Exploring Internal Quality Metric Fluctuations in Revision Histories – An Empirical Study

***Abstract*—Background: Version control is at the core of the software development workflow. More often than not, it is enough for developers to understand changes in particular versions of a code artifact and to make sense of its development history. However, understanding longer term changes that span the project's history is a daunting task, as relevant information may only be available in scattered commit messages.**

Aims: Developers thus need a baseline to understand a project's history. In this paper, we take the first step towards facilitating the detection of change patterns in revision histories. We study the changes in measurable code characteristics between revisions by classifying fluctuations in metrics across the development history.

Method: We analyze the history of 13 open source Java projects calculating the fluctuations in internal quality metrics at the revision level.

Results: By classifying these fluctuations, we show that it is possible to identify recognizable *tradeoffs*, where some metrics may improve at the expense of others, at key development contexts (implementation, release, and refactoring).

Conclusions: Our findings suggest that metric fluctuations and quality tradeoffs can form the basis for important macroscopic insights about quality concerns in a project's development history.

***Index Terms*—Software maintenance, Internal quality metrics, Software design, Revision history**

I. Introduction

Modern software development relies heavily on version control system (VCS) to archive code artifacts and structure the development workflows. Its ubiquitousness places it as a core element of contemporary software development processes for good reasons. VCS is easy to use, requires low effort, but awards great returns for developers: it enables them to look at and understand the development history of a project, it allows them to revert changes and collaborate effectively, given that software products are often too big and complex for a sole developer and collaborative development is essential.

Developers spend a considerable amount of time trying to understand how to best maintain, add new features, or fix bugs [20]. They peruse through multiple sources of information, such as code, wikis, discussions, commit messages, and others, often having to sift through potentially out of date information [25]. Ultimately, successfully contributing to a project often requires a lot of tacit, contextual design knowledge, typically accumulated over time throughout its development history.

However, developers do not always need a comprehensive knowledge of the entire revision history of a component: they need particular information about the most relevant changes for their current task or recent changes. Unfortunately, extracting relevant and pertinent information about an artifact from its revision history is a daunting task. It definitely involves relying on commit messages and descriptions of the revisions, which – when they exist – can be irrelevant or unhelpful in describing the changes. The only other alternative is to examine the *diff* – the list of changes between revisions for each file generated automatically – but this process is confusing and slow [30].

The main objective of our work is to improve developer productivity in understanding significant changes for the various code artifacts they interact with. The main argument is that only a few important changes may be required to characterize the entire history of an artifact and help future developers understand what the priorities and requirements are when changing this artifact. Ultimately we would like to generate descriptive, history-based metadata for each code artifact that could be used by developers and other stakeholders to synthesize documentation, and to empirically assess software and software projects, and make informed decisions. In this paper, we take a first step in this direction, by focusing on changes in measurable code characteristics between revisions. Our approach is to characterize revisions across the development history with information about fluctuations in code metrics. By characterizing the metric fluctuations across the history of development, we aim to provide a baseline for understanding change and to facilitate the detection of change patterns that can be relevant as documentation to developers or other stakeholders.

A key component of our approach is the notion of *quality tradeoffs*, i.e., changes where developers consciously prioritize some internal code quality characteristics at the expense of others. In practical terms, these revisions would include both metrics that have improved and metrics that have deteriorated, due to the changes. Our argument is that they are an indicator of hot spots of design. Our underlying assumption is that when contributors make tradeoffs between quality characteristics, they are deliberately or inadvertently expressing specific *design choices* in code. In this, we are following the open source principle of considering the code as the most authoritative source of design information [13]. Our approach is based on existing techniques for measuring quality metrics [12] and for using metrics to assess quality characteristics[4]. We are inspired by work on understanding tradeoffs in refactoring [2]. We envision our work as complementary to other approaches for extracting tacit and contextual design knowledge from discussions [35], commit messages [8], etc.

In this paper, we address the following research questions:

RQ0 Do metric fluctuations exist in projects' revision histories?

RQ1 How can we characterize metric fluctuations?

RQ2 Do fluctuations in metrics depend on the development context?

To answer them, we examined the development history of 13 open source software projects.

RQ0 is the basic premise of our work. By anticipating a positive response to this question, we can further investigate what the nature of this fluctuation is and how they can characterize revisions.

Specifically, for RQ1, we devised a tool-supported methodology to mine the projects' version history for metrics, compute their changes between revisions, and aggregate them for each revision. The collected data allowed us to conduct a quantitative analysis of metric changes and identify specific fluctuation patterns.

For RQ2, we focused on quality tradeoffs, a specific class of metric fluctuations, where a revision contains both positive and negative changes in metrics. We investigated whether it was possible to detect specific quality tradeoffs in revisions that followed this category. To validate the relevance of the detected tradeoffs, we looked at the results in three different contexts. Specifically, we compared the presence of tradeoffs in testing code versus the rest of the codebase; we compared tradeoffs in revisions close to and far from release dates; we studied the correlation of tradeoffs with the presence of refactoring. In each of these contexts, we found that the presence of tradeoffs follows existing empirical results and established theories.

Our study found that quality tradeoffs coincide with particular and significant development activities and milestones. We also found that, although they carry a lot of information about an artifact, quality tradeoffs are the minority of metric fluctuation patterns in commits, which actually confirms our premise that developers need only focus on a subset of the artifact's history to understand. This is study is the first step towards improving (a) the productivity of developers, by reducing the size of the material that needs to be studied to contribute to a component, and (b) the understanding of developers about a project.

In summary, we contribute:

1) a quantitative study of the metric fluctuations in 13 open source projects;
2) a public data repository of historical internal quality metric fluctuations for open source projects;
3) a systematic methodology to mine internal quality metric fluctuations from revision histories;
4) an open source toolchain that implements it.

The remainder of this paper is organized as follows: We discuss the creation of a dataset, and an exploratory analysis to answer RQ0 and RQ1 in Sec. II. In Sec. III, we answer RQ2 by analyzing the effect of three different development contexts in metric fluctuations. We discuss threats to validity in Sec. IV and related work in Sec. V. We conclude the paper with a discussion of future work in Sec. VI.

II. Exploratory Study

In this section, we aim to answer RQ0 and RQ1. We begin by specifying our methodology, then we describe our data collection process and the steps taken to analyze the data. Finally, we discuss our results and introduce the concept of *quality tradeoffs*.

A. Design

We conduct an exploratory study structured using the guidelines established by Easterbrook et al. [11], [10]. We take a post-positivist philosophical stance that knowledge is objective, outcomes are determined by causes, and use a quantitative approach to isolate complex things into manageable sizes [11]. Our approach is based on existing theories such as internal quality metric measurement [12] and quality characteristic appreciation [4], and we follow the open source principle of considering the code as the most authoritative source of design information [13]. We use a mixed-method approach, consisting of an exploratory case-driven archive analysis, and comparative explorations. We describe the components of our exploration in the following paragraphs. We specify the scope, the population and its sampling, and the data collection techniques and their utilization.

1) Unit of analysis: Our unit of analysis is the *revision*. In version control systems, a revision represents the list of differences for the files that changed in the project from a previous – also known as parent – version to the next version. Indeed, revisions correspond to groups of incremental changes between a pair of versions, each revision represents an atomic unit of work, usually containing a cohesive set of changes although it is not always the case in practice [3]. Thus, we can think of the list of revisions of a project as its evolutionary history; each revision embodies a version of the software.

2) Target population: We aim to understand the patterns of metric fluctuations in software projects written in Java under version control. The target population of our study is therefore the set of all revisions present in the version histories of such projects. The version history contains the incremental changes applied to the project until its latest state of development as explained in the previous paragraph.

3) Sampling technique: We selected revisions from the version history of a sample of 13 popular open-source Java projects. The selection was conducted using a mix of *convenience*, *maximum variation*, and *critical* sampling based on a blend of several attributes such as projects' popularity amongst developers, usage as research subjects, size, number of contributors, platform, development style, and type (e.g., library, desktop application). Each project has a website and a public repository of source code under an arbitrary version control system – The type of the VCS was not a criterion. The projects are listed in Table I with the branch, source code location, number of commits mined and size.

B. Building a Dataset

1) Approach: We used a combination of static analysis and software repository mining as our data collection technique.

TABLE I: List of the software projects retained.

Project	Code source	Branch	Commits	Size (SLOC)
Ant	https://github.com/apache/ant	master	14 234	139k
Apache Xerces-J	https://github.com/apache/xerces2-j	trunk	5 508	142k
ArgoUML	http://argouml.tigris.org/source/browse/argouml/trunk/src/	trunk	17 797	176k
Dagger2	https://github.com/google/dagger	master	1 969	74k
Hibernate ORM	https://github.com/hibernate/hibernate-orm	master	9 320	724k
jEdit	https://sourceforge.net/p/jedit/svn/HEAD/tree/jEdit/trunk/	trunk	22 873	124k
Jena	https://github.com/apache/jena	master	7 112	515k
JFreeChart	https://github.com/jfree/jfreechart	master	3 640	132k
JMeter	https://github.com/apache/jmeter	trunk	15 898	133k
JUnit4	https://github.com/junit-team/junit4	master	1 972	30k
OkHttp	https://github.com/square/okhttp	master	1 951	61k
Retrofit	https://github.com/square/retrofit	master	1 038	20k
RxJava	https://github.com/ReactiveX/RxJava	2.x	4 137	276k

a) Download: The first step is to download each project. Since the projects are under version control, we automatically get the latest version of their *default* branch as shown in Table I. A branch represents an independent line of development [6]. We focus on the default branch because it is often this rendition of the development that will be released to the public. Moreover, by downloading a version-controlled project – also known as a *clone* or *checkout* – we automatically download its entire history of changes.

b) Static analysis: We developed the tool MetricHistory to calculate the metrics for all the source code artifacts of each version for each project. This tool provides a convenient interface to enable mining software repositories with many features while encapsulating VCS concerns. In particular, we use the *collection* feature to automatically compute the metrics for each version of the project. The metrics for a version are computed internally with SourceMeter [12], a highly configurable tool that supports a wide variety of metrics for diverse source code artifacts (e.g., method, class, package). Of all the metrics available in SourceMeter, we focus our analysis on four metrics on the interval scale: *Coupling between objects* (CBO), *Depth of inheritance* (DIT), *Lack of cohesion of methods 5* (LCOM5), and *Weighted methods per class* (WMC).

At this point, it is important to emphasize that the focus of our study is not the dependencies between these specific metrics (which has been extensively studied in related work [29], [31]). Rather, we are interested in fluctuations of quality metrics in general and aim to characterize patterns of change during the course of a project's history. We selected these metrics because they are considered some of the most representative for the particular properties (respectively coupling, inheritance complexity, cohesion, and method complexity) and good indicators of design quality [21], [29]. Therefore we can draw conclusion from this small set without loss of generality. Regardless, during the data collection phase of this study, we have gathered a greater number of metrics, which we plan to exploit in the future.

2) Data Collection: For the second step, we analyze the history contained in the *default* branch of each of the 13 projects with two additional restrictions: we only keep commits until 2018-12-31 (included) and exclude commits that are the result of a *merge* operation. In VCS, "merging" means to integrate all the changes from a "source" branch to a "destination" branch in a single commit. The resulting commit is generally hard to read for humans and presents little interest for analysis as the individual changes can be found on the source branch. We used the following command to obtain the list of revisions to be analyzed for each project (where `<branch>` is the name of the default branch):

```
git fetch --all && git log <branch>
    --pretty="%H"
    --no-merges
    --until="2018-12-31"
```

For projects that use the VCS Subversion (jEdit, ArgoUML), we first migrated the version history from Subversion to Git using GitHub Importer [14].

Overall, the 13 projects represent a cumulative $107,449$ versions of projects spanning 20 years of software development history. For each version, we saved the value of 52 metrics [12] for each Java class. The data is available privately on zenodo.org for review purposes [9] and will be made public upon publication.

Calculating the metrics for thousands of revisions is a computationally intensive task, especially for large projects (e.g. jEdit, Hibernate ORM). It can be seen as the equivalent of calculating the metrics for $107,449$ projects. To accomodate this process, we built a distributed computation system based on the Akka toolkit and runtime [1]. Each computation node invokes MetricHistory tasks to calculate the metrics for individual revisions of a project. Job assignment is performed by a scheduler node that also collects and stores the results in a data repository.

As mentioned, our unit of analysis is revisions viewed as the changes between pairs of versions. The command listed previously only covers one of the version in the pair. To be able to compute the changes we also need to run a static analysis on the parent version. A portion of the parents will already be included in our initial list but there will be some missing such as some merge versions, previously excluded as units of analysis but now required to calculate the fluctuations. To fix this, we implemented another feature in MetricHistory to recover the parent component of each revision. Then, we added the parents not present in the original list of versions to the final list.

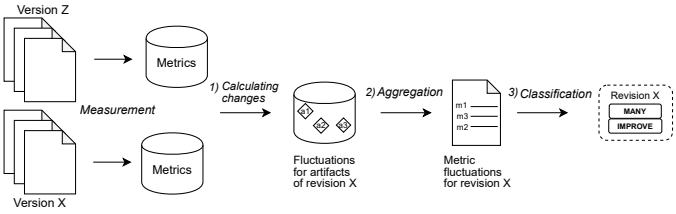


Fig. 1: Analysis procedure overview

C. Analysis Approach

We show an overview of our analysis approach in Fig. 1. (The step *Measurement* is explained in Sec. II-B1).

Let $\mathcal{V}_{all}$ be the set of all versions of a project and $\mathcal{V} \subset \mathcal{V}_{all}$ be the subset of revisions selected to be analyzed (i.e., commits to the default branch made before 2018-12-31, that were not merges). Let $\mathcal{A}_{all}$ be the set of all code artifacts in the project – in our case, Java classes. Finally, let $\mathcal{M}_{all}$ be the set of metrics supported by the static analysis and $\mathcal{M} \subset \mathcal{M}_{all}$ the set of metrics selected for analysis – in our case: DIT, CBO, WMC, and LCOM5.

We define the result of the data collection step as a collection of records x_{vam}, each representing a value resulting of the combination of the metric $m \in \mathcal{M}$ for artifact $a \in \mathcal{A}_{all}$ at version $v \in \mathcal{V}$.

In the following, we use a running example for illustration: consider a small system containing in version v_1 the classes *Dog* and *Cat* with the respective metrics [WMC: 5, CBO: 10] and [WMC: 4, CBO: 11]. For version v_2, the class *Bird* is added and measurements changed to *Dog*[WMC: 8, CBO: 3], *Cat*[WMC: 2, CBO: 1], *Bird*[WMC: 1, CBO: 7]. For example, x_{vam} with $v = v_2$, $a =$ 'Bird' and $m =$ CBO means $x_{vam} = 7$.

1) Calculating changes: The first step of this process calculates the metric fluctuations for each code artifact for each revision in V.

A metric fluctuation is calculated by the function $f(x_1, x_2) = f(x_{vam}, x_{pam}) = x_{vam} - x_{pam} = d_{vam}$ which calculates the fluctuation of metric m between the version v and p ($v \in \mathcal{V}, p \in V_{all}$, p is older than v) of artifact a. Let d_{vam} be the metric fluctuation. In our example, we calculate the fluctuation of the metric WMC in the class *Dog* for the revision v_2 with $x_1 = (v_2, \textit{Dog}, \text{WMC}), x_2 = (v_1, \textit{Dog}, \text{WMC})$ which translates to $f(x_1, x_2) = f(8, 5) = 8 - 5 = +3$ thus $d_{vam} = +3$ with $v = v_2, a = \textit{Dog}, m =$ WMC.

This operation is repeated for the metrics of every artifact that changes in the revisions of $\mathcal{V}$. We define $\mathcal{A} \subseteq \mathcal{A}_{all}$ as the set of artifacts that strictly changes in a revision and that are not test classes. We define as test code any Java class whose name has the suffix "Test". This is a common Java naming convention and is followed in all of the projects in our study. In our exemple, $\mathcal{V}_{all} = \{v_1, v_2\}$ and $\mathcal{V} = \{v_2\}$. For v_1, $\mathcal{A}$ is undefined because we have no previous parent revision. For v_2, $\mathcal{A} = \{\textit{Dog}, \textit{Cat}\}$. The class *Bird* is not included because in our analysis we do not consider created and deleted clases.

The result of this step is a matrix $D_v^{|\mathcal{A}| \times |\mathcal{M}|}$ where $|\mathcal{A}|, |\mathcal{M}|$ represent the sizes of $\mathcal{A}$ and $\mathcal{M}$, respectively. This matrix represents the metric fluctuations for each changing artifact in revision v. In our example, we compute the matrix $D_v^{2\times 2}, v = v_2$ in the previous step. It has 2 rows because $\mathcal{A} = \{\textit{Dog}, \textit{Cat}\}$ and 2 columns because we have $\mathcal{V} = \{\text{WMC}, \text{CBO}\}$. After applying $f(x_1, x_2)$ to each artifact, we obtain the (flat) matrix: $[+3, -7, -2, -10]$. Each value corresponds to an instance of d_{vam} (e.g., $a = \textit{Cat}, m =$ CBO we obtain -10). $[+3, -7]$ are respectively the fluctuation of WMC and CBO for class *Dog* and $[-2, -10]$ are the similar fluctuations for class *Cat*.

In summary, in this step, we introduce the operation $f(x_1, x_2)$ that computes the fluctuation of metric values between two versions. While our implementation of $f()$ uses real values on interval scale as input, this is not a limitation, and can be used with other scales or mathematical constructs (e.g., distributions).

2) Aggregation: In this step, we summarize to changes in the artifacts to create a general trend of fluctuation that can be understood at the revision level. Specifically, we aggregate the fluctuations of each metric across every changed artifact from $\mathcal{A}$ into one value per metric.

We define the function $g(\vec{x}) = g(\vec{d}_{vm}) = g([d_{v1m}, d_{v2m}, ..., d_{v|A|m}]) = d_{v1m} + d_{v2m} + ... + d_{v|A|m} = r_{vm}$ to aggregate the fluctuations of metric m across the artifacts $\mathcal{A}$. Thus, r_{vm} represents the trend of fluctuation of metric m for revision v. For our analysis, we choose a naive aggregation implemented as a sum. The symbol $\vec{x}$ represents a collection of values (i.e., a vector). Thus, $\vec{d}_{vm} = [d_{v1m}, d_{v2m}, ..., d_{v|A|m}]$ is the list of metric m for every artifact in $\mathcal{A}$ for version v which can be seen as a vertical slice of matrix D_v. In our example, $\vec{d}_{vm}$ with $v = v_2, m =$ WMC would be equal to $[+3, -2]$ because $+3$ is the value of WMC for class *Dog* and -2 the value for class *Cat*.

We apply this function for each metric in $\mathcal{M}$ for version v which yields us one r_{vm} per metric. This collection of aggregations is expressed as $\vec{r}_v = [r_{v1}, r_{v2}, ..., r_{v|M|}]$. Thus, $\vec{r}_v$ contains the aggregated fluctuations for the metrics of $\mathcal{M}$ for revision v. In our example, r_{v1} corresponds to the result of the aggregation for fluctuations of metric WMC in classes *Dog* and *Cat* which is equal to $r_{v1} = (+3) + (-2) = +1$. We can also calculate $r_{v2} = (-7) + (-10) = -17$ for CBO and obtain our metric fluctuations at the revision level for revision v_2 as $\vec{r}_v = [+1, -17], v = v_2$.

3) Classification: We introduce a classification taxonomy with two dimensions to characterize the changes in a revision. The aggregated results $\vec{r}_v$ are used to determine the classification of a revision. This classification was designed to reflect the macroscopic changes happening in a revision.

To develop this classification scheme, we did a pilot qualitative analysis of the revision history of JFreeChart. Two of the co-authors worked independently and then consolidated their approaches in a single schema. The consolidated schema was then further refined as the full dataset was collected.

A classification is defined as a tuple ω from the space $\Omega := (\mathcal{C}, \mathcal{D})$ where:

- $\mathcal{C} \coloneqq \{\text{ZERO}, \text{ONE}, \text{MANY}\}$ representing the cardinality of changed metrics.
- $\mathcal{D} \coloneqq \{\text{NEUTRAL}, \text{IMPROVE}, \text{DECLINE}, \text{MIXED}\}$ representing the direction of change for the changed metrics.

For the cardinality of change, we define ZERO when there is strictly no metric that changed for a revision; ONE when exactly one metric changed for a revision; and MANY when multiple metrics changed for a revision. For the direction of change, we define IMPROVE and DECLINE when all metrics go in a positive or negative direction with respect to quality respectively; NEUTRAL when the metrics of the artifacts change but balance out to zero during aggregation (e.g., $r_{vm} = g(\vec{d}_{vm}) = (+2) + (-2) = 0$); and MIXED for cases where some metrics IMPROVE and others DECLINE. The combination (ZERO, NEUTRAL) is a special case that signifies nothing changed. For example, the classification (ONE, IMPROVE) means that there was exactly one positive fluctuation of metric. Note that improvement or decline is defined with respect to quality, not value. Hence an increase in the value for, e.g., CBO, is measured as DECLINE, since increased coupling signifies quality deterioration.

We define the operation of classifying the metric fluctuations of revision v ($\vec{r}_v$) in Ω as the application of the function $c(\vec{x}) = c(\vec{r}_v) = \omega_v$. With $\omega_v \in \Omega$ being the classication for revision v. In our example, we have only have the aggregated metric fluctuation $\vec{r_v} = [+1, -17], v = v_2$ to classify. Thus, our sole classification $\omega_v, v = v_2$ is equal to $\omega_v = c(\vec{x}) = c(\vec{r_v}) = c([+1, -17]) = (\text{MANY}, \text{MIXED})$.

The classification of all of a project's revisions is $\vec{\omega} = [\omega_1, \omega_2, ..., \omega_{|V|}]$ where $|V|$ is the number of revisions in $\mathcal{V}$.

D. Results

We characterized every revision from $\mathcal{V}$ using the classification scheme proposed above. We present the results for each project in Fig. 2 and overall (average and cumulative) in Fig. 3 . The results are shown as heatmap tables, where the rows correspond to $\mathcal{C}$ and the columns to $\mathcal{D}$. Darker shades indicate higher numbers, whereas thatched lines indicate invalid combinations.

Comparing the per-project and overall heatmaps, we observe that our distribution of categories is relatively uniform with no major variations between projects for each category. Indeed, the heatmaps in Fig. 3a and Fig. 3b are the same. Individual project heatmaps in Fig. 2 follow the same pattern. Based on these observations we formulate the following answer for RQ0: *For the studied metrics, there exist observable metric fluctuations between versions of a project.* We observe some recurring tendencies:

First, we note that (ZERO, NEUTRAL) always holds the majority of revisions and represents 64.56% of occurrences on average. This means that, most of the time the four metrics we analyzed are not touched at all by the changes made in existing classes.

Second, we observe that the case where metric fluctuations cancel each other out in a revision represents only a small fraction of the overall observations. In this case, artifact metrics fluctuate but the aggregation process masks the effect (i.e., the net metric fluctuation for a revision sums up to 0). This case is represented by the revisions that were classified in the cells $\{\text{ONE}, \text{MANY}\} \times \{\text{NEUTRAL}\}$.

Third, we note that the group of four cells $\{\text{ONE}, \text{MANY}\} \times \{\text{IMPROVE}, \text{DECLINE}\}$ is well populated in all projects and that observations in these cells represent 29.76% of the occurrences. In these cells, the overall quality of the code (as viewed through metrics) is either monotonically improved or worsened in one or more aspects. While the distribution of occurrences between each of the four cells is variable, we can see that there are always more DECLINE occurrences than IMPROVE occurrences. Keeping in mind that we track the fluctuations of only a subset of metrics and that we ignore the changes induced by new or deleted classes, this observation suggests that, over the development of these applications, classes tend to decline in quality more than they improve overall. This can be explained as a reflection of the fact that as a program grows with new capabilities so does its intrinsic complexity and size and is a confirmation of the design erosion phenomenon [34].

Fourth, we observe a significant minority of occurrences clustered in the cell (MANY, MIXED). This cell represents cases when at least two quality metrics changed in opposite directions. It represents, on average, 5.46% of the occurrences, which is about a fifth of the cases where metrics change monotonically. These cases are notable because they capture revisions where code changes improve the codebase in some quality aspects at the cost of others. We consider this fluctuation pattern as a reflection of *tradeoffs* between metrics.

In summary, we answer RQ1 as follows: (a) in the majority of revisions, there are no metric fluctuations; (b) metric fluctuations rarely "cancel out"; (c) metrics change monotonically in roughly a third of all revisions; (d) quality deteriorates more often than it improves; (e) metric tradeoffs exist in a minority of revisions.

Based on observation (a), we can confirm that, at least from a metrics perspective, in the largest part of a project's history there is very little happening that is significant to the measurable design quality of the project. By extension, for most of a project's history, measurable quality characteristics (i.e., metrics) cannot help developers understand what quality attributes are of greater priority to the project. However, observation (e) leads us to conclude that there consistently exists a small number of revisions in which there are observable and *measurable* quality tradeoffs. These can be indicative of "design hot spots", and can potentially reveal a great deal of information about the design decisions and principles followed in a project. Narrowing the scope to these hot spots for future qualitative analysis of the system's design quality has the potential to provide insights for improving project awareness, onboarding and overall developer productivity. It is important to mention that, even though we focused on illustrating fluctuations along two dimensions (direction, cardinality), our approach can be easily adapted to take into account further characterization dimensions, such as the magnitude of the

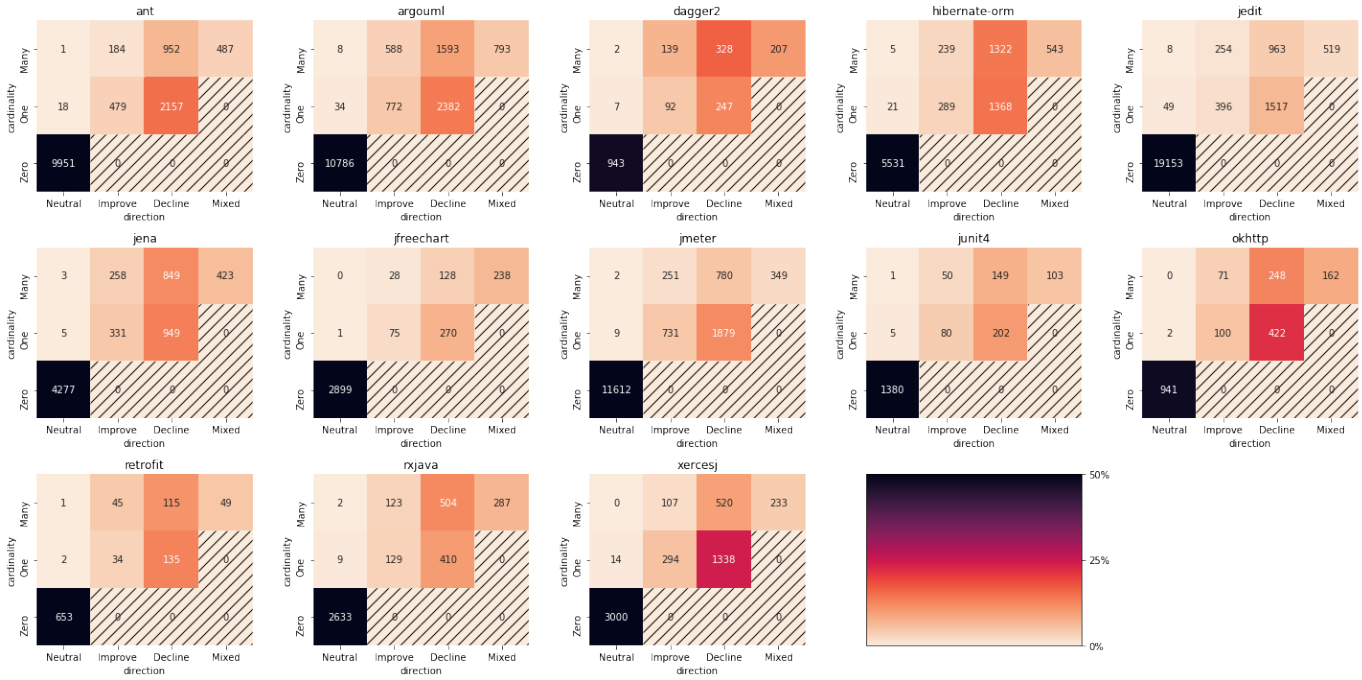
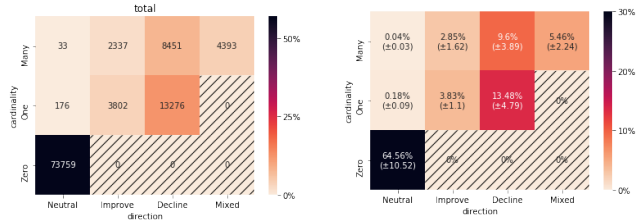


Fig. 2: Metric fluctuations for each project



(a) Cumulative sum

(b) Average

Fig. 3: Distribution of metric fluctuations in production code artifacts, averaged over all projects. The number in parentheses is the standard deviation.

metrics fluctuations.

III. Fluctuations in Context

In this section, we address RQ2 ("Do fluctuations in metrics depend on the development context?") by analyzing our dataset under the lens of three separate development contexts: (a) production versus test code, (b) refactoring, and (c) the different phases of the release cycle. We compare how the categorization of revisions according to their metric fluctuations changes for each of the contexts. Further, as our exploratory analysis pointed to the existence of revisions with measurable quality tradeoffs, we specifically focus our investigation on understanding how such tradeoffs fit with existing theories about each of the studied contexts.

A. Context 1 – Production vs. Test Code

We want to compare the distributions in metric fluctuation categories for revisions that affect production code versus those that affect test code artifacts. Tests can be considered a form of specification of the requirements of a software project. We thus expect their quality to be more stable than that of production code as shown in Fig. 3. Further, if revisions whose metric fluctuation matches the tradeoffs category are indeed "design hot spots", we expect that test code should have relatively fewer revisions in this category, as design is an activity that mostly concerns production code.

1) Methodology: To make the comparison, we modify the *Calculating changes* analysis step described in Sec. II-C in order to include test artifacts. For each revision v, we relax the definition of $\mathcal{A}$ to include strictly changing test classes (previously excluded) and we create two matrices $D_v^{|P|\times|\mathcal{M}|}$ and $D_v^{|T|\times|\mathcal{M}|}$,where $P \subseteq \mathcal{A}$ contains all the strictly changing production artifacts, $T \subseteq \mathcal{A}$ contains all the strictly changing test artifacts, and $P \cap T = \emptyset$. We then continue the rest of the analysis procedure for each set of metric fluctuations, resulting in two sets of classification for all revisions: $\vec{\omega}_p$ and $\vec{\omega}_t$, for production and test code respectively.

2) Results: The heatmaps representing the distribution of metrics fluctuations for production and test code are shown in Figures 3b and 4, respectively.

We note that the test distribution also follow the trend we saw in Sec. II-D where the majority of revisions are categorized as $(\text{ZERO}, \text{NEUTRAL})$, i.e., the metrics do not change. However, we also note that test code has a higher percentage of revisions in this category (85.06% vs 64.56%). This is consistent with our hypothesis that the quality of test code is more stable than that of production code.

We also note that production code has a significantly larger percentage of revisions in the $(\text{MANY}, \text{MIXED})$ category, which

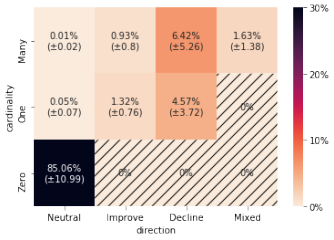


Fig. 4: Distribution of metric fluctuations in test code artifacts, averaged over all projects. The number in parentheses is the standard deviation.

TABLE II: Ratio of revisions containing a refactoring to total number of revisions

Project	Refactorings	Ratio
Ant	1702	11.96%
ArgoUML	2001	11.8%
Dagger2	568	28.91%
Hibernate ORM	1517	16.28%
jEdit	1394	6.1%
Jena	1125	15.86%
JFreeChart	159	4.37%
JMeter	1403	8.99%
JUnit4	332	16.85%
OkHttp	467	23.99%
Retrofit	191	18.47%
RxJava	608	14.84%
Apache Xerces-J	648	11.77%

is consistent with the interpretation that tradeoffs should concern production rather than test code.

As previous results suggested, we observe that NEUTRAL changes for ONE or MANY changes represent a negligible number of revisions. Finally, in the DECLINE column of both heatmaps, design erosion [34] is clearly observable.

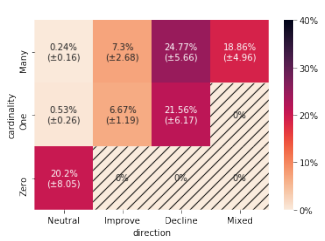
In conclusion, we see that the change of code context between production and test has a noticeable effect on the classification of the metric fluctuations.

B. Context 2 – Refactoring

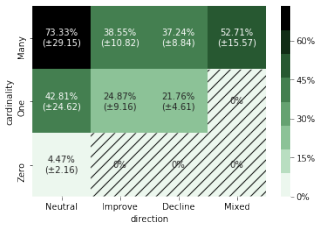
We want to compare the distribution in metric fluctuation categories for revisions that contain refactorings, henceforth: *refactoring revisions* (RRs), compared to the general case presented in Secion II-D.

Refactoring has been extensively studied for its impact on software quality [2], [29]. Researchers have focused on the identification of refactoring opportunities and on analyzing their impact on design, using code smells [15], [31] or code metrics [21]. The resulting consensus is that refactoring impacts the design quality of a system in a significant way. This relationship is not an incidental: developers purposefully use refactoring to express specific design intentions [27]. We thus expect that there should be relatively fewer RRs that have no impact in metrics.

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. Additionally, we know that changes in design, positive or negative, are reflected



(a) Average percentage of metric fluctuations for each category.



(b) Average percentage of RRs with respect to all revisions for each category.

Fig. 5: Metric fluctuation distributions for refactoring revisions averaged for all projects. The number in parentheses is the standard deviation of each average.

in software metrics [19] and that refactoring doesn't always improve monotonically the quality of an application. In other words, developers consciously make design quality tradeoffs while refactoring. It is therefore a very interesting context in which to study metric fluctuations. We thus also expect to find an increased presence of tradeoffs among RRs.

1) Methodology: To identify RRs in the projects' revision histories, we used RMiner [33], an automated tool that can automatically detect refactorings with high recall and precision. We configured RMiner to output the list of refactorings present in each revision in $\mathcal{V}$. We thus filtered the 107 449 commits of all the projects, down to 12 115 RRs, i.e., revisions with at least one refactoring. The median percentage of RRs compared to the entire revision history 14.84%. The project with the lowest percentage is JFreeChart with 4.37% and highest is Dagger2 with 28.91%. 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. The ratio for each project is shown in Table II.

2) Results: The heatmap showing the distribution of metric fluctuations for RRs, averaged for all projects is shown on Fig. 5(a). We show the percentage of RRs compared to the total number of revisions in each category in Fig. 5(b). It is immediately evident that the distribution is very different from the one in Fig. 3(b).

First we note that in the context of refactoring the category (ZERO, NEUTRAL) is notably less populated, as on average only 20.2% of RRs fall in this category. We can also see in Fig. 5(b) that RRs are more than 50% of the total number of (MANY, MIXED) revisions. These observations are consistent with our hypothesis that RRs are more likely to be affecting design quality.

Second, we note that there are about four times as many RRs that are categorized as (MANY, MIXED) and are thus potential "design hot spots" because of tradeoffs in metrics. This is also consistent with our hypothesis that refactoring is an activity during which developers are bound to be making design quality tradeoffs. Similarly to the previous context, we also observe design erosion.

We thus conclude that the refactoring context has a big

TABLE III: Analyzed minor releases for each project

Project	Release Range			Commits
	Start	End	Count	
Ant	ANT_12	ANT_1.10.0_RC1	9	13225
Dagger2	dagger-parent-1.0.0	dagger-2.20	24	1839
Hibernate ORM	4.0.0.Final	5.4.0	9	7654
Jena	jena-2.10.0	jena-3.10.0	15	6960
JMeter	v2_7	v5_0	13	7572
JUnit4	r4.6	r4.13-beta-2	8	1815
OkHttp	parent-1.1.0	parent-3.12.0	26	1811
Retrofit	parent-1.0.0	parent-2.5.0	16	944
RxJava	0.5.0	v2.2.0	23	4403
Apache Xerces-J	Xerces-J_1_1_0	Xerces-J_2_12_0	17	5470

effect on metric fluctuations.

C. Context 3 – Releases

We want to understand how the distribution in metric fluctuation categories for revisions changes relative to the development cycle. Projects are often put in a "freeze" state before important releases [22]. Moreover, Potdar et al. have found that little self-admitted technical debt is accrued close to release dates [24]. We expect this to be reflected in quality metric fluctuations closer to release dates.

1) Methodology: In order to study the distribution of metric fluctuations within releases, we partition of the set of revisions $\vec{\omega}$ of each project into a set of *release slices* $\vec{\omega}_i$. Each *release slice* $\vec{\omega}_i$ represents a part of the revision history that contributed to project evolution between successive releases R_{i-1} and R_i.

To compute the release slices for each project, we manually inspected all Git tags that annotate specific revisions in its evolution. Git tags are used to mark important points in a revision history and tags for release points usually follow a semantic versioning syntax scheme (e.g., v1.0, 2.1, r2.3). After identifying release tags, we manually sorted them in ascending order of Major, Minor and Patch numbering. Next, we processed pairs of successive releases (R_{i-1}, R_i) to identify the slice of commit history that contributed to transition of the project from release R_{i-1} to R_i. The *release slice* thus consists of the set of reachable commits from tag R_i after excluding the commits that are reachable from tag R_{i-1}. Notice that the set of reachable commits of a revision represents all its ancestors in the commit history. The last commit in each release slice $\vec{\omega}_i$ is the one that is tagged with the release tag R_i. The release slice for R_i can be generated by the following command: `git log` $R_{i-1}..R_i$ `--pretty="%H" --no-merges`

We manually recovered the release history of 10 projects in our dataset. We excluded ArgoUML, jEdit and JFreeChart due to low availability of release information in Git tags. We set the granularity of our analysis at the level of minor releases. In the projects we analyzed, patch-level releases were either too granular or too sparsely used, while major releases were too infrequent.

Within each release slice, revisions are further divided into groups, on the basis of their proximity to the release date. Given a release with tag R_i, its release date corresponds to the author date of the commit that is referenced by R_i. Thus, the proximity to release date for a commit $\omega_{i,j}$, belonging to release slice $\vec{\omega}_i$, is equal to the difference in days of its author date from release date. We divide each release slice $\vec{\omega}_i$ to four periods, demarcated by the quartiles of the proximity to release date of commits included in $\vec{\omega}_i$. For a release slice $\vec{\omega}_i$, we define the following periods:

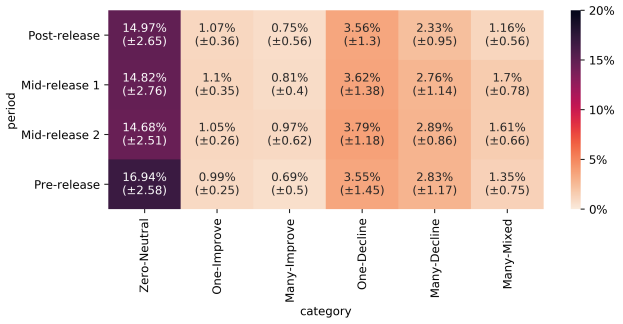
- *Post-release*: commits with proximity to release that is larger than the third quartile Q3 (75th percentile) of proximity values within $\vec{\omega}_i$. These commits correspond to the first revisions of a release slice and can be regarded as the first commits that follow release R_{i-1} (hence the term *Post-release*).
- *Mid-release 1*: commits with proximity to release that is larger than the second quartile Q2 (median) of proximity values and less or equal to Q3. *Mid-release 1* follows the post-release period and can be regarded as a period when concerns handled by the release start to become more mature towards the pre-release period.
- *Mid-release 2*: commits with proximity to release that is larger than the first quartile Q1 (25th percentile) of proximity values and less or equal to Q2.
- *Pre-release*: commits with proximity to release that is less or equal to Q1. These commits represent the last commits of the release slice that lead to release R_i.

We implemented the retrieval of release slices and release periods for commits as a feature of MetricHistory. We summarize the details of the data collected in Table III. Columns 2–3 show the first and last minor releases, while columns 4–5 show the total number of releases analyzed for each project and the overall number of commits. We exclude the first minor release of each project (as they have no "post release" phase) and releases with made beyond 2018-12-31.

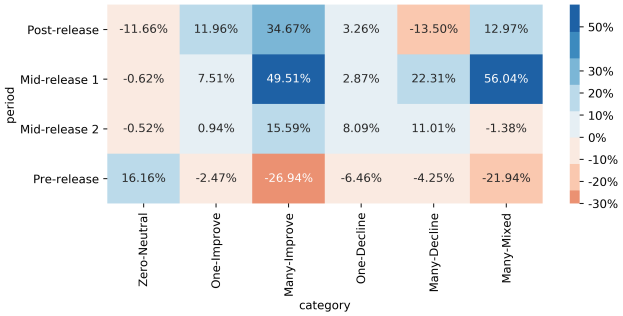
2) Results: The heatmaps representing the average ratio of project revisions over metric fluctuation categories and release periods is shown in Fig. 6(a). Each cell with row label R_i and column label C_j provides the average and standard deviation over all projects, for the ratio of project revisions that are committed in release period R_i and classified as category C_j. For brevity we omit categories with very few revisions $((\text{ONE}, \text{NEUTRAL}), (\text{MANY}, \text{NEUTRAL}))$.

We observe that in Fig. 6(a) the fluctuations in metrics do not change dramatically across the different release periods. The overall fluctuation pattern is that the majority of revisions are in the $(\text{ZERO}, \text{NEUTRAL})$ category, with a noticeable design erosion effect indicated by the DECLINE categories, and a small minority of $(\text{MANY}, \text{MIXED})$ revisions. We do however note a small observed increase in the $(\text{ZERO}, \text{NEUTRAL})$ category closer to the release date, which is consistent with the conclusions of published research mentioned above.

To further understand these observations, we also calculated the percent change of average ratio for each category of metric fluctuations over successive phases. The resulting heatmap is shown in Fig. 6(b). This representation shows more clearly the trends, normalizing the domination of the $(\text{ZERO}, \text{NEUTRAL})$ category. We make a few observations. First, we note that the increase in the $(\text{ZERO}, \text{NEUTRAL})$ category in the pre-release period is coupled with a decrease in all other categories. This is



(a) Average percentage of project revisions over release periods and metric fluctuation categories. The number in parentheses is the standard deviation.



(b) Percent change of average ratio over successive release periods. Post-release is calculated relative to Pre-release.

Fig. 6: Distribution of metric fluctuations relative to stages in release cycle, averaged over all projects.

again consistent with published research. Second, we note that there is a marked increase in all other categories during the first mid-release period. One explanation could be it is during this period that the project is most in flux, with feature additions and maintenance tasks. It is notable that the (MANY, MIXED) category is most increased during this period. This might indicate that future research should focus in this period to identify quality tradeoffs.

We conclude that the effect of the release context to metric fluctuations is not big. However we can be observe some interesting differential effects.

D. Discussion

We have studied metric fluctuations in three contexts. First we compared the fluctuations in production and test code and found that there is a noticeable difference between the two, with test code being more stable and containing fewer tradeoffs. Second, we investigated refactoring and found that it has a clearly identifiable effect on metric fluctuations. Third, we studied how metric fluctuations change over time during releases. We found a very small effect closer to release dates, as well as some interesting differential effects. These observations lead us to formulate an answer to RQ2 that fluctuations in metrics can depend on the development context. Future research should investigate additional contexts and fluctuation categories, as well as differential effects.

IV. Threats to validity

Construct validity is threatened by two 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 Sec. II-C, 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. Interestingly, we observed no statistical differences in distributions for detecting tradeoffs when taking into account added and deleted classes. 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. This way, we would see if a metric changed even if the changes of metric of two classes or more would cancel each other. For these reasons, we cannot draw general conclusions about what would happen at different granularity levels or with other metrics. 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. 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.

External validity is threatened by the generalizability of the study. We analyzed 13 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 a preliminary 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 commits).

Nevertheless, our study has a clearly defined scope; we, therefore, do not claim that our results are generalizable, but rather make an existential argument for metric fluctuations.

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 MetricHistory upon publication.

V. 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 [29], 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. [21] 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. [23]. Although, in principle, tradeoffs could be captured in detection rules, Tsantalis et al. [31] 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 [29], 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. [18] and Hegedus et al. [16] 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. [7] 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. [5] investigated the extent to which developers are successful at removing code smells while refactoring. Soetens et al. [28] analyzed the effects of refactorings on the code's complexity. Tsantalis et al. [32] 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 it did not discuss more major design decisions as part of the refactoring intent.

Recovery of design decisions has been studied from an architectural perspective. Jansen et al. [17] proposed a methodology for recovering architectural design decisions across releases of a software system. 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.[26]. 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 the lower level design and which 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.

VI. Conclusion

Our vision is to help developers better understand the the code artifacts they work with. In this paper, we take the first step towards this vision by studying fluctuations in quality metrics in projects' revision histories. By characterizing the metric fluctuations we aim to better understand change and to facilitate the detection of change patterns that can be relevant as documentation to developers or other stakeholders.

We have studied the complete revision histories of 13 open source projects, with a small set of well-known metrics that allowed us to investigate metric fluctuations without loss of generality. We have found that projects' histories contain important metric fluctuations that can be meaningfully categorized in terms of dimension and cardinality. We have identified an important category of fluctuations that can be used to help analysis focus on a small minority of revisions with high likelihood to contain important design quality tradeoffs. We further analyzed 10 of the projects in three separate development contexts, and found that there is a dependency between metric fluctuations and development context.

In the future, we intend to expand our analysis to include more quality metrics, as well as other types of static analysis. We want to specifically focus on revisions containing tradeoffs and understand if they embody design decisions. Ultimately we would like to generate descriptive, history-based metadata for each code artifact that could be used by developers and other stakeholders to synthesize documentation, and to empirically assess software and software projects, and make informed decisions.

REFERENCES

[1] Akka. https://akka.io/. [Accessed 2019-04-08].

[2] 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.

[3] Ryo Arima, Yoshiki Higo, and Shinji Kusumoto. A study on inappropriately partitioned commits: How much and what kinds of ip commits in java projects? In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 336–340, New York, NY, USA, 2018. ACM.

[4] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering*, 28(1):4–17, 2002.

[5] 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.

[6] Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkely, CA, USA, 2nd edition, 2014.

[7] 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.

[8] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

[9] Dataset. https://doi.org/10.5281/zenodo.2653236, April 2019.

[10] Steve Easterbrook and Barbara Neves. Course notes for CSC2130 Empirical Research Methods for Computer Scientists, 2014. [Accessed 2019-04-03].

[11] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. *Selecting Empirical Methods for Software Engineering Research*, pages 285–311. Springer London, London, 2008.

[12] 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.

[13] Cristina Gacek and Budi Arief. The many meanings of open source. *IEEE software*, 21(1):34–40, 2004.

[14] GitHub. https://help.github.com/en/articles/importing-a-repository-with-github-importer. [Accessed 2019-04-24].

[15] 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.

[16] 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.

[17] 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).

[18] 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.

[19] 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.

[20] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, June 1978.

[21] 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.

[22] Martin Michlmayr, Francis Hunt, and David Probert. Quality practices and problems in free software projects. In *Proceedings of the First International Conference on Open Source Systems*, pages 24–28, 2005.

[23] 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.

[24] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100. IEEE, 2014.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 51:1–51:11, New York, NY, USA, 2012. ACM.

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

[32] 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.

[33] 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.

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

[35] Giovanni Viviani, Calahan Janik-Jones, Michalis Famelis, Xin Xia, and Gail C. Murphy. What design topics do developers discuss? In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 328–331, New York, NY, USA, 2018. ACM.