Does Shortening the Release Cycle Affect Refactoring Activities: A Case Study of the JDT Core, Platform SWT, and UI projects

Olivier Nourry, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi

Principles Of Software engineering and programming Languages Group (POSL), Kyushu University, Fukuoka, Japan

Abstract

[**Context**] Several large-scale companies such as Google and Netflix chose to adopt short release cycles (e.g., rapid releases) in recent years. Although this allows these companies to provide updates and features faster for their users, it also causes developers to have less time to dedicate to development activities other than feature development. [**Objective**] In this paper, we investigate how refactoring activities were impacted by the adoption of shorter releases. [**Method**] We extract all refactorings applied over a period of two years during traditional yearly releases and almost two years during shorter quarterly releases in three Eclipse projects. We then analyze both time periods' refactoring activities to understand how refactoring activities can be impacted by shortening the release cycles. [**Results**] We observe reduced refactoring activities in one project and a decrease in more complex refactoring operations after shortening the release cycles. We also find that weekly efforts dedicated to refactoring activities was lower across all projects after shortening the release cycles. [**Conclusion**] Shorter releases may impact software development tasks such as refactoring in unintended ways. Not applying specific types of refactoring may also affect the software's quality in the long term. Using this case study and past work on shorter releases, potential short release adopters can now better plan their transition to shorter releases knowing which areas of development may be affected.

Email address: oliviern@posl.ait.kyushu-u.ac.jp, {kashiwa, kamei, ubayashi}@ait.kyushu-u.ac.jp (Olivier Nourry, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi)

Keywords: Release Policy Change, Refactoring, Release Engineering

1. Introduction

Refactoring in software systems has been investigated extensively throughout the years. Multiple aspects such as the impacts on code quality [1] and testing activities [2] have been investigated in order to determine how important it is to refactor the code in a software development project. Past studies [3, 4, 5] have concluded that refactoring is beneficial to a software development project and that it is worth dedicating some development time to refactor the code.

Modern-day software companies, however, have started changing the way they develop software systems. Over the past few years, large companies such as Google, Facebook, Firefox, and Netflix have all decided to adopt a short release cycle model for various reasons such as keeping up with customer expectations or keeping up with competitors [6].

This short release cycle model has been referred to as rapid releases (RR), or short releases (SR), and the old yearly release model as traditional releases (TR). Several studies have been conducted in order to better understand the impacts of shorter releases by comparing various metrics and activities before and after shortening the release cycles. For example, a previous study [2] showed that shorter releases' builds contain more code commits than traditional releases' builds. Another study [1] also found that short release cycles had a significant impact on the number of files being affected per commit. The code quality, number of bugs, integration, and testing in shorter releases have also all been subject to investigation in past studies [1, 2, 7] but to the best of our knowledge, the impacts of shortening the release cycles on refactoring activities have not yet been investigated. In this study, we therefore aim to further contribute to the empirical knowledge related to the impacts of shortening the release cycles on software development activities.

In this case study, we investigate if shortening the release cycles had any impacts on refactoring activities in three software development projects. Due to the possible negative impacts of not refactoring, it is important for potential short release adopters to be aware of the possible downsides related to refactoring activities that shorter releases can bring to a project. We examine four aspects of refactoring activities: the amount of refactoring done, the efforts devoted to refactoring, the type of refactoring applied in the code, and the human resources dedicated to refactoring. These aspects allow us to

draw an overall picture of refactoring during short releases and traditional releases in order to determine if the refactoring activities are impacted after adopting a shorter release cycle.

Paper Organization: Section 2 introduces previous studies involving refactoring and short releases. Section 3 describes our dataset selection, its extraction and the filtering process, and Section 4 presents our results. In Section 5, we summarize our findings, discuss their implications and address some of the threats to the validity of this study. We conclude in Section 6 where we briefly summarize our findings and make our final comments on refactoring and short releases.

2. Background

2.1. Refactoring

Refactoring has been widely defined as changes that improve code quality but preserve the software system's external behavior [8]. Thanks to the improvement of refactoring detection algorithms [9, 10, 11, 12, 13] that automatically identify refactoring activities from source code histories, many researchers have conducted empirical studies [3, 14, 15, 16, 17, 18, 19] on software evolution and software refactoring.

Refactoring and Software Quality. In a previous study [20], Lacerda et al. performed a systematic literature review and concluded that refactoring has a strong relationship with code quality attributes such as understandability, reusability, and maintainability. During their literature review, they also identified that code quality is one of the relationship linking code smells and refactoring. Code quality is therefore a good representation of how well refactoring can remove code smells.

While leading a case study [21] on the impacts of refactoring on software quality and productivity in an agile environment, Moser et al. also found empirical evidence that refactoring improves code quality factors, reduces code complexity and coupling, and improves cohesion. Their results also showed empirical evidence that refactoring increases development productivity. In another study [22], Moser et al. also proposed a new methodology to assess if refactoring improves quality. Using their new proposed approach, they conducted a case study in a near industrial environment and found that refactoring has a positive effect on reusability and promotes ad-hoc reuse of object-oriented classes.

Kim et al. [4] found that there is an increase in the number of bug fixes

after API-level refactorings while investigating the role of refactorings in three evolving open source projects. They also found that less time is required to fix bugs after API-level refactorings are applied to the code.

While refactoring has many benefits, refactoring code changes carry the same risks associated with any other codebase change such as the introduction of new defects as shown in a previous study [23] that found that refactoring can account for a significant part of API breaking changes.

Refactoring automation. To mitigate the risks of introducing new defects, it has been suggested that refactoring tools should be used whenever possible. Multiple automated tools have been developed over time to ease the process of refactoring for developers. These tools were designed to help developers in several ways ranging from automated refactoring suggestion [24], to automated refactoring validation that checks if a manual refactoring was applied correctly [25] and all the way to fully automated refactoring applications [26]. Popular IDEs such as IntelliJ IDEA and Eclipse have also added refactoring support by adding their own built-in refactoring tools to their software. These tools generally cover a wide range of refactoring patterns and can help developers save time when refactoring their code.

Xing and Stroulia [27] studied the proportion of code modifications for refactoring changes and which types are the most frequent types of refactorings being applied in the Eclipse JDT project. They found that about 70% of structural changes may be due to refactorings code changes and that 60% of refactorings can be automatically applied by a refactoring tool if the refactoring tool can gather the information relevant to the refactoring being applied.

Although refactoring tools are becoming more commonly used, Silva et al. [16] found that many developers still do not trust automated refactoring for complex cases or believe that automation does not need to be used in trivial cases. A previous study led by Brett et al. [28] also found that refactoring tools themselves can contain multiple bugs. It is therefore very important that refactoring is done properly in order to avoid introducing more defects. Because each release is usually longer when using a traditional release cycle model, the evolution of the software system is well controlled due to developers planning ahead of time how much time is to be allocated to each refactoring task. New release methodologies however reduce that planning time a lot by shortening release cycles [29]. This paper differs from previous refactoring and software evolution studies by taking into account this new shorter releases approach to software development.

2.2. Short Releases

Many software development projects have been switching from a traditional release cycle to a shorter release cycle to deliver new features faster (a few weeks to a few months to ship a major release) rather than providing them yearly. This satisfies users and also allows developers to receive quick feedback from users [30]. For instance, the Firefox and Eclipse projects shortened the release intervals for major/main releases, respectively. Specifically, the Firefox project started releasing a product in 6 weeks rather than yearly. From Eclipse 4.9, the Eclipse project abandoned service releases (i.e., patch releases) and replaced all the service releases with main releases.

Many studies have examined the impacts of shortening the release cycles on development activities (e.g., testing, bug-fixing, and integrating) to find that shortening release intervals affects development activities [2, 7, 30, 31, 32, 33, 34, 35]. For bug-fixing activities, the average bug-fixing time is reduced by adopting shorter releases [31]. As for testing activities, shorter releases narrow the scope of the tests and increase the number of test executions [2].

Research motivation. In the field of the consumer market, Tonietto et al. [36] reported about code behaviour changes when developers are under time pressure. When time is limited, people tend to perform fewer tasks and are less likely to complete time-consuming tasks because of the perception of having less time than they actually have when compared to an unbounded time [36]. In this way, while having limited time, developers change the scope of their tasks and the amount of effort put on specific tasks in order to optimize their activities.

Refactoring, for instance, is not trivial work when it must be completed in a short amount of time. In fact, past studies have found that certain types of refactoring can be quite challenging even without time limitations. For example, Garrido et al. [37] have previously studied the challenges of refactoring a C program file that uses preprocessor directives. Vassallo et al. [38] also investigated the advantages and barriers of continuous refactoring in CI and found out that *the main barrier to continuous refactoring was the lack of time*. The same study also states that this is even more true in agile projects because there is more emphasis on new changes. We use this finding by Vassallo et al. to motivate our study of the impacts of shorter releases on refactoring practices.

To the best of our knowledge, our work is the first to examine the impact

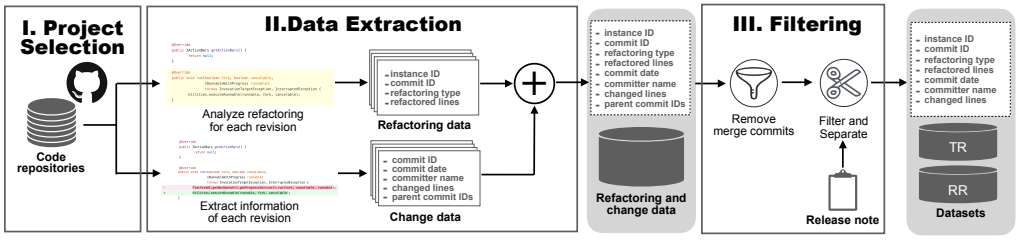

Figure 1: An overview of the data extraction procedure

of switching from traditional yearly releases to shorter releases on refactoring activities. Since refactoring plays a crucial role in software development, this case study provides further insight for projects planning to transition to shorter releases. In this study, we focus on the process change and compare the amounts, efforts, contents, and human resources of refactorings between traditional releases and shorter releases in order to investigate if shortening the release cycles had any impacts on refactoring activities.

3. Study Design

This section explains the design of our case study in order to address our research questions introduced in Section 3.1. To measure the impacts of switching to a shorter release cycle on refactoring activities, we compare two datasets derived from traditional yearly releases and shorter quarterly releases. Figure 1 depicts an overview of the data extraction process. We first apply a refactoring analysis tool to each revision in code repositories. Next, we measure the relevant metrics and merge them with the outputs of the refactoring tool. Finally, the merged outputs are divided into two datasets based on whether the commit date is before or after the cycle change date.

3.1. Research Questions

In this paper, we structure our study around the following four research questions.

RQ_1: Do developers still refactor as much? (Amount)

RQ_2: Are refactoring efforts affected by shorter releases? (Effort)

RQ_3: Do shorter releases impact what type of refactoring is applied to the code? (Contents)

RQ_4: Does shortening the release cycles affect the human resources dedicated to refactoring activities and their refactoring workload? (Human-resource)

In previous studies [1, 2, 7] that investigate the impacts of shorter releases on various development tasks, three main aspects have been used to measure the impacts: the frequency of the task (RQ1), the characteristics of the task (RQ3) and the human resources dedicated to the task (RQ4). In addition to investigating these three main aspects, in RQ2, we diverge from the main aspects used by previous studies in order to investigate the impacts of shorter releases on refactoring efforts by measuring the amplitude of the refactoring changes. In this paper, we design our research questions in a way that will help paint a broad picture of what impacts, if any, switching to shorter releases has on refactoring activities. We also try to align our analysis with previous studies so that we can compare our results afterwards. We compare the results of our empirical study with previous studies in Table 11 in the discussion section to get an overall view of the possible impacts of shortening the releases on development activities.

3.2. Project Selection

To address our research questions, we perform a case study analyzing refactoring in traditional yearly releases and shorter quarterly releases. The projects used in this study were picked because they satisfied the following three criteria:

$Criterion_1$: **Experience in shortening release cycles.** We need to compare the amount, effort, contents and human-resource of refactoring between traditional and shorter releases. Therefore, our studied systems must have development experience following both a traditional release cycle and a shorter release cycle.

$Criterion_2$: **Java System.** Since we need to analyze refactoring activities, we use a state of the art refactoring analysis tool that can detect various refactoring types (See Section 3.3). However, this tool has the limitation of only being able to analyze Java code. Thus, our studied systems must be written in Java.

$Criterion_3$: **Large-size repositories** In order to acquire reliable results, we need many instances of refactoring. Our studied systems must have at

least 1,000 commits during the traditional releases period and a significant number of commits during the shorter releases period in their master branch.

Because short release cycles are not a widely used approach yet in open-source software development, the number of open source projects that have developed for a significant period of time following both yearly traditional releases and shorter releases is limited. Considering this limiting factor and our three selection criteria, we decided to pick three Java projects from the Eclipse Foundation which are known to be participating in the simultaneous release initiative. These projects are the JDT.CORE, PLATFORM.UI, and PLATFORM.SWT projects, which are mature (over 10 years of development) and still active.

The Eclipse Foundation, which owns these projects, follows a simultaneous release scheme to deliver products in sync with all the projects in Eclipse. Figure 2 depicts the transition of Eclipse's release cycle. So far, the release cycle has undergone changes twice before the adoption of their current release model. Until Eclipse version 4.5, Eclipse delivered yearly new features and bug-fix in the main release in June and then provided patch releases twice in September and February in order to fix bugs found in the main release. In Eclipse 4.6, they added a new patch release (named service release) in December and moved the February patch release to March making all releases released quarterly. Furthermore, from Eclipse 4.8, a shorter release cycle was adopted which abolished patch releases and made every quarterly release into a main release.

3.3. Data Extraction

In order to mine the refactoring patterns, we used the state-of-the-art refactoring analysis tool developed by Tsantalis et al. called RefactoringMiner [10]. RefactoringMiner can detect up to 40 different types of refactoring patterns (as of February 1, 2020)[1]. We used this tool due to its state of the art performance (high precision and high recall [10]) as well as the large diversity of patterns that the tool can detect. The tool can detect simple patterns such as renaming and moving pieces of code or pushing down and pulling up methods. It is also able to detect more time consuming or

[1]We used build "8ccec9050ad6dcc4d575cce3713f5283d5988c5d" since RefactoringMiner is still being developed.

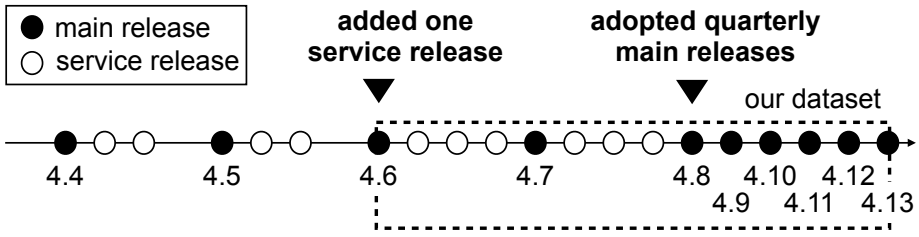

Figure 2: Range of our dataset

complex patterns such as extracting a superclass or merge and split types of refactorings such as split parameter or merge attribute. After cloning the repositories of the studied projects, we ran the RefactoringMiner tool on every commit. The output of each commit contains the information of multiple refactoring instances, including the refactoring types and the refactored lines in the source code.

Additionally, we extracted the basic information of each commit such as commit date, parent commit ids, changed lines, and committer name. We used the commit date and the parent commit ids for our filtering process and to create our datasets. Then, we utilized the changed lines and the committer name for RQ2 and RQ4, respectively. Finally, we merged the basic information with the output of the RefactoringMiner, based on the commit id.

3.4. Filtering

Merge commits. Before making datasets, we need to remove commits with duplicated refactoring instances. Code repositories contain merge commits, which will cause refactoring instances to be redundantly detected even though the same snippet has previously been marked as refactored in the parent commits. Therefore, we eliminate commits with multiple parent commits from our dataset.

Commits during early development periods. Since the goal of this study is to find the impacts of shortening the release cycle, in order to have a fair comparison between the traditional releases data and short releases data, we selected a subset of the traditional releases data that was as close as possible to the switch to shorter releases. Using this subset, we can ensure that we are not comparing early development commits and contributions that are aimed at developing the core of the application with recent contributions

Table 1: Summary of the dataset

Project	Release	Commits	Refactoring Instances	Refactoring Commits	% Refactoring Commits
jdt.core	TR	1,306	3,435	354	27.10%
	SR	618	816	146	23.60%
platform.swt	TR	1,877	3,466	168	9.00%
	SR	1,243	1,409	115	9.30%
platform.ui	TR	1,475	1,914	270	18.30%
	SR	1,340	3,182	224	16.70%

aimed at software maintenance and update [1]. Thus, we do not use the data prior to Eclipse 4.6 (June 22nd, 2016). We would like to note that the yearly release cycle (main release) was broken on Eclipse 4.8 (June 27th, 2018) where quarterly releases were adopted (See Figure 2). All commits authored prior to June 27th, 2018 (Eclipse 4.8) were categorized as traditional release commits and all commits done on June 27th, 2018, and after were categorized as short release commits. Although the release periods have a different length, our data is normalized so that our results are not affected by it. In RQ1 and RQ2, we compare weekly median data to normalize between the two release periods, in RQ3 we normalize using a ranking-based approach, and finally in RQ4 we use percentages and medians to normalize our data.

The main idea behind this choice is that the project would have the same maturity and frequency of code contribution from the contributors if no release cycle change happened. Taking a subset that was too early in the project's life would result in comparing traditional releases commits aimed at developing the core of the application against shorter releases commits which are more oriented towards maintenance and feature additions since the project is much older at that point.

Dataset Summary: The final dataset, as shown in Table 1 is made of traditional releases data and short releases data extending over a similar length of time. Henceforth, traditional releases refer to the time period between July 22nd, 2016, and June 27th, 2018, and short releases refer to the time period after June 27th, 2018.

4. Case Study Results

4.1. (RQ1) Do Developers Still Refactor as Much?

Motivation. Due to the shorter releases, developers do not have as much time to dedicate to tasks not related to feature development and maintenance

such as testing [39, 40, 41, 42] or refactoring. For example, Vassallo et al. [38] found that lack of time is the main barrier to refactoring while investigating continuous refactoring in CI. We therefore investigate if adopting shorter release cycles affects the overall project's refactoring activity by comparing how much refactoring was being done during both release periods.

Approach. Refactoring is not usually done as an everyday task but rather when it is needed to implement new features or when the code becomes unreadable [18]. For these reasons, we decided to aggregate our data to study refactoring activity on a weekly basis.

For RQ1, we focus our results around the median weekly values to compare the refactoring done during the traditional releases period and the shorter releases period. To extract the overall refactoring activity in our chosen projects, we use the following two metrics:

Refactoring Instances: We use refactoring instances as our main metric to see exactly how much refactoring was applied to the code during each period. The number of instances directly reflects how many times the code was changed for refactoring purposes.

Refactoring Commits: We use refactoring commits as a secondary metric to study refactoring activities. The number of refactoring commits gives us an insight as to how often the code is refactored.

We start our investigation by counting the number of refactoring instances per week in each project and extracting the weekly median number of refactoring instances. We then compare the median values between traditional releases and short releases to see if shorter releases impacted how much refactoring is applied to the code. We then proceed to look at refactoring commits to see if there is more weekly refactoring activities after shortening the releases. We extract the weekly median number of commits during the traditional releases period and the shorter releases period and then compare the weekly median percentage of refactoring commits during both periods.

We finally use a Mann-Whitney test for statistical tests and Cliff's delta for effect size, which are non-parametric, to verify if there is a statistically significant difference in the number of refactoring instances and refactoring commits between the traditional releases and the shorter releases. We select these non-parametric tests because refactoring activities sometimes show outliers and are not assured to follow a normal distribution.

Findings 1. We observe a decreased amount of refactoring with a lower number of refactoring instances applied in the JDT.CORE project after shortening the release cycle

As shown in Table 2, we find that the weekly median number of refactoring instances is very similar during both the traditional releases and the shortened releases for the two platform projects. The JDT.CORE project's median refactoring instances per week, however, is twice as much during the traditional releases period than the short releases period. Although both the platform projects show a small increase in the median number of weekly refactoring instances, a Mann-Whitney test on the weekly number of instances for each project confirms that out of the three projects, the JDT.CORE project is the only one showing a statistically significant difference in the number of refactoring instances per week (p-value < 0.01). It is also the only project with a non-negligible effect size for the number of refactoring instances per week. The two platform projects were found to have no statistically significant difference in the weekly number of instances.

Findings 2. We find a decrease in refactoring activities with a lower number of weekly refactoring commits in the JDT.CORE project after adopting shorter releases.

The refactoring commits data shows less of a difference than the refactoring instances data when looking at the weekly median number of refactoring commits. Out of the three projects, we find that only the JDT.CORE project has a difference in the weekly median number of refactoring commits. We however find that this measure does not tell the whole story; when looking at the percentage of weekly commits that contain refactoring changes in Table 2, we find that there was an overall decrease in median percentage of refactoring commits of almost 6% in the JDT.CORE project after adopting shorter releases. Although the decrease is much smaller than the one found in the JDT.CORE project, both platform projects also show a decrease in the median percentage of refactoring commits.

Using a Mann-Whitney test and Cliff's delta, we find that our results remain consistent with our previous observation; only the JDT.CORE project has a statistically significant difference in weekly refactoring commits and a non-negligible effect size. The PLATFORM.SWT and PLATFORM.UI projects do not show any statistically significant difference in the number of refactoring commits between the traditional releases and the shorter releases.

Table 2: Weekly refactoring activities of each project during TR and SR

Project	Release	Refactoring instances			Refactoring commits			
		Median #	SS	ES	Median #	Median %	SS	ES
jdt.core	TR	10.5	0.002	d=0.28	3.0	25.00%	0.002	d=0.28
	SR	5.0	**p<0.01**		2.0	19.09%	**p<0.01**	
platform.swt	TR	2.0	0.523	d=-0.06	1.0	7.14%	0.867	d=-0.01
	SR	2.5	p>0.01		1.0	7.02%	p>0.01	
platform.ui	TR	5.0	0.4882	d=-0.06	2.0	16.67%	0.469	d=-0.06
	SR	6.0	p>0.01		2.0	14.29%	p>0.01	

*SS: statistical significance ES: effect size

From our investigation, we conclude that adopting shorter releases may have affected the amount of refactoring done and the refactoring activities of the JDT.CORE project. We found no clear positive or negative impacts of shortening the release cycle in the platform projects.

4.2. (RQ2) Are Refactoring Efforts Affected by Shorter Releases?

Motivation. In some cases, we find commits with multiple refactoring instances affecting only a few lines of code. In other cases, we find single commits with only one instance that required changing hundreds of lines of code. Because of these different cases, we cannot determine how much effort is put into refactoring using only the number of refactoring commits and instances.

Since past studies have found that refactoring is more often done manually than using refactoring tools [18, 19, 16], we need to take into account that some refactorings require significantly more time and code changes while others can be done very quickly and easily. In this RQ, we aim to find out how much refactoring efforts developers dedicate to refactoring by comparing the refactoring code churn of refactoring commits during both the traditional releases and the shorter releases.

Approach. Because code churn has been used several times as a proxy measure of effort or as part of a subset of multiple metrics used to measure effort in past studies [43, 44, 45], we decided to use the refactoring code churn as our metric to estimate how much effort is put by developers into refactoring during the yearly releases and the quarterly releases.

The first thing we did was to aggregate all files that had been refactored at least once. We then looked at the Git history to extract every code change

and code churn from this subset of files. For every refactoring instance in our dataset, we used the output of the RefactoringMiner to find the specific source code lines involved with refactoring.

In some cases such as refactorings that move source code folders, we found that the RefactoringMiner associated some code changes with a specific refactoring instance when the changes were actually caused by a different refactoring instance. This meant that some lines of code involved with refactoring were counted twice when summing up code churn for all the refactoring instances. To avoid counting the lines of code multiple times for the same code changes, we compared the code change history with the lines flagged by the RefactoringMiner to make sure each line was only counted once.

To calculate the refactoring code churn for a specific file, we therefore compared the code change history of the file with the source code lines that were flagged by the RefactoringMiner as being involved with refactoring. We discarded all code changes that were not directly caused by refactoring in order to only keep the history of refactoring code changes for that file. We repeated this process for every file in our subset to get our final dataset for RQ2.

After extracting every file's individual refactoring code churn, we aggregated our data by project in order to measure the change in refactoring effort on a project basis. We then calculated the total refactoring churn involved during the traditional releases and the shortened releases for each project. Similarly to our approach in RQ1, we then split our data on a weekly basis. We use the median of weekly refactoring code churn to get an overview of the difference in refactoring efforts over time for each project. We then look at the weekly refactoring code churn distribution to see exactly how refactoring efforts changed between the yearly releases and the quarterly releases for each project.

Although multiple studies [16, 18, 19] find that most refactoring is being done manually, some of these studies also find that a small subset of patterns associated with "Renaming" types of refactoring are more often applied using tools than manually. Due to the high likelihood of tool usage, we excluded "Renaming" types of refactoring from RQ2's analysis.

Findings 3. The median weekly refactoring churn of the JDT. CORE project shows a statistically significant difference between the traditional releases and the shorter releases. We proceed to look at the weekly data starting with the weekly median refactoring churn as shown in Table 3. Surprisingly, we find that the highest gap in median weekly refac-

Table 3: Refactoring Churn in LoC during TR and SR

Project	Release	Total Refactoring Churn	Weekly Refactoring Churn Median
jdt.core	TR	18,498	46.5
	SR	10,973	20.5
platform.swt	TR	35,131	11
	SR	4,993	5.5
platform.ui	TR	6,207	26
	SR	45,657	20.5

toring, with more than double the weekly refactoring churn during traditional releases, is from the JDT.CORE project which had the lowest difference in total refactoring churn. The PLATFORM.SWT shows a small decrease in weekly refactoring churn median and the PLATFORM.UI project shows no impact at all.

A Mann-Whitney test on the weekly refactoring code churn confirms our findings by revealing no statistically significant difference in the two platform projects. The JDT.CORE project, unlike the platform projects, was found to have a statistically significant difference in weekly refactoring churn.

> From our investigation, we conclude that adopting shorter releases may have affected the weekly amount of efforts dedicated to refactoring in the JDT.CORE project. While we also notice an overall downward trend of refactoring efforts during the shorter releases in the platform projects, the observed decrease in refactoring efforts was not as significant in the platform projects as it was in the JDT.CORE project.

4.3. (RQ3) Do Shorter Releases Impact What Type of Refactoring is Applied to the Code?

Motivation. Past studies have suggested that testing activities are affected by time constraints [39, 40, 41, 42]. Because developers must follow stricter time constraints during the quarterly releases, we investigate if the type of refactorings applied is also affected by the new time constraints. For example, it is possible that only small and simple refactorings such as renaming would be applied; leaving large refactorings that change the structure of the code on the side due to the lack of time. In this section, we look at the

type of refactorings being done during the traditional yearly releases and the quarterly releases to see if the shorter releases allow developers to still take care of all types of refactorings after shortening the release cycle.

In RQ3, we conduct two analyses (Section 4.3.1 and 4.3.2) to evaluate which refactoring types are the most common. We first start with a fine-grain analysis between yearly releases and quarterly releases based on the refactoring patterns individually. We then conduct a coarse-grain analysis by categorizing the refactoring patterns into distinct categories and analyzing the variation of these categories between the traditional releases and the shorter releases.

4.3.1. Refactoring pattern analysis (Fine-granularity)

Approach. Previous work [46] found that the number of commits is an accurate measure to estimate workload. Therefore, to get a general idea of which patterns were the most often applied in each project, we first examined how many commits for each refactoring pattern there are in all three projects during both time periods. To better visualize the variation in popularity for each refactoring pattern, we ranked every pattern during both time periods based on the number of commits that implemented them. This allows us to normalize the number of commits between the traditional releases and the shorter releases. Because we are not interested in the amount of refactoring done but rather the type of refactoring being done in RQ3, using the ranks to compare between both time periods allows us to abstract how much refactoring is being done and focus exclusively on which refactoring types are the most common during each release period. Lower ranks represent patterns that are frequently applied to the code and high ranks represent patterns that are not often applied to the code.

To see if the same patterns tend to have a high frequency of application during both time periods or if the new release model affected which refactoring patterns are being applied by the developers, we then selected the top 10 refactoring patterns with the most commits in traditional releases and the top 10 refactoring patterns with the most commits in the shorter releases respectively. To see which patterns were the most affected by the adoption of shorter releases, we also calculated the variation in ranking (ranking Δ) of each refactoring pattern from our top 10 list of patterns with the highest number of commits.

Pattern Ranking: Each refactoring pattern is assigned a rank for the yearly

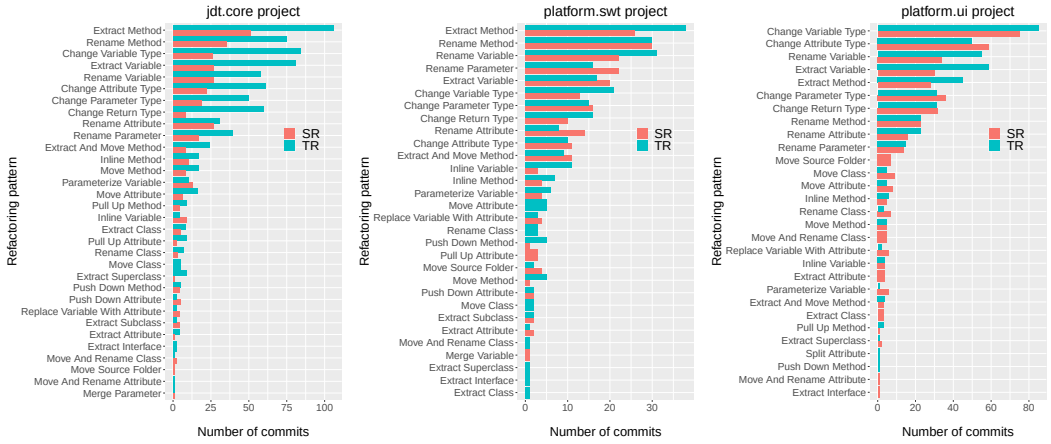

Figure 3: Number of refactoring commits per pattern during both release periods

releases period and the quarterly releases period for each project based on how many commits implement them. Common refactoring patterns are assigned low ranking values and uncommon patterns are assigned high ranking values.

Ranking Δ: We calculate the ranking Δ for each refactoring pattern by calculating the variation in ranking between yearly releases and quarterly releases. A positive Δ indicates that a pattern's rank increased and therefore became less common during the shorter releases and a negative delta indicates that a refactoring pattern's rank decreased and therefore became more common during the shorter releases.

Findings 4. We find that most refactoring patterns that were common during traditional releases were also common after shortening the release cycles. To get a general view of which patterns were popular during both time periods, we first extracted how many commits implement each refactoring pattern for each project during the traditional releases and the shorter releases. From Figure 3, we notice that the distribution of refactoring patterns that were popular in each project is similar during both time periods. Looking at the refactoring patterns with the most commits for each project, we also find that the three projects share a lot of similarities with patterns such as Extract Method being applied frequently during both the traditional releases and the shortened releases in all three projects.

From the ranking assignment process previously described, we calculate

the rank of each refactoring pattern and find that the most common refactoring patterns remain very similar even after shortening the releases. As shown in Table 4, 9 of the top 10 refactoring patterns are common in both the traditional releases and the shorter releases in the JDT.CORE project. From Table 5, we find that 8 out of 10 patterns are common in both time periods in the PLATFORM.SWT project and from Table 6, we find that all 10 patterns with the highest ranks during the traditional releases remain the same during the shorter releases in the PLATFORM.UI project. From these results, we find that most patterns that were popular during traditional releases remained popular after adopting shorter releases.

Looking at the ranking Δ of each pattern in Table 4, we notice that the *Change* refactoring patterns have significantly gone down in popularity during the shorter releases in the JDT.CORE project. We also observe that the *Rename* refactoring patterns have been oppositely affected and became much more common during the shorter release cycle. From Table 5, we find similar results as in the JDT.CORE project with *Change* refactoring patterns decreasing in popularity and *Rename* refactoring patterns increasing in popularity in the PLATFORM.SWT project. These results align with a previous study's findings [47] where they observed that Eclipse puts a heavy emphasis on backward compatibility. It is therefore not surprising that developers would be reluctant to perform refactorings that can affect the API such as changing method signatures.

While these results seem to confirm our initial assumption that simpler refactorings such as renaming are increasing in popularity due to the shorter releases, from Table 6 we find that the patterns in the PLATFORM.UI project were affected differently than in the JDT.CORE project: the *Extract* refactorings had a small decrease in popularity, the *Rename* refactorings saw no change at all and the *Change* refactorings saw a small increase in popularity. The PLATFORM.UI results therefore suggest that developers have started modifying their refactoring behaviors after adopting shorter releases by taking advantage of the more frequent major releases and applying refactorings that modify the API. Since we cannot draw a clear conclusion from our fine-grain analysis, we proceed with our coarse-grain analysis.

Table 4: Top 10 most common refactoring patterns in the JDT.CORE project during each release period

Pattern	Ranking in TR	Ranking in SR	Ranking Δ
Change Return Type (M)	6	15	+9
Change Variable Type (L)	2	6	+4
Change Attribute Type (M)	5	7	+2
Extract Method (M)	1	1	+0
Change Parameter Type (M)	8	8	+0
Rename Parameter (H)	9	9	+0
Extract Variable (L)	3	3	+0
Rename Method (H)	4	2	-2
Rename Variable (L)	7	4	-3
Rename Attribute (H)	10	5	-5
Parameterize Variable (H)	15	10	-5

(L): Low level refactoring, (M): Medium level refactoring, (H): High level refactoring

4.3.2. Categorization analysis (Coarse granularity)

Approach. To investigate the impacts of shorter releases on the different types of refactorings being applied rather than the impacts on each refactoring pattern's individual frequency, we proceeded to aggregate the refactoring patterns by categorizing them. We were then able to abstract our analysis and use our refactoring categories to continue our comparison between traditional releases and shorter releases.

We decided to use Murphy et al.'s categorization model [19] because it is less likely to be affected by subjectivity than estimating the complexity of a specific code change.

Following the categorization criteria given in the original paper [19], the patterns detected by the RefactoringMiner were categorized into three different categories: high-level refactorings which only affect the signatures of classes methods and field, medium-level refactorings which change the signature of classes, methods, and field while also significantly changing blocks of code and finally low-level refactorings which only affect blocks of code. In short, medium-level refactorings require to change the code in several places, unlike high-level and low-level refactorings that only target a specific section of the code and therefore are less likely to affect the structure of the codebase. Based on these characteristics, we hypothesize that the frequency of medium-level refactorings should be reduced during the shorter releases due to the combination of the higher complexity involved with changing the structure of the code and shorter release time.

Table 5: Top 10 most common refactoring patterns in the PLATFORM.SWT project during each release period

Pattern	Ranking in TR	Ranking in SR	Ranking Δ
Inline Variable (L)	9	17	+8
Change Variable Type (L)	4	8	+4
Change Return Type (M)	7	11	+4
Extract Method (M)	1	2	+1
Rename Variable (L)	2	3	+1
Extract Variable (L)	5	5	+0
Change Attribute Type (M)	10	10	+0
Rename Method (H)	3	1	-2
Rename Parameter (H)	6	4	-2
Change Parameter Type (M)	8	6	-2
Extract And Move Method (M)	11	9	-2
Rename Attribute (H)	12	7	-5

(L): Low level refactoring, (M): Medium level refactoring, (H): High level refactoring

Table 6: Top 10 most common refactoring patterns in the PLATFORM.UI project during each release period

Pattern	Ranking in TR	Ranking in SR	Ranking Δ
Extract Variable (L)	2	6	+4
Extract Method (M)	5	7	+2
Rename Variable (L)	3	4	+1
Rename Method (H)	8	8	+0
Rename Attribute (H)	9	9	+0
Rename Parameter (H)	10	10	+0
Change Variable Type (L)	1	1	+0
Change Attribute Type (M)	4	2	-2
Change Return Type (M)	7	5	-2
Change Parameter Type (M)	6	3	-3

(L): Low level refactoring, (M): Medium level refactoring, (H): High level refactoring

To categorize the refactoring patterns that were not previously categorized in Murphy et al.'s previous study [19], two authors separately classified each pattern detected by the RefactoringMiner. The two authors then combined the result of their individual classification to get the final classification for each pattern. For the cases where there were conflicts, the two authors discussed and agreed on the best categorization based on the criteria defined in Murphy et al.'s previous study [19]. After categorizing every refactoring pattern, we started to investigate if the type of refactorings applied changed between the traditional releases and the shorter releases.

We started by investigating which refactoring category (high-level, medium-level, low-level) was the most impacted by the change in release cycle. We

first looked at how many patterns in each refactoring category became more common after shortening the release cycles (ranking $\Delta < 0$), how many patterns became less common (ranking $\Delta > 0$) and how many patterns were not affected by the change (ranking $\Delta = 0$). We continued our investigation by looking at the overall variation in ranking of each refactoring category. To do so, we aggregated every refactoring pattern according to the refactoring category it belonged to. We then summed up the ranking Δ to see which type of refactoring (high-level, medium-level, low-level) showed the largest increase or decrease in popularity during both time periods.

Refactoring category Δ: To determine if a refactoring category became more or less common, we sum the ranking Δ for each pattern belonging to that category.

Findings 5. We find that medium-level refactoring patterns became less popular in the JDT.CORE project after shortening the release cycles.

From traditional yearly releases to quarterly releases, the JDT.CORE project has one less medium-level refactoring and one more high-level refactoring in its top 10 as shown in Table 4. From Table 5, we find that the PLATFORM.SWT project has one more low-level refactoring and one less high-level refactoring in its top 10 and the PLATFORM.UI project shows no difference between both periods as shown in Table 6. If we look at the top end of the rankings, however, we notice that in both the JDT.CORE project and the PLATFORM.SWT project, medium-level refactorings are less common than high-level and low-level refactorings during the shorter releases. This decrease in medium-level refactorings indicate that developers may not be as inclined to apply more complex refactorings during shorter releases because medium-level refactorings require to change the code in several places [19].

From the ranking Δ shown in Table 7, we find that in both the JDT.CORE project and the PLATFORM.SWT project, high-level refactorings got significantly more common after adopting shorter releases. The main difference between the two project, however, is that in the JDT.CORE project, medium-level refactorings became much less common whereas in the PLATFORM.SWT project, low-level refactorings became less common. Surprisingly, the PLATFORM.UI shows a different trend from the JDT.CORE project and the PLATFORM.SWT by having an increased amount of medium-level refactoring during the shorter releases. In fact, the PLATFORM.UI project is also the project with the smallest degree of variation in terms of rankings after adopting

Table 7: Ranking and workload variation between TR and SR

Refactoring type		jdt.core	platform.swt	platform.ui
Low-level	Sum of Δ	+1	+13	+5
Medium-level	Sum of Δ	+11	+1	-5
High-level	Sum of Δ	-12	-9	+0

shorter releases on top of being the only project that increased medium-level refactorings. From these results, it appears that only the JDT.CORE project satisfies our initial hypothesis that medium-level refactorings should see a significant decrease during the shorter releases. Our results also suggest that the PLATFORM.UI is the only one out of the three project that was able to keep up with its refactoring workload in all three categories of refactorings even after shortening the release cycle.

We therefore confirm our hypothesis and find that developers do not apply complex refactorings as much after shortening the releases in the JDT.CORE project. Although the PLATFORM.SWT project also shows a decrease in medium-level refactoring, the decrease is not significant enough to confirm our initial hypothesis. The PLATFORM.UI shows opposite results with medium-level refactoring becoming more common during the shorter releases.

Findings 6. We find that patterns that are commonly applied using refactoring tools have greatly increased in popularity during the shorter releases. If we take a look at 'Renaming' refactoring patterns which are commonly implemented using refactoring tools [16, 18], we also find that these types of pattern became way more common during the shorter releases in two of our three projects. Using the data from Tables 4, 5, and 6, we sum the Δ values of 'Renaming' refactoring patterns in each project and find a clear increase in popularity in the JDT.CORE project and the PLATFORM.SWT with a sum of Δ of -10 and -8 respectively. The PLATFORM.UI is the only project where we find a small decrease in popularity with a sum of Δ of +1. These results indicate that shorter releases may increase the use of refactoring tools by developers.

We find that developers do not apply complex refactorings as much during the shorter releases in the JDT.CORE project. We also find that patterns that are commonly implemented using refactoring tools became more common in two of the three analyzed projects after shortening the releases.

4.4. (RQ4) Does Shortening the Releases Affect the Human Resources Dedicated to Refactoring Activities and their Refactoring Workload?

Motivation. In this RQ, we aim to determine how human resources were affected after switching to shorter release cycles. A previous study about the impacts of shorter releases on testing found that switching to shorter releases led to a higher workload for testers due to fewer testers being involved with testing activities [2]. Since releases are shorter and development teams have to keep up with the development of new features, we want to see if the number of developers that refactor is affected when adopting shorter releases.

Approach. For the first part, we investigated if the number of developers that refactored was impacted by adopting shorter releases. We first extract the total number of developers, the number of developers that refactored during the traditional releases and the number of developers that refactored during the shorter releases. We then calculated the percentage of developers that refactored during each time period and compared the results between both periods. To see if shorter releases promote developer involvement with refactoring activities, we then extracted how many developers refactored during both periods and how many developers started refactoring only after shortening the release cycles.

For the second part, we investigated how the refactoring contribution of the developers was affected by adopting shorter releases. Using refactoring instances as our metric, we measured every developer's overall contribution to refactoring activities. We first extracted the median number of refactoring instances per developer. Then, to understand how the refactoring workload is shared among developers that refactor, we looked at the median percentage of refactoring instances contribution per developer during both release periods.

Findings 7. We find no clear impacts of adopting shorter releases on the number of developers that refactor.

Table 8: Developer resources during TR and SR

Project	Release	# Dev	# Refactoring Dev	% Refactoring Dev
jdt.core	TR	43	20	46.51%
	SR	35	14	40.00%
	$TR \wedge \overline{SR}$	-	8	-
	$\overline{TR} \wedge SR$	-	6	-
platform.swt	TR	55	29	52.73%
	SR	45	18	40.00%
	$TR \wedge \overline{SR}$	-	13	-
	$\overline{TR} \wedge SR$	-	5	-
platform.ui	TR	100	46	46.00%
	SR	105	55	52.38%
	$TR \wedge \overline{SR}$	-	16	-
	$\overline{TR} \wedge SR$	-	39	-

From our initial data extraction, as shown in Table 8, we find that two out of the three projects have a higher number of refactoring developers as well as a higher percentage of developers that refactor during traditional releases. We also notice that for the same two projects, namely the JDT.CORE and PLATFORM.SWT projects, most of the developers that are refactoring during the shorter releases were already refactoring during traditional releases. The PLATFORM.UI project is the only project of the three that differs. In fact, not only is there a higher percentage of developers that refactor during the shorter releases but most of the developers refactoring during that time period did not refactor during traditional releases.

Because the PLATFORM.UI project shows the exact opposite trend when compared with the JDT.CORE project and the PLATFORM.SWT project, we cannot conclude what impacts shortening the release cycles had on the number of developers that refactor. From our observations, a project can be positively affected by making more developers involved with refactoring as shown with the PLATFORM.UI project, or negatively impacted by having less developers refactor the code as shown with the JDT.CORE and PLATFORM.SWT projects.

Findings 8. The refactoring workload is more evenly distributed among developers that refactor during the shorter releases.

We investigate how the contribution of refactoring developers is affected by the change in release cycle. Using the median number of refactoring instances as our metric, we find that all three projects have less than half the number of refactoring instances per developer during the shorter releases as

Table 9: Refactoring workload contribution of developers during TR and SR

Project	Release	Median # refactoring instances	Median % refactoring instances
jdt.core	TR	28.0	0.82%
	SR	5.0	3.42%
platform.swt	TR	7.0	0.20%
	SR	3.5	3.04%
platform.ui	TR	5.5	0.29%
	SR	2.0	0.89%

shown in Table 9. In the case of the JDT.CORE project, we observe almost six times less refactoring instances per developer. These results indicate that there is a clear downward trend in the number of refactoring instances applied to the code per developer and that most of the refactoring workload may therefore be taken care of by 'refactoring experts'.

We therefore analyze how the total refactoring workload is split among developers. To measure the refactoring contribution of each developer that refactors, we measure what percentage of the total number of refactoring instances each developer has contributed. We then use the median to determine if the workload is more evenly shared among refactoring contributors or if most of them only take on a minor portion of the total refactoring workload.

We find that in both the JDT.CORE and PLATFORM.SWT projects, each developer take on a higher percentage of the total workload during the shorter releases than during the traditional releases. While the PLATFORM.SWT project also has a higher percentage of refactoring instances contribution per developer during the shorter releases, the increment is much smaller than the other two projects.

> We find no negative impacts on the amount of human resources dedicated to refactoring after shortening the release cycle. We also find that the refactoring workload is more evenly distributed among the developers during shorter releases.

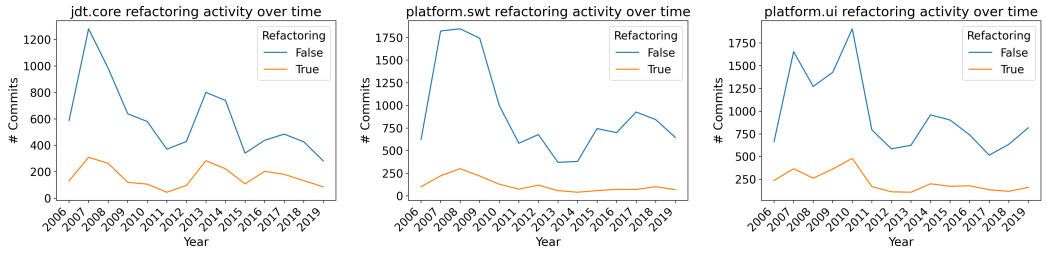

Figure 4: Evolution of refactoring activities over time

5. Discussion

5.1. Evolution of Refactoring Activities

Intuitively, it would make sense if refactoring activities diminish over time as a project ages and becomes more mature. If that was the case, then it would be less likely that the impacts observed in RQ1 and subsequent RQs would be consequences of shortening the release cycle. To verify this hypothesis, we plotted the evolution of refactoring activities since the first named version (3.2) in 2006. From Figure 4, we notice that the JDT.CORE seems to already have been on a downward trend of refactoring when the release cycle was shortened in 2018. The two platform projects, however, show a very stable amount of refactoring commits regardless of how many non-refactoring commits are being contributed to the projects.

Another concern we face is that the results and the statistical significance we observed in the JDT.CORE project in RQ1 may not be due to the event we are studying: the adoption of shorter release cycles. Since our results might be due to the way we separate our data, we decided to evaluate two separate chunks of data from the same time period (i.e., early TR and late TR, early SR and late SR). By comparing two subsets of data from the same time period, we can determine if the statistically significant difference in the number of commits that we found in RQ1 could be related to the change in release cycles or if this statistical significance is due to the seemingly decreasing refactoring activity previously observed.

We therefore used a Mann-Whitney test on the number of refactoring commits in the first half of the traditional releases period versus the second half of the traditional releases period and did the same for the shorter releases period for all 3 projects. Our results, as shown in Table 10, show that there is no statistically significant difference in the number of refactoring commits

Table 10: $p-$values for the number of refactoring commits during early versus late TR and early versus late SR

Project	early TR and late TR statistical significance	early SR and late SR statistical significance
jdt.core	0.853	0.565
platform.swt	0.236	0.407
platform.ui	0.0215	0.0193

during the traditional releases period and shorter releases period for both the JDT.CORE project and the PLATFORM.SWT project. This indicates to us that the the statistically significant difference in the number of refactoring commits found between the yearly releases period and the quarterly releases period for the JDT.CORE project may have been caused by the change in release cycles.

5.2. Refactoring and Software Quality

In RQ3, we observed that certain complex refactoring operations became less common after shortening the releases. Using a previous survey based study led by Silva et al. [16], we investigate the reasons why developers apply these patterns that became less popular. More specifically, we are interested to know how they think the quality of the code is improved by applying this type of refactoring patterns. In their survey, Silva et al. investigated the refactoring behavior of developers based on eight refactoring patterns. Two of these patterns; 'Extract Method' and 'Move Method', are classified as medium-level refactoring patterns per our classification in RQ3. We therefore use these two refactoring patterns as our basis to investigate how developers think they improve the quality of the code when they apply medium-level refactorings.

During the survey led by Silva et al., developers confessed that they usually extract methods to improve the reusability and the readability of the code. In the case of the 'Move Method' refactorings, the developers' answers point to a desire to increase cohesion and reusability. Because readability and reusability are not metrics that can easily be measured, we cannot empirically verify that the developers' refactorings achieved the intended purpose.

A past study led by Hindle et al. [48] however defined and studied the concept of software "naturalness". This code "naturalness" studies how "natural" a software's code feels similar to how a natural language can feel natural

or unnatural. A previous study led by Baishakhi and Hellendoorn [49] found that buggy code tends to be more unnatural and becomes more natural as bugs are fixed. We therefore conduct a small literature survey and find two studies led by Arima et al. [50] and Bin et al. [51] that studied the relationship between refactoring patterns and the naturalness of the code. Interestingly, both of these studies have found that the 'Extract Method' refactoring improve the naturalness of the code. The 'Move Method' refactoring, however, was found to improve the code naturalness in one study and decrease it in the other. We therefore hypothesize that the readability and reusability may indeed increase when these two refactoring patterns are applied.

To conclude, the code's quality may decrease in the long term if some refactoring patterns are ignored due to their time cost or complexity. Developers should therefore try not to modify their refactoring behaviors even if the release cycles become shorter. On the contrary, they should make sure all types of refactorings are applied to maintain the code's quality in the long term.

5.3. Summary and Implications

To get an overall view of the impacts of shorter releases, including our new findings on refactoring activities, we aggregated the results of related work studying the impacts of shortening release cycles on different development tasks as shown in Table 11. Although other studies use the Firefox project for their datasets, we notice that the impacts observed by previous studies are similar to our findings: although reducing the release cycle length has some impacts, these impacts are manageable. For example, shorter releases had an impact on how many platforms are tested in Firefox but each supported platform is tested more thoroughly [2]. From a previous study investigating short releases and bugs, one solution to keep up with the workload was to have more developers work on each release [1]. Another study investigating the impacts of short releases on the integration delay of fixed issues in the Firefox project [7] also found that developers adjusted to shorter releases by assigning an issue to a release rather than queuing issues in a backlog.

From this case study's results we were able to see how shortening the releases may affect a project's refactoring activities by looking at the JDT.CORE project's evolution. From RQ1 and RQ2, the decreased weekly number of refactoring instances, the decreased weekly percentage of commits that refactor the code and the lower weekly median number of lines refactored indicate that refactoring activities in the JDT.CORE project may have been negatively

Table 11: Impact of shorter releases on software development tasks

	Refactoring Our Paper	Integration [7]	Bugs [1]	Testing [2]
RQ1: Does adopting shorter releases affect the amount or the frequency of the task	Short releases may reduce how much refactoring is applied to the code	Issues are queued up as backlog in traditional releases versus a per-release basis in shorter releases.	Harder bugs are propagated to later releases. Fewer changes per developers but the changes touch more files.	Fewer platforms tested in total but each supported more thoroughly.
RQ2: Does adopting shorter releases affect the effort dedicated to the task	Shorter releases may impact refactoring efforts	Not studied	Not studied	Not studied
RQ3: Does adopting shorter releases affect the characteristics of the task	In some cases, refactorings that affect many parts of the code are not applied as frequently during shorter releases	Issues addressed more quickly but take more time to be integrated. The total time from the issue report date to its integration is not significantly different.	Negligible difference in the number of post-release bugs. Bugs are fixed more quickly but fewer bugs are being fixed.	More tests executions per day but the tests focus on a smaller subset of the corpus. Overall fewer tests but larger builds.
RQ4: Does the change to shorter releases affect the human resources dedicated to the task	The refactoring workload is split more evenly among developers	Not studied	More developers working on each short release.	Fewer testers but higher workload.

affected by the shorter releases. From RQ3, we also found that specific types of refactorings are becoming less common after changing the release cycle in the JDT.CORE project. Past studies have shown that refactoring improves software quality and maintainability [3, 18] which leads to fewer bugs in the long term. In Section 5.2, we also found that the refactoring patterns that were less common during the shorter releases were associated with readability and reusability; two aspects of software quality that can deteriorate in the long term if not addressed. Short releases adopters should therefore take into account the risks of shortening their release cycles on refactoring activities

and assign their resources accordingly.

By looking at the PLATFORM.UI project, however, our results also show how a project could benefit from the shorter releases while also keeping up with its refactoring workload. From RQ1 and RQ2, we found that the shorter releases did not impact the amount of refactoring being done in the PLATFORM.UI project. From RQ3, we even find that medium-level refactorings increased after shortening the releases. These results indicate that the developers are adapting their refactoring behaviors with the new release cycles. Since every release is a major release with the new release model, the developers now have the flexibility of refactoring their API during any release and our results show that they are actively taking advantage of that. From RQ4, we also find that a higher percentage of developers refactor during the shorter releases in the PLATFORM.UI project. From this case study, short releases adopters can therefore also see how their project could benefit from shortening their release cycles.

While it is not yet clear as to why some projects adapt better to shorter releases than others, we hypothesize that one of the possible reason as to why the JDT.CORE is having a harder time keeping up with its refactoring workload might be due to the nature of the code itself. Since the PLATFORM.UI project is a UI project, the chances of having to refactor the API is most likely lower than in the JDT.CORE project which is made of core application code. As we have seen previously, the Eclipse community also tends to be more reluctant to refactor their API in order to maintain a strong backward compatibility [47]. The same study also finds that the developers are aware of how many downstream users can get affected by their changes and are therefore sensitive about making breaking changes. Since the JDT.CORE project makes up the core of the Eclipse application and can affect all other aspects of the platform, we therefore assume that the developers might be less inclined to refactor their API as much as in other projects.

5.4. Threats to Validity

Internal threats: In RQ1 and RQ2, we aggregate our data on a weekly basis and use the weekly median values for our analysis. Aggregating the data on a weekly basis allows us to have enough data points to follow the evolution of refactoring activities while also keeping the time frame short enough to avoid having outliers significantly affect our results.

In RQ2, we removed all 'Rename' refactoring types because more than 50% of developers admit using refactoring tools for renaming in previous

studies investigating the usage of refactoring tools [16]. Still, several refactoring types (e.g., 'Extract Local') can be performed by tools, which might affect the results of RQ2. As of today, there is still no tool that can allow us to detect if a refactoring was performed manually or automatically. Filtering out patterns that are commonly implemented using refactoring tools based on previous studies' findings is therefore the best approach we currently have to minimize refactoring tools' impacts on our results.

In RQ3, we categorize each refactoring pattern based on a previous study authored by Murphy et al. [19]. We know that this categorization is not optimal because it does not allow us to accurately categorize refactoring patterns based on the complexity of the task or the efforts required to apply each refactoring pattern. However, it does allow us to get a hint of both complexity and efforts by taking into account which sections of the code need to be changed to apply each refactoring pattern.

External threats: Because we only used open-source software projects from the Eclipse foundation, the results shown in this paper may not be representative of other software systems. We also found that the Eclipse foundation gradually adopted shorter releases over a long period of time unlike other known projects using short releases such as Firefox. The impacts of shorter releases on the projects investigated in this study may therefore differ from other projects that chose to adopt short releases more abruptly. Because we observed different impacts of shorter release cycles on all three of our projects, it is difficult to generalize our results.

Constructive threats: Code churn has been used by multiple past studies as a measure of effort [52, 53, 54, 55]. However, it has also been suggested that code churn is not a direct measure of effort but rather one of many metrics used to measure development effort [56]. We therefore only get a general idea of the actual efforts put into refactoring by using refactoring churn. Although we used a state of the art refactoring detection tool that was used several times in previous studies [57, 58, 59, 60], we have no guarantee that all refactorings applied to the code were detected.

6. Conclusion

In this study, we investigated the impacts of shortening the release cycles on refactoring activities in three Eclipse projects which all developed using both traditional yearly releases and shorter quarterly cycles. Through this case study, we found that shortening the release cycles may impact the

refactoring activities of a software project in various ways. We also found that some of the refactoring patterns that were less common after shortening the releases are usually applied to improve the readability and reusability of the code. Not applying them could therefore affect the software's quality in the long term. This case study further contributes to the empirical knowledge related to shortening the release cycles. With the results we present in this paper, potential short releases adopters now have a better insight of the potential impacts on refactoring activities that can occur from shortening their release cycle. When aggregated with previous studies that study the impacts of shortening the release cycles, our results allow short releases adopters to have an overall picture of which area of development can be impacted when shortening the release cycles. It also allows them to dedicate resources accordingly to make the transition as smooth as possible.

Intuitively, it would make sense to observe less refactoring, more bugs, or other issues when shortening a project's development cycle due to the short amount of time available to developers. One interesting factor that however seems common across studies investigating the impacts of shorter release cycles is that developers can adapt to the shorter releases and manage software development tasks even with less time between each release. Although the impacts of shorter seem manageable, it would be interesting to study in the future what problems early adopters such as Google and Firefox have encountered when adopting shorter releases and how these issues were solved. In the case of our study, for example, while we cannot prove with certainty that developers are using more refactoring tools to keep up the with refactoring workload, our results do seem to indicate that more tools may have been used to mitigate the impacts of the shorter releases.

Acknowledgment

We gratefully acknowledge the financial support of JSPS and SNSF for the project "SENSOR" (JPJSJRP20191502), and JSPS for the KAKENHI grants (JP21H04877, JP21K17725).

References

[1] F. Khomh, B. Adams, T. Dhaliwal, Y. Zou, Understanding the impact of rapid releases on software quality, Empirical Software Engineering 20 (2) [2015] 336–373.

[2] M. V. Mäntylä, B. Adams, F. Khomh, E. Engström, K. Petersen, On rapid releases and software testing: A case study and a semi-systematic literature review, Empirical Software Engineering 20 (5) [2015] 1384–1425.

[3] F. Palomba, A. Zaidman, R. Oliveto, A. De Lucia, An exploratory study on the relationship between changes and refactoring, in: Proceedings of the 25th International Conference on Program Comprehension, 2017, p. 176–185.

[4] M. Kim, D. Cai, S. Kim, An empirical investigation into the role of api-level refactorings during software evolution, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 151–160.

[5] K. Stroggylos, D. Spinellis, Refactoring–does it improve software quality?, in: Proceedings of the 5th International Workshop on Software Quality, 2007, p. 10.

[6] A. Silva, G. Carneiro, F. Brito e Abreu, M. Monteiro, Frequent releases in open source software: A systematic review, Information 8 (3) [2017] 109.

[7] D. A. da Costa, S. McIntosh, C. Treude, U. Kulesza, A. E. Hassan, The Impact of Rapid Release Cycles on the Integration Delay of Fixed Issues, Empirical Software Engineering 23 (2) [2018] 835–904.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: improving the design of existing code, Addison-Wesley, 1999.

[9] M. Kim, M. Gee, A. Loh, 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, p. 371–372.

[10] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 483–494.

[11] D. Silva, M. T. Valente, Refdiff: Detecting refactorings in version histories, in: Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories, 2017, pp. 269–279.

[12] D. Dig, C. Comertoglu, D. Marinov, R. Johnson, Automated detection of refactorings in evolving components, in: Proceedings of the 20th European Conference on Object-Oriented Programming, 2006, p. 404–428.

[13] Z. Xing, E. Stroulia, Umldiff: An algorithm for object-oriented design differencing, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 54–65.

[14] C. Vassallo, G. Grano, F. Palomba, H. Gall, A. Bacchelli, A large-scale empirical exploration on refactoring activities in open source software projects, Science of Computer Programming 180 [2019] 1–15.

[15] A. Peruma, M. W. Mkaouer, M. J. Decker, C. D. Newman, An empirical investigation of how and why developers rename identifiers, in: Proceedings of the 2nd International Workshop on Refactoring, 2018, p. 26–33.

[16] D. Silva, N. Tsantalis, 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, p. 858–870.

[17] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, O. Strollo, When does a refactoring induce bugs? an empirical study, in: Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, 2012, p. 104–113.

[18] M. Kim, T. Zimmermann, 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.

[19] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, in: Proceedings of the 31st International Conference on Software Engineering, 2009, p. 287–297.

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

[21] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, G. Succi, A case study on the impact of refactoring on quality and productivity in an agile team, 2007.

[22] R. Moser, A. Sillitti, P. Abrahamsson, G. Succi, Does refactoring improve reusability?, 2006, pp. 287–297.

[23] D. Dig, R. Johnson, The role of refactorings in api evolution, 2005, pp. 389–398.

[24] S. Herbold, J. Grabowski, H. Neukirchen, Automated refactoring suggestions using the results of code analysis tools, 2009, pp. 104 – 109.

[25] X. Ge, E. Murphy-Hill, Manual refactoring changes with automated refactoring validation, in: Proceedings of the 36th International Conference on Software Engineering, 2014, p. 1095–1105.

[26] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, Z. Wan, Large-scale automated refactoring using clangmr, in: 2013 IEEE International Conference on Software Maintenance, 2013, pp. 548–551.

[27] Z. Xing, E. Stroulia, Refactoring practice: How it is and how it should be supported - an eclipse case study, in: Proceedings of the 22nd IEEE International Conference on Software Maintenance, 2006, pp. 458–468.

[28] B. Daniel, D. Dig, K. Garcia, D. Marinov, Automated testing of refactoring engines, 2007, pp. 185–194.

[29] M. Michlmayr, B. Fitzgerald, K.-J. Stol, Why and how should open source projects adopt time-based releases?, IEEE Software 32 (2) [2015] 55–63.

[30] F. Khomh, T. Dhaliwal, Y. Zou, B. Adams, Do faster releases improve software quality? An empirical case study of Mozilla Firefox, in: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, 2012, pp. 179–188.

[31] O. Baysal, I. Davis, M. W. Godfrey, A Tale of Two Browsers, in: Proceedings of the 8th Working Conference on Mining Software Repositories, 2011, p. 238–241.

[32] V. M. Mika, F. Khomh, B. Adams, E. Engstr, K. Petersen, On Rapid Releases and Software Testing, in: Proceedings of the 2013 IEEE International Conference on Software Maintenance, 2013, pp. 20–29.

[33] E. Kula, A. Rastogi, H. Huijgens, A. Van Deursen, G. Gousios, Releasing fast and slow: An exploratory case study at ING, in: Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 785–795.

[34] M. Claes, M. Mantyla, M. Kuutila, B. Adams, Abnormal Working Hours: Effect of Rapid Releases and Implications to Work Content, in: Proceedings of the 14th International Conference on Mining Software Repositories, 2017, pp. 243–247.

[35] N. Kerzazi, F. Khomh, Factors impacting rapid releases: An industrial case study, in: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2014, pp. 98–107.

[36] G. N. Tonietto, S. A. Malkoc, S. M. Nowlis, When an Hour Feels Shorter: Future Boundary Tasks Alter Consumption by Contracting Time, Journal of Consumer Research 45 (5) [2018] 1085–1102.

[37] A. Garrido, R. Johnson, Challenges of refactoring c programs, 2002, p. 6–14.

[38] C. Vassallo, F. Palomba, H. C. Gall, Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers, in: International Conference on Software Maintenance and Evolution, 2018, pp. 564–568.

[39] H. Do, S. Mirarab, L. Tahvildari, G. Rothermel, The effects of time constraints on test case prioritization: A series of controlled experiments, IEEE Transactions on Software Engineering 36 (5) [2010] 593–617.

[40] H. Do, G. Rothermel, An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models, in: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2006, p. 141–151.

[41] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, R. S. Roos, Timeaware test suite prioritization, in: Proceedings of the 2006 International Symposium on Software Testing and Analysis, 2006, p. 1–12.

[42] Jung-Min Kim, A. Porter, A history-based test prioritization technique for regression testing in resource constrained environments, in: Proceedings of the 24th International Conference on Software Engineering, 2002, pp. 119–129.

[43] S. Karus, M. Dumas, Predicting coding effort in projects containing xml, in: Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, 2012, p. 203–212.

[44] T. Mende, R. Koschke, Revisiting the evaluation of defect prediction models, in: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, 2009, pp. 1–10.

[45] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, A. E. Hassan, Revisiting common bug prediction findings using effort-aware models, in: Proceedings of the 2010 IEEE International Conference on Software Maintenance, 2010, p. 1–10.

[46] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Transactions on Software Engineering 43 (11) [2017] 1063–1088.

[47] C. Bogart, C. Kästner, J. Herbsleb, F. Thung, How to break an api: Cost negotiation and community values in three software ecosystems, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, p. 109–120.

[48] A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in: Proceedings of the 34th International Conference on Software Engineering, 2012, p. 837–847.

[49] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, P. Devanbu, On the "naturalness" of buggy code, in: Proceedings of the 38th International Conference on Software Engineering, 2016, p. 428–439.

[50] R. Arima, Y. Higo, S. Kusumoto, Toward refactoring evaluation with code naturalness, in: Proceedings of the 26th Conference on Program Comprehension, 2018, p. 316–319.

[51] B. Lin, C. Nagy, G. Bavota, M. Lanza, On the impact of refactoring operations on code naturalness, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, 2019, pp. 594–598.

[52] E. Arisholm, L. Briand, E. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, Journal of Systems and Software 83 (1) [2010] 2–17.

[53] S. Karus, M. Dumas, Predicting coding effort in projects containing xml, in: Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, 2012, p. 203–212.

[54] T. Mende, R. Koschke, Revisiting the evaluation of defect prediction models, in: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, 2009, pp. 1–10.

[55] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, A. Bener, Defect prediction from static code features: Current results, limitations, new approaches, Automated Software Engineering [2010] 375–407.

[56] E. Shihab, Y. Kamei, B. Adams, A. E. Hassan, Is lines of code a good measure of effort in effort-aware models?, Information and Software Technology [2013] 1981–1993.

[57] P. Thongtanunam, W. Shang, A. E. Hassan, Will this clone be short-lived? towards a better understanding of the characteristics of short-lived clones, Empirical Software Engineering 24 (2] [2019] 937–972.

[58] L. Sousa, D. Cedrim, W. Oizumi, A. Bibiano, A. Oliveira, A. Garcia, D. Oliveira, M. Kim, Characterizing and identifying composite refactorings: Concepts, heuristics and patterns, 2020.

[59] M. Iammarino, F. Zampetti, L. Aversano, M. Di Penta, Self-admitted technical debt removal and refactoring actions: Co-occurrence or more?, in: 2019 IEEE International Conference on Software Maintenance and Evolution, 2019, pp. 186–190.

[60] E. C. Neto, D. A. d. Costa, U. Kulesza, Revisiting and improving szz implementations, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2019, pp. 1–12.