An Empirical Study on Release-Wise Refactoring Patterns

SHAYAN NOEI, Queen's University, Canada
HENG LI, Polytechnique Montréal, Canada
YING ZOU, Queen's University, Canada

Refactoring is a technical approach to increase the internal quality of software without altering its external functionalities. Developers often invest significant effort in refactoring. With the increased adoption of continuous integration and deployment (CI/CD), refactoring activities may vary within and across different releases and be influenced by various release goals. For example, developers may consistently allocate refactoring activities throughout a release, or prioritize new features early on in a release and only pick up refactoring late in a release. Different approaches to allocating refactoring tasks may have different implications for code quality. However, there is a lack of existing research on how practitioners allocate their refactoring activities within a release and their impact on code quality. Therefore, we first empirically study the frequent release-wise refactoring patterns in 207 open-source Java projects and their characteristics. Then, we analyze how these patterns and their transitions affect code quality. We identify four major release-wise refactoring patterns: *early active*, *late active*, *steady active*, and *steady inactive*. We find that adopting the late active pattern—characterized by gradually increasing refactoring activities as the release approaches—leads to the best code quality. We observe that as projects mature, refactoring becomes more active, reflected in the increasing use of the *steady active* release-wise refactoring pattern and the decreasing utilization of the *steady inactive* release-wise refactoring pattern. While the *steady active* pattern shows improvement in quality-related code metrics (*e.g.,* cohesion), it can also lead to more architectural problems. Additionally, we observe that developers tend to adhere to a single refactoring pattern rather than switching between different patterns. The *late active* pattern, in particular, can be a safe release-wise refactoring pattern that is used repeatedly. Our results can help practitioners understand existing release-wise refactoring patterns and their effects on code quality, enabling them to utilize the most effective pattern to enhance release quality.

CCS Concepts: • **Software and its engineering → Maintaining software**; **Software evolution**.

Additional Key Words and Phrases: Refactoring, Refactoring Patterns, Release Cycles, Code Quality

ACM Reference Format:
Shayan Noei, Heng Li, and Ying Zou. 2025. An Empirical Study on Release-Wise Refactoring Patterns. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE019 (July 2025), 21 pages. https://doi.org/10.1145/3715734

1 Introduction

With the adoption of short release cycles, particularly driven by the growing use of continuous integration and continuous deployment (CI/CD) [22], software projects now release new versions more frequently, in shorter cycles, automatically and continuously [16, 66]. Continuous release cycles pressure practitioners to effectively integrate user feedback [21] and deliver high-quality code efficiently. The frequency of these release cycles varies across projects, ranging from a few days to weeks [1, 18, 20], whereas traditional software development often takes years to publish a new version [4, 20].

Authors' Contact Information: Shayan Noei, Queen's University, Kingston, Canada, s.noei@queensu.ca; Heng Li, Polytechnique Montréal, Montreal, Canada, heng.li@polymtl.ca; Ying Zou, Queen's University, Kingston, Canada, ying.zou@queensu.ca.

ACM 2994-970X/2025/7-ARTFSE019
https://doi.org/10.1145/3715734

Refactoring is a systematic process to improve the internal quality of software without altering its external functionalities [13, 35]. Refactoring enhances the maintainability [27] of the code by improving readability, reducing complexity, and increasing modularity [31, 38]. Previous studies have shown a relationship between refactoring and the elimination of code smells and improving the quality of the code [17, 34, 53, 60, 65]. Code smells are poor coding practices that reflect the design flaws of the code [54]; the success of refactoring is often evidenced by the reduction or elimination of code smells, which often requires multiple refactoring operations [6].

Previous studies categorize refactoring strategies into long-term refactoring strategies, which include frequent but lightweight refactoring during development (*i.e.,* floss tactic) and infrequent but heavyweight refactoring (*i.e.,* root-canal tactic) [13, 34, 55]. These strategies illustrate how refactoring is integrated into the long-term development of a project to maintain code quality over a long period. However, refactoring strategies within release cycles, which typically span shorter time frames and may depend on the release time, remain unexplored in these studies. Given the substantial amount of effort developers invest in refactoring [15, 39], it remains unclear how these efforts are distributed throughout release cycles and the corresponding impact on code quality. Shorter release cycles may prompt the adoption of specific refactoring patterns aligned with release dates. This paper aims to address this gap by analyzing refactoring strategies within release cycles and identifying patterns contributing to high-quality software delivery.

In this study, we conduct a large-scale empirical analysis of refactoring practices across 1,604 releases from 207 open-source projects. We define **release-wise refactoring patterns** to describe *how practitioners distribute their refactoring activities throughout a release.* Our goal is to identify and analyze the dominant release-wise refactoring patterns by examining changes in refactoring density throughout each release. Additionally, we assess the relationship between release-wise refactoring patterns and code quality by measuring code smells and metrics (*e.g.,* cohesion and coupling) before and after each release. Based on our findings, we aim to propose best practices for refactoring within a release, including strategies for switching between different patterns.

We aim to answer the following research questions:

RQ1. What release-wise refactoring patterns are present in open-source projects? Developers spend effort on refactoring during the software development process; however, it is not clear how they allocate refactoring tasks within a release. We study the evolution of refactoring densities throughout software releases and identify four major release-wise refactoring patterns: *early active*, *late active*, *steady active*, and *steady inactive*. Furthermore, by analyzing the types of refactoring and external factors (*e.g.,* the number of developers) associated with each release, we provide details on the characteristics of each release-wise refactoring pattern.

RQ2. What is the relationship between release-wise refactoring patterns and code quality? To identify the most effective refactoring patterns, we assess the relationship between release-wise refactoring patterns and code quality. We use code smells and code metrics as code quality indicators to evaluate the quality of the code before and after each release. Then, we rank and cluster the identified release-wise refactoring patterns based on improvements in these quality metrics. Our findings indicate that *late active* refactoring pattern is associated with a greater reduction in code smells and higher code quality compared to the other identified patterns.

RQ3. Does the usage of release-wise refactoring patterns change over time, and how do developers switch from one pattern to another? To understand how practitioners' refactoring patterns evolve across the lifecycle of a project, we analyze the distribution of various release-wise refactoring patterns during the early, middle, late, and last stages of the project lifecycles, as well as the transitions among these patterns. We find that developers tend to continue using previous refactoring patterns. Moreover, we observe an increase in the utilization of the *steady active* pattern, while the adoption of the *steady inactive* pattern decreases over time. Additionally, the usage of the

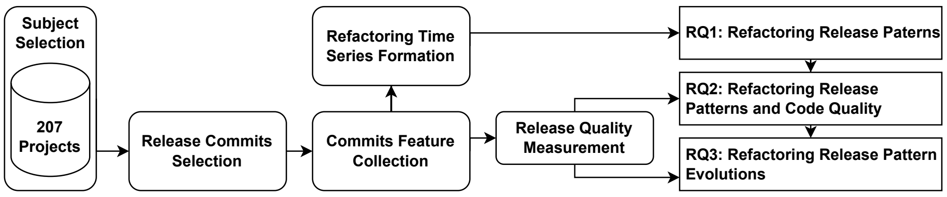
Subject Selection
207 Projects
Release Commits Selection
Commits Feature Collection
Refactoring Time Series Formation
Release Quality Measurement
RQ1: Refactoring Release Paterns
RQ2: Refactoring Release Patterns and Code Quality
RQ3: Refactoring Release Pattern Evolutions

Fig. 1. The overview of our study.

late active refactoring pattern remains consistent across all development stages. While *early active* and *steady active* patterns may offer temporarily improved release quality, the *late active* pattern contributes to higher code quality when applied continuously.

Our work makes the following main contributions:

- We provide a large-scale empirical study on the refactoring strategies corresponding to release cycles: release-wise refactoring patterns.
- We identify prevalent release-wise refactoring patterns, which can aid practitioners in identifying common refactoring patterns developers use for project management and supporting tooling developments.
- By understanding the effect of the release-wise refactoring patterns, we help developers understand the most effective way of incorporating refactoring within release cycles.
- We provide a dataset consisting of refactoring and release information for 207 open-source projects.

Organization. The remainder of our study is organized as follows. Section 2 provides the experiment setup of our study. Section 3 presents the motivation, approaches, and findings of our research questions. Section 5 discusses the implications of our study. Section 4 explains the threats to the validity of our findings. Section 6 surveys related studies. Finally, we conclude our study and present future research directions in Section 7.

2 Experiment Setup

This section details the experimental setup of our study, covering our methods for data collection, pre-processing, and analysis.

2.1 Overview of Our Approach

An overview of our study is shown in Figure 1. We conduct our experiments by systematically selecting 207 active and popular open-source Java projects with sufficient commit history. Furthermore, we select the major and minor releases within each project and identify the responsible commits for each release. From the selected commits, we extract code and refactoring-related features as well as project-related features such as the number of developers per release. We then create a time series describing the refactoring changes within each release. We answer the first research question by clustering the release-wise refactoring time series and identifying the main patterns therewithin. By measuring code smells and code metrics as code quality indicators before and after each release, we explain the best release-wise refactoring patterns for maintaining code quality, addressing the second research question. Finally, by analyzing the distribution of the identified release-wise refactoring patterns across different stages of the development lifecycle and examining the transition among them, we aim to understand how developers use and switch between these patterns and their implications for code quality. We detail the analyses specific to each individual research question in Section 3.

2.2 Subject Selection

We use a systematic approach to select the subject projects for our study. Considering that Java is one of the most popular programming languages [43, 58], and it is supported by robust refactoring detection tools, such as RMiner [60, 61], we start by collecting all Java repositories with at least one star and one fork using the GitHub advanced search,[1] which yields 490,880 Java repositories on GitHub. To ensure we include the most active and popular projects with a sufficient history of development, we select repositories that:

- are up to date with current development practices
  - have been active in the year of the data collection (with at least one push in 2023)
- are not disabled or archived
- have a healthy level of interest and activity in the repository
  - have more than 13 forks[2] (13 is calculated as the 3rd quartile of the distribution of number of forks of all the projects)
- are popular within the open-source community
  - have more than 22 stars (22 is calculated as the 3rd quartile of the number of stars)
- have a sufficient development history for our analysis
  - have at least 1,000 commits, similar to prior work [34]
- have enough releases to conduct our experiments (at least two releases), and
- provides sufficient data on the details of their releases
  - follow a standard semantic versioning format (*e.g.,* v2.0.3), which consists of three numbers in the format MAJOR.MINOR.PATCH [42]

Since the implementations of CI/CD vary across projects and are sometimes only partially adopted [49], we focus on projects with systematic versioning [42], which is often associated with the adoption of CI/CD [7, 11, 64]. This approach enables us to distinguish between major, minor, and patch releases, thereby ensuring the broader applicability of our results. The different types of releases are described below:

- *Major release:* focuses on introducing significant new features or architectural changes to the software. The versioning convention for a major release is a number followed by two zeroes (*e.g.,* v1.0.0).
- *Minor release:* is typically used to introduce a new feature or a minor improvement in the software. It is represented by the major version number followed by the minor version number and a zero (*e.g.,* v1.2.0).
- *Patch release:* is often used for a hotfix or a minor improvement in the software. It is indicated by the major version number followed by the minor version number and the patch number (*e.g.,* v1.2.3).

As a result, we select 207 repositories as the subject dataset for our study.

2.3 Release Commits Selection

Different projects choose different release strategies to release their software [41]. Some projects may choose a mainline branch to release, while others may choose different branches for their feature development or releases [40]. The projects in our dataset utilize the following dominant release strategies:

- **Mainline release strategy:** This strategy is a single-branch release strategy, in which the project uses only one branch for releases and keeps developments in a separate branch.

[1]https://github.com/search/advanced
[2]A copy of that repository under a different GitHub account.

Consequently, all releases originate from the main/master branch. 121 projects in our dataset mainly employ the *mainline* release strategy.
- **Release branch strategy:** This strategy focuses on creating a dedicated branch for each release and releasing from that specific branch. As a result, each software release has its own branch, and all versions are accessible through their respective branches. 86 projects of our dataset mainly follow the *release branch* strategy.

As we aim to analyze the refactoring patterns corresponding to each release, we focus on major and minor releases as they typically reflect significant changes and improvements. Patch releases, on the other hand, usually address single hotfixes and may not contribute to release-wise refactoring pattern analysis. To select the commits corresponding to each major/minor release, we retrieve release information from the GitHub REST API[3] for each repository to collect their major/minor release dates. Depending on the release strategy employed by the projects, we first identify the branch used for the release, including any merged branches. We then select the commits with timestamps that fall within the relevant release period from that branch, and then sort these commits based on their timestamps. We identify release strategies by analyzing the branches from which releases are made. If a project consistently uses the master/main branch, we identify it as following a mainline strategy. Projects consistently using different branches for releases are classified under the release branch strategy. If a project releases from the master/main branch and then switches to a different branch, we classify it as using both techniques. As a result, we select a total of 1,604 releases, consisting of 1,439 minor releases and 165 major releases. On median, each project includes 1 major release and 7 minor releases. Since the median time between major-to-major releases is 451 days, we exclude these from our analysis, as they align more closely with traditional development patterns (*i.e.,* not continuous release) [20, 34]. Instead, we focus on transitions involving minor-to-minor, minor-to-major, and major-to-minor releases, which have a median of 8 releases, with an interquartile range of 4 (first quartile) to 15 (third quartile). The selected releases have a median duration of 56 days, with an interquartile range of 28 to 118 days.

2.4 Commits Feature Extraction

For each commit, we select (1) the commit log information that includes details such as the author and commit date, and (2) the code change and refactoring information that contains the files changed, lines of code changes, and the frequency and types of refactoring applied in each commit. To obtain the refactoring information, we use RMiner 3.0.0, which is the most accurate state-of-the-art refactoring detection with the maximum number of refactoring types (102 types) detected [23, 34]. RMiner has demonstrated a precision of 99.7% and a recall of 94.2% [60]. We use git log to extract information about the files and lines of code affected in each commit. This approach enables us to capture not only basic commit information but also details such as the affected files, refactoring lines, and refactoring operations (*e.g.,* pull-up method).

2.5 Code Quality Measurement

Code smells have been shown as effective refactoring quality indicators, as refactorings tend to eliminate code smells [17, 34, 63]. To this end, we use Designite 2.5.5 [50, 51] to measure code smells before and after each release. Designite is a robust tool that surpasses its competitors, capable of detecting 47 types of code smells across various code levels, which fall into five higher-level code smell categories:

- **Architecture smells:** Explain the issues related to overall system structure and dependencies, such as cyclic dependencies. This category contains 7 code smell types.

[3]https://docs.github.com/en/rest

- **Design smells:** Include problems related to fundamental design principles, such as unnecessary abstractions and deficient encapsulation. This category contains 18 code smell types.
- **Testability smells:** Explain the aspects of testability of the code, such as excessive dependency. This category includes 4 code smell types.
- **Implementation smells:** Explain concerns with code implementation, like complex methods, long parameter lists, or long statements. This category has 10 code smell types.
- **Test smells:** Include the problems within the test code, such as missing assertions, and ignored tests. This category includes 8 code smell types.

If refactoring is not performed properly, it can negatively impact the overall quality of the code. Previous studies have shown that refactoring is not always motivated by the elimination of code smells [53]. Therefore, we consider common code quality metrics such as cohesion, coupling, and complexity [44] to measure code quality before and after refactoring. To achieve this, we use the Understand tool [47] and select 13 code quality metrics related to cohesion, coupling, and complexity. The explanation of each category is as follows:

- **Lack of Cohesion**: Describe the extent to which functions and classes of the code work together. We measure this by evaluating the lack of cohesion among functions and classes.
- **Coupling**: Explain the degree of interdependence between different classes and functions of the code. We assess coupling by examining the number of base classes, coupled classes, derived classes, and the inputs and outputs of functions and classes.
- **Complexity**: Describe the complexity of the code components. We assess complexity by analyzing cyclomatic complexity [28] and the max nesting degree of statements (*e.g.,* for loops inside while loops).

3 Results

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

3.1 RQ1: What release-wise refactoring patterns are present in open-source projects?

3.1.1 Motivation. Previous studies have examined long-term refactoring strategies, categorizing them into two general types: floss and root canal refactorings, with some variations in the frequency of their applications [13, 34, 55]. Additionally, these studies have identified a correlation between adopting different refactoring strategies and their impact on code quality [33, 34]. With the increased adoption of continuous release cycles, development and feature delivery are now segmented into shorter release cycles. However, there is no large-scale study on the refactoring strategies relative to release cycles and their impact on code quality. In this research question, we aim to identify the frequent patterns developers utilize to integrate refactoring tasks within software releases. By identifying these patterns, we aim to deepen our understanding of refactoring practices and measure their efficacy in the fast-release cycles. Ultimately, we aim to identify the most successful release-wise refactoring strategies to improve code quality and establish best refactoring practices in continuous release cycles.

3.1.2 Approach. To identify and gain insights into release-wise refactoring patterns, we first measure refactoring density over time, then form a normalized refactoring time series for each release. Finally, using time series clustering, we identify common refactoring patterns and their characteristics within the releases. In the following, we provide detailed explanations of each step conducted in this process:

Calculating refactoring density: To identify the refactoring trends and study changes in refactoring activities within each release, we use a metric to capture the density of daily refactoring activities with each release. Following prior work [34], we use the commits submitted on each active day of development to calculate the daily refactoring density (DRD), as defined in Equation 1.

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

DRD explains the amount of refactoring effort that is deviated from the regular development process. We use refactoring churn instead of the refactoring action count, as each action can change a varying number of lines of code. Using the DRD metric within each release, we create the refactoring-release time series, which represents the changes in refactoring density within each release.

Normalizing release time series: Due to differences in the length of release cycles, the refactoring time series of different releases vary in length, making them incomparable, as they cannot be directly mapped onto each other. To identify common release-wise refactoring patterns, we standardize the created release-wise refactoring time series within each release using linear interpolation [14, 25], which fits the time series of different releases into a fixed range of 10 data points representing 10 quartiles of refactoring densities within each release. Linear interpolation estimates the DRD at a desired point based on the values of two neighboring points on the line connecting them [2]. Therefore, linear interpolation does not alter the overall trend but scales the time series into the same range. To compare historical refactoring information across releases and better interpret release-wise patterns, we calculate DRD for the entire refactoring history before each release. This historical refactoring density provides insight into the amount of past refactoring effort from the beginning of the project until the start of a release.

Identifying refactoring release patterns: To identify release-wise refactoring patterns applied within a release, we utilize Soft Dynamic Time Warping [9] (Soft-DTW) for clustering our time series of refactoring densities (*i.e.,* DRDs). Each cluster represents a release-wise refactoring pattern, specifically reflecting common refactoring activities within each release. Soft-DTW is an extension of DTW designed for time series with variable speeds, enabling it to capture the overall pattern trends of refactoring within a release. It allows for adjustments in time series that may begin their trends slightly earlier or later. Soft-DTW helps to minimize the impact of noise in our release-wise refactoring time series (*i.e.,* random fluctuations or outliers in refactoring time series) by incorporating a smoothing function that is particularly effective in handling noisy data. This function smooths the time series, reducing sensitivity to noise during clustering without altering the original data. To determine the optimal number of clusters, we first plot t-distributed Stochastic Neighbor Embedding (t-SNE)[62] and calculate the silhouette score[45] for different numbers of clusters. The t-SNE plots help visualize the distribution of time series in a reduced 2D space, while the silhouette score, ranging from -1 to 1 [48], assesses the quality of the clustering. Then, we manually examine the cluster centroids to ensure they represent distinct and meaningful patterns.

Analyzing the release-wise refactoring patterns: Given that RMiner [60, 61] can identify 102 distinct types of refactorings, we categorize these refactorings into six mutually exclusive levels of granularity: variable, method, class, package, organization, and test levels. The description of each granularity is below:

- **Variable:** Focuses on variable levels within functions or classes (*e.g.,* rename variable).
- **Method:** Contains refactoring at the method level of the code. For example, extracting smaller methods from a larger one (*i.e.,* extract method).
- **Class:** Consists of refactorings that involve a class. For instance, extracting a superclass.
- **Package:** Includes refactorings that involve multiple classes grouped as a package. For example, merging multiple packages into one.
- **Organization:** Contains refactorings that address the overall structure of the code. For example, moving code within files.

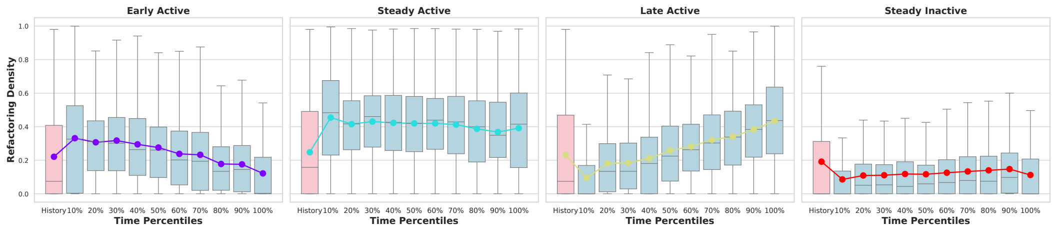

Fig. 2. Boxplot of release-wise refactoring patterns. The dotted line represents the centroid of each cluster. The blue bars indicate the refactoring density of the current release and the pink bar indicates the historical refactoring density that shows the distribution of refactoring density from the start of the project up to the current release.

- **Test:** Has refactorings for test files. For example, parameterized tests, which involve running tests with different sets of input data.

To characterize the changes in the refactoring operations applied during the time span of a release, we partition each release into several development phases. Specifically, we divide each release into three time segments using the first and third quartiles of the release duration, representing the early (before the first quartile), middle (between the first and third quartiles), and late (after the third quartile) phases of the release cycle. We are interested in determining how refactoring activities evolve across these different phases and the corresponding refactoring categories. To this end, we perform a Kruskal-Wallis [24] test for each refactoring category (*e.g.,* method level) within each release-wise refactoring pattern to identify if there is a significant difference in the adoption of various refactoring operations across different phases of a release.

Analyzing the external features: To gain a contextual understanding of the external features contributing to the adoption of each release-wise refactoring pattern, we analyze three features of: project size (*i.e.,* code size), development community size (*i.e.,* the number of contributors), and project popularity (*i.e.,* the number of stars)—thereby providing a clearer picture of how these external features relate to different refactoring patterns. We divide each feature into four equal parts and label them as *least*, *less*, *more*, and *most* [34] (*i.e.,* a release with the most contributors). We analyze each categorized feature using the chi-square test [29], which determines if there is a significant difference (p-value ≤ 0.05) between the categorized values of external features across different release-wise refactoring patterns. Additionally, we present the distribution of each feature for each refactoring pattern.

3.1.3 Results. **From the total of 1,604 analyzed releases, our clustering approach reveals four dominant refactoring patterns relative to software releases across all releases: (1) *early active* (30%), (2) *steady active* (17%), (3) *late active* (23%), and (4) *steady inactive* (30%).** Figure 2 illustrates the evolution of each identified pattern within a release. As shown in Figure 2, the *early active* pattern shows a higher refactoring density at the beginning of the release, and then it gradually decreases as it gets closer to the release. In contrast, the *late active* pattern exhibits patterns with an increased refactoring density as it approaches the release. The *steady active* pattern indicates a consistently higher density of refactoring compared to the average history, while the *steady inactive* shows a low density of refactoring throughout the release. Interestingly, more than half of the project releases (53%) exhibit varying (increasing/decreasing) refactoring intensities in their release cycles. Figure 3 shows the t-SNE plot of the identified clusters of our release-wise time series. The plot indicates a clear separation between clusters, highlighting distinct patterns within

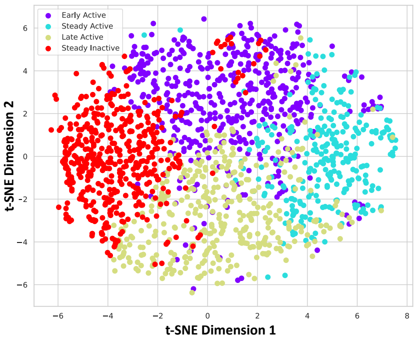

Fig. 3. t-SNE plot showing the distribution of our clustering results in two dimensions.

Table 1. Median of the sum of distributions for different refactoring types in release-wise refactoring patterns per 1,000 lines of code churn, where significant differences in distributions are observed at different phases of the release. Non-existing categories do not show significant differences across phases and are thus applied equally.

Pattern Name	Category	Early	Middle	Late	P-value
Early Active	Variable	20.06	14.93	9.41	0.000
	Method	61.13	53.17	37.91	0.000
	Class	20.60	16.32	11.82	0.005
	All	126.13	103.83	83.69	0.000
Late Active	Variable	11.93	18.35	18.58	0.001
	Method	38.41	55.31	60.06	0.001
	All	86.78	109.5	119.98	0.002
Steady Active	Variable	22.53	20.07	15.49	0.011
	All	122.35	93.19	96.54	0.032

the data. Furthermore, our approach yields a silhouette score of 0.3, suggesting a fair separation of the data [19].

***Early active* releases tend to have more class/method/variable-level refactoring earlier in a release, whereas *late active* releases tend to avoid extensive class-level refactoring late in a release.** We use the Kruskal-Wallis test [24] to examine the differences in refactoring granularities across various phases of development with the following hypothesis:

- ***H_0*: *There is no significant difference in the utilization of different granularities of refactorings across different phases of each release-wise refactoring pattern.***

Table 1 shows the results of comparing the distribution of different refactoring types across the early, middle, and late phases of development for each pattern, for the cases where the H_0 hypothesis is rejected (p-value ≤ 0.05). As it is shown, **the adoption of package, organization, and testing level refactorings does not exhibit statistically significant differences across any phases of the releases in various patterns.** Furthermore, in the *early active* pattern, with lower DRD at the beginning of the release, there is an increase in refactoring activities at the method, class, and variable levels initially, which then significantly decreases closer to the release. Conversely, in late active, method, and variable level refactorings significantly increase as the release approaches, while other categories, including the class level refactoring, remain at the same pace. In the *steady inactive* pattern, there are no significant variations in overall refactoring types, indicating a consistent low frequency of refactoring throughout the release cycle. Likewise, the *steady active* pattern

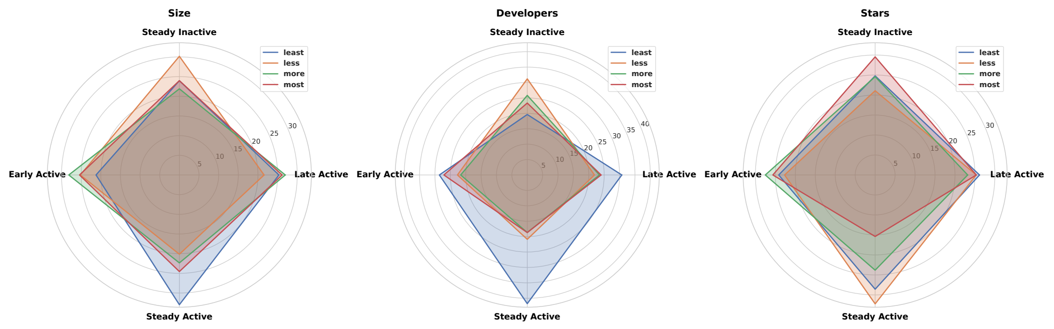

Fig. 4. Distribution of Features Across Refactoring Release Patterns.

experiences a consistently increased frequency of refactoring throughout the release, except for variable-level refactoring operations, which are considered lower-level refactorings.

Projects utilizing different release-wise refactoring patterns exhibit distinct characteristics. Figure 4 shows the distribution of the size, contributors, and stars external features among different release-wise refactoring patterns. The results of the chi-square test indicate significant differences in the distribution of each external feature among different refactoring release patterns, with p-values of 0.003 for size, 0.000 for contributors, and 0.001 for stars, respectively. We observe that:

- The *steady inactive* pattern is associated with *less* size, *less* developers, and the *most* stars.
- The *steady active* pattern is associated with *least* size, the *least* developers, and *less* stars.
- The *early active* pattern is associated with the *more* size, the *most* developers, and *more* stars.
- The late active pattern shows the *most* size, *more* developers, and *least* stars.

Smaller projects with fewer developers are mainly associated with either *steady active* or *steady inactive* patterns, where refactoring activities remain consistent throughout a release. In contrast, larger projects with more developers exhibit a variety of refactoring patterns, where refactoring activities may increase or decrease over time. This suggests that smaller projects tend to integrate refactoring into daily development activities, while larger projects may concentrate refactoring efforts in specific phases of a release, such as the early or late stages.

We identify four dominant release-wise refactoring patterns: *late active*, *early active*, *steady active*, and *steady inactive*, each associated with different types of refactoring and project characteristics.

3.2 RQ2: What is the relationship between release-wise refactoring patterns and code quality?

3.2.1 Motivation. The first research question identifies four primary refactoring patterns developers utilize. Understanding the relationship of these patterns with code quality can help developers make informed decisions to manage and distribute refactoring tasks throughout the release cycle. With the understanding, we can provide practitioners with insights into how adopting different release-wise refactoring patterns could potentially increase code quality. In this research question, we investigate the relationship between release-wise refactoring patterns and code quality measured by code smells and quality-related code metrics.

3.2.2 Approach. To measure the relationship between the detected release-wise refactoring patterns, we first use a set of code quality metrics to assess changes in quality after applying each pattern. We then rank and cluster the refactoring patterns based on these quality metric changes to evaluate their effectiveness. The following provides a detailed explanation of each step in this process:

Quality measurement: Previous studies have identified code smells as indicators of refactoring quality [17, 34, 63]. However, refactoring may not be motivated solely by the elimination of code smells [53]. Therefore, we measure the effectiveness of the release-wise refactoring patterns in eliminating or reducing code smells and use changes in code metrics (*e.g.,* low coupling) to complement these results. We measure changes in code smells categorized into six granularities and code metrics categorized into three granularities (as explained in Section 2.5), before and after each release. We then use the following equation to measure the normalized overall and category-specific differences in code smells/metrics (CSD) before and after each release [34]:

$$\mathrm{CSD}(i) = \frac{(\mathrm{ECS}(i)/\mathrm{ELC}(i) - \mathrm{ICS}(i)/\mathrm{ILC}(i))}{\mathrm{CC}(i)/\mathrm{ELC}(i)} \tag{2}$$

ECS indicates the frequency of code smells/metrics at the end of a release; ELC indicates the lines of code at the end of a release; ICS shows the initial code smells/metrics of a release; ILC indicates the initial lines of code at the beginning of a release; and CC indicates the total code churn of each release. This equation calculates the change in code smell per line from the start to the change in code smell at the end of a release, normalized by the code churn relative to the codebase size, providing a measure of how code quality has been impacted by modifications during that release.

Ranking and clustering: To measure similarities and rank the identified release-wise refactoring patterns, we use the Scott-Knott-ESD [56, 57] test. This test helps us: (1) perform statistical comparisons among the metric value changes of different patterns and rank the release-wise refactoring patterns based on the magnitude of quality changes, and (2) cluster release-wise refactoring patterns that do not exhibit significant differences in quality measures. Therefore, we can identify which groups of release-wise refactoring patterns do not show significant differences and which group represents the most effective treatment (*i.e.,* improvement in each quality measure).

3.2.3 Findings. ***Late active* and *steady active* release-wise refactoring patterns exhibit higher quality by improving cohesion and reducing coupling.** Overall lower coupling and higher cohesion (*i.e.,* lower lack of cohesion) contribute to better code quality. As shown in Table 2, *steady inactive* and *early active* release-wise refactoring patterns have higher mean values and ranks (*i.e.,* rank 1) for coupling, indicating higher coupling, and higher mean and rank in lack of cohesion, indicating lower cohesion, compared to *late active* and *steady active* release-wise refactoring patterns. Therefore, *steady inactive* and *early active* release-wise refactoring patterns exhibit lower code quality compared to *late active* and *steady active* patterns. The *steady inactive* and *early active* release-wise refactoring patterns may indicate little or no refactoring, with a focus on addressing previous developments rather than the current release. In contrast, the *steady active* and *late active* patterns reflect a more cleaning-focused phase later or throughout the release, likely targeting the current state of the code in preparation for the release.

The *steady inactive* release-wise refactoring pattern exhibits the worst performance in design, implementation, and total code smells reduction. As shown in Table 3, the *steady inactive* pattern has the highest mean (4.30) for code smell changes, indicating a greater increase in code smells after adoption compared to other release-wise refactoring patterns, leading to reduced code quality. Therefore, the lack of active refactoring and the accumulation of code smells can continuously reduce code quality and degrade its maintainability over time.

Table 2. The results of the Scott-Knott-ESD test on **code metrics** within each release-wise refactoring pattern. Patterns within the same cluster are represented by the same color. Clusters shown in red indicate lower quality compared to those highlighted in green.

Complexity			**Coupling**			**Lack of Cohesion**		
Pattern	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**
Steady Inactive	1	2.51	Steady Inactive	1	6.85	Steady Inactive	1	5.83
Early Active	1	1.77	Early Active	1	4.59	Early Active	1	3.70
Steady Active	1	1.54	Late Active	2	4.13	Late Active	2	3.41
Late Active	1	1.53	Steady Active	2	3.54	Steady Active	2	3.39

Table 3. The results of the Scott-Knott-ESD test on **code smells** within each release-wise refactoring pattern. The same color represents patterns within the same cluster. Clusters shown in red indicate lower quality compared to those highlighted in green. Positive mean values indicate an increase in the mean of code smells, while negative values represent a decrease in the mean of code smells. The mean values represent the overall mean change in each metric, respectively.

Architecture			**Design**			**Testability**		
Pattern	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**
Steady Active	1	0.03	Steady Inactive	1	-0.03	Early Active	1	0.08
Steady Inactive	2	-0.02	Steady Active	2	-0.06	Steady Inactive	1	0.07
Late Active	2	-0.03	Early Active	2	-0.07	Late Active	1	0.01
Early Active	2	-0.04	Late Active	2	-0.12	Steady Active	1	0.00
Implementation			**Test**			**Overall**		
Pattern	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**	**Pattern**	**Rank**	**Mean**
Steady Inactive	1	4.16	Early Active	1	0.43	Steady Inactive	1	4.30
Late Active	2	1.80	Late Active	1	0.15	Late Active	2	1.82
Steady Active	2	0.84	Steady Inactive	1	0.12	Steady Active	2	0.86
Early Active	2	0.26	Steady Active	1	0.03	Early Active	2	0.66

***Steady active* shows better performance in reducing coupling and improving cohesion, but it exhibits the worst release-wise refactoring pattern in terms of reducing architectural code smells.** This suggests that excessive refactoring at the implementation or design level could potentially reduce the architectural quality of the code, for example, by making dependencies unstable. It highlights the need for regular maintenance and long-term planning to eliminate architectural code smells. Therefore, it is recommended that developers, if they adopt this pattern, pay attention to the overall structure of the code and avoid making significant changes that might increase architectural code smells while addressing other code smells.

Overall, we observe that the *late active* refactoring pattern exhibits the best performance in terms of reducing code smells and improving code quality measures by code metrics compared to the others. It minimizes increases in all types of code smells while maintaining high cohesion and low coupling. This suggests that a dedicated refactoring approach might be more effective, where refactoring focuses on improving code quality before each release. Building on the state-of-the-art [34], which suggests that dedicating specific timeframes to refactoring improves code quality without considering the adoption of continuous release strategies, we find that refactoring timeframes are most effective when allocated at the end of each release in continuous release strategies. This suggests that prioritizing feature delivery or other development tasks, followed by code cleanup through refactoring, may lead to the highest code quality. Therefore, **it is recommended that developers allocate an incremental and dedicated time frame for refactoring before each release.**

Using the Mann-Whitney U test [30], we compare the frequency of different refactoring types between minor-to-major and minor-to-minor releases. Our analysis shows that package and variable level refactorings are significantly more frequent in minor-to-major releases, possibly due to package updates or the need to make incompatible API changes in major versions [42].

The *late active* release-wise refactoring pattern demonstrates the best performance in decreasing coupling, increasing cohesion, and reducing code smells. Furthermore, while the *steady active* release-wise refactoring pattern shows a good performance in reducing coupling and improving cohesion, it demonstrates the poorest performance in reducing architectural code smells. This suggests that excessive refactoring may play an opposite role for architectural code smells.

3.3 RQ3. Does the usage of release-wise refactoring patterns change over time, and how do developers switch from one pattern to another?

3.3.1 Motivation. We identify four primary refactoring patterns related to software releases: *early active*, *late active*, *steady active*, and *steady inactive*. Among these, the *late active* release-wise refactoring pattern is observed to be the most effective. However, the utilization of these patterns in different stages of development or how to switch from one pattern to another is not clearly understood, specifically in terms of their implications to code quality. This research question aims to provide deeper insights into the evolution of refactoring release patterns over time, examining how and why developers switch between these patterns and their respective impacts on refactoring quality. This understanding allows practitioners to comprehend existing patterns and evaluate how transitioning between them can enhance code quality through refactoring efforts. We aim to provide a road map that developers can follow when performing refactoring in continuous release cycles.

3.3.2 Approach. To analyze how the utilization of release-wise refactoring patterns changes over time for a project, we define four stages of development over the lifetime of a project and measure the utilization rate of each pattern throughout these stages. Additionally, we assess the probability of switching from one pattern to another and its effect on code quality. The following presents a breakdown of each step in this process:

Analyzing pattern utilization: To understand and measure the utilization of refactoring release patterns over time, we follow a similar methodology as in previous work [34]. We divide the consecutive release lengths of all projects into three time quartiles, each containing an equal number of projects, representing different stages of the project's lifecycle. Using these three time quartiles, we distribute the releases of all projects as follows:

- **Early stage:** First 3 releases (1^{st} quartile).
- **Middle stage:** Releases 4 to 6 (1^{st} to 2^{nd} quartile).
- **Late stage:** Releases 7 to 13 (2^{nd} to 3^{rd} quartiles).
- **Last stage:** More than 13 releases (3^{rd} quartile).

This approach allows us to analyze changes in pattern adoption over time by comparing the distribution of each release-wise refactoring pattern across different stages of development. This helps identify the most and least popular patterns and shows how their popularity shifts as software undergoes more releases or matures.

Measuring quality of pattern transitions: To study and analyze the transitions between patterns and their relationship with code quality, we conduct a two-step experiment. First, we calculate the transition probabilities between patterns using Markov chains [37], which model the likelihood of moving from one state (*i.e.,* release-wise refactoring pattern) to another, where each

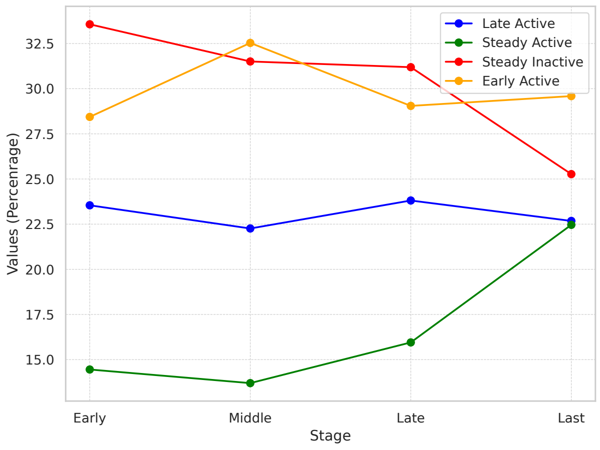

Fig. 5. The distribution of different refactoring release patterns at various stages of development. Each point represents the percentage of utilization of each pattern on the y-axis, while the x-axis indicates the development stage at which the pattern is utilized.

transition depends on the previous state. Second, we use Scott-Knott-ESD [56, 57] to measure and rank code smell changes associated with each transition, grouping them based on the significance of the reduction rate of code smells between the transitions. This two-step approach allows us to identify common shifts in refactoring practices, leading to a deeper understanding of how and why developers adopt and change refactoring patterns throughout the project's lifecycle and their impact on code quality.

3.3.3 Findings. The distribution of the utilization of release-wise refactoring patterns across different stages of development is depicted in Figure 5. Furthermore, the results of our two-step approach, which incorporates Scott-Knott-ESD with the probabilities of transition from one approach to another, are shown in Table 4.

Developers are more likely to repeat the previous refactoring pattern in later releases, as shown in Table 4, particularly in the *steady inactive* (39%) refactoring pattern, where the refactoring density is the lowest compared to other release-wise refactoring patterns. Refactoring transitions may result from developers' choices or be driven by code properties. For instance, the 39% probability of transitioning from steady inactive to steady inactive may indicate a prioritization of feature delivery over code maintenance, a higher demand for new features, or no need for refactoring. Furthermore, the 30% probability of transitioning from late active to early active may suggest postponed refactorings to the next release. Similarly, the 31% probability of switching from steady inactive to early active may indicate code preparation for development, where the current state of the code is not ideally maintainable. The *steady inactive* state is the most probable, with a 39% probability of occurring, and shows the least effectiveness in eliminating the total code smells. It ranks worst (Rank 1) in achieving high cohesion (mean 6.72), worst in reducing complexity (mean 2.73), and second worst in lowering coupling (mean 7.02). Therefore, **it is recommended to avoid staying in the steady *inactive state* if the goal is to improve quality.**

Even though the likelihood of transitioning from a *steady inactive* to a *steady active* pattern is the lowest (9%), the results from the Scott-Knott-ESD show that switching from

Table 4. Results of the Scott-Knott-ESD test results on the total changes in code smells (per 1,000 lines of code) after transitioning between release-wise refactoring patterns. Patterns within the same cluster (rank) are represented by the same color. Positive mean code smell values indicate an increase in the mean of code smells, while negative code smell values represent a decrease in the mean of code smells (*i.e.,* increase in code quality). Abbreviations used: Arch. = Architecture, Des. = Design, Impl. = Implementation, Cohes. = Cohesion, Coup. = Coupling, Compl. = Complexity. The Prob. column indicates the transition probabilities obtained from the Markov chains.

					Code Smells			**Code Metrics**		
From	**To**	**Rank**	**Prob.**	**Total**	**Arch.**	**Des.**	**Impl.**	**Coh.**	**Coup.**	**Compl.**
Steady Inactive	Steady Inactive	1	0.39	5.03	-0.05	-0.03	5.1	6.72	7.02	2.73
Early Active	Steady Inactive	2	0.27	3.98	0.01	0.12	3.61	5.10	7.24	2.19
Steady Active	Steady Inactive	3	0.19	2.80	0.02	0.09	2.46	2.41	2.54	1.07
Steady Active	Late Active	4	0.24	2.78	0.01	-0.22	3.11	2.93	3.44	1.24
Early Active	Steady Active	5	0.18	2.28	0.02	-0.01	1.92	2.95	3.04	1.27
Early Active	Early Active	6	0.32	2.11	-0.08	0.07	1.60	2.95	3.82	1.53
Steady Inactive	Late Active	7	0.21	1.73	-0.05	-0.22	2.01	3.67	4.45	1.55
Early Active	Late Active	8	0.23	1.29	0.00	0.35	0.59	2.75	3.35	1.31
Steady Inactive	Early Active	9	0.31	0.90	0.02	-0.22	0.86	3.35	4.07	1.54
Late Active	Steady Active	10	0.17	0.86	-0.01	0.01	0.74	2.29	2.72	1.08
Late Active	Steady Inactive	11	0.28	0.85	-0.05	-0.25	0.93	4.27	4.70	1.80
Steady Active	Steady Active	12	0.32	0.58	0.06	-0.05	0.60	3.71	3.83	1.69
Steady Active	Early Active	13	0.25	0.42	-0.06	0.11	0.19	2.48	2.78	1.03
Steady Inactive	Steady Active	14	0.09	-0.11	0.01	-0.08	-0.08	2.26	2.57	0.97
Late Active	Late Active	14	0.25	-0.71	-0.09	-0.36	-0.20	2.65	3.32	1.23
Late Active	Early Active	14	0.30	-0.81	-0.01	-0.33	-1.09	2.78	3.53	1.30

***steady inactive* to *steady active* can effectively eliminate code smells and improve code quality.** Specifically, this transition exhibits the second-best improvement in reducing coupling (mean of 2.57), the best improvement in achieving higher cohesion (mean of 2.26), and the best reduction in complexity (mean of 0.97), but a mean increase in architecture smells (mean of 0.01). Additionally, the 9% switch probability rate may indicate less interest or difficulties in shifting from the steady inactive to the *steady active* pattern. **It is therefore recommended when switching from the steady inactive pattern, characterized by minimal refactoring throughout the release, to the *steady active* pattern, which involves consistent more intensive refactoring throughout the release, one should be aware of potential architectural problems that may arise from the sudden increased refactoring.**

Figure 6 illustrates the possible transitions between patterns, highlighting their mean relationship with code smell reduction, reported from the Scott-Knott-ESD test. Safe states, where remaining consistently reduces code smells, are marked in green, while not-safe states, where no transition exists to another state without increasing code smells, are marked in red. As shown, switching from the *late active* to the *early active* pattern positively affects quality, reducing code smells by a mean of 0.81. However, switching back to *late active* results in a decrease in quality, with a mean increase in code smells of 1.29. Therefore, **it is recommended to avoid deferring refactorings to post-release and instead perform refactorings and cleanings closer to before release dates.**

As shown in Figure 5, **the increased utilization of the *steady active* release-wise refactoring pattern by 8% from the early to the later stages of development suggests more frequent and intensive refactoring efforts compared to earlier stages, where usage decreases by 8.3%.** This observation may indicate that as projects become mature, they require more maintenance, and the frequency of active refactoring patterns increases over time. Conversely, the reduced usage

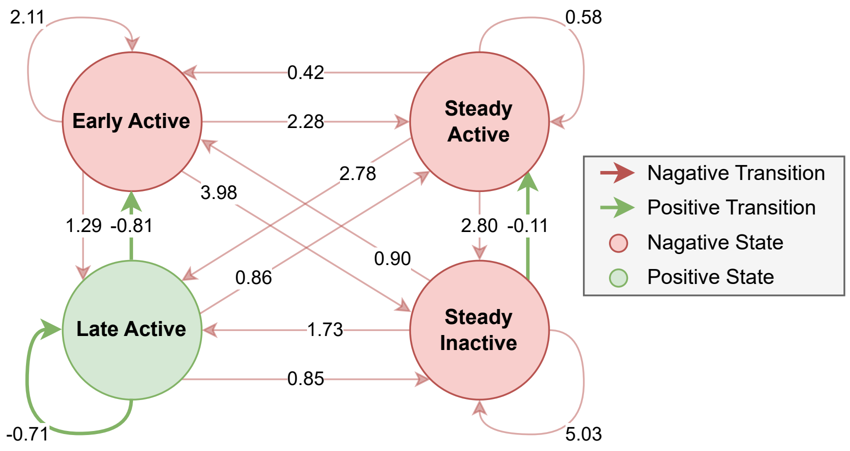

Fig. 6. Summary of positive transitions (green lines) and negative transitions (red lines). Positive transitions indicate an average increase in code smells after adopting each pattern, while negative transitions show a decrease in code smells. The green circle represents a safe transition that can be adopted repeatedly.

of the *steady inactive* release-wise refactoring pattern may be attributed to the growing need for refactoring as the project progresses.

Developers tend to repeat previously used patterns rather than frequently switching to different ones. The usage of the *steady active* pattern increases as projects progress; however, *late active* may be a better alternative, which is the most reliable release-wise refactoring pattern and is considered a safe pattern for continuous use. Moreover, postponing refactoring to the next release (*i.e.,* early active) can temporarily increase quality but may lead to decreased software quality if reverting to the *late active* state. The *steady inactive* release-wise refactoring pattern can be followed by the *steady active* pattern for improved quality.

4 Threats to Validity

In this section, we examine the potential threats to the validity of our study.

Threats to Construct validity. Indicate the data preparation and feature selection. For generating refactoring time series, we opt to calculate the refactored lines of code divided by the overall code churn. The refactored LOC is calculated from the 102 types of refactoring detected by RMiner. However, if there are additional types of refactoring that RMiner does not detect, we cannot capture them. Therefore, using a different set of refactorings would reflect different numbers of refactoring lines in our generated refactoring release time series. When selecting commits for each release, particularly in the mainline release strategy, we consider only those commits merged into the release branch. However, since commit histories can contain multiple branches evolving in parallel, other branches under development but not merged into the release branch may represent additional development efforts that were not completed for release. For the external features that impact the switching, we measure the features detectable from the commit messages and project repository. However, incorporating more socio-technical information could provide additional insights into the causes of the switching between patterns. When counting contributors, we use git log to identify contributors; however, we acknowledge that some developers may have multiple accounts or commit anonymously. Moreover, we search for the keyword *bot* in the commit history and find that 0.3% of commits contain this keyword. Therefore, we recognize that some commits may have been made by bots. Lastly, for measuring refactoring release pattern quality, we use code smells as indicators of refactoring quality. However, incorporating additional metrics, including more code metrics and attributes related to code quality—such as reliability and bug resolution rates—could provide further insights into the potential benefits of different refactoring release patterns on various aspects of the code.

Threats to Internal validity. Indicate the validity of the methods applied in our study. We choose the Java language domain for our studied projects because Java has the most reliable refactoring tool [60, 61], which detects the highest number of refactoring types. However, if comparable tools are available for other languages, studying projects written in those languages could reveal additional refactoring patterns. For the number of clusters (*i.e.,* refactoring release patterns), we opt for the cluster number with the most distinct centroids while maintaining good separation of data on the scatter plot. This approach reveals the most dominant refactoring release patterns. However, some minority refactorings could be miscategorized into these clusters. Additionally, our analysis is based on cluster centroids, so there may be some outliers that do not fit neatly into the identified patterns. To determine the quality of release-wise refactoring patterns, we rely on code smells with the addition of code metrics as quality indicators. However, other metrics may also provide valuable insights and could impact the quality assessment differently.

Threats to External validity. Concern about the generalizability of our approach. Our experiments and results are based exclusively on analyzing open-source projects that follow the standard release naming convention. Consequently, our conclusions may not apply to other projects, particularly those in different domains. Since our analysis is restricted to projects written in Java, the findings may not extend to projects written in other programming languages. Furthermore, our focus on identifying common patterns means that less common and infrequent patterns may be overlooked.

5 Implications

In this section, we provide the implications of our study for the practitioners and tool makers.

Balancing between refactoring and software development. Our results reveal that the *late active* refactoring pattern, characterized by increasing refactorings as the release approaches its end, leads to more maintainable software. Therefore, we encourage practitioners to focus on development tasks early in the release cycle and to concentrate on cleaning and performing refactoring before the release to ensure higher-quality code.

External impacts on release-wise refactoring patterns and the effects of switching between release patterns. Our results demonstrate that the use of the *steady active* release-wise refactoring pattern increases over time, while the *steady inactive* pattern decreases. This trend may be attributed to the growing refactoring needs as projects mature. However, adopting a *steady active* pattern may cause architectural problems. Therefore, it is recommended to use a refactoring pattern, preferably *late active*, to achieve higher quality code.

Providing common strategies and unique vocabulary. By identifying four refactoring release patterns, our study offers a set of unique release-wise refactoring patterns that developers choose. These refactoring release patterns also establish a common vocabulary for project managers and stakeholders when managing refactoring release strategies.

Considering different refactoring strategies when studying software development. Our results reveal that developers do not always follow the same refactoring pattern. Therefore, we encourage researchers to account for varying refactoring behaviors within releases when studying software releases or when treating releases as a factor in their analysis.

Recognizing varied refactoring patterns for toolmakers. Our study reveals various strategies in the adoption of refactoring and different transitions between patterns. We encourage refactoring tool makers to consider this aspect, recognizing that not all developers consistently prefer to perform refactoring in their daily development routines. Instead, based on the specific needs, developers may require dedicated timeframes for refactoring tasks.

6 Related Work

In this section, we review the literature related to refactoring strategies, refactoring detection, and release quality.

Refactoring patterns. Floss and Root Canal are two refactoring tactics identified in previous studies [13, 34, 55]. These terms refer to long-term refactoring practices that teams apply to respond to maintenance requests. Tsantalis *et al.* [59] perform an empirical study on refactoring in three projects and observe that refactoring activities significantly increase before the release. Noei *et al.* [34] study long-term and short-term (*i.e.,* weekly) refactoring strategies and identify that root canal refactoring patterns are correlated with higher quality code. Liu *et al.* [26] study the most frequently applied patterns and identify floss as the most frequently adopted refactoring pattern. **In this study, we explore the existence of refactoring patterns within releases and identify four variations of refactoring release strategies and their evolution over time. Moreover, we study their relationship with code quality.**

Refactoring and code smells. Previous research has investigated the connection between refactoring, code smells, and code quality. Bibiano *et al.* [6] explore batch refactoring and the support of automated refactoring tools on batch refactorings, measuring their success in reducing code smells. Fontana *et al.* [12] explore refactoring tools and measure their limitations in removing code smells. Noei *et al.* [34] measure the relationship between long-term and short-term refactoring strategies and code smells. Cinnéide *et al.* [38] report that even though refactoring is aimed at eliminating code smells, developers are not motivated to eliminate code smells by refactoring. Previous studies [6, 8, 12, 34, 53, 65] link refactoring with code smells. Thus, we use code smells as quality indicators for refactoring to measure the effectiveness of refactoring release patterns on code quality. **In this work we provide a quantitative study to explore and measure the relationship between different refactoring release patterns and code quality.**

Release and maintenance effort. Previous studies have shown that maintenance can have a relationship with releasing new versions. Baumgartner [5] utilizes a Large Language Model (LLM) and develops an AI-driven pipeline to eliminate data clumps during the release. Saidani [46] explores refactoring after adopting CI/CD and finds a drop in refactoring frequency and application after implementing CI/CD pipelines. Ding et al. [10] study the tests of the release and explain how tests cannot reflect the code's performance. **In this study, we measure the frequent practices of developers in switching and utilizing release-wise refactoring patterns over time.**

Refactoring detection. Various tools have been proposed for detecting refactoring actions in software history across different programming languages. However, the available refactoring detection tools in Java can identify the most types of refactoring. Silva *et al.* [52] present RefDiff, a refactoring detection tool designed for Java, C, and JavaScript, capable of identifying up to 10 types of refactoring. Moghadam*et al.* [32] introduce RefDetect, which can detect up to 27 refactoring types in Java and C++. Noei *et al.* [35] introduce MLRefScanner, a tool capable of detecting refactoring commits in Python machine learning projects. Atwi*et al.* [3] introduce PyRef, capable of identifying up to 11 refactoring operations in Python. Tsantalis*et al.* [60, 61] present RMiner, capable of identifying 99 refactoring operations in Java codebases. **In this work, we primarily use Java as the language for studying projects due to its comprehensive tooling support, which accurately reflects realistic refactorings during the release cycle.**

7 Conclusion

In this study, we analyze 207 popular and active open-source Java projects to identify the common refactoring patterns employed by developers within software release cycles. We first examine refactoring density frequencies and analyze changes in refactoring practices throughout the release cycle, identifying four major refactoring release patterns: *early active*, *late active*, *steady active*, and *steady inactive*. We find that the *late active* release-wise refactoring pattern, which is consistently used across different stages of development, represents the best refactoring practice to be applied continuously. This pattern is characterized by consistent class, package, and organizational-level refactoring throughout the release, with an increased focus on method and variable-level refactorings as the release date approaches—leading to better code quality post-release. However, the *steady active* pattern is increasingly adopted by practitioners as projects progress. While the *steady active* pattern, characterized by a high density of refactoring throughout the release, may improve software quality, it can also introduce architectural smells. For future work, we aim to explore automated methods for assessing releases in response to maintenance needs.

8 Data Availability

The replication package of the study is available online [36].

Acknowledgments

We sincerely thank Dr. Bram Adams for his inspiration in the initial formation of this research idea.

References

[1] Bram Adams and Shane McIntosh. 2016. Modern release engineering in a nutshell–why researchers should care. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 5. IEEE, 78–90.

[2] Hiroshi Akima. 1978. A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points. *ACM Transactions on Mathematical Software (TOMS)* 4, 2 (1978), 148–159.

[3] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. 2021. PyRef: refactoring detection in Python projects. In *2021 IEEE 21st international working conference on source code analysis and manipulation (SCAM)*. IEEE, 136–141.

[4] Richard Baskerville and Jan Pries-Heje. 2004. Short cycle time systems development. *Information Systems Journal* 14, 3 (2004), 237–264.

[5] Nils Baumgartner, Padma Iyenghar, Timo Schoemaker, and Elke Pulvermüller. 2024. AI-Driven Refactoring: A Pipeline for Identifying and Correcting Data Clumps in Git Repositories. *Electronics* 13, 9 (2024), 1644.

[6] Ana Carla Bibiano, Eduardo Fernandes, Daniel Oliveira, Alessandro Garcia, Marcos Kalinowski, Baldoino Fonseca, Roberto Oliveira, Anderson Oliveira, and Diego Cedrim. 2019. 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, 1–11.

[7] Luís Carvalho and João Costa Seco. 2021. Deep semantic versioning for evolution and variability. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming*. 1–13.

[8] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. 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*. 465–475.

[9] Marco Cuturi and Mathieu Blondel. 2017. Soft-dtw: a differentiable loss function for time-series. In *International conference on machine learning*. PMLR, 894–903.

[10] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1435–1446.

[11] V. Djurovic. 2025. Automatic Semantic Versioning in CI/CD Pipelines. Bitshift. https://www.bitshifted.co/blog/automatic-semantic-versioning-cicd-pipelines/ [Accessed 10-02-2025].

[12] Francesca Arcelli Fontana, Marco Mangiacavalli, Domenico Pochiero, and Marco Zanoni. 2015. On experimenting refactoring tools to remove code smells. In *Scientific Workshop Proceedings of the XP2015*. 1–8.

[13] Martin Fowler, Kent Beck, J Brant, W Opdyke, and D Roberts. 1999. Refactoring: Improving the Design of Existing Code Addison-Wesley Professional. *Berkeley, CA, USA* (1999).

[14] Milton Friedman. 1962. The interpolation of time series by related series. *J. Amer. Statist. Assoc.* 57, 300 (1962), 729–757.

[15] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 2021. One thousand and one stories: a large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1303–1313.

[16] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 426–437.

[17] James Ivers, Ipek Ozkaya, Robert L Nord, and Chris Seifried. 2020. Next generation automated software evolution refactoring at scale. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1521–1524.

[18] Thorn Jansen, Zeinab Abou Khalil, Eleni Constantinou, and Tom Mens. 2021. Does the duration of rapid release cycles affect the bug handling activity?. In *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*. IEEE, 17–24.

[19] Moona Kanwal, Najeed A Khan, Najma Ismat, Aftab A Khan, and Muzammil Ahmad Khan. 2024. Machine Learning Approach to Classification of Online Users by Exploiting Information Seeking Behavior. *IEEE Access* (2024).

[20] Foutse Khomh, Bram Adams, Tejinder Dhaliwal, and Ying Zou. 2015. Understanding the impact of rapid releases on software quality: The case of firefox. *Empirical Software Engineering* 20 (2015), 336–373.

[21] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. 2012. Do faster releases improve software quality? an empirical case study of mozilla firefox. In *2012 9th IEEE working conference on mining software repositories (MSR)*. IEEE, 179–188.

[22] Thijs T Klooster. 2021. *Effectiveness and Scalability of Fuzzing Techniques in CI/CD Pipelines.* Ph. D. Dissertation.

[23] Rrezarta Krasniqi and Jane Cleland-Huang. 2020. Enhancing source code refactoring detection with explanations from commit messages. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 512–516.

[24] WH Kruskal. 1952. Kruskall–Wallis one way analysis of variance. *J Am Stat Assoc* 47, 260 (1952), 583–621.

[25] Mathieu Lepot, Jean-Baptiste Aubin, and François HLR Clemens. 2017. Interpolation in time series: An introductive overview of existing methods, their performance criteria and uncertainty assessment. *Water* 9, 10 (2017), 796.

[26] Hui Liu, Yuan Gao, and Zhendong Niu. 2012. An initial study on refactoring tactics. In *2012 IEEE 36th Annual Computer Software and Applications Conference*. IEEE, 213–218.

[27] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship.* Pearson Education.

[28] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[29] Mary L McHugh. 2013. The chi-square test of independence. *Biochemia medica* 23, 2 (2013), 143–149.

[30] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[31] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.

[32] Iman Hemati Moghadam, Mel Ó Cinnéide, Faezeh Zarepour, and Mohamad Aref Jahanmir. 2021. Refdetect: A multi-language refactoring detection tool based on string alignment. *IEEE Access* 9 (2021), 86698–86727.

[33] Emerson Murphy-Hill and Andrew P Black. 2008. Refactoring tools: Fitness for purpose. *IEEE software* 25, 5 (2008), 38–44.

[34] Shayan Noei, Heng Li, Stefanos Georgiou, and Ying Zou. 2023. An Empirical Study of Refactoring Rhythms and Tactics in the Software Development Process. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5103–5119.

[35] Shayan Noei, Heng Li, and Ying Zou. 2024. Detecting Refactoring Commits in Machine Learning Python Projects: A Machine Learning-Based Approach. *ACM Transactions on Software Engineering and Methodology* (2024).

[36] Shayan Noei, Heng Li, and Ying Zou. 2024. Replication Package. https://github.com/Software-Evolution-Analytics-Lab-SEAL/FSE2025-Refactoring-Release.

[37] James R Norris. 1998. *Markov chains.* Number 2. Cambridge university press.

[38] Mel Ó Cinnéide, Aiko Yamashita, and Steve Counsell. 2016. Measuring refactoring benefits: a survey of the evidence. In *Proceedings of the 1st International Workshop on Software Refactoring*. 9–12.

[39] Anthony Peruma, Steven Simmons, Eman Abdullah AlOmar, Christian D Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. How do i refactor this? An empirical study on refactoring trends and topics in Stack Overflow. *Empirical Software Engineering* 27, 1 (2022), 11.

[40] Shaun Phillips, Guenther Ruhe, and Jonathan Sillito. 2012. Information needs for integration decisions in the release process of large-scale parallel development. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. 1371–1380.

[41] Shaun Phillips, Jonathan Sillito, and Rob Walker. 2011. Branching and merging: an investigation into current version control practices. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*. 9–15.

[42] Tom Preston-Werner. 2024. Semantic Versioning 2.0.0. https://semver.org/ [Accessed 30-05-2024].

[43] pypl. 2024. PYPL PopularitY of Programming Language index. https://pypl.github.io/PYPL.html/ [Accessed 30-05-2024].

[44] Hajo A Reijers and Irene TP Vanderfeesten. 2004. Cohesion and coupling metrics for workflow process design. In *International conference on business process management*. Springer, 290–305.

[45] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.

[46] Islem Saidani, Ali Ouni, Mohamed Wiem Mkaouer, and Fabio Palomba. 2021. On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Information and Software Technology* 138 (2021), 106618.

[47] SciTools. 2024. Understand: The Software Developer's Multi-Tool. https://scitools.com/. [Accessed 12-Jul-2024].

[48] Ketan Rajshekhar Shahapure and Charles Nicholas. 2020. Cluster quality analysis using silhouette score. In *2020 IEEE 7th international conference on data science and advanced analytics (DSAA)*. IEEE, 747–748.

[49] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access* 5 (2017), 3909–3943.

[50] Tushar Sharma. 2019. DesigniteJava (Enterprise). doi:10.5281/zenodo.3401802 http://www.designite-tools.com/designitejava.

[51] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. Designite: A Software Design Quality Assessment Tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities (BRIDGE '16)*. Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/2896935.2896938

[52] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2020. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2786–2802.

[53] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*. 858–870.

[54] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. 2012. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering* 39, 8 (2012), 1144–1156.

[55] Leonardo Sousa, Willian Oizumi, Alessandro Garcia, Anderson Oliveira, Diego Cedrim, and Carlos Lucena. 2020. When Are Smells Indicators of Architectural Refactoring Opportunities: A Study of 50 Software Projects. In *Proceedings of the 28th International Conference on Program Comprehension*. 354–365.

[56] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2017. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. 1 (2017).

[57] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2018. The Impact of Automated Parameter Optimization for Defect Prediction Models. (2018).

[58] tiobe. 2024. index | TIOBE - The Software Quality Company. https://www.tiobe.com/tiobe-index/ [Accessed 30-05-2024].

[59] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity.. In *CASCON*. 132–146.

[60] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2020).

[61] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 483–494.

[62] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[63] Aiko Yamashita and Leon Moonen. 2012. Do code smells reflect important maintainability aspects?. In *Proceedings of 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 306–315.

[64] O. Yeremenko. 2025. Semantic Versioning for CI/CD. Beetroot Careers. https://career.beetroot.co/understanding-semantic-versioning-for-ci-cd/ [Accessed 10-02-2025].

[65] Norihiro Yoshida, Tsubasa Saika, Eunjong Choi, Ali Ouni, and Katsuro Inoue. 2016. Revisiting the relationship between code smells and refactoring. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. IEEE, 1–4.

[66] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. 2021. Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 471–482.

Received 2024-09-13; accepted 2025-01-14