations are packaged in external modules that depend on the client module [14]. Regarding the application of LSP, we searched for cases where the design of subclasses changes for conformance to the contract of the superclass, e.g., narrowing down their public interface to match that of their parent, fixing pre/postcondition violations in concrete overridings.

C. Results

We summarize our findings in Table V. For each project, we show the number of RRs (Column 3) for each of the four scenarios, and the number of RRs tagged with each of the SOLID principles (Columns 4–8).

These findings allow us to first notice that SOLID principles are unevenly applied to each of the two projects. In Retrofit, we found applications of four SOLID principles, while in JFreeChart, only SRP was found (with a single exception). On the one hand, we can attribute this pattern to the different slices of commit history that were studied for each project. Specifically, the commit history of Retrofit is studied from project inception, while the commit history of JFreeChart covers only the maintenance phase of its lifecycle. On the other hand, it is clearly the case that for both projects, SRP accounts for the majority of RRs tagged with the application of some design principle, regardless of which scenario they were classified.

Regarding the application of SOLID principles relative to the different scenarios of metrics fluctuations, we note some interesting patterns. First, we observe that in both projects Scenario 1 RRs have no occurrences, while Scenario 2 RRs have very few (2 in JFreeChart and 1 in Retrofit). This confirms our initial assumption that fluctuations on a single metric is not an indicator of design intent.

Second, we observe that the percentage of RRs, where we could detect no design principle, decreases substantially for Scenario 3 and more for Scenario 4. Specifically, in JFreeChart, 47.1% of Scenario 3 RRs involve application of SRP and OCP and 66.7% of Scenario 4 RRs apply SRP for changing design. A similar pattern is observed in Retrofit, where SOLID principles are applied in 43.2% and 83.5% of Scenario 3 and Scenario 4 revisions respectively. Thus, we can conclude that RRs with Scenario 3 and 4 metric fluctuation patterns are very probably cases where developers apply design amendments on the basis of SOLID principles.

Finally, we note that for both projects, Scenario 4 RRs had the lowest number of RRs for which no design principle application was detected (33.3% and 16.7% respectively). This is consistent with the findings in Section V that Scenario 4 correlates with the presence of design intent.

The example RR from JFreeChart discussed in Section V, is also a clear example of this. Specifically, the user `matinh` applied refactoring to preserve SRP: the common functionality is the concern of the new intermediate class, leaving the subclasses with only their specialized behaviour.

The above observations, lead us to conclude that in the analyzed cases the introduction of specific design principles resulted in Scenario 4 fluctuations. We thus formulate the following answer to RQ3:

AQ3: Fluctuations of design metrics in refactoring commits can be attributed to the implementation of a design intent.

VII. Threats to validity

Construct validity is threatened by three sources. First, by using only four metrics, we are bound to miss some aspects of changes in a refactoring revision. 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 IV, we ignore the metric fluctuations generated from classes that are added or deleted. This means our pipeline does not capture metric fluctuations on the entire change set of a revision, in cases of class addition or removal. Third, we concentrated on metric fluctuations at the class level. Doing that, we might have missed variations in metrics in smaller or larger granularities. To mitigate this, we count the number of metric changed when we aggregate the metrics at the class level to represent the revision as explained in Section IV. This way, we would see if a metric changed even if the changes of metric of two classes or more would cancel each other. Finally, we do not count Rename operations as design decisions. 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 change so should the intent. However, for this study we are only interested in structural design decisions.

For these reasons, we cannot draw general conclusions about what would happen at different granularity levels, with other metrics or for all type of design decisions. Regardless, this does not impact the main contributions of this study since we do not claim generalization.

Internal validity is, potentially, threatened by the off-the-self tools used in our data processing pipeline. First, RMiner can produce false positives and miss refactorings. To mitigate the threat, we combed through the refactorings manually to remove false positives for section V and section VI. Second, we depend on SourceMeter for the calculation of metrics and are, therefore, tied to its quality. This threat will be mitigated in future reproductions of the study, since we architected our toolchain such that new versions of the tool or entirely different tools can be easily integrated. Third, the process of detecting design intent and applications of SOLID principles is subject to researcher bias. To mitigate this, a calibration analysis was exercised independently by two reviewers on all the RRs of JFreeChart. Then we proceeded to resolve any conflicts and created a common annotation protocol as described in Sections V-A and VI-B that could be used by the two reviewers to analyze the other revisions.

External validity is threatened by the generalizability of the study. We analyzed 11 open source projects and focused on a subset of their version history. Thus, our study is biased by the development practices used in these projects. An argument towards the representativeness of the selected sample of refactoring revisions are the different levels of commit hygiene revealed by our manual analysis; some revisions were very well documented and worked toward a clear, unique, defined goal while some other had misleading, vague or empty descriptions with code changes affecting different concerns (tangled commit). By being confronted to this diversity confirmed that the hygiene of the commits of a project has a direct impact on the difficulty and time necessary to understand the changes happening to it in the context of version control.

Nevertheless, our study has 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 intent using refactorings and metric fluctuations as indicators.

Empirical reliability concerns the reproducibility of our study. To ensure that our findings are reproducible, we explicitly documented each step of our study, used existing, publicly available data and tools, and we will release the source code of our classification scripts and Design Archaeologist upon publication.

VIII. Discussion

Our investigation in Section IV showed that in a relatively small minority of revisions in the history of a project, we find that not all refactoring activities are monotonic in their effect on quality. In the course of refactoring, developers cause some quality attributes to improve and others to deteriorate. Our investigation in Section V also revealed that in that minority it is common to find RRs that carry design intent. Finally, our investigation in Section VI provided us with evidence that the fluctuations in quality metrics in that minority are caused precisely by the presence of design intent.

TABLE VI: Filtering revision histories to identify revisions likely to carry design intent

Project	Total revisions	RRs	Scenario 4 RRs	(%)
JFreeChart	3 646	80	6	0.16%
Retrofit	1 591	106	18	1.13%
JUnit4 4	2 216	246	51	2.30%
OkHttp	3 157	312	78	2.47%
Dagger2	2 074	360	108	5.21%
RxJava	5 314	408	119	2.24%
Jena	7 204	752	185	2.57%
Ant	14 054	1 047	209	1.49%
jEdit	7 985	866	242	3.03%
JMeter	15 472	728	110	0.71%
ArgoUML	17 795	1 016	239	1.34%
Total	80 508	5 921	1 365	1.70%

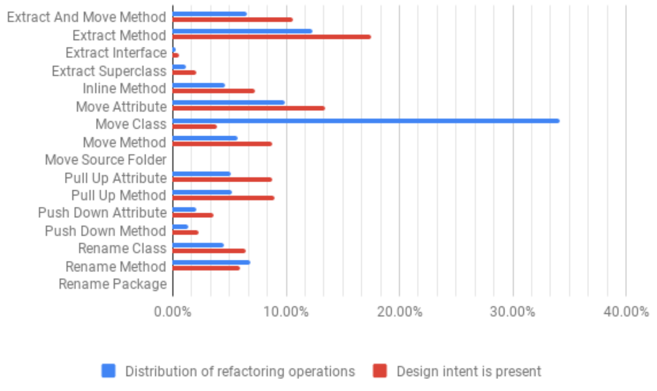

Fig. 6: Refactoring operations in the sampled revisions used to answer RQ2. The second series shows the distribution only for refactoring revisions carrying design intent.

A. *Towards the detection of design intent in revision histories*

Based on these results, we theorize that there is a close relationship between design intent, refactoring and tradeoffs in design attributes. We therefore hypothesize that it is possible to use this relationship to detect design decisions in revision histories. The tooling we created in order to perform our empirical investigation can be then be considered a proof of concept if we take classification into Scenario 4 to mean a recommendation that a particular revision contains a design decision. In this light, we can read the results of Section V as saying that the component of Design Archaeologist that does the classification of RRs into the different scenarios is a predictor of the presence of design decisions with approximately 67% precision. We can thus consider the results of Section IV as showing that a filter made up of a combination of (a) RMiner, (b) 4 metrics measured by SourceMeter, and (c) Design Archaeologist allowed us to identify among the 80 508 commits of the 11 projects, 1 365 revisions (1.7%) that have a high chance of carrying design intent. We show the degree of filtering for each of the 11 projects separately in Table VI.

Preliminary findings from our study allow us to envision various ways to improving the precision of design intent detection (see Section VIII-B why achieving high recall is secondary). For example, we found that there is a difference in the frequency of use of different kinds of refactoring operations in RRs carrying design intent compared to other RRs. As shown in Figure 6, e.g., MoveClass is rarely associated with design intent, whereas refactorings such as ExtractMethod and MoveAttribute tend to appear more frequently in RRs that carry design intent. Further investigation can reveal other such indicators that can then be used to train a machine learning based classifier.

Apart from improving the precision, we can also envision ways to only find RRs carrying design intent but to also automatically generate accurate descriptions of it. This can include automatically processing commit messages, changelogs, code comments, and posts in online discussions. We can also statistically correlate specific refactoring patterns with design intent. For example, we already have some indication from Table V that very often refactoring is used to enforce the Single Responsibility Principle (SRP). Another idea is to use time-series clustering to find relationships between current and previous changes to track design decisions through the project's history and to use association rules to cross reference them with metric fluctuations and other mined artifacts. Any advance in the generation of design intent description will also help improve the detection process by integrating it with the training of the classifier.

B. Putting automated detection in practice

A technique for the detection of design decisions in revision histories could complement existing design recovery techniques in order to combat design erosion and evaporation by using refactoring and its impact on metrics as an *indicator* for the presence of design decisions. We do not claim that RRs can reveal all or even most of the design decisions of developers. In other words, achieving a high degree of recall is not important. Instead, we propose to supplement existing techniques with an additional source of information.

Other applications could involve the creation of tools that help developers better document their contributions. For example, continuous integration tools could be augmented to offer guidance about how to make commit messages more helpful. We could provide feedback to help Agile developers make design decisions explicit at the end of a sprint by recommending what refactorings should be better explained. Other tools could automatically trigger code reviews for code commits containing refactoring that has a high likelihood to carry design intent.

From a different perspective, we could leverage the relationship between design intent, refactoring, and tradeoffs in metrics to improve existing tools, such as JDeodorant [6], that detect code smells and recommend refactoring opportunities. Given a set of stated design priorities, the tool could then propose specific refactorings such that tradeoffs between metric are resolved in an optimal way with respect to user preferences.

IX. 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 decisions. 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. Since it is known that changes in design are reflected in quality metrics, in this paper, we have studied the conjecture that refactorings that cause 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 5 921 refactoring revisions out of a pool of 80 508 commits from 11 projects. We then classified these revisions into four scenarios using software metrics. We found that refactorings do not always improve the quality of design monotonically but often embody tradeoffs. On the base of those results, we sampled refactoring revisions containing such tradeoffs from each of the 11 project and manually analyzed the resulting sample. We found that in 67% of those revisions developers also made design decisions. This suggests that the metric tradeoffs in refactoring revisions correlate with the presence of design intent. We also analyzed the complete set of refactoring revisions for JFreeChart and Retrofit to determine whether refactoring was used to introduce SOLID [14] design principles. Our qualitative analysis allowed us to conclude that fluctuation of metrics in refactorings are caused by the implementation of developers' design intent.

The results and the data we collected point to multiple directions for future work. First, we aim to move towards the realization of the approach outlined in Section VIII, i.e., the automated detection of design intent in refactoring revision histories. Second, we want to study more kinds of tradeoffs, such as between design quality and other concerns, such as security, privacy, usability, and others. We intend to include combining 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 stretch over 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 kinds of design intent, such as the removal of code smells, self-admitted technical debt and discovery of inadvertent technical debt, and documentation decay.

References

[1] Jehad Al Dallal and Anas Abdin. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Trans. Softw. Eng.*, 44(1):44–69, January 2018.

[2] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. *Recommending Refactoring Operations in Large Software Systems*, pages 387–419. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[3] Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, SBES '16, pages 73–82, New York, NY, USA, 2016. ACM.

[4] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, pages 74–83, New York, NY, USA, 2017. ACM.

[5] R. Ferenc, L. Lang, I. Siket, T. Gyimthy, and T. Bakota. Source meter sonar qube plug-in. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 77–82, Sept 2014.

[6] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039, May 2011.

[7] Yann-Gaël Guéhéneuc. Ptidej: A flexible reverse engineering tool suite. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 529–530. IEEE, 2007.

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

[9] Anton Jansen, Jan Bosch, and Paris Avgeriou. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software*, 81(4):536 – 557, 2008. Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006).

[10] 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)*, volume 1, pages 599–603, March 2016.

[11] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21, Nov 2012.

[12] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

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

[14] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

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

[16] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012.

[17] William F Opdyke. Refactoring object-oriented frameworks. 1992.

[18] Martin P. Robillard. Sustainable software design. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 920–923, New York, NY, USA, 2016. ACM.

[19] A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, and N. Medvidovic. Recovering architectural design decisions. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 95–9509, April 2018.

[20] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 858–870, New York, NY, USA, 2016. ACM.

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

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

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

[24] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 132–146, Riverton, NJ, USA, 2013. IBM Corp.

[25] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *40th International Conference on Software Engineering (ICSE 2018)*, Gothenburg, Sweden, May 27 - June 3 2018. IEEE.

[26] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105 – 119, 2002.

[27] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.