A Multidimensional Empirical Study on Refactoring Activity

Nikolaos Tsantalis[†], Victor Guana[*], Eleni Stroulia[*], Abram Hindle[*]
[†]Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
[*]Department of Computing Science
University of Alberta, Edmonton, Alberta, Canada

Abstract

In this paper we present an empirical study on the refactoring activity in three well-known projects. We have studied five research questions that explore the different types of refactorings applied to different types of sources, the individual contribution of team members on refactoring activities, the alignment of refactoring activity with release dates and testing periods, and the motivation behind the applied refactorings. The studied projects have a history of 12, 7, and 6 years, respectively. We have found that there is very little variation in the types of refactorings applied on test code, since the majority of the refactorings focus on the reorganization and renaming of classes. Additionally, we have identified that the refactoring decision making and application is often performed by individual refactoring "managers". We have found a strong alignment between refactoring activity and release dates. Moreover, we found that the development teams apply a considerable amount of refactorings during testing periods. Finally, we have also found that in addition to code smell resolution the main drivers for applying refactorings are the introduction of extension points, and the resolution of backward compatibility issues.

1 Introduction

In the past, refactoring activity has been empirically investigated with respect to its frequency [15], the adoption of refactoring tools [10], its relation with bug fixing [7] and testing [11], and its perception by developers [9]. There are still some interesting questions that have not been explored yet. For instance, what are the principal drivers behind the application of refactorings, and how do different team roles and code artifacts affect refactoring practice.

In this paper, we inspected the refactoring history of three well known projects, namely JUnit, HTTPCore, and HTTPClient, and investigated 5 research questions related to important aspects of refactoring practice. These questions are:

RQ1: Do software developers perform different types of refactoring operations on test code and production code?

RQ2: Which developers are responsible for refactorings?

RQ3: Is there more refactoring activity before major project releases than after?

RQ4: Is refactoring activity on production code preceded by the addition or modification of test code?

RQ5: What is the purpose of the applied refactorings?

It is our strong belief that answers to these research questions will help the community better reflect on the actual refactoring practice.

Furthermore, they will increase the awareness of the software maintenance community on the drivers behind the application of refactorings and help motivate the creation of refactoring tools (e.g., code smell detectors, IDEs) addressing the actual needs of developers in a principled and empirical manner. This study primarily focuses on high and medium level refactorings, as defined by Murphy-Hill et al. [10]. High level refactorings are those that change the signatures of classes, methods, and fields, while medium level refactorings change also blocks of code (e.g., Extract Method) in addition to the aforementioned changes [10]. To the best of our knowledge, this is the first empirical study that investigates the purpose of the applied refactorings with respect to the design problems being faced or the design decisions being made. Additionally, this work makes a distinction between the refactoring operations applied on production and test code.

The rest of the paper is organized as follows. Section 2 presents the related work regarding refactoring activity analysis and refactoring detection; Section 3 shows our experimental setup, selection of case studies, issues on decentralized repository analysis, and our refactoring detection methodology; Section 4 presents our experimental results along with a discussion around the research questions; Section 5 discusses the threats to the validity of the study; Section 6 summarizes the paper and discusses its main conclusions.

2 Related Work

One among the earlier empirical studies on refactoring practice was that of Xing and Stroulia [15] who examined the structural changes in the evolution of the Eclipse JDT, using the UMLDiff [14] algorithm, in order to investigate (a) what proportion of these changes are due to refactorings, (b) which are the typical refactorings applied in practice, and (c) which types of refactorings are "safe" to client applications that reuse the refactored system. They concluded that about 70% of structural changes may be due to refactorings and for about 60% of these changes, the references to the affected entities in a component-based application can be automatically updated by a refactoring-migration tool.

Murphy-Hill et al. [10] investigated the refactoring practices followed by the developers using the version history of the Eclipse code base as extracted from its CVS repository. They concluded that (a) commit comments are not useful for predicting refactoring activity, (b) refactoring is frequently used as a means to reach a specific end, such as adding a feature or fixing a bug (floss refactoring), and (c) the percentage of low-level and medium-level refactorings (i.e., sub-method level refactorings) is significant (40-60% of the total number of refactorings) compared to high-level refactorings (i.e., refactorings changing the signature of classes, methods and fields).

Kim et al. [7] investigated the role of API-level refactorings (i.e., package/-class/method moves/renames, and method signature changes) using the fine-grained evolution history (i.e., revisions) of three open source projects, namely Eclipse JDT Core, JEdit and Columba. They concluded that (a) there is an increase in the number of bug fixes after API-level refactorings, (b) the time taken to fix bugs is shorter after API-level refactorings than before, (c) a large number of refactoring revisions include bug fixes at the same time or are related to later bug fix revisions implying that refactorings introduce bugs or refactorings are applied to facilitate bug fixes, and (d) API-level refactorings occur more frequently before than after major software releases.

Kim et al. [9] surveyed developers at Microsoft regarding their use and understanding of refactoring. They found that many developers did not view refactoring solely as "semantic-preserving code transformations" but more as efforts related to code quality improvement. Some developers were motivated to refactor in order to decrease dependencies to aid code reuse. They repeated some of their prior analysis on Microsoft products [7].

Rachatasumrit and Kim [11] studied the impact of refactoring operations on regression tests. The main concern of the study was to identify whether code that has been refactored preserves its original behavior or, in some cases, introduces faulty enhances that need to be retested. Among others, the study found that

only 22% of refactorings are tested, and that at least half of the failed test affected after code modifications, included exercising code that involved refactoring edits. From our particular interest, this empirical study reports the ratio of failure-inducing changes out of all the refactoring edits in three open source projects.

With respect to refactoring detection tools several approaches have been proposed in the literature. Dig et al. [3] combined a fast syntactic analysis based on Shingles encoding to detect refactoring candidates and a more expensive semantic analysis to refine the results. Weissgerber and Diehl [13] developed a signature-based analysis to detect refactoring candidates and clone detection to rank these candidates. Kim et al. [8] proposed a rule-based program differencing approach that automatically discovers and summarizes systematic code changes as logic rules.

3 Study Design

In this section we discuss the experimental design of our empirical study. First, we discuss our rationale for selecting the particular software systems for our study. Second, we expose the technical problems we faced in our attempt to compare subsequent revisions in decentralized repositories, and the solutions we developed for these problems. Lastly, we explain our conceptual and practical framework for detecting refactorings within successive revisions of code.

3.1 Selecting Repositories

For our study we selected three medium-size projects: the JUnit testing framework, the Jakarta HTTPCore, and the Jakarta HTTPClient. These systems have a development history ranging from 6 to 12 years. We chose JUnit because it is a well studied system and the two Jakarta HTTP components because they are managed by the same team of people and the comparison of their histories would enable us to investigate if the types of applied refactorings depend on the development team, or whether they are dictated by the specific project requirements. All three selected systems were developed in Java and provide a reliable build automation system, since our refactoring detection mechanism requires to analyze compiled Java sources.

3.1.1 JUnit

JUnit is perhaps the most widely known and used testing framework for Java programs. Its development started by Erich Gamma and Kent Beck in 2000. Its development history spans over 12 years. Particularly, between releases 3.8.1 (in September 2002) and 4.0 (in February 2006) the project went through a relatively inactive period of activity. We analyzed 1018 revisions ranging from JUnit 3.5 to 4.6.

3.1.2 Jakarta HTTPComponents

HTTPCore is a set of Java components used to implement client and server side HTTP-based services. It has been mirrored in a publicly available *git* repository exposing 7 years of history (February 2005 to March 2012). We have analyzed 1588 revisions produced in this period of time. **HTTPClient** provides client side applications with authentication, connection state representation, and general connection management capabilities. It has been also mirrored in *git* repository covering its development history from December of 2005 to March 2012. We have analyzed a total 1838 revisions registered during its 6 years of development.

3.2 Determining Successive Revisions

In order to identify the refactorings applied throughout the evolution of a software system, it is necessary to extract the changes that occurred between successive revisions. However, in contrast to centralized repositories (such as SVN) which have a linear commit history, the history of decentralized repositories is a directed acyclic graph (DAG) due to the presence of *implicit branches*. In *git*, as explained by Bird et al. [2], when two collaborating developers commit changes to their local repositories, their repositories diverge (i.e., they contain different commits). When one of the developers pulls from the other, *git* creates a new commit that merges the remote sequence of commits into the pulling developer's repository.

As a result, the definition of successive revisions cannot be based on the commit time, as is possible in the case of centralized repositories. In our approach, we compare each examined revision with its parent revision in the commit history DAG. In order to avoid the duplicate report of refactorings we excluded from our analysis the merge commits. A merge commit has at least two parents and thus the comparison of the merged revision with all parent revisions from the merged branches would result in duplicate refactoring reports in the case where refactoring activity has taken place in at least one of the branches. In the example depicted in Figure 1, c_5 is a merge commit in which two branches are merged. If a refactoring took place in commit c_4, then comparing the revisions corresponding to commits c_2 and c_4 is sufficient to detect it. The comparison of the revisions corresponding to c_3 (the last commit of the branch where no refactoring activity took place) and c_5 (the merge commit) would erroneously report that the same refactoring was applied twice.

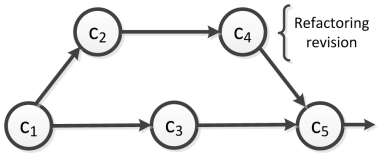


Figure 1: An example of implicit branching.

3.3 Detecting Refactorings

To detect the refactorings applied between two successive revisions, we have adopted a lightweight version of the UMLDiff [14] algorithm for differencing object-oriented models. First, our process matches the model elements in a top-down order (starting from the classes and going to the methods and fields) based on their names and the similarity of their signatures (in the case of methods). This step performs exact matching and returns a set of matched elements, a set of elements that were removed from the first model, and a set of elements that were added to the second model. Next, the removed/added elements between the two models are matched based only on the equality of their names in order to find changes in the signatures of fields and methods (in the case of multiple removed/added methods having the same name, an equal number of parameters is also required). This process returns a set of changed elements which are deleted from the corresponding sets of removed/added elements. Third, the removed/added classes are matched based on the similarity of their members at signature level. This process tolerates changes of types in the signatures of the class members and returns a set of moved/renamed classes which are deleted from the corresponding sets of removed/added classes. This process is more lightweight than the original UMLDiff algorithm in the sense that it does not take into account the dependencies between the model elements to compute their similarity.

We adopted and extended the refactoring-detection rules defined by Biegel et al. [1]. For each revision $v \in V$ we extract the following sets:

- C_v: The set of system classes/interfaces. This set contains tuples (p, n, m), where p is the name of the package to which the class belongs, n is the class name and m is the set of class members (methods and fields) belonging to the class.

- M_v: The set of system methods. This set contains tuples (c, m, p, r, b), where c is the fully qualified name of the class to which the method belongs, m is the method name, p is the parameter list of the method, r is the return type of the method and b is the set of class members (methods and fields) being accessed within the body of the method ($b = \{x\text{: } (x \in M_v)$ or $(x \in F_v)\}$).

- F_v: The set of system fields. This set contains tuples (c, f, t), where c is the fully qualified name of the class to which the field belongs, f is the field name and t is the field type.

- G_v: The set of generalization relationships in the system. This set contains tuples (c_i, c_j), where class c_i extends class c_j and c_i, c_j are fully qualified names.

- R_v: The set of realization relationships in the system. This set contains tuples (c_i,

c_j), where class c_i implements interface c_j and c_i, c_j are fully qualified names.

Next, for each transaction t, which corresponds to the comparison of two successive revisions v_1, $v_2 \in V$, we determine the entities that were added, removed and remained. For example, C_t^+ is the set of classes that were added in transaction t, C_t^- is the set of classes that were removed in transaction t and $C_t^=$ is the set of classes that remained in transaction t. The same notation (i.e., "+" for added, "-" for removed, and "=" for remained) is used for the other sets of entities as well. Using this conceptual framework, Table 1 presents a set of rules used to identify 11 different types of refactorings in successive code revisions.

Table 1: Refactoring detection rules

Refactoring	**Rule**
Extract Method	$\exists\ (c, m_i, p_i, r_i, b'_i) \in M_t^= \wedge$ $\exists\ (c, m_j, p_j, r_j, b_j) \in M_t^+ \wedge$ $b_j \subseteq b_i \wedge$ $(c, m_j, p_j, r_j, b_j) \in b'_i$ b'_i: body of m_i **after** refactoring b_i: body of m_i **before** refactoring
Move* Field	$\exists\ (c, f, t) \in F_t^- \wedge$ $\exists\ (c', f, t) \in F_t^+$
Move* Method	$\exists\ (c, m, p, r, b) \in M_t^- \wedge$ $\exists\ (c', m, p', r, b) \in M_t^+ \wedge$ $p \subseteq p'$
Extract Superclass	$\exists\ (p, n, m) \in C_t^+ \wedge$ $\exists\ (p_x, n_x, m'_x) \in C_t^= \wedge$ $\exists\ (p_x.n_x, p.n) \in G_t^+ \wedge$ $m \subseteq m_x \wedge m'_x \subseteq m_x$ m'_x: members of n_x **after** refact. m_x: members of n_x **before** refact.
Extract Interface	$\exists\ (p, n, m) \in C_t^+ \wedge$ $\exists\ (p_x, n_x, m'_x) \in C_t^= \wedge$ $\exists\ (p_x.n_x, p.n) \in R_t^+ \wedge$ $m \subseteq m_x \wedge m'_x = m_x$
Move Class	$\exists\ (p, n, m) \in C_t^- \wedge$ $\exists\ (p', n, m) \in C_t^+$
Rename Class	$\exists\ (p, n, m) \in C_t^- \wedge$ $\exists\ (p, n', m) \in C_t^+$ **Note**: constructor names and field/parameter types may be renamed accordingly
* $\exists\ (c, c') \in G_t^=$, then Move ⇒ Pull Up $\exists\ (c', c) \in G_t^=$, then Move ⇒ Push Down	

4 Results

This section discusses the findings of our exploration of the research questions we identified in the introduction and the details of the methodology used to investigate each question based on our revision extraction mechanism, and refactoring detection tool. More importantly, we discuss the observations from our empirical study, and address possible explanations of the examined behaviors.

4.1 RQ1: Do software developers perform different types of refactoring operations on test code and production code?

The first step towards answering this question is to make a distinction between classes corresponding to *test code* and those corresponding to *production code*. In JUnit, test code is placed within package `junit.tests`, since the beginning of the project. Thus refactorings on classes/methods/fields with qualified names matching `junit.tests` were classified as *test code* refactorings. Out of the total 383 detected refactorings, 219 (57%) were *production code* refactorings, while 164 (43%) were *test code* refactorings.

For the HTTPCore and HTTPClient projects we followed a different approach, since the test classes are not placed within a specific package. We noticed that the names of test classes had a prefix of `Test` or a suffix of `Mock`. Out of the total 527 detected refactorings in HTTPCore, 421 (80%) were applied to *production code*, while 106 (20%) were applied to *test code*. In HTTPClient, 161 (81%) where applied to *production code* and 37 (19%) to *test code* out of 198 total refactorings. We manually inspected the detected refactorings in all examined projects.

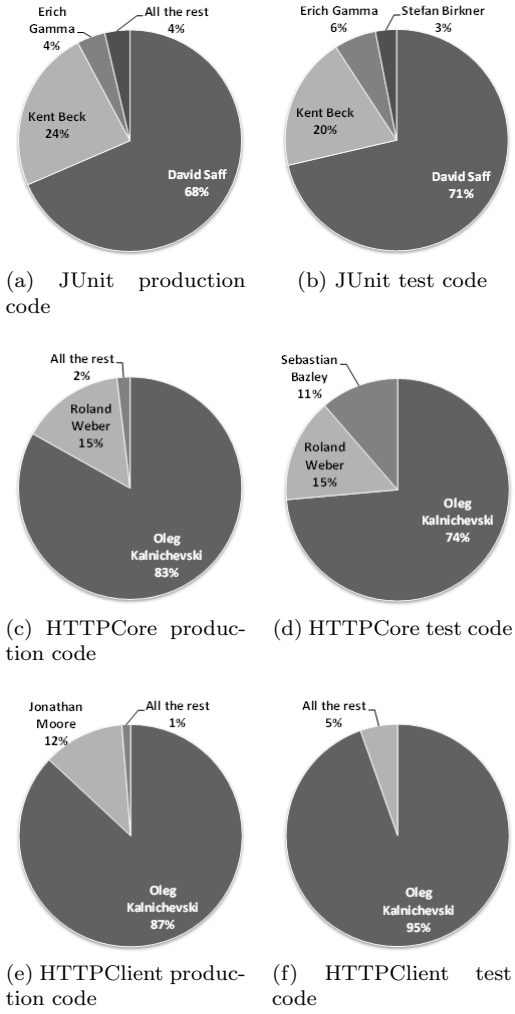
We found 8 False Positives for the Extract Method refactoring (96.4% precision) and 4 False Positives for the Rename Class refactoring (97.6% precision), while the precision for all the other types of refactorings was 100%.

Table 2 reports the number of detected refactorings on *production code* and *test code*, for each examined project. The variety of the refactorings applied to *production code* and *test code* differed: in *test code* of the examined projects we found only 3 types of refactorings, namely Move Class, Rename Class and Move Method. The goal of the applied refactorings on *test code* was often to organize the tests into new packages, reorganize inner classes, rename some test classes by giving a more meaningful name and move methods between test classes for conceptual reasons. Conversely, the refactorings that were applied to *production code*, of the examined systems, were intended to improve the design of the system by modularizing the code and removing design problems. The three most dominant refactorings on *production code* were Move Class, Extract Method and Rename Class followed by refactorings related to the reallocation of system's behavior (Move and Pull Up Method). The refactorings related to the reallocation of system's state (Move and Pull Up Field) as well as the introduction of additional inheritance levels (Extract Superclass/Interface) are less frequent. Finally, the refactorings that move state or behavior to subclasses (Push Down Method/Field) are rarely applied.

4.2 RQ2: Which developers are responsible for refactorings?

The JUnit project version control system references 32 developers, although most of them are not big contributors, they have at least one commit. Figures 2a and 2b show the percentage of the refactoring activities performed by JUnit committers on *production* and *test code*, respectively. It is evident that the refactorings are performed by a limited number of developers (7 contributors on *production code* and 4 on *test code*). Additionally, the proportion of the refactorings contributed by each developer is almost identical between *production* and *test code*. The top two refactoring contributors, namely David Saff and Kent Beck (the current and the former project managers), are also the two top committers with 63% and 12% of the commits, respectively.

Figures 2c and 2d show the distribution of



(a) JUnit production code

(b) JUnit test code

(c) HTTPCore production code

(d) HTTPCore test code

(e) HTTPClient production code

(f) HTTPClient test code

Figure 2: Refactoring contributors.

the refactoring activities contributed to the HTTPCore *production code* and *test code*, respectively. In this case, only 4 developers actively applied *production code* refactorings, while 3 applied *test code* refactorings. We have identified Oleg Kalnichevski (top committer with 78% of the commits), and Roland Weber (third in the list of top committers with 8% of the commits) as the two main refactoring contributors of this project. The second in the top list of committers (Sebastian Bazley) with 11% of the total commits has contributed a small number of refactorings only on *test code*.

Figures 2e and 2f present the refactoring distribution for the HTTPClient *production* and

Table 2: Number of detected refactorings on the production and test code of the examined projects.

	Production code			Test code		
Refactoring/ Project	*JUnit*	*HTTP Core*	*HTTP Client*	*JUnit*	*HTTP Core*	*HTTP Client*
Move Class	94	199	52	131	48	28
Extract Method	66	81	71	2	0	0
Rename Class	22	75	14	22	25	7
Move Method	15	14	8	9	33	2
Pull Up Method	17	11	2	0	0	0
Move Field	1	10	7	0	0	0
Extract Interface	1	12	1	0	0	0
Extract Superclass	4	7	1	0	0	0
Pull Up Field	0	7	1	0	0	0
Push Down Method	1	2	0	0	0	0
Push Down Field	0	3	0	0	0	0

test code, respectively. HTTPClient shares the same development team with HTTPCore except for one additional developer (Jonathan Moore, fourth in the list of top committers with 6% of the commits) who is the second most active refactoring contributor after Oleg Kalnichevski. Similarly to the HTTPCore project, Oleg Kalnichevski (top committer with 58% of the commits) is also the top refactoring contributor. However, Sebastian Bazley and Roland Weber (the second and third in list of top committers with 18% and 16% of the total commits, respectively) have not contributed any refactorings in the HTTPClient project.

It is worth noticing that although HTTPCore (with 8 committers) and HTTPClient (with 9 committers) share almost the same team of developers, only Oleg Kalnichevski contributes refactorings in both projects. This means that the rest of the developers, although they commit code in both projects, they contribute refactorings in only one of them (probably the project they are more familiar with).

Based on these observations, we can conclude that most of the applied refactorings are performed by specific developers that usually have a key role in the management of the project. Moreover, we did not notice any substantial change in the distribution of the refactoring commits by the developers to the *product code* and *test code* of the three projects under study.

4.3 RQ3: Is there more refactoring activity before major project releases than after?

We inspected the refactoring activity around the release points along the lifetime of the three projects under study. We have tracked each project major release dates (based on the public software release dates) and manually examined their relevance in order to filter minor releases such as isolated application of patches, major documentation migration, or change of license types. In order to perform a homogeneous study, we have selected windows of 80 days around each release date. Each window was divided in two groups of 40 days in order to split the analysis in the periods before and after a release point. In our three case studies the window size met two fundamental conditions. First, a high volume of change activity was registered in the repository during the framed time period, and second, the time window did not overlap with others. While in some cases the HTTPCore project had major releases within 2 months of distance, in the JUnit project the commit activity decreased substantially 30 days after the release dates. We found that a window of 80 days is a practical time frame around the release dates, and cohesive with the development style of the three projects under observation.

Figures 3, 4, and 5 present the refactoring activity around the release dates of JUnit (13

releases), HTTPCore (9 releases), and HTTPClient (7 releases), respectively. The distribution of the refactoring frequency around the release dates of each project has been summarized with violin plots [6]. The advantage of a violin plot over a box plot is that the former also includes frequency density information. First, for each project using a single window, we projected the distribution of the refactoring activity by adding the daily refactoring activity counts. This projection allowed us to condense the information of the total refactoring activity per day in a single dimension, and to analyze each project refactoring activity trends around a generic release point. Figure 3b for example, presents the distribution of the refactoring activity around the generic release day for the JUnit project. On the left, the violin plot shows the distribution of the refactoring activity before the release dates of the project, while on the right it portrays the after release counterpart.

In the JUnit project, Figure 3a shows a refactoring activity peak around 20 days before the release day. Moreover, it shows constant refactoring activity 10 days before the release day in multiple release points. On the right side (after the release day) no significant refactoring activity was detected in less than 25 days. Figure 3b shows a wide and tall distribution of refactoring activity before the projected release day. A very small distribution body was spotted on the right, exposing the small significance of the activity after the release point.

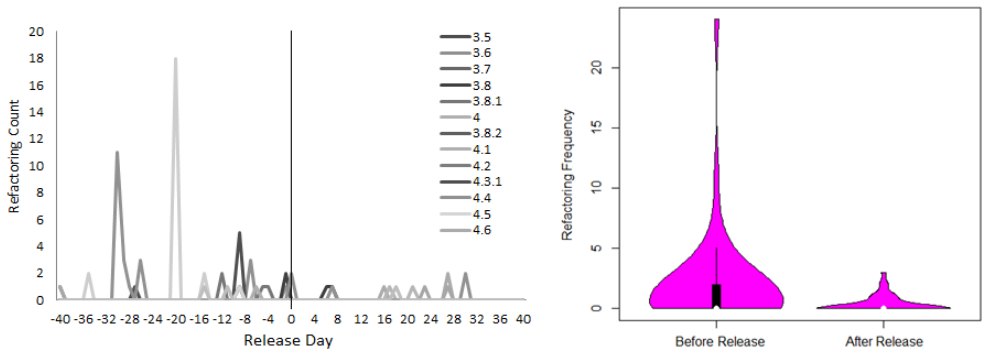
Regarding the HTTPCore project, Figure 4 shows a similar behavior. In this case, most of the refactoring activity that occurred before the release day is situated within a single week. Figure 4a shows significant activity after 10 days of the release day. However, this activity has a much minor scale compared to the one registered before the same release. The violin plot presented in Figure 4b shows the high refactoring distribution before the release days of the project. However, an important but smaller in comparison body was found for the period after the release point.

With respect to the HTTPClient project, Figure 5a shows a common refactoring zone in the period between 36 and 28 days before the project release date. Another common considerable peak can be observed just one week before the release day. In general, the refactoring activity after the release point is very small in comparison to the window before the release. This is also observable in Figure 5b where both distribution bodies are compared.

In the three projects we studied, we observed a common behavior where the refactoring activity is high before the release dates. Moreover, our results expose that the refactoring activity after the releases is almost non-existent, and that in the cases where some activity was found, it was very small compared to the activity observed before the releases. The analysis of commits before and after the release dates revealed that the developers were always committing changes after the releases, although in most cases the number of commits is lower in comparison to the number of commits before the releases. However, this means that the observed absence of refactoring activity after the releases is not due to the absence of commits.

4.4 RQ4: Is refactoring activity on production code preceded by the addition or modification of test code?

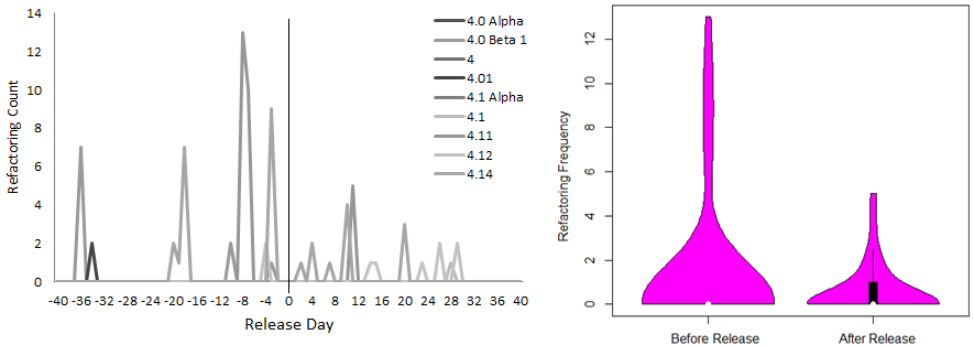
Our methodology for studying this question involved three steps: (i) test activity detection, (ii) manual inspection of test hotspots, and (iii) window analysis construction. In the first step, we built an algorithm that explored the revision history of the projects and detected the addition and modification of test files, counting its occurrences along the lifetime of each project. We followed the same approach presented in Section 4.1 to differentiate *production code* from *test code*. Next, based on the time periods identified as test hotspots, we manually inspected the commit comments, and their respective new or modified sources. Through a manual inspection process we found a number of false positives that were excluded from the analysis. In these cases several *test classes* were modified, but the actual changes were: reorganization of the import headers, addition of copyright notices, or documentation of specific cases. Lastly, in the third step, we grouped the contiguous test hotspots and identified the actual test periods of each project. Taking the



(a) JUnit Refactoring Multi-Release Count

(b) JUnit Refactoring Density – Release Point Comparison

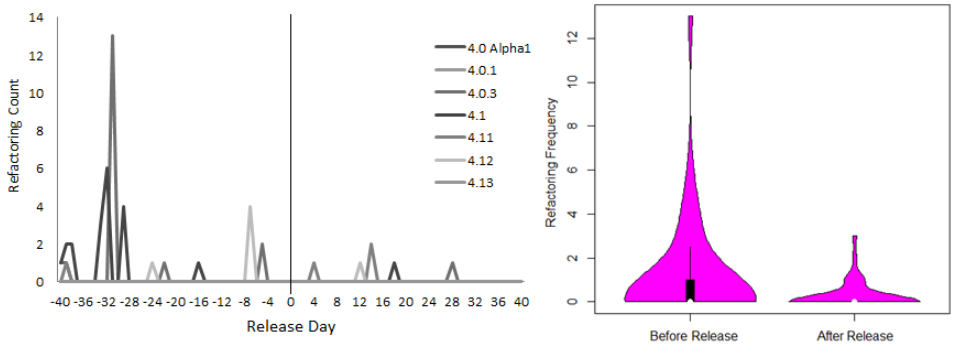
Figure 3: Refactoring Activity Comparison (Release) - JUnit (40 days before and after release)



(a) HTTP-Core Refactoring Multi-Release Count

(b) HTTP-Core Refactoring Density – Release Point Comparison

Figure 4: Refactoring Activity Comparison (Release) - HTTPCore (40 days before and after release)



(a) HTTP-Client Refactoring Multi-Release Count

(b) HTTP-Client Refactoring Density – Release Point Comparison

Figure 5: Refactoring Activity Comparison (Release) - HTTPClient (40 days before and after release)

last day of each period as our pivot point, we followed the same window analysis as the one presented in Section 4.3. Figures 6, 7, and 8 present the refactoring activity of our three case studies around the end of each testing period. The pivot of the window (the zero point) is labeled as End of Testing Period (E.T.P).

In the case of the JUnit project, Figure 6a summarizes the refactoring activity around the end of 10 testing periods. Here, we can observe a common pattern where the refactoring activity was particularly high during the testing phases of the project. Furthermore, in almost all cases, the refactoring activity substantially increased the last days before (and even the same day) the end of testing periods. In the context of the same project, Figure 6b presents the frequency distribution bodies of the project during (left) and after (right) testing periods. A higher refactoring density can be easily observed during the testing phases of the project.

Figure 7a shows the results obtained for the HTTPCore project with 8 testing periods under analysis. In this case, a very peculiar pattern emerged. In almost all cases, the significant refactoring activity peaks occurred the same day the testing period ended. This phenomenon exposed for this project that both testing and refactoring activities were linked and specially ordered. Since the E.T.P day is included in the left portion of the window frequency density analysis, Figure 7b portrays a large density body for the period under the project testing phases.

The refactoring activity around testing periods for the HTTPClient project is very similar to the one observed in the JUnit project. Figure 8a shows two interesting sections. In the first section, around 30 and 10 days before the end of the testing activities, the project presents a considerably high refactoring activity, while the refactoring activity after the end of testing periods is almost non-existent. Consequently, the density summary in Figure 8b is flat and no significant volume of refactorings is observed after the testing end point.

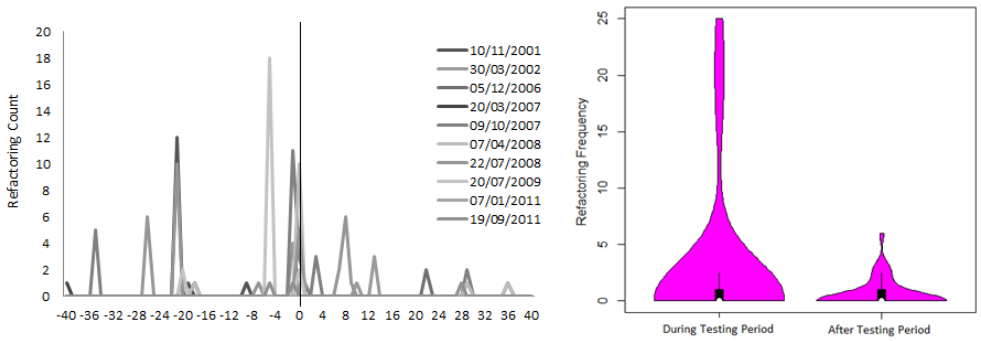
The three projects under examination demonstrate a strong alignment of refactoring activities with testing periods. In each of the analyzed windows, significant refactoring operations were applied along with test modifications. Especially in the HTTPCore project, the majority of refactoring activity was performed on the ending day of testing periods. We believe that one possibility for the alignment between refactoring and test modifications is the adoption of test-driven development practices. In some cases, we observed that the developers updated their test code first and then performed refactorings.

4.5 RQ5: What is the purpose of the applied refactorings?

To answer this question two of the authors (Tsantalis and Guana) inspected together the source code involved in each detected refactoring, in order to postulate the main drivers that motivated the developers towards the application of the refactorings. The process was performed in two rounds. In the first round, a subset of randomly selected refactorings was examined by inspecting the relevant source code before and after the application of the refactoring with a text diff tool. Next, they defined rules (described in the rest of the section) that should apply in order to classify a refactoring instance in each of the defined categories in the first round. In the second round, they performed a systematic labeling of all refactoring instances using the same inspection method by applying the aforementioned rules. For each refactoring instance, the authors inspected the text diff report as well as the corresponding commit logs and suggested a label. In the case of disagreement, a discussion with arguments from both sides took place in order to reach a consensus. In total, 210 Extract Method and 70 Inheritance related refactorings were inspected over the span of approximately 40 hours (i.e., 5 working days). The inspection time for each refactoring was between 5 to 10 minutes depending on the complexity of the diff report and the number of files affected by the corresponding refactoring.

4.5.1 Extract Method refactoring

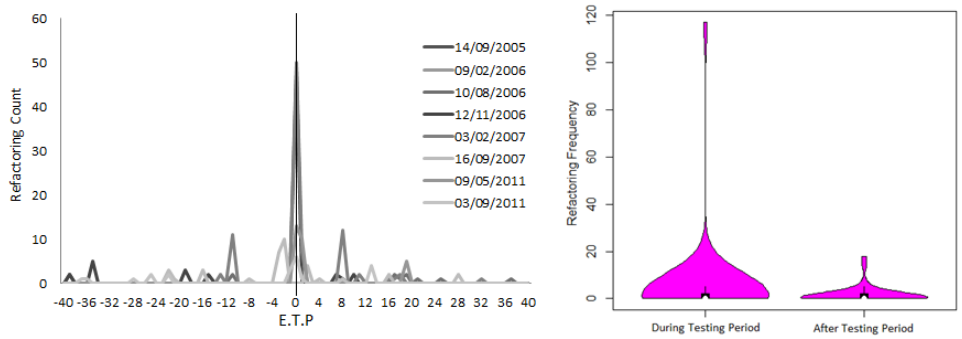
The Extract Method refactoring can be applied to resolve a large variety of code smells [4] either directly or as a part of a more complex transformation, as well as to facilitate the



(a) JUnit Refactoring Multi-Test Count

(b) JUnit Refactoring Density – Testing Point Comparison

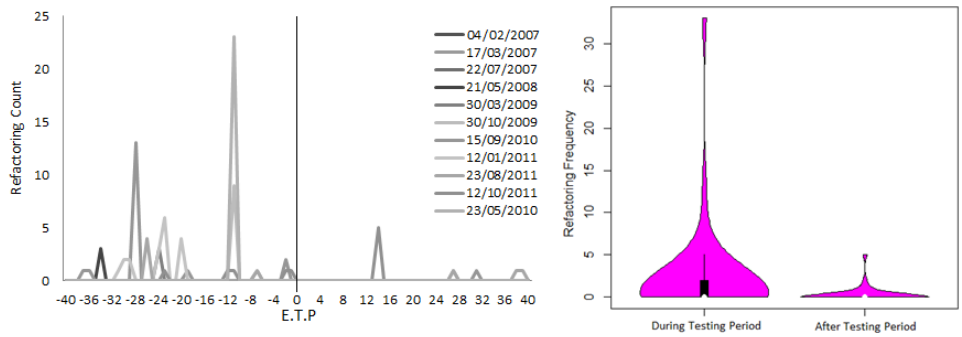
Figure 6: Refactoring Activity Comparison (Test) - JUnit (40 days before and after release)



(a) HTTP-Core Refactoring Multi-Test Count

(b) HTTP-Core Refactoring Density – Testing Point Comparison

Figure 7: Refactoring Activity Comparison (Test) - HTTPCore (40 days before and after release)



(a) HTTP-Client Refactoring Multi-Test Count

(b) HTTP-Client Refactoring Density – Testing Point Comparison

Figure 8: Refactoring Activity Comparison (Test) - HTTPClient (40 days before and after release)

introduction of new functionalities and bug fixes. To provide an insight to this diversity, we inspected in total 210 instances (detected in the three examined projects) and found three main motivations, namely *Code smell resolution*, *Extension* and *Backward compatibility*, which are further divided into more fine-grained sub-categories:

Code smell resolution:

- *Remove duplicate code*: Duplicate code fragments are replaced with a single method call to the extracted method.
- *Decompose method*: A code fragment having a distinct functionality is extracted into a new separate method in order to decompose and simplify a long or complex method (Long Method [4]).
- *Hide message chain*: A chain of method calls is extracted and replaced with a single method call in order to simplify the original method (Message Chain [4]).
- *Encapsulate field*: A field access/assignment is replaced with a call to a newly introduced getter/setter method and the visibility of the field is reduced in order to enforce the design principle of encapsulation or data hiding.

Extension:

- *Facilitate functionality extension*: The extracted/original method contains newly added code apart from the extracted/removed one. The goal is to simplify the code in the original method and make easier the addition of the new functionality.
- *Introduce polymorphism*: The extracted method is abstract and the extracted code is moved to a subclass providing a concrete implementation for the newly introduced abstract method. The purpose is to enable future extension of the system through polymorphism.
- *Self encapsulate field*: A field access/assignment is replaced with a call to a newly introduced getter/setter method, but the access modifier of the field remains the same. The goal is to enable the future extension of the system, since *indirect variable access* allows subclasses to override the superclass getter/setter and manage the data with more flexibility (e.g., lazy initialization).
- *Introduce factory method*: The creation of an object is replaced with a call to a newly introduced factory method creating the same object. The goal is to enable future extension of the system, since constructors can only return an instance of the object that is asked for, while a factory method can return instances of subclass types as well [4].

Backward compatibility:

- *Extract Delegate*: The entire body of a method or a single method call (without chain) is extracted into a new method. The goal is (a) to improve the name of a method, (b) to remove unnecessary or unused parameters from a method, or (c) to deprecate a method and at the same time preserve the public API of the class.

Table 3: Motivations for the application of Extract Method refactorings.

Motivation	**Sub-category**	**Count (%)**
Code smell	Decompose method	32 (15%)
	Remove duplication	21 (10%)
	Hide Message chain	19 (9%)
	Encapsulate field	20 (10%)
Extension	Facilitate extension	23 (11%)
	Use Polymorphism	9 (4%)
	Self encapsulate field	24 (11%)
	Factory method	37 (18%)
Backward compatibility	Extract Delegate	25 (12%)
	Total	210 (100%)

As one can observe from Table 3, the Extract Method refactoring serves a large variety of different purposes. With respect to the code smells being resolved in the examined projects, the use of the Extract Method refactoring for the decomposition of methods is more dominant compared to the other three code smells.

Additionally, there is an almost equal use of the Extract Method refactoring for the purposes of removing duplicate code, hiding message chains and encapsulating fields. With respect to the extension motivation, the Extract method refactoring has been primarily used to introduce factory methods. Moreover, it has been used to facilitate the introduction of new functionality and self encapsulate fields, while it has been more rarely used for the introduction of polymorphism (actually instances of the latter motivation were found only in HTTPCore project). Finally, the Extract Method refactoring has been used a significant number of times across the three examined projects for preserving the backward compatibility and public API of the classes. By analyzing the results for each project separately, we found out that the application of the Extract Method refactoring is driven more by code smells in JUnit (where 53% of the refactorings were applied for the resolution of code smells) and HTTPClient (where 63% of the refactorings were applied for the resolution of code smells), while it is more extension driven in HTTPCore (where 74% of the refactorings were applied for extension purposes).

4.5.2 Inheritance related refactorings

For the inheritance related refactorings (Pull Up/Push Down Method/Field, Extract Superclass/Interface) we inspected in total 70 instances (detected in the three examined projects) and found three main motivations, namely *Code smell resolution*, *Extension* and *Abstraction level refinement*:

Code smell resolution:

- *Remove duplicate code*: Duplicate methods or fields declared in multiple classes are moved to an already existing or newly introduced common superclass. The goal is to make the subclasses inherit and not copy the duplicate behavior/state.

Extension:

- *Form template method*: The Template Method design pattern [5] is introduced by "pulling up" into a template method the behavior of methods having the same signature but different functionality. The body of the template method calls a newly introduced abstract method in the superclass and the subclasses provide a concrete implementation for the abstract method by copying the bodies of the pulled up methods.

Abstraction level refinement:

- *Generalize by pulling up code*: A method-/field is moved from a subclass to an already existing or newly introduced superclass. The goal is to reallocate behavior/state to a higher level of abstraction.

- *Specialize by pushing down code*: A method/field is moved from a superclass to a subclass. The goal is to reallocate the behavior/state that is not used by a superclass to a lower level of abstraction.

- *Generalize contract*: A new interface is created containing methods which are already implemented in existing classes. The goal is to introduce an additional level of abstraction to capture the behavior being common between different classes.

- *Decompose interface*: A portion of the methods declared in an existing interface is moved to a new interface that extends the original. The goal is to make the original interface "thinner" and cover future classes that do not need to implement the removed methods.

As one can observe from Table 4, the application of inheritance-related refactorings is mainly motivated by the removal of duplicate code and the generalization of abstraction (either by pulling up code or extracting common superclasses and interfaces). On the other hand, the specialization of abstraction (by pushing down code) and the decomposition of interfaces are rarely applied practices and have been mainly observed in the HTTPCore project. Finally, the application of inheritance-related refactorings in order to form instances of the Template Method design pattern was observed only in the JUnit project and thus we cannot draw any general conclusion about this specific motivation.

Table 4: Motivations for the application of Inheritance related refactorings.

Motivation	Sub-category	Count (%)
Code smell	Remove duplication	22 (31%)
Extension	Template method	14 (20%)
Abstraction level refinement	Generalize (pull up)	12 (17%)
	Generalize contract	12 (17%)
	Specialize (push down)	6 (9%)
	Decompose interface	4 (6%)
	Total	70 (100%)

5 Threats to Validity

Internal validity is threatened by the presence of false negatives (i.e., undetected actual refactorings). Our refactoring detection technique, as mentioned in Section 4, has a small number of false positives and thus high precision, yet it was not possible to compute its recall due to the absence of the actual refactorings applied in the examined projects. Furthermore, we tried to mitigate inconsistencies in the labeling of refactorings into the intent categories through a consensus building process.

External validity is threatened by the scope of the study. The selected projects are developed in Java, are managed by relatively small teams and constitute libraries (in the case of HTTP Components) or frameworks (in the case of JUnit). Consequently, we cannot claim that the results of the study can be generalized to other programming languages where refactoring tool support may be different, projects with a significantly larger number of developers or different team organization, and different software systems (e.g., applications) where the need for backward compatibility and extensibility might be less important.

6 Conclusions

In this empirical study we investigated five questions addressing different refactoring activity in three projects.

RQ1: Do software developers perform different types of refactoring operations on test code and production code? We concluded that there is a wider variety of refactoring types applied to *production code*. Additionally, we observed that this activity was primarily focused on design improvements such as, resolution of code smells and modularization refinement. In terms of the refactorings observed on *test code*, there is a clear focus on internal reorganization of the classes of each project into packages and renaming of classes for conceptual reasons.

RQ2: Which developers are responsible for refactorings? We observed that specific developers have taken over the responsibility of planning and performing refactorings. We have found for all of the projects that we studied this role was mainly fulfilled by a single developer acting as a refactoring manager.

RQ3: Is there more refactoring activity before major project releases than after? In our three case studies, we observed that the refactoring activity is significantly more frequent before a release than after. We believe this activity is motivated by the desire to improve the design, and prepare the code for future extensions before stable versions of the API are released to the public.

RQ4: Is refactoring activity on production code preceded by the addition or modification of test code? We found a tight alignment between refactorings and active periods of test code changes. We detected intense refactoring activity during the testing periods of the projects that declined immediately after the test period ended. Our results and posterior manual inspection led us to conclude that testing and refactoring are dependent implying the adoption test-driven development practices in the examined projects.

RQ5: What is the purpose of the applied refactorings? We found a wide variety of reasons motivating the application of the Extract Method refactoring. With respect to code smells, the decomposition of methods was the most dominant motivation for applying Extract Method refactorings. With respect to facilitating extension, the primary motivation was the introduction of Factory Methods. Regarding inheritance related refactorings, the main motivation was the removal of duplicate code and the generalization of abstraction, while the decomposition of interfaces and specialization of abstraction were rarely applied.

Lessons learned: Continuous Code Quality Management platforms (e.g., Sonar) have mainly incorporated code convention checkers (e.g., FindBugs, Checkstyle, and PMD) to assess and improve code quality. There is also evidence that open-source projects fix code convention violations before public releases [12]. We believe that the results of the study highlight the need for incorporating code smell detection and resolution tools in Code Quality Management platforms. Additionally, research around the detection of refactoring opportunities has mainly focused on refactorings that remove code smells. Based on the results of the study, developers refactor their code for other reasons as well (e.g., extension, and backward compatibility). Consequently, current and future refactoring recommendation tools should support refactorings serving multiple purposes.

Acknowledgments: The authors would like to acknowledge the generous support of NSERC, AITF (former iCORE), and IBM.

References

[1] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 53–62, 2011.

[2] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, 2009.

[3] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 404–428. 2006.

[4] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, Boston, MA, USA, 1999.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] J.L. Hintze and R.D. Nelson. Violin plots: a box plot-density trace synergism. *American Statistician*, 52(2):181–184, 1998.

[7] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160, 2011.

[8] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, 2007.

[9] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 50:1–50:11, 2012.

[10] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[11] Napol Rachatasumrit and Miryung Kim. An empirical investigation into the impact of refactoring on regression testing. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pages 357–366, 2012.

[12] Michael Smit, Barry Gergel, H. James Hoover, and Eleni Stroulia. Code convention adherence in evolving software. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pages 504–507, 2011.

[13] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, 2006.

[14] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, 2005.

[15] Zhenchang Xing and Eleni 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*, pages 458–468, 2006.