Exploring Fluctuations in Internal Quality Metrics

***Abstract*—Background: Developers rely on the version history of a project to understand its key priorities. However, understanding long term changes that span the project's history is a daunting task, as relevant information may only be available in scattered commit messages. Aims: We take the first step towards detecting informative change patterns in revision histories by studying the fluctuations in measurable code characteristics between revisions. Methods: We analyze the history of 13 open source Java projects calculating the fluctuations in internal quality metrics at the revision level. Results: Across all projects, we classify more than 100 000 revisions in function of their fluctuations of metrics and observe consistent behaviors. Conclusions: We show that it is possible to identify recognizable *tradeoffs*, where some metrics may improve at the expense of others, at key development contexts.**

I. Introduction

Developers expend a lot of effort trying to understand how to best maintain software systems, add new features, or fix bugs [13]. They leverage multiple sources of information, such as code, wikis, discussions, commit messages, and others, often having to sift through potentially out of date information [15]. Ultimately, successfully contributing to a project often requires a lot of tacit, contextual design knowledge, typically accumulated over time throughout its development history. However, extracting relevant and pertinent information about an artifact from its revision history is a daunting task. When they exist, commit messages are crucial sources of information. However they are not always relevant or helpful when trying to understand the effect of changes. Diffs, i.e., exhaustive lists of changed lines of source code, are sometimes the only other alternative, but the process of understanding them is confusing and slow [19].

Developer productivity could be thus improved by helping them better understand significant changes in the history of artifacts they interact with. Ideally, developers could be presented with a short list of the most important revisions that best characterize the key decisions about a project. This would be especially useful for onboarding new contributors, helping them understand the main priorities and quality requirements when considering making changes. We envision a development workflow supported by descriptive, history-based metadata that could be used by developers and other stakeholders to synthesize documentation, and to empirically assess artifacts, 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. We propose to characterize the revisions of a project with information about fluctuations in code metrics. This is the basis for understanding change and for finding useful change patterns. We are particularly interested in *quality tradeoffs*, i.e., changes where developers 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. Inspired by work on understanding tradeoffs in refactoring [2], we assume that this change pattern is a likely indicator of design hot spots. In other words, we assume that deliberate or inadvertent *design choices* can be witnessed by tradeoffs in the quality characteristics of the code. In this, we are following the open source principle that code is the most authoritative source of design information [10]. We envision our approach as complementary to other approaches for extracting tacit and contextual design knowledge from discussions [23], commit messages [6], etc.

We are thus interested in understanding what metric fluctuations exist in projects' revision histories **(RQ1)** and whether there is evidence that supports our underlying assumption, i.e., whether we can correlate the presence of design hot spots with quality tradeoffs **(RQ2)**.

To answer these two questions, we examined the development history of 13 open source software projects. 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 investigated the prevalence of tradeoffs in two development contexts where design choices can be reasonably anticipated. Specifically, we compared the presence of tradeoffs in testing code versus the rest of the codebase; and 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 regarding design activity.

Overall, we found that quality tradeoffs coincide with particular and significant development activities. We also found that, although they carry a lot of information about an artifact, quality tradeoffs can be found in a minority of commits. In summary, we contribute: (a) a quantitative study of the metric fluctuations in 13 open source projects; (b) a public data repository of historical internal quality metric fluctuations for open source projects; (c) a systematic methodology to mine internal quality metric fluctuations from revision histories; and (d) 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 RQ1 in Sec. II. In Sec. III, we answer RQ2 by analyzing studying metric fluctuations in two different development contexts. We discuss threats to validity in Sec. IV, related work in Sec. V, and conclude in Sec. VI.

II. Exploratory Study

In this section, we aim to answer 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*.

Design: We conduct an exploratory study structured using the guidelines established by Easterbrook et al. [8]. We use a mixed-method approach, consisting of an exploratory case-driven archive analysis, and comparative explorations.

Our unit of analysis is the *revision*. Revisions, in version control systems, 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]. We can think of the list of revisions of a project as its evolutionary history; each revision embodies a version of the software. Since we aim to understand the patterns of metric fluctuations in the version history of software projects written in Java, the target population of our study is the set of all revisions in their version history.

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). The projects are listed in Table I with the *default* branch, source code location, number of commits mined, size, number of revisions containing refactorings – henceforth refered as *refactoring revisions* –, and the percentage of refactoring revisions relative to the number of commits.

Building a Dataset: We used a combination of static analysis and software repository mining as our data collection technique. At first, we downloaded the default branch of each project's version history, as shown in Table I. We focus on the default branch because it is often this rendition of the development that will be released to the public.

In a next stage, we calculated the metrics for all source code artifacts, for a subset of the *default* branch revisions of each project. Specifically, we imposed two additional restrictions for revision selection: (a) we only keep commits until 2018-12-31 (included) and (b) we exclude commits that are the result of a *merge* operation. *Merge* commits are generally hard to read for humans and present little interest for analysis as the individual changes can be found on the source branch.

We developed the MetricHistory tool to calculate the metrics for all source code artifacts of each project revision. The tool provides a convenient interface to enable mining software repositories, while encapsulating VCS concerns. We use its *collection* feature to automatically compute the metrics for each project revision. Metrics are computed internally with SourceMeter [9], a highly configurable tool that supports a

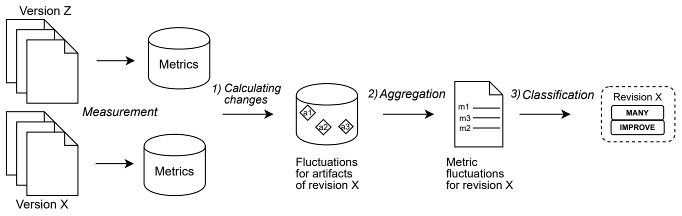


Fig. 1: Analysis procedure overview

wide variety of metrics for diverse source code artifacts (e.g., method, class, package).

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 [18], [20]). 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 [14], [18]. Therefore, this subset is big enough to be able to detect if metric fluctuations exhibit notable behavior without a loss of generality.

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 [9] for each Java class. To accomodate this process, We also built a distributed computation system based on the Akka toolkit and runtime [1]. The data is available privately on zenodo.org for review purposes [7] and will be made public upon publication.

Analysis Approach: The first stage of the analysis procedure (shown in Fig. 1) involves the calculation of metric changes for source code artifacts (classes) across revisions.

Let $\mathcal{V} = \{v_0, v_1, \dots v_N\}$ be the set of analyzed revisions for a given project. Moreover, let $v_i \in \mathcal{V}$ be a specific revision and v_j its parent. The change set of v_i comprises classes that were either added, changed or removed, with respect to parent revision v_j. Let A_i be the set of changed classes for revision v_i and $a_k \in A_i$ a given changed class. We calculate for a_k the change (difference) in each one of the four metrics, with respect to their values for the same class a_k in parent revision v_j. Let $\delta_{ik} = (\delta_{CBO_{ik}}, \delta_{DIT_{ik}}, \delta_{LCOM_{ik}}, \delta_{WMC_{ik}})$ be a tuple of these metric changes for a_k and $\Delta_i = \cup\{\delta_{ik}\}$ the set of metric changes for all changed classes of v_i.

In the second stage of data analysis, we process the tuple-set Δ_i of each individual revision v_i and reduce it to a single tuple r_i that aggregates metric changes for all changed classes of v_i. Thus, it holds that:

$$r_i = (\sum_k \delta_{CBO_{ik}}, \sum_k \delta_{DIT_{ik}}, \sum_k \delta_{LCOM_{ik}}, \sum_k \delta_{WMC_{ik}})$$

We will, henceforth, refer to r_i as *metric fluctuation* for revision v_i. The second stage of analysis produces a tuple-set

TABLE I: List of the software projects retained.

Project	Code source	Branch	Commits	Size (SLOC)	Refactorings	Refact. Ratio
Ant	https://github.com/apache/ant	master	14 234	139k	1702	11.96%
ArgoUML	http://argouml.tigris.org/source/browse/argouml/trunk/src/	trunk	17 797	176k	2001	11.8%
Dagger2	https://github.com/google/dagger	master	1 969	74k	568	28.91%
Hibernate ORM	https://github.com/hibernate/hibernate-orm	master	9 320	724k	1517	16.28%
jEdit	https://sourceforge.net/p/jedit/svn/HEAD/tree/jEdit/trunk/	trunk	22 873	124k	1394	6.1%
Jena	https://github.com/apache/jena	master	7 112	515k	1125	15.86%
JFreeChart	https://github.com/jfree/jfreechart	master	3 640	132k	159	4.37%
JMeter	https://github.com/apache/jmeter	trunk	15 898	133k	1403	8.99%
JUnit4	https://github.com/junit-team/junit4	master	1 972	30k	332	16.85%
OkHttp	https://github.com/square/okhttp	master	1 951	61k	467	23.99%
Retrofit	https://github.com/square/retrofit	master	1 038	20k	191	18.47%
RxJava	https://github.com/ReactiveX/RxJava	2.x	4 137	276k	608	14.84%
Xerces-J	https://github.com/apache/xerces2-j	trunk	5 508	142k	648	11.77%

$\mathcal{R} = \{r_0, r_1, \dots r_N\}$ comprising the metric fluctuations for the analyzed version history of the project.

The third and final stage of analysis aims at classifying each revision v_i to a *metric fluctuation category*, on the basis of the values of the respective tuple r_i. This classification was designed to reflect the macroscopic changes happening in a revision and is based on a two-dimension classification taxonomy.

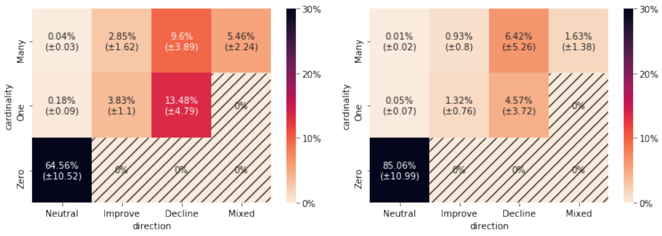
To develop this classification scheme, we did a pilot qualitative analysis of the revision history of JFreeChart. Two of the co-authors worked independently and then consolidated their approaches in a single schema. The consolidated schema was then further refined as the full dataset was collected.

A classification for revision v_i is defined as a tuple ω_i from the space $\mathcal{C} \times \mathcal{D}$ where:

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

For the cardinality of change, we define ZERO as the case where there is strictly no metric change across the changed classes of a revision, i.e., $\delta_{ik} = (0, 0, 0, 0), \forall a_k \in A_i$ for a given revision v_i. Cardinality ONE represents the case where exactly one metric has non-zero aggregate change, i.e., one component of r_i is non-zero for v_i. MANY represents the case where two or more components of r_i are non-zero for v_i.

As concerning the direction of change, we define IMPROVE and DECLINE when all components of r_i go in a positive or negative direction with respect to quality respectively. NEUTRAL represents the case where δ_{ik} is non-zero for two or more changed classes a_k in v_i, but balance out to zero during aggregation, i.e. $r_i = (0, 0, 0, 0)$. Finally, MIXED matches cases where some components of r_i IMPROVE, while 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. The resulting classification of a project's revisions is $\Omega = \{\omega_0, \omega_1, \dots \omega_N\}$ where N is the number of the project's analyzed revisions.



(a) Production (b) Test

Fig. 2: Distribution of metric fluctuations in production and test code artifacts, averaged over all projects. The number in parentheses is the standard deviation.

Results: We classified all revisions in $\mathcal{V}$, for the production code of each project in Table I. Fig. 2a presents overall results with standard deviation as a heatmap table, where the rows correspond to $\mathcal{C}$ and the columns to $\mathcal{D}$. Darker shades indicate higher numbers of revisions, whereas thatched lines indicate invalid combinations. We, also, compared per-project distributions of revisions and found similar fluctuation patterns among projects, as is indicated by the low standard deviation of each category in Fig. 2a.

We observe some recurring tendencies in the distribution of project revisions to fluctuation categories. 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 out each other 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 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. 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 [22].

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 metrics fluctuations.

III. FLUCTUATIONS IN CONTEXT

In this section, we address RQ2 (whether design hot spots and quality tradeoffs correlate) by analyzing our dataset under the lens of two separate development contexts: (a) production versus test code, and (b) refactoring. Specifically, we compare how the categorization of revisions according to their metric fluctuations changes for each of the contexts. We focus our investigation on understanding how quality tradeoffs fit with existing theories about each of the studied contexts.

Context 1 – Production vs. Test Code: We compare the distribution of revisions to metric fluctuation categories by separating the analysis of production and test code. To make the comparison, we adjust the analysis of Section II to include only test classes. Since tests can be considered a form of requirements specification for a software project, we expect their quality to be more stable than that of production code. Further, if revisions falling into tradeoffs category are indeed "design hot spots", we expect that test code should have relatively fewer revisions in this category, as the design activity mostly concerns production code.

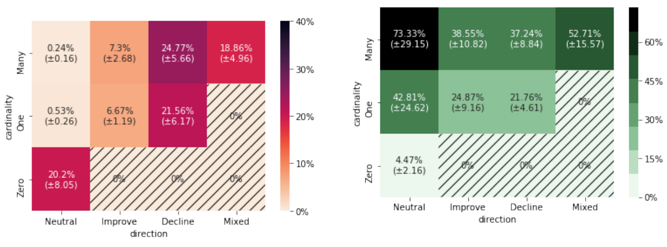
The heatmap representing the distribution of metrics fluctuations for production and test code is shown in Figure 2. We note that the test distribution also follow the trend we saw in Section II where the majority of revisions are categorized as $(\text{ZERO}, \text{NEUTRAL})$, i.e., the metrics do not 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 (none of the projects are following Test Driven Development). We also note that production code has a significantly larger percentage of revisions in the $(\text{MANY}, \text{MIXED})$ category, which 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 [22] 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.

Context 2 – Refactoring: We compare the distribution in metric fluctuation categories for refactoring revisions (RRs), compared to the general case presented in Section II.

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, changes in design, positive or negative, are reflected in software metrics [12] and refactoring doesn't always improve monotonically the quality of an application. Refactoring also impacts the design quality of a system in a significant way [18], [20] [14] and developers purposefully use refactoring to express specific design intentions [16]. It is therefore a very interesting context in which to study metric fluctuations. We thus expect that there should be relatively fewer RRs that have no impact in metrics and an increased presence of tradeoffs among RRs.

To identify RRs in the projects' revision histories, we used RMiner [21], 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. The median percentage of RRs compared to the entire revision history is 14.84%. The project with the lowest percentage is JFreeChart with 4.37%



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

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

Fig. 3: Metric fluctuation distributions for refactoring revisions averaged for all projects. The number in parentheses is the standard deviation of each average.

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 the last column of Table I.

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

Discussion: We have studied metric fluctuations in two 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. 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 three sources. The first threat comes from our selection of four metrics which is inevitably going to miss some interesting fluctuations. This threat is mitigated by the scope of the study: this is an exploratory work that serves as a proof of concept for metric fluctuations. We are not trying to enumerate all instances of tradeoffs. Secondly, 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 when taking into account added and deleted classes.

Thirdly, we use a sum operation to aggregate metrics to the revision level. This choice supports our exploration process by providing a baseline and a compelling proof of concept. In our next study, we're planning to use more refined methods.

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 (we explored the default branch). Thus, our study is biased by the development practices used in these projects. For mitigation, we selected a diverse set of projects and also investigated their commit hygiene during a preliminary manual analysis.

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 [18], 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.

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. [14] 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. Although, in principle, tradeoffs could be captured in detection rules, Tsantalis et al. [20] went one step further and defined a new metric, *Entity Placement*, to capture a tradeoff. Quality assessment using Entity Placement is supposed to give more global results with respect to detection and improvement after refactoring. Hegedus et al. [11] 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. [5] 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. [4] investigated the extent to which developers are successful at removing code smells while refactoring. Soetens et al. [17] analyzed the effects of refactorings on the code's complexity.

VI. Conclusion

Our vision is to help developers better understand 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 revision histories of 13 open source projects, with a subset of well-known metrics that allowed us to investigate metric fluctuations. We have found that projects' histories contain important metric fluctuations that can be meaningfully categorized in terms of direction and cardinality. We have identified an important category that can be used to help analysis focus on a small minority of revisions with high likelihood to contain important design quality tradeoffs.

In the future, we intend to expand our analysis to include more quality metrics, other types of static analysis, and refine our approach. We also 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] 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.

[5] 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.

[6] 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.

[7] Dataset. https://doi.org/10.5281/zenodo.3242739, June 2019.

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

[9] R. Ferenc, L. Langó, I. Siket, T. Gyimóthy, 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.

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

[11] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.