Why We Refactor? Confessions of GitHub Contributors

Danilo Silva
Universidade Federal de
Minas Gerais, Brazil
danilofs@dcc.ufmg.br

Nikolaos Tsantalis
Concordia University
Montreal, Canada
tsantalis@cse.concordia.ca

Marco Tulio Valente
Universidade Federal de
Minas Gerais, Brazil
mtov@dcc.ufmg.br

ABSTRACT

Refactoring is a widespread practice that helps developers to improve the maintainability and readability of their code. However, there is a limited number of studies empirically investigating the actual motivations behind specific refactoring operations applied by developers. To fill this gap, we monitored Java projects hosted on GitHub to detect recently applied refactorings, and asked the developers to explain the reasons behind their decision to refactor the code. By applying thematic analysis on the collected responses, we compiled a catalogue of 44 distinct motivations for 12 well-known refactoring types. We found that refactoring activity is mainly driven by changes in the requirements and much less by code smells. EXTRACT METHOD is the most versatile refactoring operation serving 11 different purposes. Finally, we found evidence that the IDE used by the developers affects the adoption of automated refactoring tools.

CCS Concepts

•**Software and its engineering → Software evolution;** *Maintaining software; Software maintenance tools;*

Keywords

Refactoring, software evolution, code smells, GitHub

1. INTRODUCTION

Refactoring is the process of improving the design of an existing code base, without changing its behavior [27]. Since the beginning, the adoption of refactoring practices was fostered by the availability of refactoring catalogues, as the one proposed by Fowler [10]. These catalogues define a name and describe the mechanics of each refactoring, as well as demonstrate its application through some code examples. They also provide a *motivation* for the refactoring, which is usually associated to the resolution of a code smell. For example, EXTRACT METHOD is recommended to decompose a large and complex method or to eliminate code duplication.

As a second example, MOVE METHOD is associated to smells like Feature Envy and Shotgun Surgery [10].

There is a limited number of studies investigating the real motivations driving the refactoring practice based on interviews and feedback from actual developers. Kim et al. [17] explicitly asked developers "in which situations do you perform refactorings?" and recorded 10 code symptoms that motivate developers to initiate refactoring. Wang [40] interviewed professional software developers about the major factors that motivate their refactoring activities and recorded human and social factors affecting the refactoring practice. However, both studies were based on general-purpose surveys or interviews that were not focusing the discussion on specific refactoring operations applied by the developers, but rather on general opinions about the practice of refactoring.

Contribution: To the best of our knowledge, this is the first study investigating *the motivations behind refactoring based on the actual explanations of developers on specific refactorings they have recently applied.* To fill this gap on the empirical research in this area, we report a large scale study centered on 463 refactorings identified in 222 commits from 124 popular, Java-based projects hosted on GitHub. In this study, we asked the developers who actually performed these refactorings to explain the reasons behind their decision to refactor the code. Next, by applying thematic analysis [6], we categorized their responses into different themes of motivations. Another contribution of this study is that we make publicly available[1] the data collected and the tools used to enable the replication of our findings and facilitate future research on refactoring.

Relevance to existing research: The results of this empirical study are important for two main reasons.

First, having a list of motivations driving the application of refactorings can help researchers and practitioners to infer rules for the automatic detection of these motivations when analyzing the commit history of a project. Recent research has devised techniques to help in understanding better the practice of code evolution by identifying frequent code change patterns from a fine-grained sequence of code changes [26], isolating non-essential changes in commits [13], and untangling commits with bundled changes (e.g., bug fix and refactoring) [7]. In addition, we have empirical evidence that developers tend to interleave refactoring with other types of programming activity [24], i.e., developers tend to *floss refactor*. Therefore, *knowing the motivation behind a refactoring can help us to understand better other related changes in a commit.* In fact, in this study we found

---

[1]http://aserg-ufmg.github.io/why-we-refactor

several cases where developers extract methods in order to make easier the implementation of a feature or a bug fix.

Second, having a list of motivations driving the application of refactorings can help researchers and practitioners to develop refactoring recommendation systems tailored to the actual needs and practices of the developers. Refactoring serves multiple purposes [10], such as improving the design, understanding the code [9], finding bugs, and improving productivity. However, research on refactoring recommendation systems [1] has mostly focused on the design improvement aspect of refactoring by proposing solutions oriented to code smell resolution. For example, most refactoring recommenders have been designed based on the concept that developers extract methods either to eliminate code duplication, or decompose long methods [36, 31, 34, 12, 18, 38]. In this study, we found 11 different reasons behind the application of EXTRACT METHOD refactorings. Each motivation requires a different strategy in order to detect suitable refactoring opportunities. Building refactoring recommendation systems tailored to the real needs of developers will help to promote more effectively the practice of refactoring to the developers, by recommending refactorings helping to solve the actual problems they are facing in maintenance tasks.

2. RELATED WORK

Refactoring is recognized as a fundamental practice to maintain a healthy code base [10, 4, 27, 19]. For this reason, vast empirical research was recently conducted to extend our knowledge on this practice.

Studies on refactoring practices: Murphy et al. [20] record the first results on refactoring usage, collected using the Mylar Monitor, a standalone framework that collects and reports trace information about a user's activity in Eclipse. Murphy-Hill et al. [24] rely on multiple data sources to reveal how developers practice refactoring activities. They investigate nine hypotheses about refactoring usage and conclude for instance that commit messages do not reliably indicate the presence of refactoring, that programmers usually perform several refactorings within a short time period, and that 90% of refactorings are performed manually. Negara et al. [25] provide a detailed breakdown on the manual and automated usage of refactoring, using a large corpus of refactoring instances detected using an algorithm that infers refactorings from fine-grained code edits. As their central findings, they report that more than half of the refactorings are performed manually and that 30% of the applied refactorings do not reach the version control system.

Studies based on surveys & interviews: Kim et al. [16, 17] present a field study of refactoring benefits and challenges in a major software organization. They conduct a survey with developers at Microsoft regarding the cost and risks of refactoring in general, and the adequacy of refactoring tool support, and find that the developers put less emphasis on the behavior preserving requirement of refactoring definitions. They also interview a designated Windows refactoring team to get insights into how system-wide refactoring was carried out, and report that the binary modules refactored by the refactoring team had a significant reduction in the number of inter-module dependencies and post-release defects. Wang [40] interviews 10 professional software developers and finds a list of intrinsic (i.e., self-motivated) and external (i.e., forced by peers or the management) factors motivating refactoring activity.

Studies on refactoring tools: Vakilian et al. [39] reveal many factors that affect the appropriate and inappropriate use of refactoring tools. They show for example that novice developers may underuse some refactoring tools due to lack of awareness. Murphy-Hill et al. [21] investigate the barriers in using the tool support provided for the EXTRACT METHOD refactoring [10]. They report that users frequently made mistakes in selecting the code fragment they want to extract and that error messages from refactoring engines are hard to understand. Murphy-Hill et al. [24] show that 90% of configuration defaults in refactoring tools are not changed by the developers. As a practical consequence of these studies, refactoring recommendation systems [1] have been proposed to foster the use of refactoring tools and leverage the benefits of refactoring, by alerting developers about potential refactoring opportunities [35, 36, 2, 30, 3, 31].

Studies on refactoring risks: Kim et al. [14] show that there is an increase in the number of bug fixes after API-level refactorings. Rachatasumrit and Kim [29] show that refactorings are involved in almost half of the failed test cases. Weißgerber and Diehl show that refactorings are sometimes followed by an increasing ratio of bug reports [41].

However, existing studies on refactoring practices do not investigate in-depth the *motivation* behind specific refactoring types, i.e., why developers decide to perform a certain refactoring operation. For instance, Kim et al. [17] do not differentiate the motivations between different refactoring types, and Wang [40] does not focus on the technical motivations, but rather on the human and social factors affecting the refactoring practice in general. The only exception is a study conducted by Tsantalis et al. [37], in which the authors themselves manually inspected the relevant source code before and after the application of a refactoring with a text diff tool, to reveal possible motivations for the applied refactorings. Because they conducted this study without asking the opinion of the developers who actually performed the refactorings, the interpretation of the motivation can be considered subjective and biased by the opinions and perspectives of the authors. In addition, the manual inspection of source code changes is a rather tedious and error-prone task that could affect the correctness of their findings. Finally, the examined refactorings were collected from the history of only three open source projects, which were libraries or frameworks. This is a threat to the external validity of the study limiting the ability to generalize its findings beyond the characteristics of the selected projects. In this study, we collected refactorings from 124 different projects, and asked the developers who actually performed these refactorings to explain the reasons behind their decision to refactor the code.

3. RESEARCH METHODOLOGY

3.1 Selection of GitHub Repositories

First, we selected the top 1,000 Java repositories ordered by popularity in GitHub (stargazers count) that are not forks. From this initial list, we discarded the lower quartile ordered by number of commits, to focus the study on repositories with more maintenance activity. The final selection consists of 748 repositories, including well-known projects, like JETBRAINS/INTELLIJ-COMMUNITY, APACHE/CASSANDRA, ELASTIC/ELASTICSEARCH, GWTPROJECT/GWT, and SPRING-PROJECTS/SPRING-FRAMEWORK.

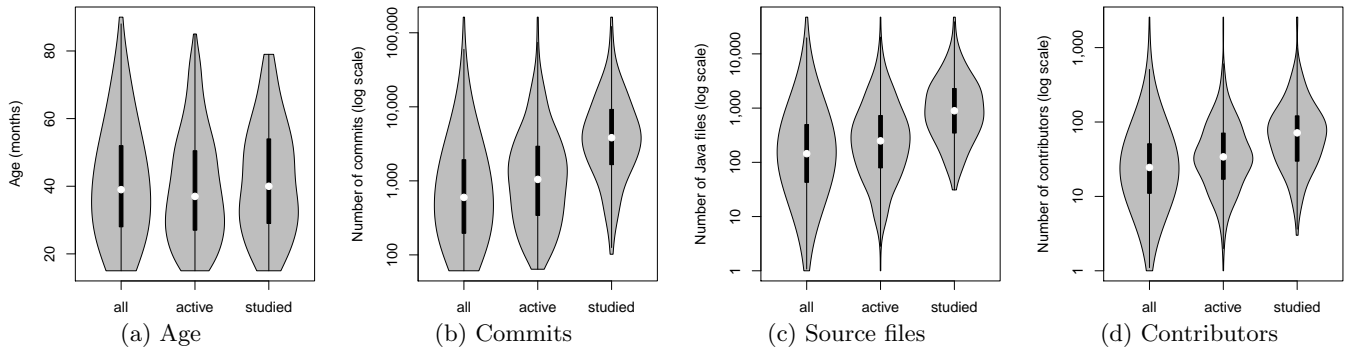Figure 1 shows violin plots [11] with the distribution of

Figure 1: Distribution of (a) age, (b) commits, (c) Java source files, and (d) contributors of repositories

age (in months), number of commits, size (number of ∗.java files), and number of contributors of the selected repositories. We provide plots for all 748 systems (labeled as *all*), for the 471 systems (63%) with at least one commit during the study period (labeled as *active*), and for the 124 systems (17%) effectively analyzed in the study (labeled as *studied*), which correspond to the repositories with at least one refactoring detected in the commits during the study period (61 days), along with answers from the developers to our questions about the motivation behind the detected refactorings. We can observe in Figure 1 that the *active* systems tend to have a higher number of commits, source files, and contributors than the initially selected systems (*all*). The same holds when comparing the *studied* systems with the *active* systems. These observations are statistically confirmed by applying the one-tailed variant of the Mann-Whitney U test.

3.2 RefactoringMiner Tool

In the study, we search for refactorings performed in the version history of the selected GitHub repositories by analyzing the differences between the source code of two revisions. For this purpose, we use a refactoring detection tool proposed in a previous work [37]. The tool, named RefactoringMiner in this paper, implements a lightweight version of the UMLDiff [42] algorithm for differencing object-oriented models. This algorithm is used to infer the set of classes, methods, and fields added, deleted or moved between successive code revisions. After executing this algorithm, a set of rules is used to identify different types of refactorings. Unlike other existing refactoring detection tools, such as Ref-Finder [15] and JDevAn [43], RefactoringMiner provides an API and can be used as an external library independently from an IDE, while Ref-Finder and JDevAn can be executed only within the Eclipse IDE. The strong dependency of Ref-Finder and JDevAn to the Eclipse IDE prevented us from using these tools in our study, since as it will be explained in Section 3.3, our study required a high degree of automation, and this could be achieved only by being able to use RefactoringMiner programmatically through its API.

In the study, we analyze 12 well-known refactoring types detected by RefactoringMiner, as listed in the first column of Table 2. The detection of RENAME CLASS/METHOD/FIELD refactorings is not currently supported by RefactoringMiner, because it requires a more advanced source code analysis that examines changes in usage patterns (i.e., changes in class instantiations, method call sites, field accesses, respectively) to verify the consistency of the renaming operation. Typically, these refactorings are performed to give a more meaningful name to the renamed code element. Previous

studies show that they are usually performed automatically, using the refactoring tool support of popular IDEs [24, 25].

3.2.1 RefactoringMiner Precision and Recall

As we rely on RefactoringMiner to find refactorings performed in the version history of software repositories, it is important to estimate its recall and precision. For this reason, we evaluated RefactoringMiner using the dataset reported in a study by Chaparro et al. [5]. This dataset includes a list of refactorings performed by two Ph.D. students on two software systems (ArgoUML and aTunes) along with the source code before and after the modifications. There are 173 refactoring instances in total, from which we selected all 120 instances corresponding to 8 of the refactoring types considered in this study (8×15 instances per type). The dataset does not contain instances of EXTRACT SUPER-CLASS/INTERFACE, MOVE CLASS, and RENAME PACKAGE refactorings. We compared the list of refactorings detected by RefactoringMiner with the known refactorings in those systems to obtain the results of Table 1, which presents the number of true positives (TP), the number of false positives (FP), the number of false negatives (FN), the recall and precision for each refactoring type. In total, there are 111 true positives (i.e., existing refactoring instances that were correctly detected) and 9 false negatives (i.e., existing refactoring instances that were not detected), which yield a fairly high recall of 0.93. Besides, there are 2 false positives (i.e., incorrectly detected refactoring instances), which yield a precision of 0.98. The lowest observed recall is for PULL UP METHOD (0.80), while the lowest observed precision is for EXTRACT METHOD (0.88).

In conclusion, the accuracy of RefactoringMiner is at acceptable levels, since Ref-Finder (the current state-of-the-art refactoring reconstruction tool) has an overall precision of 79% according to the experiments conducted by its own authors [28], while an independent study by Soares et al. [33] has shown an overall precision of 35% and an overall recall of 24% for Ref-Finder.

3.3 Study Design

During 61 days (between June 8th and August 7th 2015), we monitored all selected repositories to detect refactorings. We built an automated system that periodically fetches commits from each remote repository to a local copy (using the `git fetch` operation). Next, the system iterates through each commit and executes RefactoringMiner to find refactorings and store them in a relational database.

As in a previous study [37], we compare each examined commit with its parent commit in the directed acyclic graph

Table 1: RefactoringMiner Recall and Precision

Refactoring	TP	FP	FN	Recall	Prec.
EXTRACT METHOD	15	2	0	1.00	0.88
INLINE METHOD	13	0	2	0.87	1.00
PULL UP ATTRIBUTE	15	0	0	1.00	1.00
PULL UP METHOD	12	0	3	0.80	1.00
PUSH DOWN ATTRIBUTE	15	0	0	1.00	1.00
PUSH DOWN METHOD	13	0	2	0.87	1.00
MOVE ATTRIBUTE	15	0	0	1.00	1.00
MOVE METHOD	13	0	2	0.87	1.00
Total	111	2	9	0.93	0.98

(DAG) that models the commit history in git-based version control repositories. Furthermore, we exclude merge commits from our analysis to avoid the duplicate report of refactorings. Suppose that commit C_M merges two branches containing commits C_A and C_B, respectively. Suppose also that a refactoring *ref* is performed in C_A, and therefore detected when we compare C_A with its parent commit. Because the effects of *ref* are present in the code that resulted from C_M, *ref* would be detected again if we compared C_M with C_B. Therefore, we assume that discarding merge commits from our analysis does not lead to any refactoring loss, but rather avoids duplicate refactoring reports.

On each working day, we retrieved the recent refactorings from the database to perform a manual inspection, using a web interface we built to aid this task. In this step, we filter out false positives by analyzing the source code diff of the commit. In this way, we avoid asking developers about false refactorings. Additionally, we also marked commits that already include an explanation for the detected refactoring in the commit description, to avoid asking an unnecessary question. For instance, in one of the analyzed commits we found several methods extracted from a method named `onCreate`, and the commit description was:

"Refactored `AIMSICDDbAdapter::DbHelper#onCreate` for easier reading"

Thus, it is clear that the intention of the refactoring was to improve readability by decomposing method `onCreate`. Therefore, it would be unnecessary and inconvenient to ask the developer.

This process was repeated daily, to detect the refactorings as soon as possible after their application in the examined systems. In this way, we managed to ask the developers shortly after they perform a refactoring, to increase the chances of receiving an accurate response. We send at most one email to a given developer, i.e., if we detect a refactoring by a developer who has been already contacted before, we do not contact her again, to avoid the perception of our messages as spam email. The email addresses of the developers were retrieved from the commit metadata.

In each email, we describe the detected refactoring(s) and provide a GitHub URL for the commit where the refactoring(s) is(are) detected. In the email, we asked two questions:

1. Could you describe why did you perform the listed refactoring(s)?
2. Did you perform the refactoring(s) using the automated refactoring support of your IDE?

With the first question, our goal is to reveal the actual motivation behind real refactorings instances. With the second question, we intend to collect data about the adequacy and usage of refactoring tools, previously investigated in other empirical studies [24, 25]. In this way, we can check whether the findings of these studies are reproduced in our study. We should clarify that by "automated refactoring" we refer to user actions that trigger the refactoring engine of an IDE by any means (e.g., through the IDE menus, keyboard shortcuts, or drag-and-drop of source code elements).

During the study period, we sent 465 emails and received 195 responses, achieving a response rate of 41.9%. Each response corresponds to a distinct developer and commit. The achieved response rate is significantly larger than the typical 5% rate found in questionnaire-based software engineering surveys [32]. This can be attributed to the *firehouse interview* [22] nature of our approach, in which developers provide their feedback shortly after performing a refactoring and have fresh in their memory the motivation behind it. Additionally, we included in our analysis all 27 commits whose description already explained the reasons for the applied refactorings, totaling a set of 222 commits. This set of commits covers 124 different projects and contains 463 refactoring instances in total.

After collecting all responses, we analyzed the answers using thematic analysis [6], a technique for identifying and recording patterns (or "themes") within a collection of documents. Thematic analysis involves the following steps: (1) initial reading of the developer responses, (2) generating initial codes for each response, (3) searching for themes among codes, (4) reviewing the themes to find opportunities for merging, and (5) defining and naming the final themes. These five steps were performed independently by the first two authors of the paper, with the support of a simple web interface we built to allow the analysis and tagging of the detected refactorings. At the time of the study, the first author (Author#1) had 3 years of research experience on refactoring, while the second author (Author#2) had over 8 years of research experience on refactoring.

After the generation of themes from both authors, a meeting was held to assign the final themes. In 155 cases (58%), both authors suggested semantically equivalent themes that were rephrased and standardized to compose the final set of themes. The refactorings with divergent themes were then discussed by both authors to reach a consensus. In 94 cases (35%), one author accepted the theme proposed by the other author. In the remaining 18 cases (7%), the final theme emerged from the discussion and was different from what both authors previously suggested. Figure 2 shows a case of an EXTRACT METHOD refactoring instance that was required to reach a consensus between the authors. The developer who performed the refactoring explained that the reason for the refactoring was to support a new feature that required pagination, as described in the following comment:

"Educational part of PyCharm uses stepic.org courses provider. This server recently decided to use pagination in replies."

By inspecting the source code changes, we can see that a part of the original method `getCourses()` (left-hand side of Figure 2) was extracted into method `addCoursesFromStepic()` (right-hand side of Figure 2). After the refactoring, the extracted method is called twice, once before the `while` loop added in the original method, and once inside the `while` loop. For this reason, Author#1 labeled this case as "Avoid duplication", since the extracted method is reused two times after the refactoring. However, the extracted method contains additional new code to compute properly the URL based on the page number passed as a
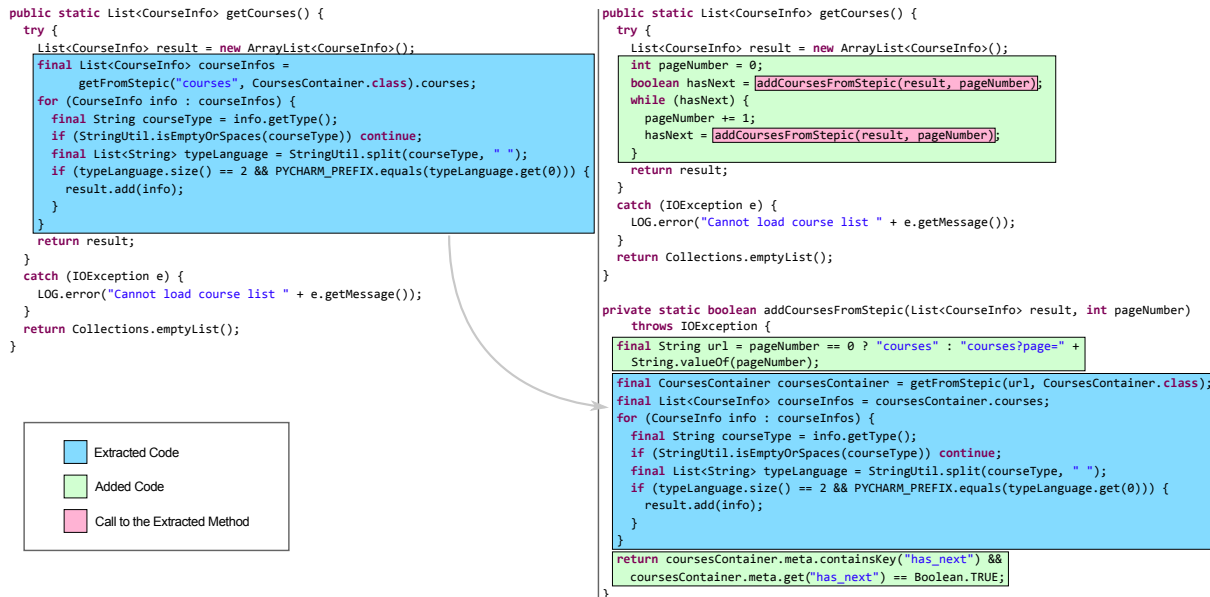
```
public static List<CourseInfo> getCourses() {
  try {
    List<CourseInfo> result = new ArrayList<CourseInfo>();
    final List<CourseInfo> courseInfos =
        getFromStepic("courses", CoursesContainer.class).courses;
    for (CourseInfo info : courseInfos) {
      final String courseType = info.getType();
      if (StringUtil.isEmptyOrSpaces(courseType)) continue;
      final List<String> typeLanguage = StringUtil.split(courseType, " ");
      if (typeLanguage.size() == 2 && PYCHARM_PREFIX.equals(typeLanguage.get(0))) {
        result.add(info);
      }
    }
    return result;
  }
  catch (IOException e) {
    LOG.error("Cannot load course list " + e.getMessage());
  }
  return Collections.emptyList();
}
```

```
public static List<CourseInfo> getCourses() {
  try {
    List<CourseInfo> result = new ArrayList<CourseInfo>();
    int pageNumber = 0;
    boolean hasNext = addCoursesFromStepic(result, pageNumber);
    while (hasNext) {
      pageNumber += 1;
      hasNext = addCoursesFromStepic(result, pageNumber);
    }
    return result;
  }
  catch (IOException e) {
    LOG.error("Cannot load course list " + e.getMessage());
  }
  return Collections.emptyList();
}

private static boolean addCoursesFromStepic(List<CourseInfo> result, int pageNumber)
    throws IOException {
  final String url = pageNumber == 0 ? "courses" : "courses?page=" +
    String.valueOf(pageNumber);
  final CoursesContainer coursesContainer = getFromStepic(url, CoursesContainer.class);
  final List<CourseInfo> courseInfos = coursesContainer.courses;
  for (CourseInfo info : courseInfos) {
    final String courseType = info.getType();
    if (StringUtil.isEmptyOrSpaces(courseType)) continue;
    final List<String> typeLanguage = StringUtil.split(courseType, " ");
    if (typeLanguage.size() == 2 && PYCHARM_PREFIX.equals(typeLanguage.get(0))) {
      result.add(info);
    }
  }
  return coursesContainer.meta.containsKey("has_next") &&
    coursesContainer.meta.get("has_next") == Boolean.TRUE;
}
```

Extracted Code
Added Code
Call to the Extracted Method

Figure 2: Example of Extract Method refactoring that was required to reach a consensus.

parameter (first line in the extracted method), and to return a `boolean` indicating if there exists a next page (last line in the extracted method). For this reason, Author#2 labeled this case as "Facilitate extension", since the extracted method also helps to implement the new pagination requirement. After deliberation, the authors reached a consensus by keeping both theme labels, since the extracted method serves both purposes of reuse and extension.

3.4 Examined Refactorings

We monitored 748 Java projects during the study period, and found commits in 471 projects (63%), i.e., 277 projects remained inactive. We also found 285 projects with refactoring activity, as detected by RefactoringMiner (including false positives). In these projects, 2,241 refactoring instances were detected (in 729 commits), and were manually inspected by the first author of the paper to confirm whether they are indeed true positives.

Table 2: Refactoring activity

Refactoring	TP	FP	Prec.	Com.	Proj.
EXTRACT METHOD	468	135	0.78	312	136
MOVE CLASS	432	512	0.46	85	60
MOVE ATTRIBUTE	129	44	0.75	45	38
RENAME PACKAGE	105	0	1.00	25	24
MOVE METHOD	99	48	0.67	40	31
INLINE METHOD	58	67	0.46	44	36
PULL UP METHOD	33	1	0.97	18	17
PULL UP ATTRIBUTE	23	1	0.96	11	11
EXTRACT SUPERCLASS	22	11	0.67	18	16
PUSH DOWN METHOD	16	1	0.94	6	6
PUSH DOWN ATTRIBUTE	15	1	0.94	7	7
EXTRACT INTERFACE	11	8	0.58	10	9
Total	1411	830	0.63	539	185

Table 2 shows the number of true positives (TP), false positives (FP), and precision (Prec.) of RefactoringMiner by refactoring type, as computed after the manual inspection of the detected refactorings. In general, our tool achieves very high precision for RENAME PACKAGE (100%), PULL UP/PUSH DOWN ATTRIBUTE/METHOD refactorings (over 94%), and relatively high precision for EXTRACT METHOD and MOVE ATTRIBUTE refactorings (over 75%), while the precision for MOVE METHOD and EXTRACT SUPERCLASS is 67%. However, for some refactorings the precision is closer to 50%, namely EXTRACT INTERFACE (58%), INLINE METHOD (46%), and MOVE CLASS (46%). We observed several cases of inner classes falsely detected as moved, because their enclosing class was simply renamed. By supporting the detection of RENAME CLASS refactoring, we could improve MOVE CLASS precision. It should be emphasized that we asked the developers only about the true positives detected by RefactoringMiner. In comparison to the results presented in Section 3.2.1, the precision is lower, because the commits analyzed from GitHub projects may include tangled changes, while the commits analyzed in Section 3.2.1 include only refactoring operations. Tangled changes make the detection of refactorings more challenging, thus resulting in more false positives. Finally, Table 2 shows the number of distinct commits (Com.) and projects (Proj.) containing at least one true positive refactoring (539 out of 729 commits and 185 out of 285 projects with detected refactorings contain at least one true positive refactoring).

4. WHY DO DEVELOPERS REFACTOR?

In this section, we present the results for the first question answered by the developers, regarding the reasons behind the application of the refactorings we detected. Based on the results of the thematic analysis process (Section 3.3), we compile a catalogue of 44 distinct motivations. We dedicate Section 4.1 to discuss EXTRACT METHOD, which is the most frequently occurring refactoring operation in our study, and also the one with the most observed motivations (11). Section 4.2 presents the motivations for the remaining refactorings.

4.1 Motivations for Extract Method

Table 3 describes 11 motivations for EXTRACT METHOD refactoring and the number of occurrences for each of them. The most frequent motivation is to extract a reusable method (43 instances). In this case, the refactoring is motivated by

Table 3: EXTRACT METHOD **motivations**

Theme	Description	Occurrences
Extract reusable method	Extract a piece of reusable code from a single place and call the extracted method in multiple places.	43
Introduce alternative method signature	Introduce an alternative signature for an existing method (e.g., with additional or different parameters) and make the original method delegate to the extracted one.	25
Decompose method to improve readability	Extract a piece of code having a distinct functionality into a separate method to make the original method easier to understand.	21
Facilitate extension	Extract a piece of code in a new method to facilitate the implementation of a feature or bug fix, by adding extra code either in the extracted method, or in the original method.	15
Remove duplication	Extract a piece of duplicated code from multiple places, and replace the duplicated code instances with calls to the extracted method.	14
Replace Method preserving backward compatibility	Introduce a new method that replaces an existing one to improve its name or remove unused parameters. The original method is preserved for backward compatibility, it is marked as deprecated, and delegates to the extracted one.	6
Improve testability	Extract a piece of code in a separate method to enable its unit testing in isolation from the rest of the original method.	6
Enable overriding	Extract a piece of code in a separate method to enable subclasses override the extracted behavior with more specialized behavior.	4
Enable recursion	Extract a piece of code to make it a recursive method.	2
Introduce factory method	Extract a constructor call (class instance creation) into a separate method.	1
Introduce async operation	Extract a piece of code in a separate method to make it execute in a thread.	1

the immediate reuse of a piece of code in multiple other places, in addition to the place from which it was originally extracted. We often observe a concern among developers to reuse code wherever possible, by extracting pieces of reusable code. This is illustrated by the following comments:

"These refactorings were made because of code reusability. I needed to use the same code in new method. I always try to reuse code, because when there's a lot of code redundancy it gets overwhelmingly more complicated to work with the code in future, because when something change in code that has it's duplicate somewhere, it usually needs to be changed also there."

"The reason for me to do the refactoring was: Don't repeat yourself (DRY)."

The second most frequent motivation is to introduce an alternative method signature for an existing method (25 instances), e.g., with extra parameters. To achieve that, the body of the existing method is extracted to a new one with an updated signature and additional logic to handle the extended variability. The original method is changed to delegate to the new one, passing some default values for the new parameters. The following comment illustrates this case:

"The extracted method `values(names List<String>, values List<Object>)` could be of help for some users using Lists instead of arrays, and because the implementation already transformed the provided arrays into Lists internally."

Decomposing a method for improving readability (21 instances) is the third most frequent motivation. Typically, this corresponds to a *Long Method* code smell [10], as illustrated in this comment:

"The method was so long that it didn't fit onto the screen anymore, so I moved out parts."

The next two motivations are to facilitate extension (15 instances) and to remove duplication (14 instances). In the first case, a method is decomposed to facilitate the implementation of a new feature or the fix of a bug by adding

code either in the extracted or in the original method, as illustrated in this comment:

"I was fixing an exception, in order to do that I had to add the same code to 2 different places. So I extracted initial code, replace duplicate with the extracted method and add the 'fix' to the extracted method."

In the second case (i.e., remove duplication), a piece of duplicated code is extracted from multiple places into a single method, as illustrated in the following comments:

"I refactored shared functionality into a single method."

"I checked how other test methods create testing User objects and noticed that it takes two lines of code that were repeated all over the test class. So I abstracted these two lines of code into a method for better readability and then reused the method in all the places that had the same code."

Finally, two other important motivations are to improve testability (6 instances) and to replace a method by preserving backward compatibility (6 instances). In the first case, the decomposition enables the developer to test parts of the code in isolation, as illustrated in this comment:

"I wanted to test the part of `authenticate()` which verifies that a member is element of a set, and that would have been more complex using `authenticate` directly."

In the second case, the goal is to introduce a method having the same functionality with an already existing one, but a different signature (e.g., improved name, or removed unused parameter), and at the same time preserve the public API by making the original method delegate to the new one. This motivation is best illustrated in the following comment:

"I did that refactoring because essentially I wanted to rename the functions involved - you'll see the old functions just forward straight to the new ones. But I didn't just rename because other code in other projects might be referring to the old functions, so they would need to still be present (I guess they should have been marked as @deprecated then, but I was a bit lazy here)."

Table 4: Motivations for Move Class, Attribute, Method (MC, MA, MM), Rename Package (RP) Inline Method (IM), Extract Superclass, Interface (ES, EI), Pull Up Method, Attribute (PUM, PUA), Push Down Attribute, Method (PDA, PDM)

Type	Theme	Description	Occurrences
MC	Move class to appropriate container	Move a class to a package that is more functionally or conceptually relevant.	13
MC	Introduce sub-package	Move a group of related classes to a new subpackage.	7
MC	Convert to top-level container	Convert an inner class to a top-level class to broaden its scope.	4
MC	Remove inner classes from deprecated container	Move an inner class out of a class that is marked deprecated or is being removed.	3
MC	Remove from public API	Move a class from a package that contains external API to an internal package, avoiding its unnecessary public exposure.	2
MC	Convert to inner class	Convert a top-level class to an inner class to narrow its scope.	2
MC	Eliminate dependencies	Move a class to another package to eliminate undesired dependencies between modules.	1
MC	Eliminate redundant sub-package	Eliminate a redundant nesting level in the package structure.	1
MC	Backward compatibility	Move a class back to its original package to maintain backward compatibility.	1
MA	Move attribute to appropriate class	Move an attribute to a class that is more functionally or conceptually relevant.	15
MA	Remove duplication	Move similar attributes to another class where a single copy of them can be shared, eliminating the duplication.	4
RP	Improve package name	Rename a package to better represent its purpose.	8
RP	Enforce naming consistency	Rename a package to conform to project's naming conventions.	3
RP	Move package to appropriate container	Move a package to a parent package that is more functionally or conceptually relevant.	2
MM	Move method to appropriate class	Move a method to a class that is more functionally or conceptually relevant.	8
MM	Move method to enable reuse	Move a method to a class that permits its reuse by other classes.	3
MM	Eliminate dependencies	Move a method to eliminate dependencies between classes.	3
MM	Remove duplication	Move similar methods to another class where a single copy of them can be shared, eliminating duplication.	1
MM	Enable overriding	Move a method to permit subclasses to override it.	1
IM	Eliminate unnecessary method	Inline and eliminate a method that is unnecessary or has become too trivial after code changes.	13
IM	Caller becomes trivial	Inline and eliminate a method because its caller method has become too trivial after code changes, so that it can absorb the logic of the inlined method without compromising readability.	2
IM	Improve readability	Inline a method because it is easier to understand the code without the method invocation.	1
ES	Extract common state/behavior	Introduce a new superclass that contains common state or behavior from its subclasses.	7
ES	Eliminate dependencies	Introduce a new superclass that is decoupled from specific dependencies of a subclass.	1
ES	Decompose class	Extract a superclass from a class that holds many responsibilities.	1
PUM	Move up common methods	Move common methods to superclass.	8
PUA	Move up common attributes	Move common attributes to superclass.	7
EI	Facilitate extension	Introduce an interface to enable different behavior.	1
EI	Enable dependency injection	Introduce an interface to facilitate the use of a dependency injection framework.	1
EI	Eliminate dependencies	Introduce an interface to avoid depending on an existing class/interface.	1
PDA	Specialized implementation	Push down an attribute to allow specialization by subclasses.	2
PDA	Eliminate dependencies	Push down attribute to subclass so that the superclass does not depend on a specific type.	1
PDM	Specialized implementation	Push down a method to allow specialization by subclasses.	1

4.2 Motivations for Other Refactorings

Table 4 presents the motivations for the remaining refactorings studied in the paper. We found nine different motivations for MOVE CLASS. The two most frequent motivations are to move a class to a package that is more functionally or conceptually related to the purpose of the class (13 instances), and to introduce a sub-package (7 instances). The first one is illustrated by the following comment:

"This refactoring was done because common interface for those classes lived in `org.neo4j.kernel.impl.store.record`, while most of it's implementors lived in `org.neo4j.kernel.impl.store` which did not make sense because all of them are actually records."

For MOVE ATTRIBUTE, the most common motivation is also to move the attribute to an appropriate class that is more functionally or conceptually relevant (15 instances), as in the example below:

"In this case, each of these fields was moved as their relevance changed. As `UserService` already handles the login process, it makes sense that changes to the login process should be encapsulated within `UserService`."

Remove duplication is another motivation for moving an attribute, as illustrated by the following comment:

"The attributes were duplicated, so I moved them to the proper common place."

For RENAME PACKAGE, the most common motivation is to update the name of a package to better represent its purpose (8 instances), as in the example below:

"This was a simple package rename. `test` seems to fit better than `tests` here as a single test can be executed too."

We found three main reasons for a MOVE METHOD refactoring: move a method to an appropriate class (8 instances), move a method to enable reuse (3 instances), and move a method to eliminate dependencies (3 instances). The most frequent motivation for INLINE METHOD is to eliminate an unnecessary or trivial method, as illustrated in the comment:

"Since the method was a one-liner and was used only in one place, inlining it didn't make the code more complex. On the other hand, it allowed to lessen calls to `getVirtualFile()`."

EXTRACT SUPERCLASS is usually applied to introduce a new class with state or behavior that can be shared by subclasses (7 instances). PULL UP METHOD/ATTRIBUTE is performed to move common code to an existing superclass (8 and 7 instances, respectively). EXTRACT INTERFACE and PUSH DOWN ATTRIBUTE/METHOD are less popular refactorings and thus their motivations have at most two instances.

5. REFACTORING AUTOMATION

In this section, we discuss the results drawn from the second question answered by the developers, regarding the use (or not) of automatic refactoring tools provided by their IDEs to apply the refactorings we presented. First, in Section 5.1, we present how many of the interviewed developers applied the refactoring(s) automatically. We also present which refactoring types are more frequently applied with tool support. In Section 5.2, we discuss some insights drawn from developers' answers that explain why refactoring is still applied manually in most of the cases. Last, in Section 5.3, we present additional details regarding which IDE developers most often used for refactoring.

5.1 Are refactoring tools underused?

Table 5 shows the results for this question. 95 developers (55% of valid answers) answered that the refactoring was performed manually without tool support; 66 developers (38%) answered that the refactoring engine of an IDE was used; 13 developers (7%) answered that the refactoring was partially automated. In summary, refactoring is probably more often applied manually than with refactoring tools.

Table 5: Manual vs. automated refactoring

Modification	Occurrences	
Manual	95	▆▆▆
Automated	66	▆▆
Not answered	48	▆
Partially automated	13	▪

We also counted the percentage of automated refactorings by refactoring type, as presented in Table 6. RENAME PACKAGE is the refactoring most often performed with tool support (58%), followed by MOVE CLASS (50%). Three other refactorings are performed automatically in around a quarter of the cases: EXTRACT METHOD (29%), MOVE METHOD (26%), and MOVE ATTRIBUTE (24%). INLINE METHOD follows with 18% of automatic applications. Finally, for the remaining refactorings, we do not have a large number of instances to draw safe conclusions (maximum 9 instances), but there is a consistent trend showing that inheritance-related refactorings are mostly manually applied.

Table 6: Refactoring automation by type

Refactoring Type	Occurrences		Automated %	
EXTRACT METHOD	118	▆▆▆	29%	▪
MOVE CLASS	36	▆	50%	▆▆
MOVE ATTRIBUTE	21	▪	24%	▪
MOVE METHOD	19	▪	26%	▪
INLINE METHOD	17	▪	18%	▪
RENAME PACKAGE	12	▪	58%	▆▆
EXTRACT SUPERCLASS	9	▪	11%	▪
PULL UP METHOD	9	▪	11%	▪
PULL UP ATTRIBUTE	7	▪	14%	▪
EXTRACT INTERFACE	3	▏	0%	
PUSH DOWN ATTRIBUTE	3	▏	33%	▆
PUSH DOWN METHOD	2	▏	0%	

5.2 Why do developers refactor manually?

29 developers explained in their answers why they did not use a refactoring tool. Table 7 shows five distinct themes we identified in these answers.

Table 7: Reasons for not using refactoring tools

Description	Occurrences	
The developer does not trust automated support for complex refactorings.	10	▆▆▆
Automated refactoring is unnecessary, because the refactoring is trivial and can be manually applied.	8	▆▆
The required modification is not supported by the IDE.	6	▆▆
The developer is not familiar with the refactoring capabilities of his/her IDE.	3	▪
The developer did not realize at the moment of the refactoring that he/she could have used refactoring tools.	2	▪

Lack of trust (10 instances) was the most frequent reason. Some developers do not trust refactoring tools for complex operations that involve code manipulation and only use them for renaming or moving:

"I don't trust the IDE for things like this, and usually lose other comments, notation, spacing from adjacent areas."

"I'd say developers are reluctant to let a tool perform anything but trivial refactorings, such as the ones you picked up on my commit."

On the other hand, some developers also think that tool support is unnecessary in simple cases (8 instances). Sometimes the operation may involve only local changes and is trivial to do by hand. Thus, calling a special operation to do it is considered unnecessary, as illustrated by this comment:

"Automated refactoring is overkill for moving some private fields."

Additionally, developers also mentioned: lack of tool support for the specific refactoring they were doing (6 instances), not being familiar with refactoring features of the IDE (3 instances), and not realizing they could use refactoring tools at the moment of the refactoring (2 instances).

5.3 What IDEs developers use for refactoring?

When answering to our emails, 83 developers spontaneously mentioned which IDE they use. Therefore, we decided to investigate these answers, specially because our study is not dependent on any IDE, and thus differs from previous studies which are usually based only on Eclipse data [24, 25]. Table 8 shows the most common IDEs mentioned in these answers and the percentage of refactorings performed automatically in these cases. 139 developers (63%) did not explicitly mention an IDE when answering this question. Considering the answers citing an IDE, IntelliJ IDEA is the most popular one. It also has the highest ratio of refactorings performed automatically (71%). Since 11 JetBrains/-intellij-community (and related plug-ins) developers answered to our questions, we also investigated the answers separately in two groups, namely answers from IntelliJ IDEA developers and from IntelliJ IDEA users. We observed that the ratio of automated refactorings in both groups is very similar (73% vs. 70%). Therefore, the responses from these 11 IntelliJ IDEA developers do not bias the percentage of automated refactoring reported for IntelliJ IDEA.

Table 8: IDE popularity

IDE	Occurrences		Automated %	
Editor not mentioned	139	▮▮▮	12%	▮
IntelliJ IDEA	51	▮	71%	▮▮▮
Eclipse	18	▮	44%	▮▮
NetBeans	8	▏	50%	▮▮
Android Studio	4	▏	25%	▮
Text Editor	2	▏	0%	

6. DISCUSSION

In this section, we discuss the main findings of our study.

Refactoring Motivations: Our study confirms that Extract Method is the "Swiss army knife of refactorings" [37]. It is the refactoring with the most motivations (11 in total). In comparison to [37], there is an overlap in the reported motivation themes for Extract Method. We found some new themes, such as *improve testability* and *enable recursion*, but

we did not find any instances of the themes *encapsulate field* and *hide message chain*, reported in [37], which are related to code smell resolution. We assume these different themes are due to the nature of the examined projects, since [37] examined only three libraries and frameworks, while in this study we examined 124 projects from various domains including standalone applications. By comparing to the code symptoms that initiate refactoring reported in the study by Kim et al. [17], we found the *readability*, *reuse*, *testability*, *duplication*, and *dependency* motivation themes in common.

Most of the refactoring motivations we found have the intention to facilitate or even enable the completion of the maintenance task that the developer is working on. For instance, *extract reusable method*, *introduce alternative method signature*, and *facilitate extension* are among the most frequent motivations, and all of them involve enhancing the functionality of the system. Therefore, Extract Method is a key operation to complete other maintenance tasks, such as adding a feature or fixing a bug. In contrast, only two out of the 11 motivations we found (*decompose method to improve readability* and *remove duplication*) are targeting code smells. This finding could motivate researchers and tool builders to design refactoring recommendation systems [36, 31, 34, 12, 18, 38] that do not focus solely on detecting refactoring opportunities for the sake of code smell resolution, but can support other refactoring motivations as well.

We also observe that developers are seriously concerned about avoiding code duplication, when working on a given maintenance task. They often use refactorings—especially Extract Method—to achieve this goal, as illustrated by the following comments:

"I need to add a check to both the then- and the else-part of an if-statement. This resulted in more duplicated code than I was comfortable with."

"There was already code duplication, but the bug fix required another cut-and-paste, which made it code triplication. That was above my pain level so I decided to group the replicated code out into `bail()`."

The other refactorings we analyzed are typically performed to improve the system design. For example, the most common motivation for Move Class, Move Attribute, and Move Method is to reorganize code elements, so that they have a stronger functional or conceptual relevance.

Automated vs. Manual Refactoring: In a field study with Eclipse users, Negara et al. [25] report that most refactorings (52%) are manually performed. In our study, involving developers using a wider variety of IDEs, we found that 55% of refactorings are manually performed. However, we also found that IntelliJ IDEA users tend to use more the refactoring tool support than other IDE users. Moreover, the results for automated Extract Method refactorings are very similar in both studies: 28% in our study, against 30% in their study. While the total percentages of manually performed refactorings are very similar, we should keep in mind that Negara et al. counted simple refactorings, like renamings, which are more often applied with tool support. Compared to the study by Murphy-Hill et al. [24], where they report that 89% of refactorings are performed manually (considering also renamings), we detected significantly more automated refactorings. We suspect this difference may be due to two reasons. First, automated refactoring tools may have become more popular and reliable over the last years.

Second, our study involves developers using a broader range of IDEs, which may also influence how developers use refactoring tool support.

Regarding the reasons for not using automated refactoring, our results are in line with the three main factors found in the study by Murphy-Hill et al. [24]: *awareness*, *opportunity*, and *trust*. The exception is the argument that tool support is unnecessary in simple cases, which is not closely related to any of the three aforementioned factors. However, the same argument can be observed in the study by Kim et al. [17], in which some developers mention that they do not feel a great need for automated refactoring tools.

Refactoring Popularity: In this study we detected refactorings in 285 of the monitored repositories in a time window of 61 days. Given that only 471 out of the 748 monitored repositories were active during that period, we found refactoring activity in 60.5% of the repositories with at least one commit. This shows that refactoring is a common practice, especially considering that frequent refactorings such as RENAME CLASS/METHOD/FIELD were not considered.

The top-5 most popular refactorings detected in our study are EXTRACT METHOD, MOVE CLASS, MOVE ATTRIBUTE, RENAME PACKAGE, and MOVE METHOD. MOVE METHOD is the third most popular refactoring in the study by Negara et al. [25]. The top-2 refactorings in this study (RENAME LOCAL VARIABLE and EXTRACT LOCAL VARIABLE) are low-level refactorings, which have not been considered in our study. We focused on high-level refactorings, because they can be motivated by multiple factors.

Using a sample of 40 commits with manual and automated refactorings, Murphy-Hill et al. [24] report that the two most popular refactorings are RENAME CONSTANT and PUSH DOWN. However, PUSH DOWN refactorings are among the least popular ones in our study. This difference may be related to the number of commits analyzed in the studies (40 vs. 539 commits in our study), and the specialized nature of the software (i.e., the Eclipse IDE) examined in [24].

7. THREATS TO VALIDITY

External Validity: This study is restricted to open source, Java-based, GitHub-hosted projects. Thus, we cannot claim that our findings apply to industrial systems, or to systems implemented in other programming languages. However, we collected responses from 222 developers contributing in 124 different projects, which is one of the largest samples of systems used in refactoring studies.

Internal Validity: First, we use in the study a tool that detects refactorings by comparing two revisions of the code. We evaluated the recall of this tool using a sample of 120 documented refactoring operations. We achieved a recall of 0.93. However, we cannot guarantee a similar recall in the studied GitHub projects, because some commits might contain tangled changes making more difficult to isolate (or untangle [7]) the changes related to refactorings. In addition, it is known that this kind of detection approach may miss refactorings that do not reach the version control system (e.g., sequences of overlapping refactorings applied to the same piece of code). We claim this threat should be tolerated in large scale studies, where we cannot assume that the developers would be willing to install an external monitoring tool in their IDEs [25]. Furthermore, as we showed in this study, developers nowadays use IDEs from multiple vendors.

In order to cover as many IDEs as possible and strengthen the external validity, a study based on monitoring would require to develop a separate version of this tool for each IDE. Second, we cannot claim that the catalogue of motivations we propose is exhaustive. Notably, we have a limited number of motivation themes for less frequent refactoring types, such as PUSH DOWN METHOD/ATTRIBUTE and EXTRACT INTERFACE. Third, to mitigate inconsistencies in the proposed themes, we rely on an initial classification performed independently by two authors of the paper, followed by a consensus building process. We also make publicly available the responses collected from the developers and the proposed refactoring motivation themes to provide a means for replication and verification.

8. CONCLUSIONS

In summary, the main conclusions and lessons learned are:

1. Refactoring activity is mainly driven by changes in the requirements (i.e., new feature and bug fix requests) and much less by code smell resolution. Only 2 out of the 11 motivations for EXTRACT METHOD were related to code smell resolution (*remove duplication*, *decompose method*) covering only 25% (35/138) of the motivation instances.

2. EXTRACT METHOD is a key operation that serves multiple purposes, specially those related to code reuse and functionality extension. It is also used to improve the testability of code, and deprecate public API methods.

3. The elimination of dependencies is the most common motivation among the *move/abstract* related refactorings.

4. Manual refactoring is still prevalent (55% of the developers refactored manually the code). In particular, inheritance related refactoring tool support seems to be the most under-used (only 10% done automatically), while MOVE CLASS and RENAME PACKAGE are the most trusted refactorings (over 50% done automatically).

5. The IDE plays an important role in the adoption of refactoring tool support. IntelliJ IDEA users perform more automated refactorings (71% done automatically) than Eclipse users (44%) and Netbeans users (50%).

6. Compared to the study by Murphy-Hill et al. [24], it seems that developers apply more automated refactorings nowadays. Our findings confirm Negara et al. [25] who collected data only from Eclipse IDE users, but our study covers developers using more IDEs.

Based on our findings, we propose that future research on refactoring recommendation systems should refocus from code-smell-oriented to maintenance-task-oriented solutions. This could be achieved by leveraging the recent advancements in feature location [8] and requirements tracing to automatically locate the code associated with a feature or bug fix request, or a requirement change, and recommend suitable refactorings that will make easier the completion of the maintenance task. We strongly believe this will boost the adoption of recommendation systems by the developers.

9. REFERENCES

[1] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Recommending refactoring operations in large software systems. In M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, editors, *Recommendation Systems in Software Engineering*, pages 387–419. Springer Berlin Heidelberg, 2014.

[2] G. Bavota, A. De Lucia, and R. Oliveto. Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. *J. Syst. Softw.*, 84(3):397–414, Mar. 2011.

[3] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. Methodbook: Recommending Move Method refactorings via relational topic models. *IEEE Trans. Softw. Eng.*, 40(7):671–694, July 2014.

[4] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.

[5] O. Chaparro, G. Bavota, A. Marcus, and M. Di Penta. On the impact of refactoring operations on code quality metrics. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 456–460, 2014.

[6] D. S. Cruzes and T. Dyba. Recommended steps for thematic synthesis in software engineering. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, pages 275–284, 2011.

[7] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. *CoRR*, abs/1502.06757, 2015.

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[9] B. Du Bois, S. Demeyer, and J. Verelst. Does the "refactor to understand" reverse engineering pattern improve program comprehension? In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 334–343, 2005.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

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

[12] K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting Form Template Method refactoring opportunities with program dependence graph. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pages 53–62, 2012.

[13] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 351–360, 2011.

[14] M. Kim, D. Cai, and 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*, pages 151–160, 2011.

[15] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-Finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 371–372, 2010.

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

[17] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Trans. Softw. Eng.*, 40(7), July 2014.

[18] N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering*, May 2015.

[19] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[20] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, July 2006.

[21] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: Observations and tools for Extract Method. In *Proceedings of the 30th International Conference on Software Engineering*, pages 421–430, 2008.

[22] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering*, 41(1):65–81, Jan 2015.

[23] E. R. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, 2008.

[24] E. R. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Trans. Softw. Eng.*, 38(1):5–18, 2012.

[25] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP)*, pages 552–576, 2013.

[26] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, pages 803–813, 2014.

[27] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[28] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

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

[30] V. Sales, R. Terra, L. Miranda, and M. Valente. Recommending Move Method refactorings using

dependency sets. In *Proceedings of the 20th Working Conference on Reverse Engineering*, pages 232–241, 2013.

[31] D. Silva, R. Terra, and M. T. Valente. Recommending automated Extract Method refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 146–156, 2014.

[32] J. Singer, S. E. Sim, and T. C. Lethbridge. *Guide to Advanced Empirical Software Engineering*, chapter Software Engineering Data Collection for Field Studies, pages 9–34. Springer London, London, 2008.

[33] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *J. Syst. Softw.*, 86(4):1006–1022, Apr. 2013.

[34] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Inf. Softw. Technol.*, 54(12):1297–1307, Dec. 2012.

[35] N. Tsantalis and A. Chatzigeorgiou. Identification of Move Method refactoring opportunities. *IEEE Trans. Softw. Eng.*, 35(3):347–367, May 2009.

[36] N. Tsantalis and A. Chatzigeorgiou. Identification of Extract Method refactoring opportunities for the decomposition of methods. *J. Syst. Softw.*, 84(10):1757–1782, Oct. 2011.

[37] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 132–146, 2013.

[38] N. Tsantalis, D. Mazinanian, and G. P. Krishnan. Assessing the refactorability of software clones. *IEEE Trans. Softw. Eng.*, 41(11):1055–1090, Nov 2015.

[39] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, pages 233–243, 2012.

[40] Y. Wang. What motivate software engineers to refactor source code? evidences from professional developers. In *IEEE International Conference on Software Maintenance*, pages 413–416, Sept 2009.

[41] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 112–118, 2006.

[42] Z. Xing and E. 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.

[43] Z. Xing and E. Stroulia. The JDEvAn tool suite in support of object-oriented evolutionary development. In *Companion of the 30th International Conference on Software Engineering*, pages 951–952, 2008.