An Empirical Study of Refactoring Rhythms and Tactics in the Software Development Process

Shayan Noei, Heng Li, Stefanos Georgiou, Ying Zou

Abstract—It is critical for developers to develop high-quality software to reduce maintenance cost. While often, developers apply refactoring practices to make source code readable and maintainable without impacting the software functionality. Existing studies identify development rhythms (i.e., weekly development patterns) and their relationship with various metrics, such as productivity. However, existing studies focus entirely on development rhythms. There is no study on refactoring rhythms and their relationship with code quality. Moreover, the existing studies categorize the refactoring tactics (i.e., long-term refactoring patterns) into two general concepts of consistent and inconsistent refactoring. Nevertheless, the existence of other tactics and their relationship with code quality is not explored. In this paper, we conduct an empirical study on the refactoring practices of 196 Apache projects in the early, middle, and late stages of development. We aim to identify (1) existing refactoring rhythms, (2) further refactoring tactics, and (3) the relationship between the identified tactics and rhythms with code quality. The recognition of existing refactoring strategies and their relationship with code quality can assist practitioners in recognizing and applying the appropriate and high-quality refactoring rhythms or tactics to deliver a higher quality of software. We find two frequently used refactoring rhythms: work-day refactoring and all-day refactoring. We also identify two deviations of floss and root canal refactoring tactics as: intermittent root canal, intermittent spiked floss, frequent spiked floss, and frequent root canal. We find that root canal-based tactics are correlated with less increase in the code smells (i.e., higher quality code) compared to floss-based tactics. Moreover, we find that refactoring rhythms are not significantly correlated with the quality of the code. Furthermore, we provide detailed information on the relationship of each refactoring tactic to each code smell type.

Index Terms—Refactoring, Code Quality, Code Smells, Refactoring Rhythms, Refactoring Tactics.

1 Introduction

Refactoring is a systematic process of improving the internal quality of software without affecting its functionalities [1]. Many studies show that refactoring is a widely engaged part of the software maintenance process [2], [3], [4], [5]. Refactoring facilitates the extensibility and maintainability of a software system [6], [7], [8]. Various reasons drive developers' refactoring activities, such as improving software design [9], making software systems easier to understand [10], enhancing reusability [11], removing dependencies among attributes, methods, classes, interfaces, and packages [12], as well as eliminating code smells [12], [13], [14], [15], [16], [17]. A code smell is a design flaw that violates the principles of design standards and impairs the maintainability of software [18], [19]. As a result, a code smell can affect the internal quality of software. For instance, a broken hierarchy code smell happens when a subtype and its supertype do not share an "IS-A" relationship [20]. The presence of code smells is a good indicator for code quality checks [18], and multiple refactoring operations are typically needed to eliminate code smells [17].

Several factors contribute to the quantity of refactoring operations performed to improve code quality, such as developer perceptions, team experience, development schedule, software characteristics, and so forth [21]. Different teams may apply different refactoring strategies in short-term or long-term periods [22], [23]. Identifying patterns in refactoring practices and their relationship with code quality can help software developers adopt the most suitable patterns in their projects. More specifically refactoring patterns have two perspectives: (1) refactoring rhythms and (2) refactoring tactics. Refactoring rhythms describe how refactoring operations split across the weekdays and usually focus on existing tasks. Refactoring tactics are referred to as long-term refactoring more focused on future development [24].

In the context of refactoring rhythms, existing studies focus on development rhythms and categorize development rhythms as work-day (*i.e.,* Monday to Friday) development and all-day development. Moreover, they correlate development rhythms with the measures, such as task performance and productivity [25], [26], [27]. However, to the best of our knowledge, no study has investigated the identification of refactoring rhythms and their relationship with code quality.

In terms of refactoring tactics, existing studies divide refactoring tactics into *floss* and *root canal* [1], [28], [29]. *Floss* refactoring is distinguished by frequent refactoring along with the development process. *Root canal* refactoring is identified by occasional refactoring aside from the development process. While the terms *floss* and *root canal* are widely used as refactoring tactics, the existence of other possible tactics and their relationship with code quality is not explored in the existing work.

In this work, we study developers' refactoring activities (rhythms and tactics) and their impact on code quality in terms of code smells. To identify refactoring rhythms and other possible refactoring tactics, we study 196 Apache projects and introduce two metrics: daily refactoring density (DRD) and weekly refactoring density (WRD). Using the introduced metrics, we divide each project into daily and weekly time frames to identify frequent refactoring tactics and rhythms during the lifetime of a project. Then, we investigate the relationship between different rhythms and tactics with code quality. Such information can guide

developers in selecting the most suitable and high-quality refactoring rhythms or tactics for their projects. To this end, we aim to answer the following research questions:

RQ1. What are the rhythms of refactoring?— To identify existing refactoring rhythms, we analyze if the refactoring rhythms fit into the software development rhythms introduced by previous studies (*work-day* and *all-day*). By utilizing the DRD metric and performing Kruskal-Wallis test [30], we observe that the majority of projects (95%) apply two primary rhythms: (1) work-day refactoring (11%) and (2) all-day refactoring (84%).

RQ2: What are the most frequent refactoring tactics used in projects?— To identify refactoring tactics, we utilize the WRD metric and form a time series of refactoring activities for every project. Using dynamic time warping (DTW), we cluster refactoring time series and we observe four variations of floss and root canal refactoring tactics:

- **Intermittent spiked floss:** Regular and consistent refactoring with fewer sudden increases (spikes) compared to frequent spiked floss.
- **Frequent spiked floss:** Consistent refactoring but with more spikes in refactoring density compared to intermittent spiked floss.
- **Intermittent root canal:** Once in a while refactoring in high densities, but with most weeks having no refactoring densities.
- **Frequent root canal:** More frequent refactoring with more spikes in refactoring density compared to intermittent root canal with most weeks having no refactoring activities.

RQ3: What is the relationship of different refactoring rhythms and tactics with code quality?— In the two first research questions, we identify frequently used refactoring rhythms and tactics. Furthermore, we are interested in understanding how different rhythms and tactics are associated with code quality improvement (*i.e.*, reducing code smells). To examine the relationship between refactoring rhythms with code quality metrics, we use a Scott-Knott-ESD [31], [32] test to rank and cluster code smell changes after adopting each refactoring rhythm and tactic. We observe that root canal-based tactics are more targeted refactoring operations, therefore, are correlated with more reduction of code smells compared to floss-based tactics and deliver higher quality code. Furthermore, we observe that refactoring rhythms are not significantly correlated with software quality. Consequently, refactoring rhythms are chosen based on the project assets and the development team's comforts. Finally, we provide some guidelines on positive and negative relationships between different refactoring tactics and different types of code smells.

In conclusion, we make the following contributions:

1) We identify refactoring rhythms and tactics that are used in the software development process, which can provide insights for practitioners to understand existing refactoring practices and develop tooling to support such practices.
2) We understand the relationship between the used tactics and rhythms with software quality, which helps developers to adopt the most suitable approach.

The replication package can be accessed online [33].

Paper organization. The remainder of our study is organized as follows. Section 2 describes the setup of this study. Section 3 presents our approaches and results for answering our research questions. Section 4 discusses the threats to the validity of our findings and Section 5 provides the implications of out study. Section 6 surveys related studies and compares them to our work. Finally, we conclude our paper and present future research directions in Section 7, and acknowledge contributions in Section 8.

2 Experiment Setup

This section presents the setup of our study, including our data collection and data analysis approaches.

2.1 Overview of Our Approach

Figure 1 gives an overview of our study. We conduct our research using the projects with a reasonable amount of development activities from the 20-MAD Apache dataset [34]. We extract the refactoring history of these projects and calculate refactoring density metrics along with their lifespan. By utilizing refactoring density metrics, we compare refactoring distributions at different time periods to answer the first and the second research questions that aim to identify refactoring rhythms and tactics. Furthermore, we use the characteristics of the projects and developers to provide insights into the rationale of using such rhythms and tactics. Finally, by measuring quality changes after adopting each rhythm and tactic, we identify the relationship of the identified rhythms and tactics with the quality of code.

2.2 Data Collection

To perform our experiments, we perform several steps to collect our dataset.

2.2.1 Project Selection and Pre-processing

The 20-MAD Apache dataset [34] contains information about commits and issues related to 765 Mozilla and Apache projects with a timespan of 20 years. In particular, the dataset contains 3.4M commits, 2.3M issues, and 17.3M issue comments. Considering that Java is one the most popular programming languages [35], [36] and it is best supported by refactoring extraction tools (*e.g.*, Rminer [14]), we limit our study to Java projects. To select projects with enough data that help identify the different refactoring tactics and rhythms, we exclude the projects that:

- have less than 80% of Java source code;
- have less than the 1^{st} quantile of commit counts (*i.e.*, $<$ 1,021 commits); and
- have a short lifespan (*i.e.*, $<$ one-year of commit history).

As a result, we obtain 196 Java projects that have sufficient commit history and lifespan for our analysis.

The studied projects in our dataset have varying lifespans and therefore, possess different development histories. Furthermore, refactoring habits or requirements may change over time. For example, as a project ages, design issues may be fixed less frequently [37]. Hence, we cannot compare

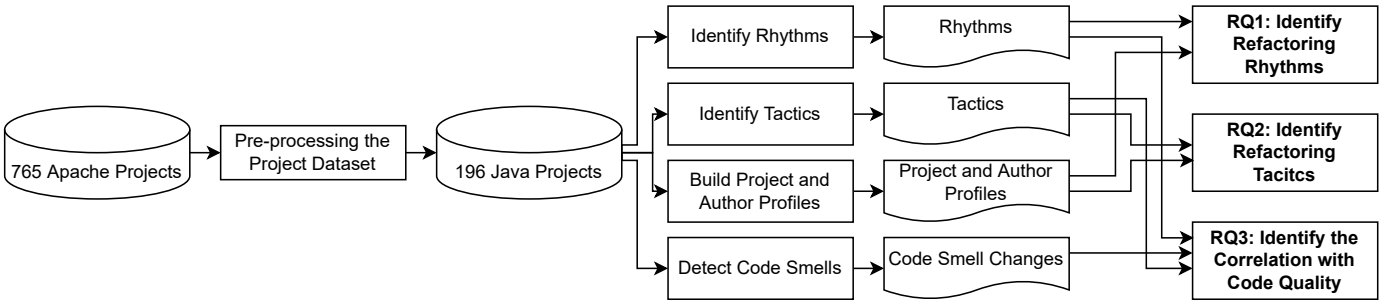

Fig. 1: An overview of our study.

refactoring practices unless we have projects with a similar lifetime. To be able to analyze the refactoring activities in the projects, we partition longer projects into multiple stages. We utilized the first, second, and third quartiles of project ages to establish thresholds and divided all projects into four age groups, with each group containing an equal number of projects. The age groups are categorized as follows: (1) younger than 4.5 years (*i.e.,* the first quartile), (2) between 4.5 (first quartile) and 7 years (second quartile), (3) between 7 (second quartile) and 8.5 years (third quartile), and (4) older than 8.5 years (third quartile) of activities. We excluded projects with less than 4.5 years of activities from our study because such projects have a wide variety of ages, making it difficult to compare similar activities among them, unless they have a similar lifespan. By using the identified age groups, we define different stages of software development: (1) early stage: start of a project until the 4.5^{th} year, (2) middle stage: 4.5^{th} year of a project until the 7^{th} year, and (3) late stage: 7^{th} year of a project until the 8.5^{th} year which is the longest age considered in all studied projects. As a result, we observe that 50 projects have ages between 4.5 and 7 years, 49 projects have ages between 7 to 8.5 years, and 50 projects have ages of more than 8.5 years. To compare the old projects (*e.g.,* 10 years old) with young ones (*e.g.,* 5 years old), we divide the older projects into two or three stages using the thresholds of the age groups. For instance, if a project has 6 years of activities, it has only the early stage of development (*e.g.,* the first 4.5 years of activities), while a project with 10 years of activities has the early, middle, and late stages of development. Doing so allows us to identify similar rhythms and tactics that might appear among the projects at different stages. We use these three development stages throughout the analyses performed in this study.

2.2.2 Refactoring Extraction

To extract the refactoring operations, we use the Rminer 2.0.3 tool [14], which is an AST-based algorithm that finds up to 59 Java refactoring types from the commit history without the need for user-defined thresholds [14], [38]. Furthermore, Rminer is a superior refactoring detection tool compared to its opponents and identifies refactoring operations with an overall precision of 99.7% and a recall of 94.2% [14]. To measure the accuracy of the tool on our dataset, with a confidence level of 95% and a margin of error of 5% on the total number of commits, we select 385 commits to perform our manual validation. In each commit, we reviewed all types of refactorings that the tool could identify and determined if and how many of them were present in the results of the tool. We then compared our results with the tool's results. For example, if we identified a pull-up method but the tool did not, we marked it as a tool failure and vice versa. The manual validation is done by one of the authors and one undergraduate computer science student. The results of the manual validation show an overall precision of 97%, recall of 96%, and F1 score of 95% respectively. Furthermore, we calculate Cohen's kappa coefficient [39] from the participant's manual validation results and achieve a score of 0.91, which suggests a strong agreement. Therefore, we run Rminer on every selected project for each commit to extract refactoring activities in the history of development.

2.2.3 Code Smells Extraction

To identify the relationship between the refactoring rhythms or tactics and code quality, we need to analyze how these refactoring patterns reduce or increase the frequency of code smells. As we use code smells as code quality indicators, we analyze how refactoring rhythms and tactics are associated with the reduction or increase in the frequency of code smells.

We use the Designite tool [40] to extract code smells. Designite can identify the most types of code smells compared to its alternatives and detects numerous code smells in large codebases [40], [41]. For instance, Arcan [42] and Hotspot detector [43] can detect only 4 types of code smells. Similarly, Jdeodorant [44] detects 5 and Arcade [45] detects 11 types of code smells. Designite can identify 35 code smell types including 7 architecture smells, 18 design smells, and 10 implementation smells as listed in Table 1. The different categories of code smells have specific causes and impacts on the software system. Architecture code smells arise from poor software designs within the system architecture and can negatively impact system quality, performance, and lifespan [46]. Design code smells result from inadequate system design and have a negative impact on code quality [19]. Implementation code smells, on the other hand, stem from poor implementation decisions made by contributors and can negatively affect the quality of the code [1]. To address all three types of code smells, refactoring has been shown to be an effective solution [1], [19], [46], [47]. Being able to identify the various types of code smells allows us to study the impact of different refactoring strategies and techniques in a more comprehensive manner. Moreover, Designite is open-source and can be used on cloned projects at the code

TABLE 1: The list of code smell metrics used in the study.

Category	Metrics
Architecture Smells	Cyclic Dependency, Unstable Dependency, Ambiguous Interface, God Component, Feature Concentration, Scattered Functionality, and Dense Structure
Design Smells	Imperative Abstraction, Wide Hierarchy, Broken-Modularization, Cyclic Hierarchy, Hub like Modularization, Multipath Hierarchy, Unnecessary Abstraction, Missing Hierarchy, Multifaceted Abstraction, Feature-Envy, Unutilized Abstraction, Rebellious-Hierarchy, Deficient Encapsulation, Broken Hierarchy, Unexploited Encapsulation, Insufficient-Modularization, Cyclically Dependent-Modularization, and Deep Hierarchy.
Implementation Smells	Long Method, Complex Method, Long-Parameter List, Long Identifier, Long-Statement, Complex Conditional, Abstract-Function Call from Constructor, Empty Catch-Clause, Magic Number, and Missing Default.

TABLE 2: The list of author metrics used in the study and their descriptions.

Author Metrics	Description
Contribution	Defines the code churn of the developer.
Timezone	Describes the primary timezone of a developer.
Experience	Describes the time that a developer contributes to a project.
Commits	Defines the number of commits submitted by a developer.
Work Time	Explains the primary time of commit submission (*e.g.,* 14:00).
Refactoring Density	Describes the density of contribution toward refactoring.

level. Therefore, we use Designite to measure code smells at the start and end of each development stage. This helps us evaluate the frequency and types of code smells before and after implementing each refactoring tactic or rhythm. Given that development activities are minimal in the early days of a project, we use the first quartile of the early stage as the starting point and consider it the initial quality point for extracting code smells.

2.3 Author and Project Profiles Identification

Different teams may have different author formations with distinct skills. Furthermore, the characteristics of the projects, such as their size, may lead to the adoption of different refactoring strategies. Therefore, it is essential to identify the types of authors and projects in different stages of development to gain insights into the adopted refactoring tactics and rhythms. To this end, we pick a set of metrics to explain the characteristics of the authors and projects. Furthermore, we cluster authors and projects based on the collected metrics and form different profiles for projects and authors. The selected metrics are listed in Tables 2 and 3.

We collect contribution, timezone, experience, commits, work time, contributors count, and the age of the projects by writing a python script and traversing the commit logs. To calculate the refactoring density, we used Rminer to obtain the number of refactoring lines and a bash script to calculate the total code churn (*i.e.,* all lines of code added or deleted in a commit). We then divided the number of refactoring lines by the total code churn. Moreover, we obtain information about files, comments, and lines of code from the Cloc tool [48] and we use GitHub API [49] to fetch stars.

TABLE 3: The list of project metrics used in the study and their descriptions.

Project Metrics	Description
Files	Describes the total number of files.
Comments	Defines the lines of comments added to the codebase.
Lines of Code	Describes all lines of codes written in Java.
Contributors	Describes the total number of known contributors.
Timezones	Defines the number of different timezones of the developers contributing to the project.
Commits	Describes the total number of commits.
Age	Defines the length between the first commit and the last commit in days.
Stars	Describes the popularity of a project in terms of stars gained.
Refactoring Density	Describes the density of refactoring in a repository.

Highly correlated metrics are linearly related and can be expressed by each other. Furthermore, redundant metrics can be derived from other metrics. Having highly correlated or redundant metrics makes it difficult to analyze the impact of the metrics [50]. Thus, we perform correlation analysis and redundancy analysis to remove correlated and redundant metrics.

- ***Correlation analysis:*** We find that the author and project profile metrics do not follow a normal distribution, thus we use Spearman's rank correlation coefficient to find the correlation between the computed metrics. A coefficient of > 0.7 represents a high correlation [51]. For each pair of highly correlated metrics, we remove one metric and keep the other in our model. Figure 2 shows a dendrogram representing the correlation between the project and author metrics.
- ***Redundancy analysis:*** R-square is a measure that shows how variance of a dependent variable can be explained by independent variables [52]. We use an R-squared cut-off of 0.9 to identify redundant metrics that can be estimated from other metrics.

We measure highly correlated and redundant metrics in the project and developer profiles metrics and exclude them from the studied metrics. The correlation (Figure 2-A) analysis reveals that author's *commits* and *experience* are highly correlated. Likewise, in project metrics, *files*, *comments*, and *lines of codes* are highly correlated (Figure 2-B). Hence, we remove project's *comments*, project's *lines of codes*, and author's *experience*. After removing highly correlated metrics from the clustering set, we performed redundancy analysis, but no redundant metrics were identified.

After removing the highly correlated and redundant metrics, based on the distribution of each metric in different quartiles, we divide them into four groups and label them as *Least*, *Less*, *More*, and *Most* [53]. We use k-mods clustering, an extension of the k-means [54] clustering algorithm, which is suitable for clustering categorical data, to cluster the labeled metrics and cluster them into different profiles. We use elbow method [55] to find the optimal number of clusters (k) and manually validate and check if

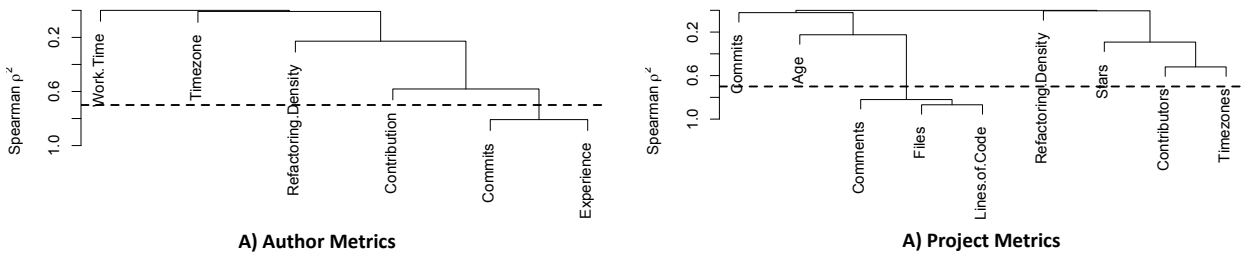

Fig. 2: The results of the correlation analysis of author and project profile metrics.

the clustering results provide distinct centroids. The elbow method involves plotting a graph that displays the number of clusters versus the sum of squared errors (SSE) for each cluster. The optimal number of clusters is identified by the point on the graph where the SSE begins to level off and form an elbow shape [56].

For Author profiles, we identify 3 as the optimal number of clusters with the optimal cost value of 62,872. Therefore, we apply k-mods with 3 clusters (k=3) and identify three major profiles. Based on the selected metrics (*i.e.*, timezone, contribution, refactoring density, commits, and work time), we label the three identified clusters as *main authors*, *casual contributors*, and *core authors*. The *core authors* make the most contributions while the *casual contributors* make the least contributions. The *main* and *core* authors are primarily located in America and Western Europe with commits between 12:33 to 17:06 at their local time (14:57-17:06 for *main authors* and 12:33-14.57 for *core authors*). Casual contributors are primarily located in North America and commit from 17:06 to 24:00 (midnight).

For the Project profiles, we find four clusters that provide different meaningful clusters with the optimal cost value of 919,000. Hence, we apply k-mods clustering with k=4 and identify four major profiles, namely, *vibrant*, *maintaining*, *obsolete*, and *growing* projects. *Vibrant* projects have the most commits, contributors, and stars while *obsolete* projects experience the least refactoring density along with less commits, most age, and more stars. Furthermore, *growing* projects experience least refactorings with the least stars while having more commits and least contributors. Moreover, *maintaining* projects share least commits and contributors with the most refactoring density. Tables 4 and 5 summarize the results of author and project clustering.

2.4 Research Methods

This section presents the research methods applied to answer the research questions.

2.4.1 Refactoring rhythms identification

To identify the refactoring rhythms of the projects, we require a measure to describe the amount of refactoring activities (*i.e.*, refactoring churn) applied on each day of the week. As the refactoring activities could be reflected by the amount of code changes caused by refactoring, we use the refactoring churn to quantify the amount of refactoring activities and normalize it by the actual code churn. Therefore, we introduce daily refactoring density (DRD), which indicates the amount of refactoring activities deviated from the overall development (*e.g.*, the total code churn) of each day of development. The DRD is calculated as below:

$$\text{DRD}(i) = \frac{\text{Refactoring churn of the day } (i)}{\text{Total code churn of the day } (i)} \quad (1)$$

We measure the daily refactoring densities (*i.e.*, DRD) and compare them throughout the week. We form seven groups, each of which represents one day of the week and contains all refactoring activities that occurred on that particular day. Doing so allows us to find the similarities and differences of refactoring activities from one day to another. As our data does not follow a normal distribution, to measure the significance of the differences or similarities of the measured DRDs among different days, we use the Kruskal-Wallis test, an extension of Mann–Whitney U test [57] that evaluates if two or more samples come within the same distribution [30]. The Kruskall-Wallis test does not assume that the data is normally distributed or not. We use p-value > 0.05 to decide if a test of null-hypothesis is significant [58]. The null-hypothesis is a statistical theory that measures if a significant relationship exists between two sets of data [59].

2.4.2 Refactoring tactics identification

Previous studies show that the majority of projects utilize agile methodologies, which require small tasks to be finished within one week of development [58], [60]. To understand the long-term refactoring activities applied by developers in the long run, we need a measure to understand the amount of refactoring churns compared to the development per week of the development. Hence, we propose a weekly refactoring density WRD metric, which reflects the amount of refactoring activities per week of development. The WRD metric is computed using the following formula:

$$\text{WRD}(i) = \frac{\text{Refactoring churn of the week } (i)}{\text{Code churn of the week } (i)} \quad (2)$$

For each project, we create a time series of WRDs within every stage of development. Each data point of the time series includes a week of development and the corresponding WRD metric for that week. Accordingly, the refactoring time series data depicts refactoring behaviors over time. Therefore, similar refactoring time series between different stages of the projects represent a similar refactoring habit. The created refactoring time series share different lengths and they vary in speed. For instance, the development period of Project A is 10 weeks, while that of Project B is 20 weeks, indicating a variation in their length. Additionally, Project A experiences a refactoring spike in the second week

TABLE 4: The clusters identified by K-mods clustering to identify author profiles.

Label	Timezone	Contribution	Refactoring Density	Commits	Work Time
Main Authors	Less (-5.00, 0.00]	More (577.00, 9,848.0]	Most (0.19, 1.00]	More (7.0, 63.0]	More (14:57, 17:06)
Casual Contributors	Least [-12.00, -5.00]	Least [0.00, 23.00]	Least [0.00, 0.00]	Least [0, 2]	Most (17:06, 24:00]
Core Authors	Less (-5.00, 0.00]	Most (9,848.0, ∞)	More (0.06, 0.19]	Most (63.0, ∞)	Less (12:33, 14:57]

TABLE 5: The clusters identified by K-mods clustering to identify project profiles.

Label	Files	Contributors	Timezones	Commits	Age	Stars	Refactoring Density
Vibrant	Most (2157.75, ∞)	Most (44.00, ∞)	Most (10.75, ∞)	Most (3,450.75, ∞)	Less (902.77, 1,684.16]	Most (237.25, ∞)	More (0.15, 0.20]
Maintaining	Least [0.00, 530.75]	Least [0.00, 15.00]	Least [0.00, 1.00]	Least [0.00, 951.00]	Less (902.77, 1,684.16]	Less (1.00, 38.00]	Most (0.20, 1.00]
Obsolete	Less (530.75, 1,242.00]	More [27.00, 44.00]	More [5.00, 10.75]	Less (951.00, 1,702.00]	Most (2,096.51, ∞)	More (38.00, 237.25]	Least [0.00, 0.10]
Growing	More [1242.00, 2157.75]]	Least (15.00, 27.00]	Least [0.00, 1.00]	More (1,702.00, 3,450.75]	More (1,684.16, 2,096.51]	Least [0.00, 1.00]	Least (0.00, 0.10]

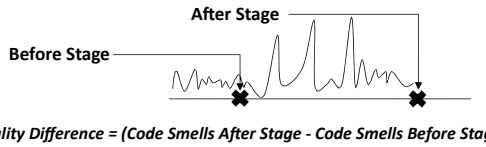

Fig. 3: How changes in code smells are calculated after each stage of development in each project.

of development, while Project B experiences the same spike in the fifth week of development, indicating a variation in the speed of refactoring activities. Therefore, comparing the refactoring time series with point-to-point measures, such as euclidean distance [61], could not overcome the limitations of speed and length variation and could not identify similarities optimally. Therefore, we use the dynamic time warping (DTW) algorithm to measure the similarity between the refactoring time series of project stages as refactoring tactics. DTW overcomes the limitation of point-to-point comparisons with the step pattern that allows transitions and weights between two pairs [62]. Moreover, DTW is an algorithm for calculating the similarity between two time series that vary in speed [63].

2.4.3 Quality Changes Measurement

For identifying the relationship between the refactoring rhythms or tactics and code quality, we need to analyze how these refactoring rhythms or tactics reduce or increase the frequency of each type of code smell. To do this, we use the boundary points before and after each stage of development. As it is shown in Figure 3, we measure the frequency of each code smell type before and after each stage (*i.e.,* between two consecutive stage boundaries) and measure the difference (reduction or increase) in each type of code smell.

Since a larger codebase could contain more code smells, we normalize the frequency of code smells by the lines of code (LOC) in the codebase. Additionally, since more code changes (*i.e.,* larger code churn) may lead to larger differences in code smells, we normalize the difference in code smells by the code churn within each stage. Finally, we calculate the differences in each type of code smell and label it according to the identified rhythm and tactic adopted in that stage. We utilize the frequency of code smells at the end of each stage (ECS), the lines of code at the end of each stage (ELC), the initial frequency of code smells in each stage (ICS), the lines of code at the beginning of each stage (ILC), and the total code churn of each stage (CC) to measure the normalized differences of the total frequency of code smells of each stage (CSD). To measure CSD at each stage (i), we use the following equation:

$$\mathrm{CSD}(i) = \frac{(\mathrm{ECS}(i)/\mathrm{ELC}(i) - \mathrm{ICS}(i)/\mathrm{ILC}(i))}{\mathrm{CC}(i)/\mathrm{ELC}(i)} \quad (3)$$

An increase in CSD indicates an increase in the number of code smells (*i.e.,* decreased code quality) and a decrease in CSD indicates an increase in the number of code smells (*i.e.,* increased code quality). To measure the smell difference after utilizing each refactoring rhythm or tactic, we require a multiple comparison method to cluster and rank the identified rhythm or tactic into statistically significant groups and rank them based on CSD metric (*i.e.,* changes in code quality). Therefore, we use the Scott-Knott-ESD [31], [32] test that uses a multiple comparison method that divides and ranks a set of input distributions into statistically distinct groups [32]. Scott-Knott-ESD is an extension of the Scott-Knott [64] test with the addition of effect size difference. The effect size examines the strength of the difference between different groups of data [65]. Therefore, we cluster and rank the refactoring rhythms and tactics based on the CSDs and identify the rhythms or tactics leading to a codebase with an increased or decreased amount of code smells.

3 Results

In this section, we provide the motivation, approach, and findings for each of our research questions.

3.1 RQ1. What are the rhythms of refactoring?

3.1.1 Motivation

In software development, developers can have various working rhythms. For example, some developers prefer

to work only on workdays; however, others do not mind working even on weekends. Existing studies focus on development rhythms and categorize development rhythms as *work-day* (*i.e.,* Monday to Friday) development and *all-day* (*i.e.,* Monday to Sunday) development [25], [26], [27]. Prior research reports that the state of getting recovered during the weekend from working on the workdays is correlated with an increase in weekly task performance and personal initiative [25]. Moreover, the state of working overtime is associated with a decrease in productivity [66]. Furthermore, previous studies have suggested that deviating from regular development to perform refactoring may help address unhealthy code and potentially improve code quality [28]. Inspired by prior work, our intuition is that providing dedicated time for refactoring outside of regular development cycles enables developers to focus more on addressing unhealthy code through refactoring, resulting in improved code quality. Thus, we study the refactoring rhythms based on their deviations from the development rhythms. Understanding the refactoring deviations from the regular development rhythms and their relationship with the code quality improvement can assist software teams and developers to (1) understand the existing refactoring rhythms and (2) adopt/apply the most effective refactoring rhythms. In this research question, we investigate and characterize different refactoring rhythms to help developers understand the existing refactoring rhythms and identify which one suits their needs.

3.1.2 Approach

As described in Section 2.4.1, to identify the refactoring rhythms of the studied projects, we form seven groups and measure DRDs on every day of development. Moreover, we compare DRDs throughout the week to discover refactoring rhythms using the Kruskal-Wallis test [30].

Prior studies [25], [26], [27] divide the software development process into two groups— *work-day* development and *all-day* development (including workdays and weekends). To identify the refactoring rhythms adopted in different stages and different projects, we need to compare different days of refactoring. Hence, we group DRDs into seven groups based on the weekdays, where each group represents a weekday and depicts the overall DRD distribution on the corresponding day of the week. Using the Kruskal-Wallis test, we first identify whether we can fit the majority of the rhythms adopted from the selected project stages into two groups, namely *work-day* refactoring and *all-day* refactoring. To do this, we perform two individual tests using the following hypothesis:

- ***$H_{0\text{-}1}$:*** *Refactoring densities are similar among all days of the week.*
- ***$H_{0\text{-}2}$:*** *Refactoring densities are similar among all workdays of the week*

To this end, after running the first test, we exclude the stages of projects that have a similar distribution of refactoring on all days of the week and perform the second test. We accept a hypothesis if the p-value is higher than 0.05 and reject otherwise.

Clustering project and developer profiles in terms of refactoring activities. To understand the distribution and significance of the refactoring rhythms across different project and author profiles, we rank the combinations of the different refactoring rhythms and project or author profiles. Using the Scott-Knott-ESD [31], [32], we group combinations of refactoring rhythms and project or author profiles into statistically significant clusters. Specifically, we perform two separate Scott-Knott-ESD clustering analyses: (1) for combinations of project profiles and refactoring rhythms, and (2) for combinations of author profiles and refactoring rhythms.

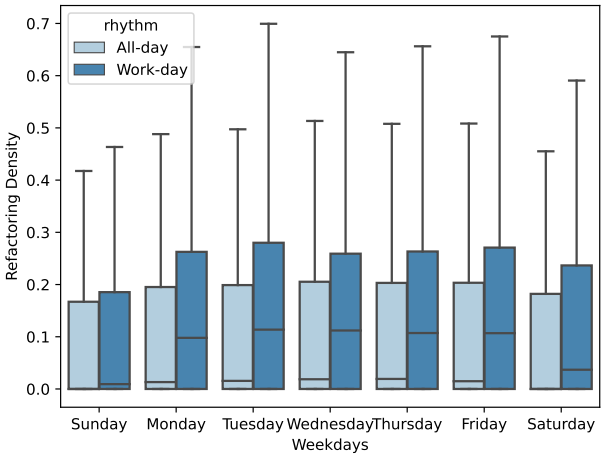

Fig. 4: Comparison of refactoring density between *work-day* and *all-day* refactoring rhythms.

In the clustering method, each clustered item (*i.e.,* a node) represents the distribution of projects or authors that are associated with a specific refactoring rhythm. Moreover, each clustered item is represented by a vector of the same length as the number of projects or authors, with each project or author being assigned a value of 1 if it is associated with the specific project or developer profile and the specific refactoring rhythm, and a value of 0 otherwise. For example, All-day-Vibrant refers to the distribution of the *all-day* refactoring rhythm across all projects that fall under the *vibrant* project profile. Each cluster corresponds to a statistically significant distribution of the various combinations of refactoring rhythms and project or author profiles across the dataset. The results of the Scott-Knott-ESD test provide insights into how refactoring rhythms are distributed across different profiles.

Identifying refactoring rhythms characteristics. To study the differences between refactoring operations (*e.g.,* pull up method) performed on weekends and those performed on workdays, we use the Mann–Whitney U test [57] to compare the distribution of each refactoring operation on the weekend and workdays. We utilize Cliff's Delta to measure the effect size of the differences. We consider the operations that obtain a p-value < 0.05 and an effect size > 0.33 [67], indicating a medium or large magnitude of difference, as the operations that are performed significantly differently between the weekends and the workdays.

3.1.3 Findings

The majority (95%) of project stages follow one of the work-day or all-day refactoring rhythms. We accept the null hypothesis $H_{0\text{-}1}$ for 84% of the project stages and the null

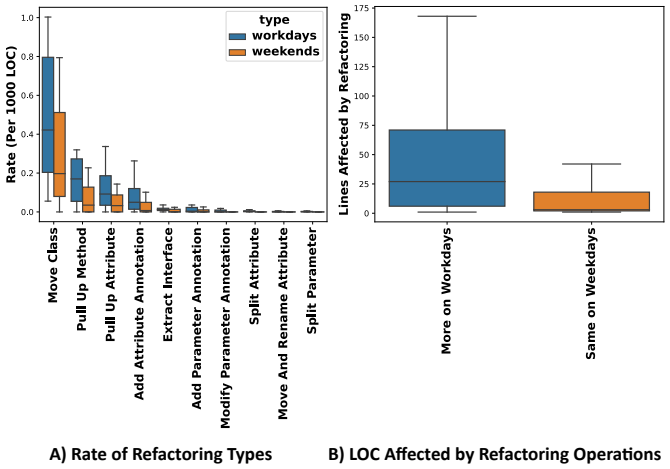

Fig. 5: The different refactoring operations and the lines of refactored code applied in weekend compared to weekdays in *all-day* refactoring rhythm.

hypothesis $H_{0\text{-}2}$ for 11% of the remaining project stages. For the 5% of the project stages, both null hypotheses $H_{0\text{-}1}$ and $H_{0\text{-}2}$ are rejected. Therefore, our analysis shows that only a few project stages (*i.e.,* 5%) do not follow any of the initial refactoring rhythms. Specifically, we find that 11% of the project stages perform *all-day* refactoring, whereas 84% of the project stages perform the *work-day* refactoring rhythm.

In the *work-day* refactoring rhythm, we observe a significant difference in refactoring densities between workdays and weekends, with the median refactoring densities being higher in workdays compared to weekends, as illustrated in Figure 4. Additionally, certain types of refactoring are applied differently between weekends and workdays, as shown in Figure 5-A. These types of refactoring include *move class*, *pull up method*, *pull up attribute*, *add attribute annotation*, *extract interface*, *add parameter annotation*, *modify parameter annotation*, *split attribute*, *move and rename attribute*, and *split parameter*. These refactoring actions mainly relate to the class/method level and play a crucial role in shaping the overall system design. Therefore, developers may prefer to perform complex design-level refactorings on workdays, leaving weekends for less risky modifications. We observe that the median lines of the codes affected by the workday refactoring types (*i.e.,* the refactoring types that are applied more on workdays) are higher compared to weekday refactoring types (*i.e.,* the refactoring types that are applied similarly during workdays and weekends) (Figure 5-B). Therefore, developers apply more heavy-weight refactoring operations that involve more code changes (*e.g., move class*) during the workdays compared to the weekends.s In contrast, in the *all-day* refactoring rhythm, the Kruskal-Wallis test results demonstrate that there is no statistically significant difference in the density of refactoring among the different days of the week, and the median refactoring densities are consistent across all days of the week, as depicted in Figure 4. Additionally, we do not observe a significant difference in the refactoring operations performed on weekends compared to workdays. Therefore, it appears that developers exert an equal amount of effort towards refactoring throughout the week in the *all-day* rhythm.

TABLE 6: Scott-Knott-ESD test results on the refactoring rhythms and associated project and author profiles.

Project Profiles			Author Profiles		
Rhythm Profile	**Cluster Rank**	**Mean (%)**	**Rhythm Profile**	**Cluster Rank**	**Mean (%)**
All-day Maintaining	1	0.89	**All-day Core**	1	0.79
All-day Growing	1	0.83	**All-day Main**	1	0.80
All-day Vibrant	1	0.83	**All-day Casual**	1	0.76
All-day Obsolete	1	0.82	**Work-day Casual**	2	0.19
Work-day Vibrant	2	0.14	**Work-day Main**	2	0.16
Work-day Obsolete	2	0.12	**Work-day Core**	2	0.15
Work-day Growing	2	0.11			
Work-day Maintaining	3	0.06			

Table 6 shows the results of the Scott-Knott test, providing further insights into the relationship between the project and author profiles with their corresponding rhythms:

Among project profiles: In the *maintaining*, *obsolete*, *growing*, and *vibrant* project stages, the *all-day* refactoring rhythm (Cluster 1) is often used with a similar distribution than the *work-day* refactoring rhythm with over 82% utilization practice (Clusters 2 and 3). In *vibrant*, *obsolete*, and *growing* project stages *work-day* refactoring rhythm (Cluster 2) is more frequently used compared to the *maintaining* project stages. *Maintaining* project stages experience the most refactoring density and the least number of commits (Table 3). Moreover, in *Maintaining* project stages, the usage of the *all-day* refactoring rhythm is highest, at 89% (Cluster 1), compared to the *work-day* refactoring rhythm, which is 0.06% (Cluster 3). Therefore, in *maintaining* project stages where there is less development and more refactoring, developers are likely to perform the *work-day* rhythm less frequently and focus on refactoring whenever they have time. The observation that the distribution of the *work-day* refactoring rhythm is similar in *vibrant*, *obsolete*, and *growing* projects (Cluster 2), and the distribution of the *all-day* rhythm is also similar in these project stages (Cluster 1), indicates that the project profile does not have a significant impact on the choice of refactoring rhythm in *vibrant*, *obsolete*, and *growing* project stages.

Among author profiles: *Core*, *main*, and *casual* authors often utilize the *all-day* refactoring rhythm with a similar distribution (Cluster 1). Moreover, the distribution of the *work-day* rhythm is similar in all author profiles (*i.e., core*, *main*, and *casual*) (Cluster 2). Since the distribution of different author profiles in *all-day* and *work-day* rhythms separately are similar, the choice of specific refactoring rhythm is not influenced by the type of developer. Therefore, it is likely that authors choose different rhythms based on their preferences.

Software projects follow two major refactoring rhythms: *work-day* and *all-day* refactoring. The *work-day* refactoring rhythm tends to have higher densities of refactoring to the code base from Monday to Friday. In the *all-day* refactoring rhythm, there is no significant difference in refactoring activities on different days of the week. In *maintaining* project stages, the *all-day* refactoring rhythm is more prevalent compared to other project stages. The choice of refactoring rhythms (*all-day* or *work-day*) is not influenced by the type of authors.

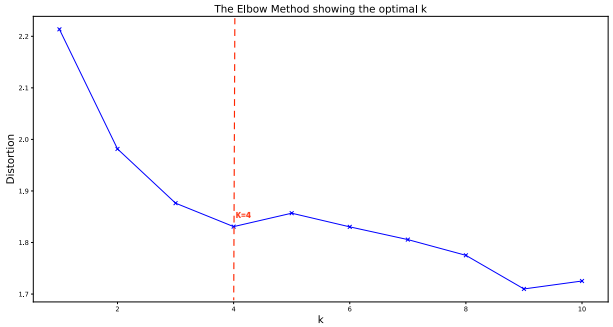

Fig. 6: The results of the elbow curve, showing the optimal number of clusters using DTW.

3.2 RQ2: What are the most frequent refactoring tactics used in projects?

3.2.1 Motivation

Previous studies have only classified refactoring tactics as either *floss* or *root canal* [1], [28], [29]. *Floss* refactoring is distinguished by frequent refactoring along with the development process. On the other hand, *root canal* refactoring is identified by occasional refactoring aside from the development process. While the terms *floss* and *root canal* tactics have been useful in understanding the general patterns of refactoring, there may be other potential refactoring tactics that have not yet been identified. Moreover, understanding the distinctive features of each refactoring tactic can offer valuable insights into developers' decision-making processes when choosing a particular tactic. Additionally, recognizing various refactoring tactics can establish a common vocabulary for describing them, facilitating communication among practitioners. This, in turn, helps developers comprehend the refactoring tactics they use and choose or switch to the most appropriate tactic for their project. In this research question, by considering different stages of development in different projects, we investigate whether there are more refactoring tactics other than *floss* and *root canal*.

3.2.2 Approach

As described in Section 2.4.2, to understand the refactoring tactics in the studied projects, we first cluster refactoring time series of the project stages (*i.e.,* in terms of WRD). Using the DTW algorithm, we measure the similarity between the WRD time series of each pair of project stages as part of the clustering process.

Clustering common refactoring practices. To identify refactoring tactics, we utilize WRD and form a time series that represent the refactoring history of each project. Subsequently, using DTW we cluster the projects based on the similarities of their refactoring activities represented by the time series. As the selected projects do not share a similar life cycle and they may experience different refactoring practices in different stages of development, we measure the similarities of refactoring activities between projects in different stages of development (*i.e.,* early, middle, late). Therefore, if a project has multiple development stages, we break its time series into multiple smaller time series, each of which represents one stage of the project.

We use Dynamic Time Warping (DTW), a clustering technique for temporal sequences based on their similarity, to cluster refactoring time series as refactoring tactics. We identify the optimal number of clusters using the elbow method [55]. The elbow method measures the sum of squared errors (SSE) and selects the smallest value of k (*i.e.,* the number of clusters) with the lowest SSE as the optimal number of clusters. This is determined by identifying the point on the graph where the SSE begins to level off and form an elbow shape [56]. Moreover, we manually validate the optimal number of clusters (k) identified by the elbow method and check if our clustering results provide distinct centroids. Based on the elbow curve analysis depicted in Figure 6, four is identified as the optimal number of clusters. Furthermore, we utilize the silhouette score with the existing criteria [68], [69], [70], [71], [72] of the silhouette method to verify the optimal number of clusters. This identification is based on two conditions: (1) average silhouette score greater than 0.5 and (2) absence of clusters exhibiting all silhouette scores below the average. The silhouette score analysis points towards the optimal cluster numbers being 3 and 4, yielding average silhouette scores of 0.56 and 0.52, respectively. All clusters in both cases exhibit scores above the average threshold. This observation supports the idea that both 3 and 4 clusters could be considered as optimal solutions. However, as k=4 leads to a more even distribution of the sizes (*i.e.,* thicknesses) of the clusters [68], we opt for k=4. In summary, both the elbow curve and silhouette score analyses suggest that 4 clusters are the preferred number of clusters.

We utilize DTW to identify the similarities of refactoring time series by analyzing all stages of development in all projects together. Analyzing all stages together at the same time allows us to compare and identify the unified common behaviors in all stages of development despite their different life spans. By considering the optimal number of clusters as four and performing DTW, we identify four common behaviors based on the cluster centroids to represent the common tactics as variations of the *root canal* and *floss* refactoring tactics.

Identifying refactoring spikes. To provide more insights on the identified tactics, we measure the number of spikes that happen in each refactoring tactic centroid. Our intuition is that a higher number of spikes within a refactoring tactic time series indicates more deviation of refactoring densities from regular refactoring densities. To determine a spike we apply the Median Absolute Deviation (MAD) method. Compared to the standard deviation, MAD is a robust estimator of scale. MAD can also be used as a scaling quantity instead of the standard deviation, which is vulnerable to

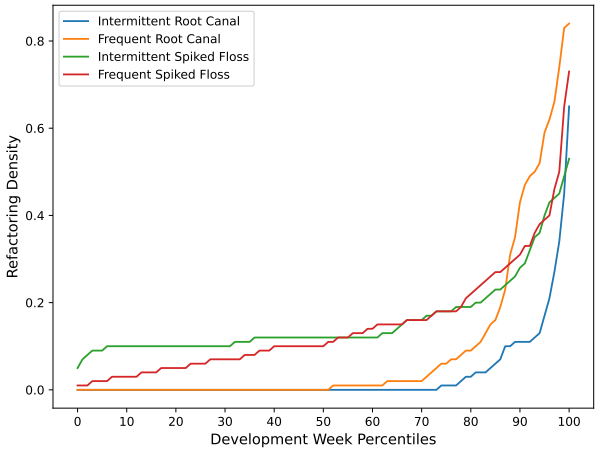

Fig. 7: The relationship between development weeks percentiles and refactoring density.

the influence of extreme values [73], [74]. MAD can be calculated using the formula below where n is each data point and ñ is the median of all data points in a window:

$$MAD = median\left(|n - median(\tilde{n})|\right) \quad (4)$$

Therefore, to detect refactoring spikes, we iterate through the data points in our centroid time series within a window of four weeks (one month) before and after a given index, which represents a week of development. For each window, we calculate the median absolute deviation (MAD). Then, we check if the absolute deviation of the data point at the current index from the median of the window is greater than three times the MAD [75]. If the condition is true, we consider it a refactoring spike.

Clustering refactoring tactics in terms of project and developer profiles. Using a similar approach to Section 3.1.2, we use the Scott-Knott-ESD [31], [32] to cluster the distribution of the project and author profiles associated with the refactoring tactics into statistically significant groups. We perform two separate clusterings for (1) for combinations of project profiles and refactoring tactics, and (2) for combinations of author profiles and refactoring tactics. Each cluster represents the distribution of project or author profiles in project stages corresponding to the identified tactic. For example, RC-Vibrant indicates the distribution of the *root canal* tactic across *vibrant* project stages. Hence, each group represents a statistically significant distribution of the project or author profiles associated with the refactoring tactics. The results of the Scott-Knott-ESD test reveal more insights into the identified refactoring tactics.

3.2.3 Findings

Our clustering approach uncovers four primary refactoring tactics. We define the intermittent spiked floss and the frequent spiked floss as two variations of the floss refactoring tactic, and the intermittent root canal and the frequent root canal as two variations of the root-canal refactoring tactic. The behavior (*i.e.,* changes in refactoring density over time) of refactoring tactics is illustrated in Figure 8. The main difference between the floss-based and root canal-based tactics is that: floss-based tactics mix refactoring with regular development activities (*i.e.,* the refactoring densities are consistently higher than zero, as indicated in Figure 8 (B and C)); in comparison, root canal-based tactics involve refactoring activities once in a while (*i.e.,* the refactoring densities are at or near zero for most of the time periods, as indicated in Figure 8 (A and D). Additionally, Figure 7 shows the refactoring density percentiles of root canal-based and floss-based tactics. As is shown root canal-based tactics have zero refactoring densities for more than half of the development cycle. Specifically, intermittent root canal have zero refactoring density until the 74th percentile of development weeks (*i.e.,* more than 74% of the weeks have zero refactoring densities), and frequent root canal have zero refactoring until the 52nd percentile (*i.e.,* more than 52% of the weeks have zero refactoring densities). In contrast, floss-based tactics have a non-zero median at all percentile points. Moreover, *frequent root canal* and *frequent floss* tend to have more frequent high-density periods than *intermittent spiked floss* and *intermittent root canal*. Furthermore, as it is shown in Table 7, by comparing refactoring spikes count in each tactic, we find that *intermittent spiked floss* has fewer spikes (35) compared to *frequent spiked floss* (59). Similarly, *frequent root canal* has more spikes (66) compared to *intermittent root canal*, which has 35 spikes. Overall, the *frequent root canal* and *frequent spiked floss* tactics exhibit more frequent high-density periods (*i.e.,* refactoring spikes) than the *intermittent root canal* and *intermittent spiked floss* tactics. We define each refactoring tactic as follows:

TABLE 7: Summary of the number of refactoring spikes for each refactoring tactic centroid.

	Floss		Root Canal	
	Intermittent Spiked	**Frequent Spiked**	**Intermittent**	**Frequent**
Spikes Count	35	59	35	66

- **Intermittent spiked floss:** with refactoring consistently in all development weeks, developers perform refactoring on a regular basis along with fewer refactoring spikes compared to *frequent spiked floss.*
- **Frequent spiked floss:** with refactoring consistently in all development weeks, developers perform refactoring with more drops and increases (*i.e.,* spikes) in refactoring density compared to *intermittent spiked floss*.
- **Intermittent root canal:** with the majority of the weeks having zero refactoring densities, developers tend to perform refactoring irregularly but in high densities when they do perform it.
- **Frequent root canal:** with the majority of the weeks having zero refactoring densities, developers tend to perform more frequent refactorings with more spikes in refactoring density compared to *intermittent root canal*.

Software projects undergo different refactoring tactics during their lifetime. Table 8 shows the distribution of the identified tactics in different stages of development. In particular, in the early stage of development, the majority of refactoring tactics are floss-based (55%). This observation is aligned with the previous studies showing that the majority

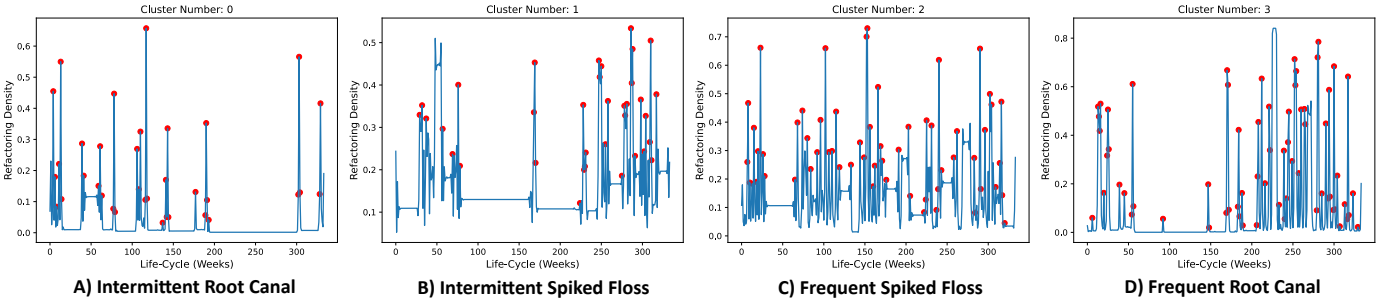

Fig. 8: Clustering centroids that represent refactoring tactics identified in this study, which are labeled as: *intermittent root canal*, *intermittent spiked floss*, *frequent spiked floss*, and *frequent root canal*. The red dots show refactoring spikes in each tactic.

TABLE 8: The distribution of refactoring tactics in different stages of development

	Floss		Root Canal	
	Intermittent Spiked	Frequent Spiked	Intermittent	Frequent
Early	3%	52%	19%	26%
Middle	35%	12%	41%	12%
Late	21%	0%	78%	1%

of refactoring tactics are floss-based refactoring [76], [77]. However, in the middle stage, the utilization of floss-based tactics drops to 47%. Finally, in the late stage of development, the majority of refactorings are observed to be root canal-based tactics (79%). ***Therefore, the amount of floss-based refactoring tactics reduce while the projects enter their later stages of development: the developers aim for more targeted refactoring operations as the projects grow over time.*** Table 9 shows the results of the Scott-Knott-ESD test, which reveals more information on the distribution of refactoring tactics associated with different project and author profiles:

Among project profiles: In *maintaining* project stages, intermittent root canal (cluster 1) is frequently used (60%). While, in vibrant project stages, floss-based tactics (*frequent spiked floss* and *intermittent spiked floss*) (cluster 2) are more frequently used (79%). Moreover, the *obsolete* project stages mainly use intermittent root canal (cluster 2) and *growing* project stages (cluster 2) utilize *frequent spiked floss* and *intermittent root canal* as the main refactoring tactics. As Vibrant project stages have more and most contributors and commits, they have more active development activities; thus they are more likely to experience frequent refactoring during the development process (*i.e.,* floss-based tactics). However, *maintaining* and *obsolete* project stages with the least contributors try to maintain the code and keep it working by doing targeted refactoring from time to time.

Among author profiles: *casual* developers mainly do floss-based refactoring tactics (cluster 1 and 2) (*i.e., frequent spiked floss* and *intermittent spiked floss*), while *main* authors often utilize *frequent spiked floss* (cluster 1). Moreover, *core* authors utilize both *frequent spiked floss* and *intermittent root canal* (cluster 2). As *core* authors have the most and more contributions to the repository, they are more likely to contribute more to critical refactoring activities; therefore, they are more likely to do root canal-based refactoring, while letting *casual* contributors perform floss-based refactoring during the development process.

TABLE 9: Scott-Knott-ESD test results on the refactoring tactics and associated project and author profiles.

Project Profiles			Author Profiles		
Tactic Profile	Cluster Rank	Mean (%)	Tactic Profile	Cluster Rank	Mean (%)
IR-Maintaining	1	0.60	**FF-Casual**	1	0.47
IR-Obsolet	2	0.42	**FF-Main**	1	0.42
FF-Growing	2	0.42	**FF-Core**	2	0.35
FF-Vibrant	2	0.41	**IR-Core**	2	0.31
IR-Growing	2	0.39	**IF-Casual**	2	0.30
IF-Vibrant	2	0.38	**IF-Main**	3	0.25
FF-Obsolete	3	0.30	**IR-Main**	3	0.24
FR-Maintaining	4	0.25	**IR-Casual**	4	0.19
FR-Obsolete	4	0.19	**IF-Core**	4	0.18
IR-Vibrant	4	0.18	**FR-Core**	4	0.16
FR-Growin	5	0.11	**FR-Main**	5	0.09
FF-Maintaining	5	0.09	**FR-Casual**	6	0.04
IF-Obsolete	5	0.09			
IF-Growing	5	0.08			
IF-Maintaining	6	0.06			
FR-Vibrant	6	0.03			

IR: intermittent root canal, IF: intermittent spiked floss, FF: frequent spiked floss, FR: frequent root canal

Apart from *floss* and *root canal* refactoring tactics, software developers use more diverse refactoring tactics, such as *intermittent root canal*, *intermittent spiked floss*, *frequent spiked floss*, and *frequent root canal*. Among project stages categorized as *obsolete* or *maintaining*, root canal-based tactics are prevalent, whereas floss-based tactics are more commonly employed in *vibrant* stages. Additionally, *core* authors tend to use more root canal-based tactics, whereas casual contributors are inclined towards floss-based tactics.

3.3 RQ3: What is the relationship of different refactoring rhythms and tactics with code quality?

3.3.1 *Motivation*

In the first and second research questions, we identify different refactoring tactics and rhythms applied by different projects. Apart from finding different tactics and rhythms, identifying the relationship between refactoring tactics and code quality is crucial as it helps developers prioritize their efforts, improve development processes, and deliver high-quality software. Therefore, we utilize code smells as quality metrics for refactoring [13], [16] to compare the changes in quality after adopting each tactic or rhythm. Understanding the relationship of different rhythms and tactics with code quality can help practitioners and project teams to (1) discover the positive and negative aspects of the different

refactoring rhythms or tactics, and (2) adopt or switch to the most suitable refactoring rhythm or tactic.

3.3.2 Approach

To understand the relationship of the identified refactoring rhythms and tactics with code quality, we utilize code smells listed in Table 1 as code quality metrics. Using Scott-Knott-ESD test [31], [32], we cluster the magnitude of code smell changes (*i.e.,* increase/decrease) after adopting each tactic or rhythm. The Scott-Knott-ESD complements the Scott-Knott test [64] by taking the effect size difference into account when identifying different clusters. We first identify the relationship between the identified (1) refactoring tactics and (2) refactoring rhythms with overall increase or decrease in code smells as quality measures. Moreover, we identify the relationship of each refactoring tactic and rhythm with different types of code smells. We describe our detailed approach below.

Measuring code smell changes. As discussed in Section 2, to measure the relationship of the identified refactoring rhythms and tactics with code quality, we use code smells. We set three stages (early, middle, and late) for the lifetime of projects and then collect the code smell metrics, which are normalized by the project size, at the beginning and the end of each stage respectively.

The relationship of refactoring rhythms and tactics with the overall code quality. In Section 3.1, we classify refactoring rhythms as *all-day* and *work-day*. Besides, in Section 3.2, we identify four major refactoring tactics: *intermittent root canal*, *intermittent spiked floss*, *frequent spiked floss*, and *frequent root canal*. To identify the relationship between the above refactoring rhythms and tactics with code quality, we utilize the normalized frequency of code smell changes after adopting each rhythm and tactic (listed in Table 1). Therefore, a higher frequency of changes indicates an increase in code smells and a decrease in software quality. To analyze the overall relationship of refactoring tactics and rhythms with the frequency of code smell changes, we use the normalized sum of all code smell changes as the overall changes in code smells and label them with the corresponding rhythm and tactic. Using the Scott-Knott-ESD test we cluster and rank the refactoring rhythms and tactics based on the code smell changes to identify the rhythms and tactics leading to more smelly code. We use $p - values < 0.05$ to identify the statistical significance and use means to rank the identified clusters.

The relationship of refactoring rhythms and tactics with each code smell type. To provide more insights and details on the identified rhythms and tactics, we conduct separate analyses to assess the relationship between the frequency of different types of code smells and each refactoring rhythm and tactic. We use 35 code smells (listed in Table 1) and the changes after adopting each rhythm and tactic. To this end, we utilize the Scott-Knott-ESD test to cluster ($p - values < 0.05$ as the significance threshold) and rank (using means) the rhythms and tactics based on each type of code smells separately. Therefore, we perform 35 individual tests for rhythms and 35 separate tests for tactics. Therefore, each Scott-Knott-ESD test is responsible for one type of code smells. Doing so allows us to identify how the refactoring rhythms and tactics impact each type of code smells.

TABLE 10: Scott-Knott-ESD test results on the overall changes in the frequency of code smells associated with the refactoring tactics.

	Floss		**Root Canal**	
	Intermittent Spiked	**Frequent Spiked**	**Intermittent**	**Frequent**
Cluster Rank	1	2	3	3
Mean	0.198276	0.012329	-0.025959	-0.029701

TABLE 11: Scott-Knott-ESD test results on the overall changes in the frequency of code smells associated with the refactoring rhythms.

	All Day	**Work Day**
Cluster Rank	1	1
Mean	0.01352	0.01991

3.3.3 Findings

In this section, we provide the findings on the relationship of both refactoring rhythms and tactics with code quality.

Overall relationship: We use the sum of all types of code smell changes (*i.e.,* the total number of code smell changes regardless of the code smell type) to measure the overall code smell changes after adopting each refactoring rhythm and tactic. As the results suggest, the identified rhythms belong to the same cluster and do not significantly affect overall changes in the number of code smells (Table 11), however, the *all-day* refactoring rhythm is associated with the lowest mean in overall code smell changes, which indicates a higher code quality. Overall, refactoring rhythms are not statistically associated with the overall changes in the code smells. For refactoring tactics, on the other hand, *intermittent spiked floss* and *frequent spiked floss* are in the first and second ranked clusters, hence, they are associated with more increase in the overall changes of code smells compared to the *frequent root canal* and the *intermittent root canal* tactics. ***In fact, on average, floss-based tactics are associated with an increase in the frequency of code smells (positive mean as shown in Table 10), while root canal-based tactics are associated with a decrease in the frequency of code smells (negative mean as shown in Table 10).*** Therefore, root canal-based tactics (*i.e., frequent root canal* and *intermittent root canal*) are associated with a higher code quality compared to floss-based tactics. A possible explanation may be that floss-based refactoring is typically integrated with addressing daily maintenance tasks, such as bug fixes and the implementation of new features, while root canal-based refactoring focuses on improving the overall quality of the design.

Relationship with specific code smells: The results from our analysis of the overall changes in the number of code smells show a significant difference in the code smell changes between the floss-based and the root canal-based tactics. However, rhythms do not show a significant difference in the changes in the frequency of code smells. Therefore, we cluster the frequency of code smell changes in each code smell type after adopting each refactoring tactic separately. Figure 9 shows the results of the individual Scott-Knott tests applied for each type of code smell. We observe that, for 6% (2 out of 35 code smell types),

Code Smell	Clusters				Code Smell	Clusters				Code Smell	Clusters			
Cyclic Dependency	IF(1)	FF(2)	IR(3)	FR(3)	Deficient Encapsulation	IF(1)	FF(2)	IR(3)	FR(3)	Wide Hierarchy	IF(1)	FF(2)	IR(3)	FR(3)
God Component	IF(1)	FF(2)	IR(3)	FR(3)	Unexploited Encapsulation	IF(1)	FF(2)	FR(3)	IR(3)	Abstract Function Call from Constructor	IF(1)	FF(2)	FR(3)	IR(3)
Ambiguous Interface	FF(1)	FR(1)	IF(1)	IR(2)	Broken Modularization	FF(1)	IF(1)	FR(2)	IR(2)	Complex Conditional	IF(1)	FF(2)	FR(3)	IR(3)
Feature Concentration	IF(1)	FF(2)	FR(3)	IR(3)	Cyclically Dependent Modularization	IF(1)	FF(2)	IR(3)	FR(3)	Complex Method	IF(1)	FF(2)	IR(3)	FR(3)
Unstable Dependency	IF(1)	FF(2)	IR(3)	FR(3)	Hub-like Modularization	IF(1)	FF(2)	IR(3)	FR(3)	Empty Catch Clause	IF(1)	FF(1)	FR(1)	IR(1)
Scattered Functionality	IR(1)	IF(1)	FF(1)	FR(1)	Insufficient Modularization	IF(1)	FF(2)	IR(3)	FR(3)	Long Identifier	IF(1)	FF(2)	IR(3)	FR(4)
Dense Structure	IF(1)	FF(2)	IR(2)	FR(2)	Broken Hierarchy	IF(1)	FF(2)	FR(3)	IR(3)	Long Method	IF(1)	FF(2)	IR(3)	FR(3)
Imperative Abstraction	IF(1)	FR(2)	FF(2)	IR(3)	Cyclic Hierarchy	IF(1)	FF(2)	IR(3)	FR(3)	Long Parameter List	IF(1)	FF(2)	IR(3)	FR(3)
Multifaceted Abstraction	IF(1)	FR(1)	FF(1)	IR(1)	Deep Hierarchy	FF(1)	IF(1)	IR(2)	FR(2)	Long Statement	IF(1)	FF(2)	FR(3)	IR(3)
Unnecessary Abstraction	IF(1)	FF(1)	FR(1)	IR(2)	Missing Hierarchy	IF(1)	FF(2)	FR(3)	IR(3)	Magic Number	IF(1)	FF(2)	IR(3)	FR(3)
Unutilized Abstraction	IF(1)	FF(1)	FR(2)	IR(2)	Multipath Hierarchy	IF(1)	FF(1)	IR(2)	FR(3)	Missing Default	IF(1)	FF(2)	IR(3)	FR(3)
Feature Envy	IF(1)	FF(1)	FR(2)	IR(2)	Rebellious Hierarchy	FF(1)	IF(2)	IR(3)	FR(3)	Overall	IF(1)	FF(2)	IR(3)	FR(3)

Intermittent Spiked Floss | Frequent Spiked Floss | Intermittent Root Canal | Frequent Root Canal

Fig. 9: Results from the Scott-Knott-ESD tests that cluster and rank the refactoring tactics for overall and different types of code smell changes. A higher rank indicates a larger increase (or smaller decrease) in code smells.

namely empty catch clause and multifaceted abstraction, the different refactoring tactics are not associated with a statistically significant difference in the frequency of the corresponding code smell. This was determined through the Scott-Knott-ESD tests resulting in a single cluster. However, root canal-based tactics result in statistically smaller increases in the frequency of code smells, indicating higher quality, for 80% (28 out of 35) of code smell types. This includes 90% of the implementation smell types (9 out of 10 types), 83% of the design smell types (15 out of 17 types), and 57% of the architecture smell types (4 out of 7 types). The five remaining code smell types (*i.e.,* ambiguous interface, scattered functionality, dense structure, imperative abstraction, and unnecessary abstraction) show slightly different clustering results (Figure 9) from the majority of the code smells (74%). Therefore, adopting root canal-based tactics results in the majority of improvements in code smells across all three categories (*i.e.,* implementation, design, and architecture smells) of code smells. Overall, our results suggest that more dedicated refactoring efforts (i.e., using root canal-based tactics) can better help remove or fix most types of code smells.

Root canal-based tactics are associated with a greater decrease (or smaller increase) in the number of code smells, and thus higher code quality, compared to floss-based tactics, which suggests more dedicated refactoring operations. However, refactoring rhythms are not associated with the changes in the number of code smells, suggesting that the choice of rhythm may be driven more by project-specific factors and team preferences rather than their impact on code quality.

4 THREATS TO VALIDITY

In this section, we discuss the possible threats to the validity of our study.

Internal validity. Concerning our project selection and selected approaches, in the second research question, for clustering refactoring time series and finding refactoring tactics, we analyze the refactoring densities in three stages of development. We choose the mentioned time frames so that we could compare the refactoring behaviors of the project stages with similar length of development history. We admit having more projects with different lengths of time frames could reveal more refactoring tactics. Moreover, due to the varying lengths of the life cycles of projects in stages after the late refactoring stage, time series clustering could not be applied, and we had to exclude them from our study. Thus, it is possible that some patterns may emerge in later stages that we were unable to capture. In the second research question, We have categorized the data into four distinct clusters, namely *intermittent root canal*, *intermittent spiked floss*, *frequent spiked floss*, and *frequent root canal*. The number of clusters chosen may impact the quality and comprehensibility of the clustering outcomes, as well as the insights and conclusions derived from them. If there are too many clusters, it may result in overfitting, whereas if there are too few, important information may be lost. To avoid bias, we use the elbow method [55], silhouette score [72], and manual inspection to identify the optimum number of clusters. However, different numbers of clusters could reveal less or more refactoring tactics. In the third research question, we use code smells as a code quality measure to study the relationship between refactoring rhythms or tactics with code quality. Nevertheless, we agree that code quality can be characterized by other measures, such as the number of bugs or maintenance costs. Furthermore, we admit that other socio-technical metrics, such as the way refactoring is applied (*e.g.,* manually or automatically) and regulations of the development team could affect our code quality measurement. Future work that explores the relationship between refactoring activities and other characteristics of code quality could complement our results.

External validity. Concerning the generalization of our findings, our experiments and results are based solely on the analysis of the 196 Apache projects we studied, and therefore, our conclusions may not necessarily apply to other projects, such as those in different domains. Additionally, since our analysis was limited to projects written in Java, the findings may not be applicable to projects written in other programming languages.

Construct validity. Concerning our measurement accuracy, in the third research question, to study the relationship between refactoring rhythms and code quality (*i.e.,* in terms of the frequency of code smell changes), we measure the code smells in different stages of the projects, because calculating code smells every week takes approximately 25 days for each project, and computing them for all projects requires a significant amount of time. Nevertheless, extracting quality changes every week could provide more accurate results.

5 IMPLICATIONS

In this section, based on the results of our study, we provide implications for practitioners, developers, and tool builders to improve their understanding of different refactoring rhythms and tactics.

Our findings help practitioners understand the patterns of refactoring activities and their impact on code quality, which can help practitioners make more informed decisions in their refactoring adoption. In this study, we identify the deviations of code refactoring from the regular development rhythms and measure how different refactoring rhythms and tactics are associated with the increase or decrease in code quality. Our findings can help practitioners understand practical refactoring rhythms and tactics in different real-world projects and observe their relationship with code quality. Hence, practitioners can leverage our findings to (1) understand the refactoring rhythms or tactics that they use and the impact on code quality, and (2) adopt or apply the most effective refactoring rhythms or tactics for their projects.

Refactoring rhythms do not have a significant impact on code quality. In this study, we observe two dominant refactoring rhythms as *all-day* and *work-day* refactoring rhythms. By measuring the code smell changes after adopting each rhythm, we don't observe significant differences in the code quality after applying either *work-day* or *all-day* rhythms. Therefore, we recommend that practitioners choose a refactoring rhythm that aligns with their project objectives and milestones, whether *work-day* or *all-day* rhythm. Considerations such as the size of the contribution and the developers' comfort could be taken into account when making this decision.

Different refactoring tactics have different impacts on code quality: root canal-based refactoring is more likely than floss-based refactoring to be associated with better code quality. As shown in our results, the two root canal-based tactics are ranked in the first place in terms of the overall frequency of code smell reduction, more specifically for 28 out of 35 types of code smell types. Root canal-based tactics lead to an average decrease in the frequency of code smells while floss-based tactics lead to an average increase in the frequency of code smells. Hence, root canal-based tactics outperform floss-based tactics by reducing the total amount of code smells. Therefore, we encourage practitioners to apply higher-level refactorings once in a while to keep the code maintainable with less code smells.

Establishing a common vocabulary for describing various refactoring rhythms and tactics can help improve communication among practitioners, developers, project managers, and other stakeholders. This work formulated several common patterns of refactoring rhythms and tactics. Such common patterns (or common vocabulary) facilitate communication among practitioners, enabling them to identify and implement effective refactoring patterns and techniques. By gaining a better understanding of the existing refactoring patterns, team members can communicate more effectively with each other and with stakeholders. This is especially useful in large and complex projects where there may be multiple teams and stakeholders involved.

Future research that performs empirical studies on refactoring should be aware of different refactoring tactics adopted in their studied projects. Our study reveals a distinction in the adoption of floss or root canal-based refactoring over the long term and their different impact on code quality. We strongly recommend researchers consider different types of refactoring tactics for the project selection when they investigate refactoring evolution, practices, and their impact on software quality. Opting for projects exclusively aligned with floss-based or root canal-based approaches could substantially influence or bias the outcomes of their studies. On the other hand, we recommend researchers distinguish the projects that adopt different refactoring tactics when they study the characteristics of refactoring in these projects.

Our findings promote the adoption of root canal-based refactorings in the development process instead of solely relying on floss-based refactoring. In fact, a large portion (42%) of our studied projects only perform floss-based refactoring. Our research underscores the importance of incorporating root canal-based refactoring as a means to minimize code smells. Consequently, we advocate for developers to move beyond solely relying on ad hoc refactoring during development (*i.e.,* floss-based). Instead, we propose the inclusion of dedicated refactoring tasks within the development timeline. While this approach may necessitate additional time investment, it proves beneficial for developers to conduct long-term maintenance tasks on code quality.

6 RELATED WORK

In this section, we review the literature related to refactoring rhythms, tactics, and studies related to refactoring detection.

Working rhythms. Zhang *et al.* [26] conduct a survey study on developers for working overtime. The authors find that working overtime is a common behavior among software practitioners. Developers who work more often on weekends believe that working overtime could increase their productivity. Similarly, Claes *et al.* [27] study the frequency of the commit messages on 86 open-source projects and find that one-third of developers work overtime either at night or during the weekends. In terms of researchers, Wang *et al.* [78] study the download information of scientific papers and find that many researchers work on weekends. However, the amount of overtime work differs among countries. Binnewies *et al.* [25] conducts a survey study on 133 employees and shows that psychological detachment, relaxation, and mastery experiences during the weekend are associated with being recovered for the upcoming week. Being recovered affects the weekly task performance, personal initiative, organizational citizenship behavior, and low perceived effort. To summarize, most of the prior work has been conducted to find different working rhythms during the week and correlate them with productivity. **However, there is no study related to the refactoring rhythms and their relationship with code quality. Our goal is to identify various refactoring rhythms employed by developers and determine which rhythms are positively associated with higher code quality.**

Refactoring tactics. *Floss* and *root canal* are two refactoring tactics identified in previous studies [1], [28]. *Floss* refactor-

ing is distinguished by frequent refactoring, blended with the software development process, while the *root canal* is identified by occasional periods of refactoring which is not consistent with the software development process. Liu *et al.* [76] investigate refactoring histories on data collected from 753,367 engineers and suggest that between *floss* and *root canal*, the most frequently adopted refactoring tactic by engineers is *floss*. Sousa *et al.* [29] classify refactoring as *floss* and *root canal* and conduct a study on software projects to examine refactoring opportunities indicated by code smells. **In this study, we identify new variations of refactoring tactics in addition to the previously mentioned tactics. Moreover, we study their relationship with code quality in terms of code smells.**

Refactoring detection. Several tools and approaches are introduced to automatically identify refactoring operations [38], [79]. The main idea behind these approaches is to compare different versions of the code fragments stored in a version control system and point out refactoring operations. These tools can help us study refactoring activities on a large scale in the software maintenance process. Kim *et al.* [79] introduce Ref-Finder, which takes two versions of a program as input from workspace snapshots or subversion of a repository and extracts logical facts about the syntactic structure of a program. Nevertheless, Soares *et al.* [80] conducts a study and show that Ref-Finder has low precision and recall which leads to false-positive results, which means it is inaccurate in detecting refactorings. However, Tsantalis *et al.* [38] design a tool, Rminer, that overcomes the above constraints. Similarly to Ref-Finder [79], Rminer [14], [38] takes two revisions of source code from the commit history in the version control system of a Java project and returns a list of refactoring operations applied between two versions. Using a similar approach, Alizadeh et al. [81] introduce a bot integrated into a version control system that monitors software repositories and identifies refactoring opportunities by analyzing recently changed files through pull requests. It then finds the best series of refactorings to fix the quality issues. **In this work, we employ the refactoring detection approach Rminer, which was developed in previous research [14], to extract refactoring operations from our dataset. We also validate its effectiveness in our context.**

Refactoring and code quality. Prior work has performed studies regarding the relationship between refactoring and code quality. Almogahed *et al.* [82] examine the studies that identify the impacts of code refactoring on software quality. It identifies that researchers agree that refactoring has a positive impact on both internal and external quality attributes. Moreover, Lacerda *et al.* [83] conduct a literature review on refactoring tools and common code smells to measure the relationship between refactoring operations and code smells. By analyzing the initial and final code smells after refactoring, the study finds that a significant proportion of code smells get eliminated after performing refactoring, which in turn preserves or enhances software quality during the maintenance process. Moreover, it notices that code smells and refactoring are linked by quality attributes and quality attributes that affect code smells are the same ones that affect refactoring. Bibiano *et al.* [17] correlate and study the effect of batch refactoring on code smells. It identifies that there is usually more than one refactoring operation required to eliminate the code smells. Cinn'eide *et al.* [9] conduct a survey study on the benefits of refactoring and argue that, although refactoring is commonly believed to aim at removing code smells, developers are not strongly motivated by the desire to eliminate them. Murphy *et al.* [28] define two refactoring tactics, floss and root canal, using a dental metaphor. Floss involves frequent refactoring with other program changes, while root canal involves infrequent, longer periods of refactoring with few other program changes. Murphy *et al.* propose five principles and evaluate tools for alignment with floss tactics. Murphy *et al.* find that the tools are not aligned with floss tactics and are therefore not suitable for floss refactoring. It suggests that floss refactoring is likely to result in higher quality and lower costs in the long run. However, it does not propose a quantitative approach to measure this claim. Previous studies [12], [13], [14], [15], [16], [17] link refactoring with code smells and code quality which makes code smells a good quality indicator of code after performing refactoring operations. Therefore, we use code smells to measure the relationship between the identified rhythms and tactics with code quality. **Different from the existing studies, our work is the first to quantitatively study the relationship between refactoring rhythms/tactics and code quality.**

7 Conclusion

In this study, we investigate the refactoring activities on a dataset consisting of 196 Apache projects to identify refactoring tactics and rhythms that developers and projects adopt in the software development process. We also examine their relationship with code quality in terms of code smells. Comparing both refactoring and development activities, we first determine that in more than 95% of project stages developers use a systematic refactoring rhythm on weekdays. Two major rhythms are identified as 1) *work-day* refactoring and (2) *all-day* refactoring. By considering the relationship between refactoring rhythms and the quality metrics (*i.e.*, code smells), we observe that different refactoring rhythms do not make a statistically significant difference to the code quality. Moreover, by clustering the life-cycle of refactoring activities we find four variations of existing refactoring tactics: (1) *frequent spiked floss*, (2) *intermittent spiked floss*, (3) *frequent root canal*, and (4) *intermittent root canal* refactoring tactics. We observe that root canal-based tactics (*frequent root canal* and *intermittent root canal*) are associated with a larger reduction in the frequency of code smells compared to floss-based tactics (*frequent spiked floss* and *intermittent spiked floss*). Our findings can help researchers and practitioners understand practical refactoring activities in real-world projects and their relationship with code quality. Practitioners can leverage our findings to choose the appropriate refactoring patterns for their projects based on their resources and code quality requirements. For future work, we plan to conduct experiments for other programming languages and focus more on automatic vs. manual refactoring operations.

8 Acknowledgments

We would like to express our sincere gratitude to Yucan Li for participating in the manual validation of our results. Their assistance was invaluable in ensuring the accuracy and reliability of our findings.

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the design of existing code addison-wesley professional," *Berkeley, CA, USA*, 1999.

[2] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[3] M. Kaya, S. Conley, Z. S. Othman, and A. Varol, "Effective software refactoring process," in *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*. IEEE, 2018, pp. 1–6.

[4] A. Arif and Z. A. Rana, "Refactoring of code to remove technical debt and reduce maintenance effort," in *2020 14th International Conference on Open Source Systems and Technologies (ICOSST)*. IEEE, 2020, pp. 1–7.

[5] J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, 2017.

[6] J. Kerievsky, "Refactoring to patterns (addison-wesley signature series)," 2005.

[7] T. Sharma, "Quantifying quality of software design to measure the impact of refactoring," in *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. IEEE, 2012, pp. 266–271.

[8] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[9] M. Ó Cinnéide, A. Yamashita, and S. Counsell, "Measuring refactoring benefits: a survey of the evidence," in *Proceedings of the 1st International Workshop on Software Refactoring*, 2016, pp. 9–12.

[10] B. Du Bois, S. Demeyer, and J. Verelst, "Does the" refactor to understand" reverse engineering pattern improve program comprehension?" in *Ninth european conference on software maintenance and reengineering*. IEEE, 2005, pp. 334–343.

[11] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?" in *International conference on software reuse*. Springer, 2006, pp. 287–297.

[12] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, 2016, pp. 858–870.

[13] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects," in *Proceedings of the 2017 11th Joint Meeting on foundations of Software Engineering*, 2017, pp. 465–475.

[14] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020.

[15] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Faultbuster: An automatic code smell refactoring toolset," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 253–258.

[16] N. Yoshida, T. Saika, E. Choi, A. Ouni, and K. Inoue, "Revisiting the relationship between code smells and refactoring," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–4.

[17] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, and D. Cedrim, "A quantitative study on characteristics and effect of batch refactoring on code smells," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11.

[18] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Proceedings of 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 2012, pp. 306–315.

[19] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[20] L. Aversano, U. Carpenito, and M. Iammarino, "An empirical study on the evolution of design smells," *Information*, vol. 11, no. 7, p. 348, 2020.

[21] L. Chen and M. A. Babar, "Towards an evidence-based understanding of emergence of architecture through continuous refactoring in agile software development," in *2014 IEEE/IFIP Conference on Software Architecture*. IEEE, 2014, pp. 195–204.

[22] A. Martini and J. Bosch, "An empirically developed method to aid decisions on architectural technical debt refactoring: Anacondebt," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 31–40.

[23] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[24] V. Khorikov, "Short-term vs long-term perspective in software development." [Online]. Available: https://enterprisecraftsmanship.com/posts/short-term-vs-long-term-perspective/

[25] C. Binnewies, S. Sonnentag, and E. J. Mojza, "Recovery during the weekend and fluctuations in weekly job performance: A week-level study examining intra-individual relationships," *Journal of Occupational and Organizational Psychology*, vol. 83, no. 2, pp. 419–441, 2010.

[26] J. Zhang, Y. Chen, Q. Gong, X. Wang, A. Y. Ding, Y. Xiao, and P. Hui, "Understanding the working time of developers in it companies in china and the united states," *IEEE Software*, vol. 38, no. 2, pp. 96–106, 2020.

[27] M. Claes, M. V. Mäntylä, M. Kuutila, and B. Adams, "Do programmers work at night or during the weekend?" in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 705–715.

[28] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE software*, vol. 25, no. 5, pp. 38–44, 2008.

[29] L. Sousa, W. Oizumi, A. Garcia, A. Oliveira, D. Cedrim, and C. Lucena, "When are smells indicators of architectural refactoring opportunities: A study of 50 software projects," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 354–365.

[30] W. Kruskal, "Kruskall–wallis one way analysis of variance," *J Am Stat Assoc*, vol. 47, no. 260, pp. 583–621, 1952.

[31] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2016.

[32] ——, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 683–711, 2018.

[33] S. Noei, H. Li, S. Georgiou, and Y. Zou, "Replication package," https://github.com/seal-tse-2023/replication-package , 2023.

[34] M. Claes and M. V. Mäntylä, "20-mad: 20 years of issues and commits of mozilla and apache development," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 503–507.

[35] berkeley bootcamp, "11 Most In-Demand Programming Languages in 2021," Dec. 2020. [Online]. Available: https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/

[36] tiobe, "index | TIOBE - The Software Quality Company." [Online]. Available: https://www.tiobe.com/tiobe-index/

[37] I. Ahmed, U. A. Mannan, R. Gopinath, and C. Jensen, "An empirical study of design degradation: How software projects get worse over time," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–10.

[38] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 483–494.

[39] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed. New York: Routledge, Jul. 1988.

[40] T. Sharma, P. Mishra, and R. Tiwari, "Designite: A software design quality assessment tool," in *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, 2016, pp. 1–4.

[41] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.

[42] F. A. Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 282–285.

[43] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2015, pp. 51–60.

[44] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *2008 12th European conference on software maintenance and reengineering*. IEEE, 2008, pp. 329–331.

[45] U. Azadi, F. A. Fontana, and D. Taibi, "Architectural smells detected by tools: a catalogue proposal," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 2019, pp. 88–97.

[46] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 255–258.

[47] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 740–740.

[48] AlDanial, "AlDanial/cloc," Jul. 2021, original-date: 2015-09-07T03:30:43Z. [Online]. Available: https://github.com/AlDanial/cloc

[49] GitHub, "GitHub API," https://docs.github.com/en/rest, accessed: April 3, 2023.

[50] F. E. Harrell *et al., Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer, 2015, vol. 3.

[51] E. Noei, F. Zhang, and Y. Zou, "Too many user-reviews, what should app developers look at first?" *IEEE Transactions on Software Engineering*, 2019.

[52] J. Miles, "R-squared, adjusted r-squared," *Encyclopedia of statistics in behavioral science*, 2005.

[53] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 182–191.

[54] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[55] R. L. Thorndike, "Who belongs in the family?" *Psychometrika*, vol. 18, no. 4, pp. 267–276, 1953.

[56] M. Syakur, B. Khotimah, E. Rochman, and B. D. Satoto, "Integration k-means clustering method and elbow method for identification of the best customer profile cluster," in *IOP Conference Series: Materials Science and Engineering*, vol. 336, no. 1. IOP Publishing, 2018, p. 012017.

[57] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[58] T. Dahiru, "P-value, a true test of statistical significance? a cautionary note," *Annals of Ibadan postgraduate medicine*, vol. 6, no. 1, pp. 21–26, 2008.

[59] S. K. Haldar, "Statistical and geostatistical applications in geology," *Mineral exploration*, pp. 167–194, 2018.

[60] L. Rising and N. S. Janoff, "The scrum software development process for small teams," *IEEE software*, vol. 17, no. 4, pp. 26–32, 2000.

[61] J. C. Gower, "Properties of euclidean and non-euclidean distance matrices," *Linear algebra and its applications*, vol. 67, pp. 81–97, 1985.

[62] M. Kljun and M. Ters˘ek, "A review and comparison of time series similarity measures," in *29th International Electrotechnical and Computer Science Conference (ERK 2020). Portoroz˘*, 2020, pp. 21–22.

[63] S. K. Gaikwad, B. W. Gawali, and P. Yannawar, "A review on speech recognition technique," *International Journal of Computer Applications*, vol. 10, no. 3, pp. 16–24, 2010.

[64] E. G. Jelihovschi, J. C. Faria, and I. B. Allaman, "Scottknott: a package for performing the scott-knott clustering algorithm in r," *TEMA (São Carlos)*, vol. 15, no. 1, pp. 3–17, 2014.

[65] C. J. Ferguson, "An effect size primer: a guide for clinicians and researchers." 2016.

[66] I. Spieler, S. Scheibe, C. Stamov-Roßnagel, and A. Kappas, "Help or hindrance? day-level relationships between flextime use, work–nonwork boundaries, and affective well-being." *Journal of Applied Psychology*, vol. 102, no. 1, p. 67, 2017.

[67] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys," in *Annual meeting of the Florida association of institutional research*, 2006.

[68] "Scikit-learn k-means silhouette analysis," https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html, accessed: August 23, 2023.

[69] E. S. Dalmaijer, C. L. Nord, and D. E. Astle, "Statistical power for cluster analysis," *BMC bioinformatics*, vol. 23, no. 1, pp. 1–28, 2022.

[70] U. Rani and S. Sahu, "Comparison of clustering techniques for measuring similarity in articles," in *2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT)*. IEEE, 2017, pp. 1–7.

[71] T. Yatsunenko, F. E. Rey, M. J. Manary, I. Trehan, M. G. Dominguez-Bello, M. Contreras, M. Magris, G. Hidalgo, R. N. Baldassano, A. P. Anokhin *et al.*, "Human gut microbiome viewed across age and geography," *nature*, vol. 486, no. 7402, pp. 222–227, 2012.

[72] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[73] T. L. Wahl, "Discussion of "despiking acoustic doppler velocimeter data" by derek g. goring and vladimir i. nikora," *Journal of Hydraulic Engineering*, vol. 129, no. 6, pp. 484–487, 2003.

[74] M. Parsheh, F. Sotiropoulos, and F. Porté-Agel, "Estimation of power spectra of acoustic-doppler velocimetry data contaminated with intermittent spikes," *Journal of Hydraulic Engineering*, vol. 136, no. 6, pp. 368–378, 2010.

[75] R. Q. Quiroga, Z. Nadasdy, and Y. Ben-Shaul, "Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering," *Neural computation*, vol. 16, no. 8, pp. 1661–1687, 2004.

[76] H. Liu, Y. Gao, and Z. Niu, "An initial study on refactoring tactics," in *2012 IEEE 36th Annual Computer Software and Applications Conference*. IEEE, 2012, pp. 213–218.

[77] E. Fernandes, A. Chávez, A. Garcia, I. Ferreira, D. Cedrim, L. Sousa, and W. Oizumi, "Refactoring effect on internal quality attributes: What haven't they told you yet?" *Information and Software Technology*, vol. 126, p. 106347, 2020.

[78] X. Wang, S. Xu, L. Peng, Z. Wang, C. Wang, C. Zhang, and X. Wang, "Exploring scientists' working timetable: Do scientists often work overtime?" *Journal of Informetrics*, vol. 6, no. 4, pp. 655–660, 2012.

[79] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 371–372.

[80] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson, "Comparing approaches to analyze refactoring activity on software repositories," *Journal of Systems and Software*, vol. 86, no. 4, pp. 1006–1022, 2013.

[81] V. Alizadeh, M. A. Ouali, M. Kessentini, and M. Chater, "Refbot: intelligent software refactoring bot," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 823–834.

[82] A. Almogahed, M. Omar, and N. H. Zakaria, "Impact of software refactoring on software quality in the industrial environment: A review of empirical studies," 2018.

[83] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, p. 110610, 2020.