Refactorings Recommendation via Commit Message Analysis

Soumaya Rebai[a], Marouane Kessentini[a], Vahid Alizadeh[a], Oussama Ben Sghaier[a], Rick Kazman[b]

[a] *University of Michigan, Dearborn, Michigan, USA*
[b] *University of Hawaii*

Abstract

Context: The purpose of software restructuring, or refactoring, is to improve software quality and developer productivity. **Objective:** Prior studies have relied mainly on static and dynamic analysis of code to detect and recommend refactoring opportunities, such as code smells. Once identified, these smells are fixed by applying refactorings which then improve a set of quality metrics. While this approach has value and has shown promising results, many detected refactoring opportunities may not be related to a developer's current context and intention. Recent studies have shown that while developers document their refactoring intentions, they may miss relevant refactorings aligned with their rationale. **Method:** In this paper, we first identify refactoring opportunities by analyzing developer commit messages and check the quality improvements in the changed files, then we distill this knowledge into usable context-driven refactoring recommendations to complement static and dynamic analysis of code. **Results:** The evaluation of our approach, based on six open source projects, shows that we outperform prior studies that apply refactorings based on static and dynamic analysis of code alone. **Conclusion:** This study provides compelling evidence of the value of using the information contained in existing commit messages to recommend future refactorings.

Email addresses: `srebal@umich.edu` (Soumaya Rebai), `marouane@umich.edu` (Marouane Kessentini), `alizadeh.umich.edu` (Vahid Alizadeh), `oussama@umich.edu` (Oussama Ben Sghaier), `kazman@hawaii.edu` (Rick Kazman)

Keywords:

Commit Message, Refactoring Recommendation, Quality attributes.

1. Introduction

Software restructuring or refactoring [1] is critical to improve software quality and developer's productivity, but it can be complex, expensive, and risky. As projects evolve, developers in a rush to deliver new features frequently postpone necessary refactorings until a crisis occurs [2]. By that time it often results in degraded performance, an inability to support new features, or even a failed system and significant losses [**?** 3, 4]. Thus, several studies have been proposed to (semi-) automate the recommendation of refactorings to help developers improving the quality of their systems in a more timely fashion [5, 6, 7, 8, 9, 10, 11, 12, 13, 14].

While code-level refactoring is widely studied and well supported by tools [15, 16, 13, 17, 18], it remains a human activity which is hard to fully automate and requires developer insights. Such insights are important because developers understand their problem domain intuitively and may have a clear target end-state in mind for their system. A majority of existing tools and approaches rely on the use of quality metrics such as coupling, cohesion, and the QMOOD quality attributes [19] to first identify refactoring opportunities, and then to recommend refactorings to fix them. Many of the quality issues detected using structural metrics are known as code smells or antipatterns [20]. However, recent studies have shown that developers are not primarily interested in fixing antipatterns when they are performing refactoring [8].

In a recent survey of Alizadeh et al. [17, 21] with several software companies, 84% of interviewees confirmed that most of the automated refactoring tools recommend hundreds of code-level quality issues and refactorings, but these tools fail to adequately explain how these refactorings are relevant to a developer who is combining refactorings with other tasks such as fixing bugs and enhancing features. This observation is consistent with other studies [22, 23, 24] showing that refactorings rarely happen in isolation. Without a rigorous understanding of the rationale for refactoring, recommendation tools may continue to suffer from a high false-positive rate and limited relevance to developers [25, 26, 27]. However, if a refactoring rationale can be automatically identified, this can guide refactoring recommendations to be more relevant and less ad hoc. Recent empirical studies show that while developers document their refactoring intention, they may miss relevant refactorings aligned with their rationale [25, 24]. One of the main reasons is that manual refactoring is a tedious and time-consuming task which also explains the tendency of the developers to perform the minimum possible number of refactorings [17, 28]. Thus, it is critical to provide developers a semi-automated refactorings support that can understand their rationale and translate it into actionable refactorings recommendation.

In this paper, we start from the observation that a majority of inconsistencies between documented and applied refactorings were due to poor refactoring decisions taken manually by developers [25, 24]. Therefore, we think that there is a need for linking documentation to refactoring recommendations as well as a need for an automated system that can not only check the consistency of the developer-created descriptions of refactoring but also recommend further refactoring to meet their rationale. However, none of the existing studies have used this knowledge to guide the process of refactoring recommendation. Thus, we propose a novel approach, called **RefCom**, to capitalize on this previously unused resource.

RefCom includes the following steps. First, we filtered a large corpus of commit messages to extract the ones containing quality issues or refactorings based on a list of 87 keywords which are already defined in the literature [29, 30, 28]. We also used an existing tool, RefactoringMiner [31], to detect the refactorings applied in commits to confirm or extend the ones detected using our set of keywords. Second, we automatically identified the changed files in these selected commits and detected the impacted code fragments. Third, we checked the quality improvements in these files to detect the quality attributes that developers aimed to improve. Finally, we recommended *more* refactorings to developers based on the rationale extracted from the commits: the locations of the intended refactorings and the quality attributes to be improved. Furthermore, our tool will generate warnings to developers if their commit messages are not matching the manually applied refactorings.

Our ultimate goal is to recommend a set of refactoring solutions that enhance the improvements described in the commit messages or provide developers better ways to refactor their code based on the rationale found in the commits. RefCom identifies potential inconsistencies between developer intentions and actual applied refactorings and recommends an additional set of refactorings that better meet developer intentions and expectations. In fact, the paper validated the first hypothesis that commit messages document refactorings applied by developers including their intention by answering the following research question:

RQ1: *To what extent are refactorings documented in commit messages?*

The second hypothesis validated in this paper is the inconsistencies (or incomplete refactorings) between documented and applied refactorings in terms of expected impact/intention via answering the following research question:

RQ2: *To what extent do developers accurately document their refactoring and its rationale?*

These observed inconsistencies/gaps (RQ2) along with the fact that refactoring documentation is available at the commit level (RQ1) are the main motivations to refine existing refactoring recommendation tools. Thus, we selected our previous multi-objective refactoring recommendation tool [32] as a case study for this purpose while answering our following third research question:

RQ3: *To what extent can our approach recommend relevant refactorings based on commit analysis compared to existing refactoring techniques?*

However, it is possible to expand the outcomes of RQ1 and RQ2 to build better refactoring recommendation tools in general. To summarize, our contributions are not limited to recommending refactorings solutions using a straightforward multi-objective technique. We believe that RQ1 and RQ2 can advance the knowledge within the refactoring community. For the first two contributions RefCom uses NLP and static and dynamic analysis to detect developers' intentions, the actual refactorings and the quality attributes improvement. For the third contribution, we used a multi-objective algorithm to recommend refactoring solutions to enhance the applied refactorings (after extracting developer's intention) or fix the detected inconsistencies. We validated our approach on six open source projects containing a large number of commits. Our validation shows that RefCom outperforms both the actual refactorings applied by developers in their commits and existing refactoring tools based on antipatterns and static and dynamic analysis [32, 33]. Thus, the use of the knowledge extracted from commit messages is critical to better understand developer preferences.

The primary contributions of this paper can be summarized as follows:

1. The paper introduces, for the first time, an approach, **RefCom**, based on commit messages to recommend refactorings. Thus, the recommendations are based on understanding the developers' intention to refactor the code from the commit messages rather than fixing antipatterns and improving the majority of quality metrics.

2. The proposed technique can either: (a) enhance some of the previously refactored files in the commits by providing better alternatives after extracting the refactoring rationale; or (b) recommend refactorings to address the quality issues mentioned in the commit messages when we did not find an actual improvement when checked the files before and after the commit.

3. The paper reports the results of an empirical study on the implementation of our approach. The obtained manual evaluation results provide evidence to support the claim that our proposal is more efficient, on average, than existing refactoring techniques based on a benchmark of 6 open source systems in terms of the relevance of recommended refactorings especially for the case of incremental refactorings.

The remainder of this paper is structured as follows. Section 2 presents the relevant background details. Section 3 describes our approach while the results obtained from our experiments are presented and discussed in Section 4. Threats to validity are discussed in Section 5. Section 6 provides an account of related work. Finally, in Section 7, we summarize our conclusions and present some ideas for future work.

2. Problem Statement

2.1. Background

Quality attributes. QMOOD is a widely used quality model, based on the ISO 9126 product quality model [34]. We selected this model because it is a widely accepted quality model in industry and it has been validated based on hundreds of industrial projects[34, 35, 36, 37, 17]. Each quality attribute in QMOOD is defined using a combination of low-level metrics as detailed in Tables 1 and 2. The QMOOD model has been used in many studies [19, 38, 39] to estimate the effects of proposed refactoring solutions on software quality. QMOOD defines six high-level design quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness) that can be calculated using 11 lower-level design metrics.

Commits and refactoring. Refactoring documentation has two major parts: pull requests for "high-level" refactorings [40] and commit messages for code-level refactorings. The individual commit messages describe refactorings applied by a developer. A refactoring process typically starts with a new branch. In this branch, each commit should correspond to a code-level refactoring. After developers commit all the code-level refactorings (i.e., finish the refactoring process), developers make a pull request in which they write a description of the overall refactoring. If the refactorings are accepted, the branch is merged into the master branch.

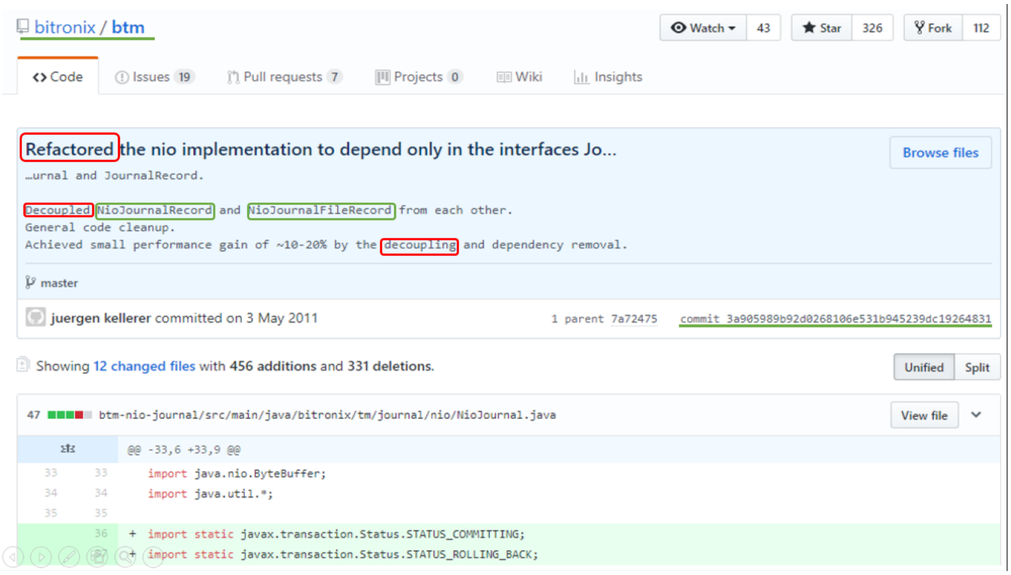
Figure 1 shows an example of a commit extracted from an open source project. The refactorings applied by the developers are summarized in Figure

Table 1: QMOOD metrics description.

Design Metric	**Design Property**	**Description**
Design Size in Classes (DSC)	Design Size	Total number of classes in the design.
Number Of Hierarchies (NOH)	Hierarchies	Total number of ”root” classes in the design (*count(MaxInheritenceTree (class)=0)*)
Average Number of Ancestors (ANA)	Abstraction	Average number of classes in the inheritance tree for each class.
Direct Access Metric (DAM)	Encapsulation	Ratio of the number of private and protected attributes to the total number of attributes in a class.
Direct Class Coupling (DCC)	Coupling	Number of other classes a class relates to, either through a shared attribute or a parameter in a method.
Cohesion Among Methods of class ($CAMC$)	Cohesion	Measure of how related methods are in a class in terms of used parameters. It can also be computed by: *1 − LackOfCohesionOfMethods()*
Measure Of Aggregation (MOA)	Composition	Count of number of attributes whose type is user defined class(es).
Measure of Functional Abstraction (MFA)	Inheritance	Ratio of the number of inherited methods per the total number of methods within a class.
Number of Polymorphic Methods (NOP)	Polymorphism	Any method that can be used by a class and its descendants. Counts of the number of methods in a class excluding private, static and final ones.
Class Interface Size (CIS)	Messaging	Number of public methods in class.
Number of Methods (NOM)	Complexity	Number of methods declared in a class.

Table 2: Quality attributes and their equations.

Quality attributes	**Definition**
	Computation
Reusability	A design with low coupling and high cohesion is easily reused by other designs.
	$0.25 * \mathit{Coupling} + 0.25 * \mathit{Cohesion} + 0.5 * \mathit{Messaging} + 0.5 * \mathit{DesignSize}$
Flexibility	The degree of allowance of changes in the design.
	$0.25 * \mathit{Encapsulation} - 0.25 * \mathit{Coupling} + 0.5 * \mathit{Composition} + 0.5 * \mathit{Polymorphism}$
Understandability	The degree of understanding and the easiness of learning the design implementation details.
	$0.33 * \mathit{Abstraction} + 0.33 * \mathit{Encapsulation} - 0.33 * \mathit{Coupling} + 0.33 * \mathit{Cohesion} - 0.33 * \mathit{Polymorphism} - 0.33 * \mathit{Complexity} - 0.33 * \mathit{DesignSize}$
Functionality	Classes with given functions that are publicly stated in interfaces to be used by others.
	$0.12 * \mathit{Cohesion} + 0.22 * \mathit{Polymorphism} + 0.22 * \mathit{Messaging} + 0.22 * \mathit{DesignSize} + 0.22 * \mathit{Hierarchies}$
Extendibility	Measurement of a design's ability to incorporate new functional requirements.
	$0.5 * \mathit{Abstraction} - 0.5 * \mathit{Coupling} + 0.5 * \mathit{Inheritance} + 0.5 * \mathit{Polymorphism}$
Effectiveness	Design efficiency in fulfilling the required functionality.
	$0.2 * \mathit{Abstraction} + 0.2 * \mathit{Encapsulation} + 0.2 * \mathit{Composition} + 0.2 * \mathit{Inheritance} + 0.2 * \mathit{Polymorphism}$



Refactored the nio implementation to depend only in the interfaces Jo…

```
…urnal and JournalRecord.

Decoupled NioJournalRecord and NioJournalFileRecord from each other.
General code cleanup.
Achieved small performance gain of ~10-20% by the decoupling and dependency removal.
```

juergen kellerer committed on 3 May 2011 1 parent 7a72475 commit 3a905989b92d0268106e531b945239dc19264831

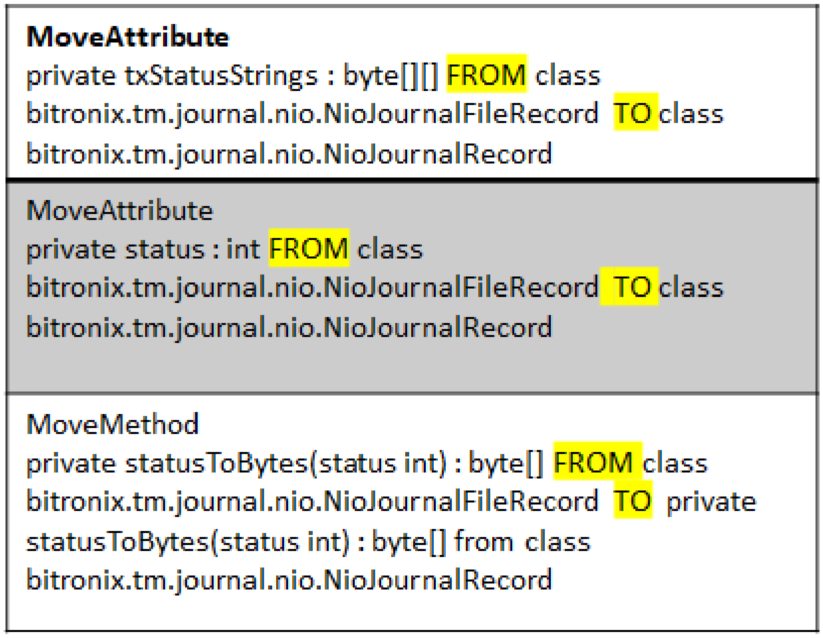
Showing 12 changed files with 456 additions and 331 deletions.

47 btm-nio-journal/src/main/java/bitronix/tm/journal/nio/NioJournal.java

```
@@ -33,6 +33,9 @@
  import java.nio.ByteBuffer;
  import java.util.*;

+ import static javax.transaction.Status.STATUS_COMMITTING;
+ import static javax.transaction.Status.STATUS_ROLLING_BACK;
```

Figure 1: An example commit from the “btm” project.

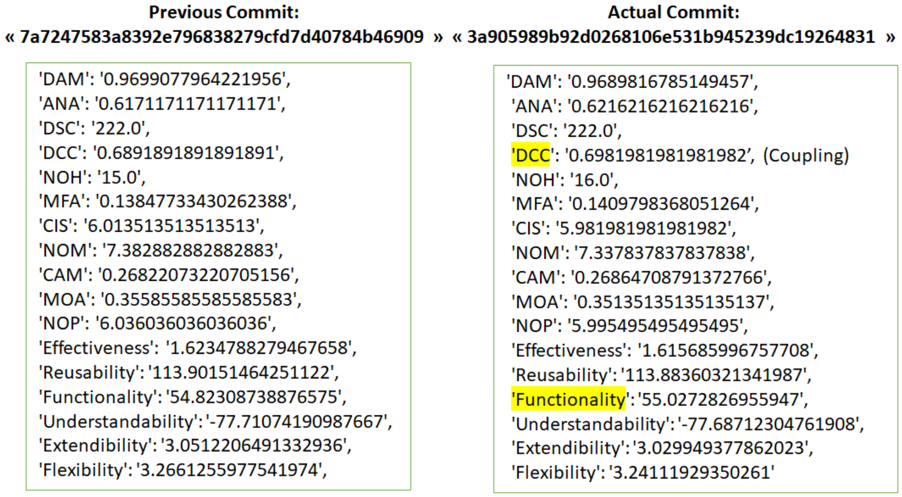


MoveAttribute
private txStatusStrings : byte[][] FROM class bitronix.tm.journal.nio.NioJournalFileRecord TO class bitronix.tm.journal.nio.NioJournalRecord

MoveAttribute
private status : int FROM class bitronix.tm.journal.nio.NioJournalFileRecord TO class bitronix.tm.journal.nio.NioJournalRecord

MoveMethod
private statusToBytes(status int) : byte[] FROM class bitronix.tm.journal.nio.NioJournalFileRecord TO private statusToBytes(status int) : byte[] from class bitronix.tm.journal.nio.NioJournalRecord

Figure 2: The list of refactorings applied in the commit.



Previous Commit:
« 7a7247583a8392e796838279cfd7d40784b46909 »

```
'DAM': '0.9699077964221956',
'ANA': '0.6171171171171171',
'DSC': '222.0',
'DCC': '0.6891891891891891',
'NOH': '15.0',
'MFA': '0.13847733430262388',
'CIS': '6.013513513513513',
'NOM': '7.382882882882883',
'CAM': '0.26822073220705156',
'MOA': '0.35585585585585583',
'NOP': '6.036036036036036',
'Effectiveness': '1.6234788279467658',
'Reusability':'113.90151464251122',
'Functionality':'54.82308738876575',
'Understandability':'-77.71074190987667',
'Extendibility':'3.0512206491332936',
'Flexibility':'3.2661255977541974',
```

Actual Commit:
« 3a905989b92d0268106e531b945239dc19264831 »

```
'DAM': '0.9689816785149457',
 'ANA': '0.6216216216216216',
 'DSC': '222.0',
 'DCC': '0.6981981981981982', (Coupling)
 'NOH': '16.0',
 'MFA': '0.1409798368051264',
 'CIS': '5.981981981981982',
 'NOM': '7.337837837837838',
 'CAM': '0.26864708791372766',
 'MOA': '0.35135135135135137',
 'NOP': '5.995495495495495',
 'Effectiveness': '1.615685996757708',
 'Reusability':'113.88360321341987',
 'Functionality':'55.0272826955947',
 'Understandability':'-77.68712304761908',
 'Extendibility':'3.029949377862023',
 'Flexibility':'3.24111929350261'
```

Figure 3: The quality metric changes in the commit.

2, and the changes in the coupling (DCC) metric can be seen in Figure 3. Of course, refactoring rarely happen in isolation and most of commits and pull-requests contain a sequence of refactorings as described in the example Figure 2 that shows a sequence of three refactorings applied in one commit.

2.2. *Motivation*

The primary motivation for our work emerged from our interactions, as part of an NSF I-Corps project, with 127 professional developers at 38 medium and large-size companies including eBay, Amazon, Google, IBM, and others. The main goal of that study was to identify the challenges associated with current refactoring tools. These are discussed next.

Understanding the refactoring rationale is a key for relevant recommendations. Developers lack knowledge of why they should apply the refactorings recommended by existing tools and are frequently overwhelmed by hundreds of automatically generated antipatterns to fix and quality attributes to improve without any indication of their impact on their current context [41, 42, 32, 18]. While existing refactoring approaches are mainly based on static and dynamic analyses to find refactoring opportunities [33, 43], developers may not have the time and motivation to fix every quality issue. For instance, several developers we interviewed [17, 21, 44] mentioned that they are reluctant to apply refactorings on files that they do not ”own” or that are not related to their current tasks. Without understanding and detecting developer intentions when they choose to refactor their code, refactoring recommendation techniques will continue to be underutilized [45].

Developers describe and document refactoring opportunities in commit messages.

While several empirical studies [46, 47] have shown that over 62% of code reviews discuss maintainability issues to be addressed by refactoring, and only 23% are focused on bug-fixing, most existing work still relies primarily on static and dynamic analyses to identify refactoring opportunities and to explain the need for them. During our survey of industrial partners (for three projects) we found that an average of 38% of quality issues discussed in code reviews and commit messages could *not* be detected using existing traditional static and dynamic analysis tools for code smell detection. As described in Figures 1, 2 and 3, the developer documented their refactoring rationale in terms of improving the coupling that was detected both in the metrics change and the detected refactorings in that commit. Thus, a recommendation refactoring tool can use this information of both the quality attribute to improve and the improved code location (files) to find more refactorings that may fit with the current intention of the developer. But none of the existing studies have used commit message analysis to detect refactoring opportunities or to infer recommendations.

Developers may not manually find the best refactoring strategy meeting their needs. Developers need documentation to comprehend refactoring and understand quality changes for code reviews, and to assess technical debt. We found that 46% of the commits in JHotDraw, Xerces, and three projects of one of our industrial partners, eBay, were related to refactoring, as detected using RefactoringMiner [31]. However, 39% of the documentation of their pull-request descriptions or commit messages was inconsistent with the actual quality changes observed in the systems after refactoring. We found

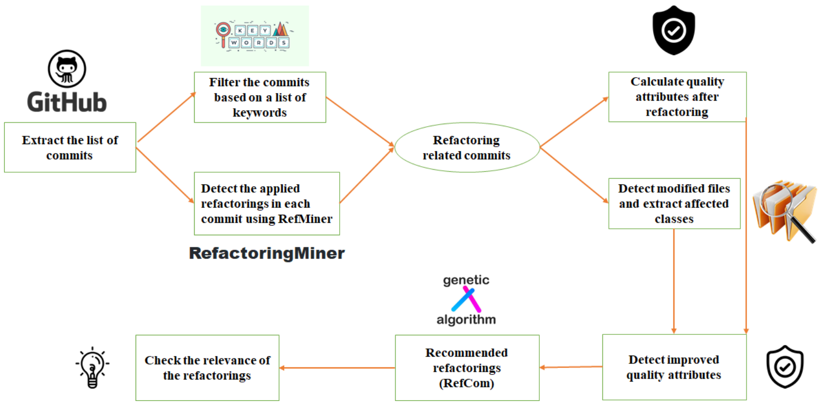

GitHub
Extract the list of commits
KEY WORDS
Filter the commits based on a list of keywords
Detect the applied refactorings in each commit using RefMiner
RefactoringMiner
Refactoring related commits
Calculate quality attributes after refactoring
Detect modified files and extract affected classes
genetic algorithm
Detect improved quality attributes
Recommended refactorings (RefCom)
Check the relevance of the refactorings

Figure 4: Approach Overview: RefCom.

that a majority of the inconsistencies in these projects was attributable to poor refactoring decisions taken manually by developers rather than to wrong documentation. Thus we need to link documentation with refactoring recommendations and we need an automated system that can check the consistency of the developer-created descriptions of refactorings and which can also recommend further refactorings for quality changes.

3. RefCom: Commit-Based Refactoring Recommendations

Operation	Source/entity	Target entity
Move Method	ctrl.booking.BookingController::handleLodgingViewEvent (java.awt.event.ActionEvent):void	ctrl.booking.LodgingModel
Extract Class	ctrl.booking.SelectionModel:: -flightList+ addFlight():void+clearFlight():void	ctrl.booking.FlightList
Move Method	ctrl.booking.BookingController::createBookings():void	ctrl.CoreModel

Table 3: An example of a solution: sequence of refactorings recommended by RefCom

Figure 4 gives an overview of our RefCom approach consisting of three main components: the extraction of refactoring-related commits, the identification of refactoring rationale from commits (where and why developers applied refactorings) and the recommendation of refactorings based on the extracted rationale from the commits to address the identified quality issues and meet the developer's intention. We describe, in the following, these three main components.

Algorithm 1 Commit-based multi-objective refactoring

```
Input
$Sys$: system to evaluate, $Pt$: parent population, $Files$: detected files from the commits analysis, $Quality\ Attributes$: detected quality attributes to improve from the commits analysis
Output
$P_{t+1}$
**Begin**
/* Test if any user interaction occurred in the previous iteration */
$S_t \leftarrow \emptyset, i \leftarrow 1;$
$Q_t \leftarrow Variation(P_t);$
$R_t \leftarrow P_t \cup Q_t;$
$P_t \leftarrow evaluate(P_t, C_t, Sys);$
$(F_1, F_2, ...) \leftarrow NonDominatedSort(R_t);$
**repeat**
    $S_t \leftarrow S_t \cup F_i;$
    $i \leftarrow i + 1$
**until** $(|S_t| \geq N)$
$F_l \leftarrow F_i;$                                          ▷ //Last front to be included
**if** $|S_t| = N$ **then**
    $P_{t+1} \leftarrow S_t;$
**else**
    $P_{t+1} \leftarrow \cup_{j=1}^{l-1} F_j;$
    /*Number of points to be chosen from $F_l$*/
    $K \leftarrow N - |P_{t+1}|;$
    /*Crowding distance of points in Fl */
    $Crowding - Distance - Assignment(F_l);$
    $Quick - Sort(F_l);$
    /*Choose $K$ solutions with largest distance*/
    $P_{t+1} \leftarrow P_{t+1} \cup Select(F_l, k);$
**end if**
**if** $CommitsAnalysis \leftarrow TRUE$ **then**

    /* Select and rank the best front */
    $Filter - Solution(F_1, Files, Quality_Attributes);$
    $Recommend - Solution(Commit)$
**end if**
**End**
```

3.1. Refactoring Related Commit Extraction

The first step of our approach is to filter the set of commits of a project by keeping only those related to refactorings. This filtering step will help constructing a set of commits that are related to refactoring. First, we created a set of 87 keywords via combining different predefined refactoring related keywords from previous work [48, 28, 29]. Thus, the input for the keyword extraction step is the set of commits along with the list of keywords and the output is a filtered set of refactoring related commits. The full list of considered keywords can be found in the website appendix [49]. Second, we used the latest version of RefactoringMiner [31], which supports 38 types of refactoring, to identify the actual refactorings applied between commits by the developers. We selected RefactoringMiner based on the high precision and recall score of over 90%, as reported in their study. Third, we calculated the QMOOD quality attributes of each commit to check whether there were improvements in the quality between commits (which would suggest that a set of refactorings had taken place). Qmood improvement evaluation step takes as an input a set of commits and outputs a filtered set of commits that contain observed actual improvements in at least one of the qmood quality attributes.

The refactoring related commits are the union of the results of the RefactoringMiner detection, keywords extraction and QMOOD improvement evaluation. We decided to unify the data from these sources for the following reasons: (1) RefactoringMiner can help to identify the applied refactorings even if they did not improve quality metrics or they were not documented, (2) the keywords extraction can help to detect commits related to refactorings even there were no refactorings detected by RefactoringMiner or no observed quality improvements (inconsistencies detection), and (3) the QMOOD improvements can help not only in identifying commits related to refactoring even if they were not documented in the commits but also in understanding the impact of the applied refactorings. Additionally, we determined that the combination of the keywords, quality changes, and RefactoringMiner is sufficient to filter the commits since we have also manually inspected some of them as well. In fact, we selected the commits that are identified by only one of the three strategies (RefactoringMinder, QMOOD improvements or keywords). We considered commits that are confirmed by at least two out of these three strategies as having already a very high probability to be related to refactorings. Thus, we inspected manually all the commits that are only detected with exclusively one of the three strategies. The total number of commits in that category are around 23% (319 commits).

RefactoringMiner can detect non-documented refactorings in the commit messages, and the use of the keywords is useful to identify the claims and intentions of developers which may not be translated into actual refactorings. The automated check of quality changes can also help to identify refactoring-related commits and check if the developers actually addressed the quality issues described in the commit messages. To summarize, the documented refactorings are in general the ones that are described in the commit messages and eventually could be detected using the keywords. Furthermore, we are able to detect the refactorings related commits using both RefactoringMinder and the QMOOD improvements. In fact, these refactorings related commits may not be described in the commits message but they are detected because they contained identified refactorings or they improved the quality.

3.2. Identifying Refactoring Rationale from Commits

Identifying refactoring rationale has two parts. The first part is the detection of the files that are refactored by developers in a commit. The second part is the identification of changes in the QMOOD quality attributes then comparing these changes with the information in the commit message.

For the first part, we used the GitHub API to identify the changed files in each commit. In the second part, we compared the QMOOD quality attribute values before and after the commit to capture the actual quality changes for each file. Once the changed files and quality attributes were identified, we checked if the developers *intended* to actually improve these files and quality attributes. In fact, we preprocessed the commit messages and we used the names of code elements in the changed files and the changed quality metrics as keywords to match with words in the commit message. Once the refactoring rationale is automatically detected using this procedure, we continue with the next step to find better refactoring recommendations that can fully meet the developer's intentions and expectations. In case that no quality changes were identified at all then a warning will be generated to developers that the manually applied refactorings are not addressing the quality issues described in his commit message.

3.3. Refactoring Recommendations

After the identification of the refactoring rationale from the history of commits as described in the previous step, we adopted an existing multi-objective algorithm for refactoring [32] to search for relevant refactoring solutions improving both the detected files and changed quality attributes. A refactoring solution, as shown in Table 3, consists of a sequence of n refactoring operations involving one or multiple source code elements of the system to refactor. For every refactoring, pre- and post-conditions are specified to ensure the feasibility of the operation [50]. We selected multi-objective algorithm adaptation due to the conflicting quality attributes that are considered in this study. In fact, our adaptation of multi-objective algorithm takes as objectives the 6 QMOOD quality attributes. Furthermore, multi-objective search has the advantage of generating a diverse set of solutions, thus we can filter the recommendations automatically based on the preferred files and quality attributes of the developer (extracted from the commits as described in the previous step) without the need to run the refactoring recommendation algorithm multiple times. For instance, if the refactoring rationale extracted from commits focused on improving both understandability and reusability in specific Class A and Class B, we execute our multi-objective algorithm using all the 6 quality attributes then we filter the Pareto front based on the two main criteria that are contained in the extracted refactoring rationale. First, we make sure that the selected solution is the one that provides the highest improvement in the quality attributes extracted from the commits during our analysis step (e.g. understandability and reusability).

Second, the optimal solution should also refactor the detected changed files in the commits (e.g. Class A, Class B.

For more details about the multi-objective refactoring algorithm, the reader can refer to [32].

The adopted multi-objective refactoring tool is based on the non-dominated sorting genetic algorithm (NSGA-II) [51] to find a trade-off between the six QMOOD quality attributes. A multi-objective optimization problem can be formulated as follow :

$$\begin{aligned} &Minimize && F(x) = (f_1(x), f_2(x), ..., f_M(x)),\\ &Subject\ to && x \in S,\\ & && S = \{x \in R^m : h(x) = 0, g(x) \geq 0\}; \end{aligned}$$

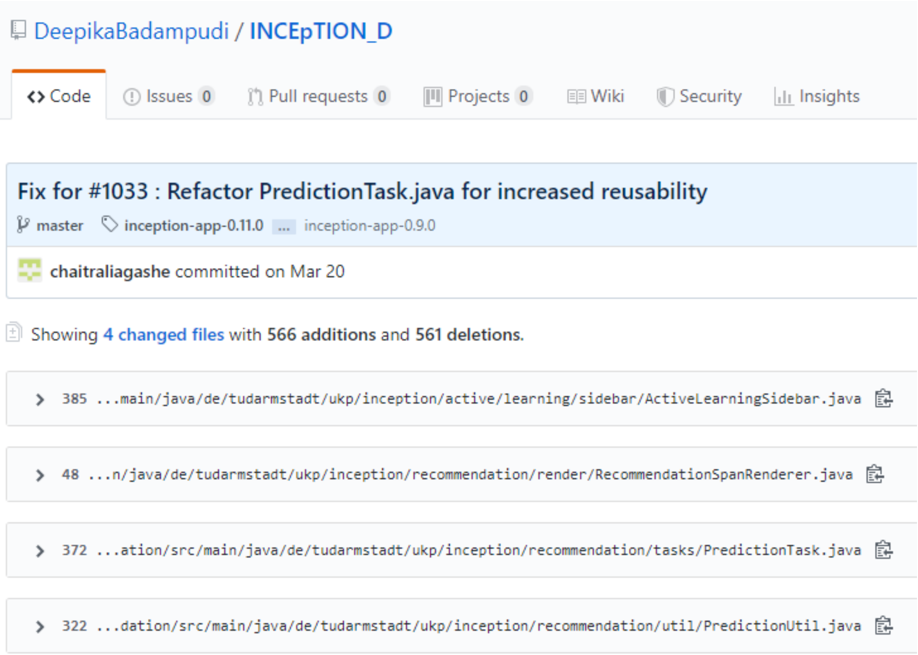
where S is the set of inequality and equality constraints and the functions f_i are *objective* or *fitness* functions. In multi-objective optimization, the quality of a solution is recognized by dominance. The set of feasible solutions that are not dominated by any other solution is called *Pareto-optimal* or *Non-dominated* solution set.

NSGA-II is a multi-objective evolutionary algorithm operating on a population of candidate solutions which are evolved toward the Pareto-optimal solution set. As described in Algorithm 1, the first iteration of the process begins with the complete execution of NSGA-II adapted to our refactoring recommendation problem based on the fitness functions representing each of the quality attributes. In the beginning, a random population of encoded refactoring solutions, P_0, is generated as the initial parent population. Then, the children population, Q_0, is created from the initial population using crossover and mutation. Parent and children populations are combined to form R_0. Finally, a subset of solutions is selected from R_0 based on the crowding distance and domination rules. This selection is based on elitism which means keeping the best solutions from the parent and child population. Elitism does not allow an already discovered non-dominated solution to be removed. After the identification of the non-dominated refactoring solutions, we apply a filter on them consisting of the detected changed files from the commit(s) and the desired quality attributes, also extracted from the commit(s). These identified refactorings are assigned to each of the commits that have been modified by the developers.

3.4. Running Example

To demonstrate a practical example of our proposed approach, we analyzed a real-world software repository on GitHub. For this purpose, we executed our tool on a repository called "Inception_D". This project provides a semantic annotation platform offering intelligent annotation assistance and knowledge management. It is a large project including over 5000 commits.

As a first step of our approach, we analyzed and filtered the commits of the mentioned repository and we extracted the refactoring-related commits. Figure 5 represents the commit where the developer(s) documented the changes as *" Refactor PredictionTask.java for increased reusability"*. It is clear from the developer's documentation that his intention was to improve the reusability of that class. This information helped in identifying the refactoring rationale. Our refactoring recommendation component takes as an input the modified classes which is, in this commit, "PredictionTask.java" and "Reusability" as a quality attribute to improve. Figure 7 shows the list of refactorings that were recommended by our tool to enhance/extend the developer's list of applied refactoring as shown in Figure 6. Three out of the four recommended refactoring solutions contained the specific modified file as a parameter. To show the usefulness and the impact of our recommended solutions, RefCom generates charts for comparaison between *the before developer's changes*, *the after developer's changes* and *the after RefCom refactorings* values of each QMOOD quality attributes. Figure 8 highlights that RefCom clearly provided much better alternatives than the actual manual refactorings applied by the developer. For instance, the reusability attribute was significantly improved—almost 15 times more than the improvement introduced by the developer's changes.



Fix for #1033 : Refactor PredictionTask.java for increased reusability

master inception-app-0.11.0 ... inception-app-0.9.0

chaitraliagashe committed on Mar 20

Showing 4 changed files with 566 additions and 561 deletions.

```
385 ...main/java/de/tudarmstadt/ukp/inception/active/learning/sidebar/ActiveLearningSidebar.java
48 ...n/java/de/tudarmstadt/ukp/inception/recommendation/render/RecommendationSpanRenderer.java
372 ...ation/src/main/java/de/tudarmstadt/ukp/inception/recommendation/tasks/PredictionTask.java
322 ...dation/src/main/java/de/tudarmstadt/ukp/inception/recommendation/util/PredictionUtil.java
```

Figure 5: The analyzed commit message from ”Inception_D”

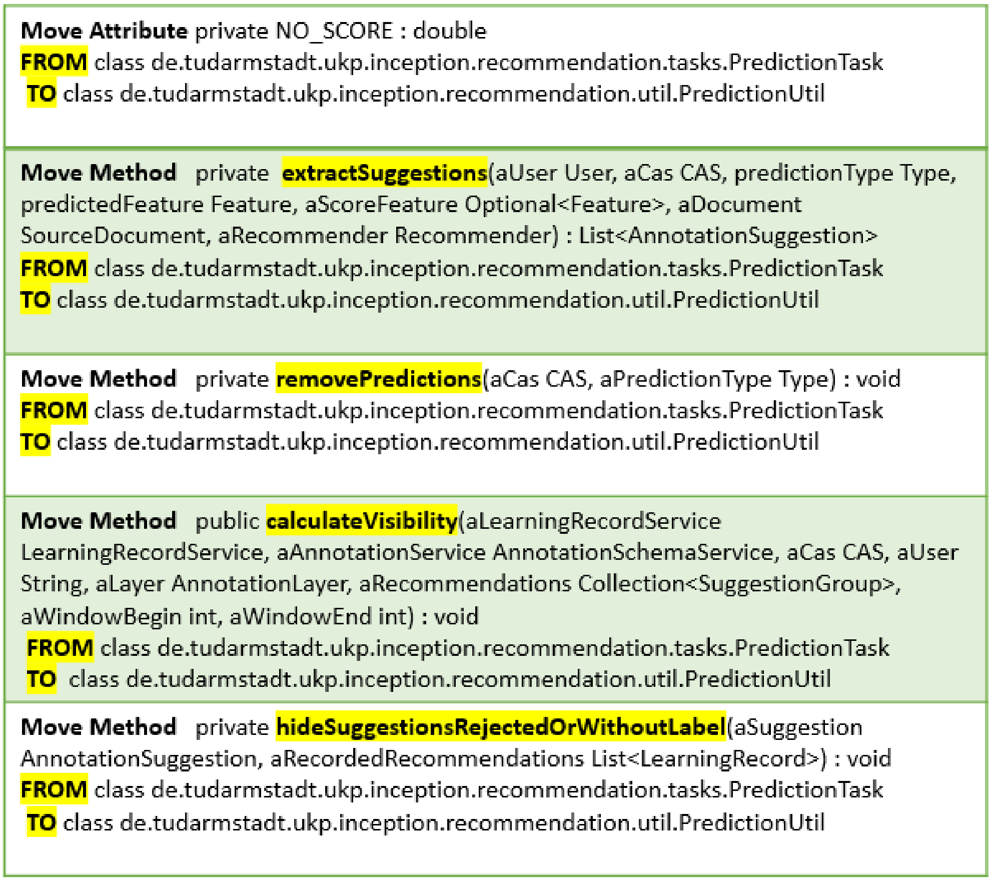
4. Evaluation

4.1. Research Questions

To validate our proposed approach, we defined the following three research questions:

- **RQ1.** To what extent are refactorings documented in commit messages?
- **RQ2.** To what extent do developers accurately document their refactoring and its rationale?
- **RQ3.** To what extent can our approach recommend relevant refactorings based on commit analysis compared to existing refactoring techniques?

While the *first* research question will validate our first hypothesis about developers document their refactoring rationale in commit messages, the *second* research question will validate the second hypothesis that developers spend the



Move Attribute private NO_SCORE : double
FROM class de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask
TO class de.tudarmstadt.ukp.inception.recommendation.util.PredictionUtil

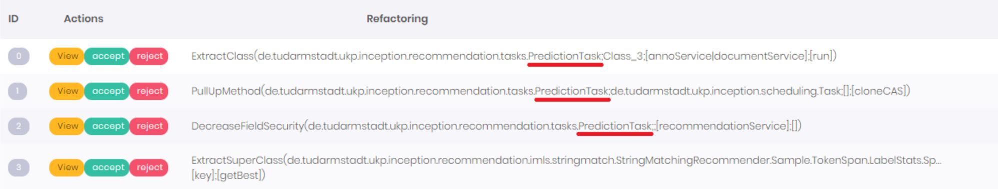
Move Method private **extractSuggestions**(aUser User, aCas CAS, predictionType Type, predictedFeature Feature, aScoreFeature Optional<Feature>, aDocument SourceDocument, aRecommender Recommender) : List<AnnotationSuggestion>
FROM class de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask
TO class de.tudarmstadt.ukp.inception.recommendation.util.PredictionUtil

Move Method private **removePredictions**(aCas CAS, aPredictionType Type) : void
FROM class de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask
TO class de.tudarmstadt.ukp.inception.recommendation.util.PredictionUtil

Move Method public **calculateVisibility**(aLearningRecordService LearningRecordService, aAnnotationService AnnotationSchemaService, aCas CAS, aUser String, aLayer AnnotationLayer, aRecommendations Collection<SuggestionGroup>, aWindowBegin int, aWindowEnd int) : void
FROM class de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask
TO class de.tudarmstadt.ukp.inception.recommendation.util.PredictionUtil

Move Method private **hideSuggestionsRejectedOrWithoutLabel**(aSuggestion AnnotationSuggestion, aRecordedRecommendations List<LearningRecord>) : void
FROM class de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask
TO class de.tudarmstadt.ukp.inception.recommendation.util.PredictionUtil

Figure 6: The manual refactoring applied by the developer in the commit



ID	Actions	Refactoring
0	View accept reject	ExtractClass(de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask;Class_3;[annoService|documentService];[run])
1	View accept reject	PullUpMethod(de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask;de.tudarmstadt.ukp.inception.scheduling.Task;[];[cloneCAS])
2	View accept reject	DecreaseFieldSecurity(de.tudarmstadt.ukp.inception.recommendation.tasks.PredictionTask;;[recommendationService];[])
3	View accept reject	ExtractSuperClass(de.tudarmstadt.ukp.inception.recommendation.imls.stringmatch.StringMatchingRecommender.Sample.TokenSpan.LabelStats.Sp... [key];[getBest])

Figure 7: The List of refactorings recommended by RefCom

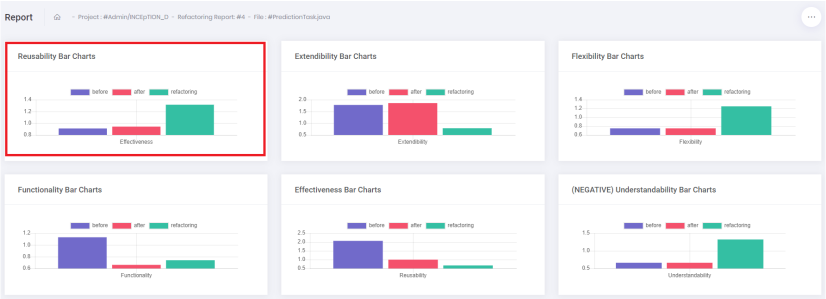


Figure 8: QMOOD quality before and after the commits comparing the manual refactorings and RefCom

minimum of manual refactorings effort to fix the identified quality issues, thus there are inconsistencies (or incomplete refactorings) between documented and applied refactorings in terms of expected impact/intention. The third question will evaluate the relevance of the recommended refactorings after integrating the two above insights into our refactoring tool to make actionable recommendations. A demo of our refactoring tool, Refcom, can be found in [49].

4.2. Experimental setting

To address the research questions, we analyzed the six open source systems in Table 4. Atomix is a fault-tolerant distributed coordination framework. Btm is a distributed and complete implementation of the JTA 1.1 API. Jgrapht is a graph library that provides mathematical graph-theory objects and algorithms. JSAT is a set of algorithms for pre-processing, classification, regression, and clustering with support for multi-threaded execution. Pac4j is a security engine. Tablesaw includes a data-frame, an embedded column store, and hundreds of methods to transform, summarize, or filter data. We selected these projects because of their size, number of commits, and applied refactorings.

To answer RQ1, we computed the ratio of the number of refactoring related commits to the total number of commits. Then, we counted the number of documented refactorings among these identified refactoring related commits. Documented refactorings are the commit messages that contain documentation

Table 4: Summary of the evaluated systems.

N	Project Name	LOC	Number of Classes	Total Commits	Refactoring related commits	Total number of refactorings
1	atomix	182280	1459	4237	343	12909
2	btm	34232	187	975	150	522
3	jgrapht	158665	526	2902	204	2202
4	JSAT	182267	436	1561	236	1457
5	pac4j	31916	302	2282	127	3130
6	tablesaw	52837	224	1930	327	3143

about refactoring. These documented refactorings are detected using keywords. However, refactoring related commits are the commits found after the union of the results of RefactoringMiner [31] detection, keywords extraction (same list of keywords previously mentioned) and the observed quality attribute changes between commits detected using our dedicated parser. A commit can be considered as a “refactoring related commit” , while it does not contain refactoring documentation (in the commit message) because it may contain either refactorings detected by RefactoringMiner or included quality improvements (when comparing before/after refactoring). In addition to evaluate the number of refactoring related commits and documented refactorings, we have also evaluated the main quality attributes that are documented in refactoring related commits to understand the most important ones that developers document. The detection of the documented quality attributes is carried out by searching for quality attributes names and their roots in the commit messages. Finally, we investigated the number of commits that introduce significant changes in the quality attributes, but which developers did not document.

To answer RQ2, we checked all the quality attributes by analyzing the code, and not only the ones claimed/documented by developers in their commits. There are two main reasons for checking all the quality attributes improvement. First, it helped identifying the refactoring related commits that contain documented quality attributes but there were no actual observed improvement of the quality attributes before and after the commit. Second, checking all the quality attributes improvement helps detecting the commit that does not claim a quality attribute but still is related to refactoring. In fact, we have used RefactoringMiner [31] and our tool for code analysis to detect the situations where quality attributes changes and applied refactorings were not documented. These are opportunities for refactoring solutions that better address these quality attributes.

To answer RQ3, we used the outcomes of the two prior research questions to identify developer refactoring rationale per commit: what files did they want to refactor? And what quality attributes did they want to improve? Then, we used that rationale to guide and filter the refactoring recommendations generated using our approach based on multi-objective search. We compared the automated refactorings using RefCom to the manual refactorings applied by the developers in the commits in terms of quality improvements. Then, we compared the recommended refactorings to two existing studies [33, 32] using a relevance measure. The relevance of the refactorings is defined as the number of refactoring recommendations accepted by developers participating in our experiments divided by the total number of recommended refactorings.

We asked 24 developers to evaluate the meaningfulness of the refactorings recommended by Refcom and by the approach of Ouni [32] and JDeodorant [33] for pull-requests on the six subject systems. We followed a random order of the three tools when the results were manually inspected. All the experimental techniques generate sequences of refactoring operations that make sense when considered together rather than when looking at them in isolation. However, it is not an option to ask a developer to assess the meaningfulness of all the refactoring operations generated for a given system. For this reason, we started by filtering for each system the sequences of refactoring operations impacting the files of a set of pull-requests to make a fair comparison between both tools. Then, the developers manually evaluated the outcomes of both tools for the commits of each pull-request.

Each participant was then asked to assess the meaningfulness of the sequences of refactoring operations. We made sure that each participant only evaluated refactoring sequences recommended by the three competitive techniques on one system. The rationale for such a choice is that an external developer would need time to acquire system knowledge by inspecting its code, and we did not want participants to have to comprehend the code from multiple systems since this would introduce a training effect in our study.

To support such a complex experimental design, we built a Java Web-app that automatically assigns the refactored pull-requests to be evaluated to the developers. The Web-app showed each participant one sequence of refactoring operations on a single page, providing the developer with (i) the list of refactorings (move method m_i to class C_j, then push down field f_k to subclass C_j,), (ii) the code of the classes impacted by the sequence of refactorings, and (iii) the complete code of the system subject of the refactoring with the generated refactoring sequence. The web page showing the refactoring sequence asked participants the question *Would you apply the proposed refactorings?* with a choice between *no* (the refactoring sequence is not meaningful), or *yes* (the refactoring sequence is meaningful and should be implemented). Moreover, participants were optionally allowed to leave a comment justifying their assessment. The Web-app was also in charge of:

Balancing the evaluations per system. We made sure that each system received roughly the same number of participants evaluating the different refactored pull-requests/commits (files associated/modified by these pull-requests) by the three approaches.

Keeping track of the time spent by participants in the evaluation of each refactoring sequence/pull-request. The time spent by participants was counted in seconds since the moment the Web-app showed the refactoring on the screen to the moment in which the participant submitted their assessment. This feature was done to remove participants from our data set who did not spend a reasonable amount of time in evaluating the refactorings. We consider less than 90 seconds a reasonable threshold to remove noise (we removed all evaluation sessions in which the participant spent less than 90 seconds in analyzing a single refactoring sequence).

Collecting demographic information about the participants. We asked their

Table 5: Participants involved to answer RQ3.

System	#Partic.	Avg. Prog. Experience	Avg. Java Experience	Avg. Refact. Exp.(1-5)
atomix	4	9	9	4.0 (high)
btm	4	8	7	3.5 (medium)
jgrapht	4	10	9	3.8 (medium)
JSAT	4	9	7	3.5 (high)
pac4j	4	7.5	7	4.5 (very high)
tablesaw	4	9	9	3.5 (high)

programming experience (in years) overall and in Java, and a self-assessment of their refactoring experience (from very low to very high). All of the participants were hired based on our current and previous extensive industry collaborations on refactoring. Despite that we contacted open source developers, we did not receive from them a timely response or did not answer at all which is a common challenge and threat in human studies within software engineering research [52]. We made sure that all the selected participants from industry are experienced in refactoring and used before these open source systems/libraries.

Table 5 shows the participants involved in our study and how they were distributed in the evaluation of the refactoring sequences generated for the six systems.

4.3. Results

Results for RQ1. Since our work is based on the assumption that developers write commit messages to document some of the applied refactorings, we identified first the commits related to refactorings then we checked those that documented the applied refactorings in the commit messages.

Table 6 summarizes our findings. It is clear that all the six open source projects have extensive refactorings applied in previous commits: an average of over 30% of all commits. The Atomix system has the highest number of commits related to refactoring. We found that 211 commit messages documented the applied refactorings, which is more than 60% of commits containing refactorings. The same observation can be applied to the remaining systems. While developers extensively apply refactorings, they may not document all of them.

Still there are enough commits including refactoring documentation to identify further opportunities for refactoring.

Project	Total number of commits	Commits related to refactoring	Docuented commits related to refactoring	Commits identified with RefacotoringMiner	Commits identified with Quality Improvements
atomix	4237	343	211	233	174
btm	975	150	52	55	46
jgraphft	2902	204	107	87	40
jsat	1561	236	113	58	65
pac4j	2282	127	84	65	33
tablesaw	1930	327	159	116	63

Table 6: An overview of the documented commits related to refactoring on the six open source systems.

We also investigated the main quality attributes of QMOOD that were documented by developers in the commit messages when refactorings were applied to improve those attributes. As described in Figure 9, we found understandability to be the most common quality documented by developers in commit messages. In 4 of the 6 open source systems it is the most common quality attribute documented by developers. For instance, the developers mentioned the rationale of understandability in messages in 53% of the commits improving the Atomix system. Reusability is the second most documented rationale, on average, in the six systems. It is also normal that developers document the rationale of the refactorings in combination with the features that were modified (functionality).

To conclude, we found that developers do document refactorings and they extensively apply refactorings over the commits of all six open source systems. Our results show that developers mention quality attributes as a rationale for their refactorings in over 50% of commits related to refactoring that are documented, which is enough to find opportunities for enhanced refactorings.

Results for RQ2. Figure 10 shows that developers are documenting their intention to refactor the code to address quality issues in the commit messages; however we did not find any quality improvements when we analyzed the quality changes in the files of these commits. For the Btm system, we found that only 32 out of 149 commits related to refactoring have actual quality changes. Only 60 out 236 commits related to refactorings have actual quality changes despite

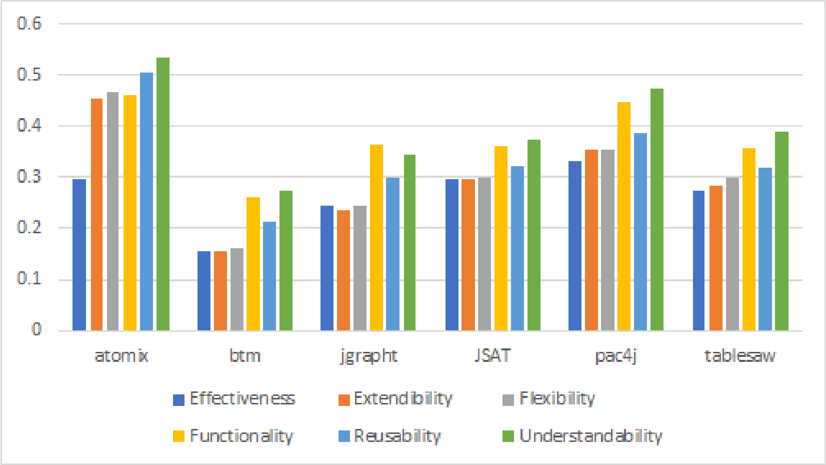


Figure 9: The percentage of documented quality attributes per system among the commits improving the quality attributes.

developers commenting on applying refactorings in their commit messages.

It is clear that developers highlight their intention to refactor the code with its rationale; however no actual quality improvements have been observed in many commits. This conclusion is one of the main motivations for RQ3.

Results for RQ3. After validating the two hypotheses of the previous research questions, we implemented our Refcom tool for improving the QMOOD quality attributes by integrating a filter to guide the refactoring recommendations based on rationale identified in the previous research questions (what quality attributes and which files do developers want to improve?). Figure 9 shows that developers documented refactorings with the intention of improving all the 6 quality attributes but with different levels of frequency. For instance, it is clear that developers focused on improving both understandability and reusability in project atomix. Thus, we executed our multi-objective algorithm using all the 6 quality attributes then we filter the Pareto front based on the two main criteria that are contained in the extracted refactoring rationale. First, we make sure that the selected solution is the one that provides the highest improvement in the quality attributes extracted from the commits during our

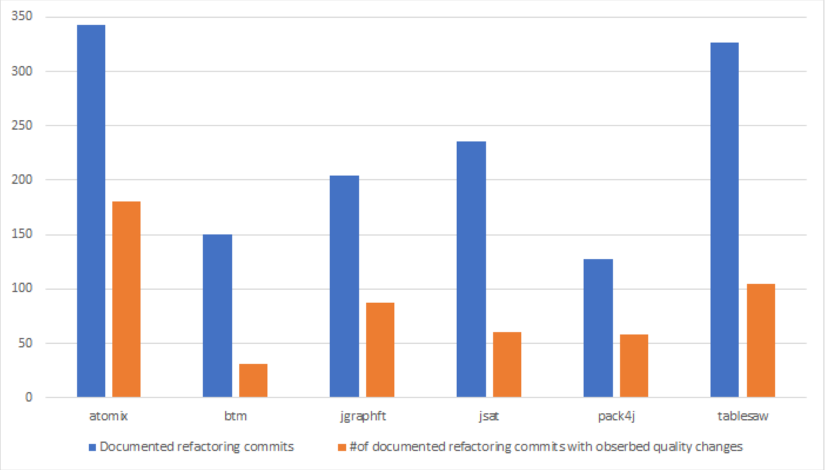


Figure 10: Missed documented refactoring opportunities in the 6 systems.

analysis step (e.g. understandability and reusability in project atomix). Second, the optimal solution should also refactor the detected changed files in the commits. We compared our results with two existing refactoring tools. Ouni [32] proposed a multi-objective refactoring formulation based on NSGA-II that generates a solution to maximize the design coherence and refactoring reuse from previous releases. JDeodorant [33] is an Eclipse plugin to detect bad smells and apply refactorings.

Figure 11 highlights the out-performance of RefCom compared to the tools of Ouni et al. [32] and JDeodorant [33]. In fact, most refactorings recommended by our approach are relevant, and all of them were successfully applied for the case Atomix system on the expected files and achieved high-quality improvements, based on the feedback from the participants.

By looking at the comments left by participants when justifying their assessments, thirteen out of the twenty four developers highlighted in their comments about the refactoring sequences that they found the refactorings relevant because they are completing the effort started by the submitter of the developer as described in the commit messages. For example, one of the developers wrote in a comment: *"I found these refactorings really improving the reusability of*

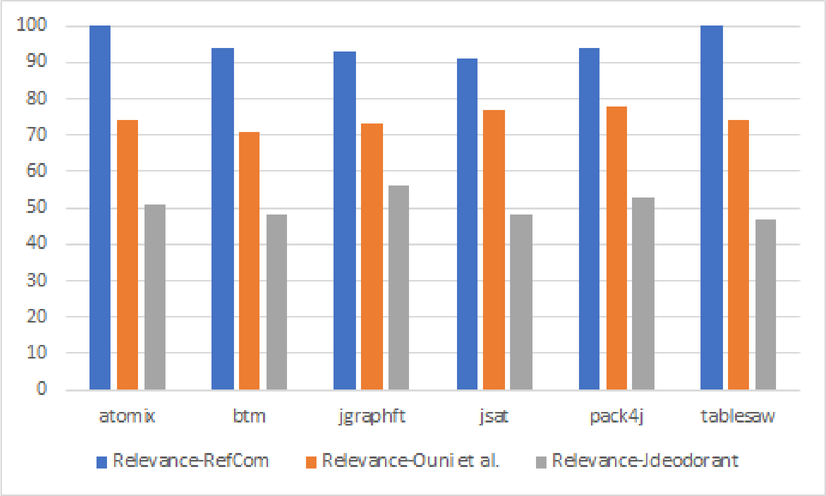


Figure 11: The relevance of the recommended refactorings by RefCom compared to existing refactoring approaches.

this class which is the main intention of the developer but he just applied couple of move methods. I found the tool recommendation even better to improve the reusability.”. We found this comment as important qualitative evidence of only the value of RefCom in terms of analyzing the recently closed pull-requests to identify changed files and fix the identified quality issues in these files.

Thus RefCom provided relevant refactoring recommendations based on the commit analysis, outperforming existing approaches to recommend refactorings.

5. Threats to Validity

We discuss in this section the different threats related to our experiments.

The threats to internal validity can be related to the list of keywords that we used to identify the commits where developers documented refactorings. However, the impact of this threat was limited by considering the use of Refactoring-Miner to identify the actual refactorings applied by developers. The parameters tuning of the optimization algorithm used in our experiments may create an internal threat that needs to be evaluated in future work since the parameter values used in our experiments were found by trial and error.

Construct validity is concerned with the relationship between theory and what is observed. We have used the QMOOD quality attributes to capture the quality changes between commits. While the QMOOD model is already empirically validated by existing studies [53], it is possible that some quality changes may not be detected using QMOOD.

External validity refers to the generalizability of our findings. We performed our experiments on 6 open-source systems belonging to different domains. However, we cannot assert that our results can be generalized to other applications and other developers. Moreover, we found that only 32 out of 149 commits related to refactoring have actual quality changes which limits the generalizability of our findings and requires more experiments. Another threat could be the number of subjects (24 developers) used for validation. Future replications of this study are necessary to confirm our findings.

6. Related Work

6.1. Detection Refactoring Opportunities

Several approaches have been proposed to automatically detect design flaws (anti-patterns, code smells) [54, 55, 56, 57, 58, 59, 60, 61, 62, 63]. We only discuss a few representative works and refer the interested reader to the recent survey by Sharma and Spinellis [64] for a complete overview.

Marinescu [7] proposes a metric-based mechanism to capture deviations from good design principles and heuristics, called “detection strategies”. Such strategies are based on the identification of *symptoms* characterizing a particular smell and *metrics* for measuring such symptoms.

Moha et al. [65] exploit a similar idea in their DECOR approach, proposing a Domain-Specific Language (DSL) for specifying smells using high-level abstractions. Four design smells are identified by DECOR, namely *Blob*, *Swiss Army Knife*, *Functional Decomposition*, and *Spaghetti Code*.

Design flaw detection can also be formulated as an optimization problem, as pointed out by Kessentini et al. [37]. They present a cooperative parallel search-based approach for identifying code smell instances. The idea here is that many evolutionary algorithms are executed in parallel to solve a common goal (the detection of code smells). The empirical evaluation reported in the paper shows the high accuracy of the proposed approach (recall and precision higher than 85%).

Besides metrics exploiting structural information extracted from the code, Palomba et al. [66] provide evidence that historical data can be successfully exploited to identify code smells; not only smells that are intrinsically characterized by their evolution across the program history but also smells such as Blob and Feature Envy.

Despite the extensive studies on the detection of refactoring opportunities [64, 67], none of them considered the use of commit messages to understand developer intentions during refactoring and the type of quality issues they want to address. The main assumption of most of these approaches is that developers want to fix code smells and antipatterns. However, we found that developers largely did not use terms related to antipatterns or code smells when describing and documenting refactoring opportunities in practice.

6.2. Refactoring Recommendation

Much effort has been devoted to the definition of approaches supporting refactoring. One representative example is JDeodorant, the tool proposed by Tsantalis and Chatzigeorgiou [68]. We point the interested reader to the survey by Bavota et al. [69] for an overview of approaches supporting code refactoring.

O'Keeffe and Cinnéide [70] presented the idea of formulating the refactoring task as a search problem in the space of alternative designs, generated by applying a set of refactoring operations. Such a search is guided by a quality evaluation function based on eleven object-oriented design metrics that reflect refactoring goals. Harman and Tratt [71] were the first to introduce the concept of Pareto optimality to search-based refactoring. They used it to combine two metrics, namely CBO (Coupling Between Objects) and SDMPC (Standard Deviation of Methods Per Class), into a fitness function and showed its superior performance as compared to a mono-objective technique [71].

The two aforementioned works [70, 71] paved the way to several search-based approaches aimed at recommending refactoring operations [72, 73, 74, 75, 42, 32]. A representative example of these techniques is the recent work by Ouni et al. [32], who propose a multi-criteria code refactoring approach aimed at optimizing five objectives: (i) minimizing the number of code smells; (ii) minimizing the refactoring cost (the number of recommended refactorings); (iii) preserving the design semantics (meaning considering textual information embedded in code identifiers and comments in the refactoring recommendation); and (iv) maximizing the consistency with code changes performed over the system's change history.

Murphy-Hill et al. [8] show that semi-automated tools for refactorings have been underutilized. In fact, fully automatic refactoring usually does not lead to the desired architecture and thus a designer's feedback should be included. Other studies also highlighted that developers are mainly interested in incremental refactoring and they are combining regular code changes such as bug-fixing with refactoring [18]. We proposed, in this paper, another perception to the way that refactorings can be recommended by extracting relevant information from commit messages and providing better suggestions to refactor the files related to the interests of the developers.

6.3. Empirical Studies on Refactoring

Empirical studies on software refactoring mainly aim at investigating the refactoring habits of software developers and the relationship between refactoring and code quality.

Murphy-Hill et al. [41] investigated how developers perform refactorings. Examples of the exploited datasets are usage data from 41 developers using the Eclipse environment and information extracted from versioning systems. Among their findings they show that developers often perform *floss refactoring*, namely they interleave refactoring with other programming activities, confirming that refactoring is rarely performed in isolation. Kim et al. [28] present a survey of software refactoring with 328 Microsoft engineers. To investigate when and how they refactor code and developer perception of the benefits, risks, and challenges of refactoring. They show that the major risk factor perceived by developers is the introduction of bugs and one of the main benefits they expect is to have fewer bugs in the future, thus indicating the usefulness of refactoring for code components exhibiting high fault-proneness. A recent empirical study [76] shows that developers have a misperception of quality metrics, as compared to terms used in academia, when documenting refactorings which motivates our work where we look at the actual metric changes rather than just the term in the commit messages, when recommending refactorings.

7. Conclusion

We presented a first attempt to recommend refactorings by analyzing commit messages. The salient feature of the proposed **RefCom** approach is its ability to capture developers need, from their commit messages, and propose to them refactorings to enhance their changes to better address quality issues. To evaluate the effectiveness of our technique, we applied it to six open-source projects and compared it with state-of-the-art approaches that rely on static and dynamic analysis. Our results show promising evidence on the usefulness of the proposed commit-based refactoring approach.

Future work will involve validating our technique with additional refactoring types, programming languages and a more extensive set of projects and commits to investigate the general applicability of the proposed methodology. We will also check the relevance of integrating commit messages in finding and recommending refactoring opportunities then fixing them based on different refactoring recommendations tools beyond our previous work.

References

[1] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., 1999 (1999).

[2] M. Feathers, Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1, Prentice Hall Professional, 2004 (2004).

[3] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, A. Shapochka, A case study in locating the architectural roots of technical debt, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2, IEEE, 2015, pp. 179–188 (2015).

[4] J. Carriere, R. Kazman, I. Ozkaya, A cost-benefit framework for making architectural decisions in a business context, in: 2010 ACM/IEEE 32nd International Conference on Software Engineering, Vol. 2, IEEE, 2010, pp. 149–157 (2010).

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

[6] D. Batory, J. N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Transactions on Software Engineering 30 (6) (2004) 355–371 (2004).

[7] R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., IEEE, 2004, pp. 350–359 (2004).

[8] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering 38 (1) (2012) 5–18 (2012).

[9] D. Dig, C. Comertoglu, D. Marinov, R. Johnson, Automated detection of refactorings in evolving components, in: European Conference on Object-Oriented Programming, Springer, 2006, pp. 404–428 (2006).

[10] J. Kim, D. Batory, D. Dig, M. Azanza, Improving refactoring speed by 10x, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 1145–1156 (2016).

[11] A. Ouni, M. Kessentini, H. Sahraoui, M. Boukadoum, Maintainability defects detection and correction: a multi-objective approach, Automated Software Engineering 20 (1) (2013) 47–79 (2013).

[12] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, M. Ó Cinnéide, Recommendation system for software refactoring using innovization and interactive dynamic optimization, in: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, ACM, 2014, pp. 331–336 (2014).

[13] B. Du Bois, S. Demeyer, J. Verelst, Refactoring-improving coupling and cohesion of existing code, in: 11th working conference on reverse engineering, IEEE, 2004, pp. 144–151 (2004).

[14] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, K. Deb, Multi-criteria code refactoring using search-based software engineering: An industrial case study, ACM Transactions on Software Engineering and Methodology (TOSEM) 25 (3) (2016) 23 (2016).

[15] I. H. Moghadam, M. Ó Cinnéide, Code-imp: a tool for automated search-based refactoring, in: Proceedings of the 4th Workshop on Refactoring Tools, ACM, 2011, pp. 41–44 (2011).

[16] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, S. Yang, Refactoring android java code for on-demand computation offloading, in: ACM Sigplan Notices, Vol. 47, ACM, 2012, pp. 233–248 (2012).

[17] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, Y. Cai, An interactive and dynamic search-based approach to software refactoring recommendations, IEEE Transactions on Software Engineering (2018).

[18] V. Alizadeh, M. Kessentini, Reducing interactive refactoring effort via clustering-based multi-objective search, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 464–474 (2018).

[19] M. O'Keeffe, M. O. Cinnéide, Search-based refactoring for software maintenance, Journal of Systems and Software 81 (4) (2008) 502–516 (2008).

[20] W. Brown, R. Malveau, S. McCormick, T. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, Wiley, 1998 (1998).

[21] V. Alizadeh, M. Kessentini, Reducing interactive refactoring effort via clustering-based multi-objective search, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, ACM, New York, NY, USA, 2018, pp. 464–474 (2018). doi:10.1145/3238147.3238217.
URL http://doi.acm.org/10.1145/3238147.3238217

[22] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, W. Zhao, Interactive and guided architectural refactoring with search-based recommendation, in: Proc. 24th.

[23] G. B. V. A. Jeffrey Yackley, Marouane Kessentini, B. Maxim, Simultaneous refactoring and regression testing: A multi-tasking approach, in: Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM2019, 2019, p. 12 pages (2019).

[24] V. A. M. K. Soumaya Rebai, Ousaama Ben Sghaier, M. Chater, Interactive refactoring documentation bot, in: Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM2019, 2019, p. 12 pages (2019).

[25] J. Pantiuchina, M. Lanza, G. Bavota, Improving code: The (mis) perception of quality metrics, in: 2018 IEEE International Conference on Software

Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018, 2018, pp. 80–91 (2018). doi:10.1109/ICSME.2018.00017.
URL https://doi.org/10.1109/ICSME.2018.00017

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

[27] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, O. Strollo, When does a refactoring induce bugs? an empirical study, in: 12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM, 2012, pp. 104–113 (2012).

[28] M. Kim, T. Zimmermann, N. Nagappan, An empirical study of refactoringchallenges and benefits at microsoft, Software Engineering, IEEE Transactions on 40 (7) (2014) 633–649 (July 2014).

[29] E. A. AlOmar, M. W. Mkaouer, A. Ouni, Can refactoring be self-affirmed?: An exploratory study on how developers document their refactoring activities in commit messages, in: Proceedings of the 3rd International Workshop on Refactoring, IWOR '19, IEEE Press, Piscataway, NJ, USA, 2019, pp. 51–58 (2019). doi:10.1109/IWoR.2019.00017.
URL https://doi.org/10.1109/IWoR.2019.00017

[30] G. Soares, R. Gheyi, T. Massoni, Automated behavioral testing of refactoring engines, IEEE Transactions on Software Engineering 39 (2) (2013) 147–162 (2013).

[31] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history, in: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, 2018, pp. 483–494 (2018). doi:10.1145/3180155.3180206.
URL https://doi.org/10.1145/3180155.3180206

[32] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, K. Deb, Multi-criteria code refactoring using search-based software engineering: an industrial case study, ACM Transactions on Software Engineering and Methodology (TOSEM) 25 (3) (2016) 23 (2016).

[33] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, Jdeodorant: identification and application of extract class refactorings, in: 33rd International Conference on Software Engineering (ICSE), 2011, pp. 1037–1039 (2011).

[34] J. Bansiya, C. G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transactions on software engineering 28 (1) (2002) 4–17 (2002).

[35] M. O'Keeffe, M. O. Cinnéide, Search-based refactoring: An empirical study, Journal of Software Maintenance and Evolution 20 (5) (2008) 345–364 (2008).

[36] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, I. Hemati Moghadam, Experimental assessment of software metrics using automated refactoring, in: International Symposium on Empirical Software Engineering and Measurement (ESEM), 2012, pp. 49–58 (2012).

[37] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, A. Ouni, A cooperative parallel search-based software engineering approach for code-smells detection, IEEE Transactions on Software Engineering 40 (9) (2014) 841–861 (2014).

[38] A. C. Jensen, B. H. Cheng, On the use of genetic programming for automated refactoring and the introduction of design patterns, in: Proceedings of the 12th annual conference on Genetic and evolutionary computation, ACM, 2010, pp. 1341–1348 (2010).

[39] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, Y. R. Kwon, Automated scheduling

for clone-based refactoring using a competent ga, Software: Practice and Experience 41 (5) (2011) 521–550 (2011).

[40] R. Khatchadourian, H. Masuhara, Automated refactoring of legacy java software to default methods, in: Proceedings of the 39th International Conference on Software Engineering, IEEE Press, 2017, pp. 82–93 (2017).

[41] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering (TSE) 38 (1) (2011) 5–18 (2011).

[42] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, A. Ouni, Many-objective software remodularization using nsga-iii, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (3) (2015) 17:1–17:45 (2015).

[43] A. Alali, H. Kagdi, J. I. Maletic, What's a typical commit? a characterization of open source software repositories, in: Proc. 16th, 2008, pp. 182–191 (Jun. 2008).

[44] H. F. Vahid Alizadeh, M. Kessentini, Less is more: From multi-objective to mono-objective refactoring via developers knowledge extraction, in: Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM2019, 2019, p. 12 pages (2019).

[45] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, A. Bernstein, The missing links: Bugs and bug-fix commits, in: Proc. 16th, 2010 (Nov. 2010).

[46] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, C. Chockley, Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft, IEEE Transactions on Software Engineering 43 (1) (2017) 56–75 (2017).

[47] M. Beller, A. Bacchelli, A. Zaidman, E. Juergens, Modern code reviews in open-source projects: Which problems do they fix?, in: Proceedings of the

11th working conference on mining software repositories, ACM, 2014, pp. 202–211 (2014).

[48] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, IEEE Transactions on Software Engineering 38 (1) (2011) 5–18 (2011).

[49] Recommending Rfactorings via Commit Message Analyis.
URL `https://sites.google.com/view/istrefcom`

[50] M. Fowler, Refactoring: Improving the Design of Existing Code, 1999 (Jul. 1999).

[51] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multi-objective genetic algorithm: Nsga-ii, IEEE transactions on evolutionary computation 6 (2) (2002) 182–197 (2002).

[52] A. J. Ko, T. D. Latoza, M. M. Burnett, A practical guide to controlled experiments of software engineering tools with human participants, Empirical Software Engineering 20 (1) (2015) 110–141 (2015).

[53] O. Baysal, R. Holmes, A qualitative study of mozilla's process management practices, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada, Tech. Rep. CS-2012-10 (2012).

[54] A. Ghannem, M. Kessentini, G. El Boussaidi, Detecting model refactoring opportunities using heuristic search, in: Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research, 2011, pp. 175–187 (2011).

[55] M. Kessentini, P. Langer, M. Wimmer, Searching models, modeling search: On the synergies of sbse and mde, in: 2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE), IEEE, 2013, pp. 51–54 (2013).

[56] M. Kessentini, R. Mahaouachi, K. Ghedira, What you like in design use to correct bad-smells, Software Quality Journal 21 (4) (2013) 551–571 (2013).

[57] A. Ghannem, G. El Boussaidi, M. Kessentini, Model refactoring using examples: a search-based approach, Journal of Software: Evolution and Process 26 (7) (2014) 692–713 (2014).

[58] A. Ouni, M. Kessentini, S. Bechikh, H. Sahraoui, Prioritizing code-smells correction tasks using chemical reaction optimization, Software Quality Journal 23 (2) (2015) 323–361 (2015).

[59] M. Kessentini, A. Ouni, P. Langer, M. Wimmer, S. Bechikh, Search-based metamodel matching with structural and syntactic measures, Journal of Systems and Software 97 (2014) 1–14 (2014).

[60] B. Amal, M. Kessentini, S. Bechikh, J. Dea, L. B. Said, On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring, in: International Symposium on Search Based Software Engineering, Springer, Cham, 2014, pp. 31–45 (2014).

[61] A. Ghannem, G. El Boussaidi, M. Kessentini, On the use of design defect examples to detect model refactoring opportunities, Software Quality Journal 24 (4) (2016) 947–965 (2016).

[62] H. Wang, M. Kessentini, A. Ouni, Bi-level identification of web service defects, in: International Conference on Service-Oriented Computing, Springer, 2016, pp. 352–368 (2016).

[63] A. Ouni, M. Kessentini, M. Ó Cinnéide, H. Sahraoui, K. Deb, K. Inoue, More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells, Journal of Software: Evolution and Process 29 (5) (2017).

[64] T. Sharma, D. Spinellis, A survey on software smells, Journal of Systems and Software 138 (2018) 158 – 173 (2018).

[65] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. L. Meur, Decor: A method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering 36 (1) (2010) 20–36 (2010).

[66] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, A. De Lucia, Mining version histories for detecting code smells, IEEE Transactions on Software Engineering 41 (5) (2015) 462–489 (2015).

[67] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Oź CinneźIde, K. Deb, On the use of many quality attributes for software refactoring: A many-objective search-based software engineering approach, Empirical Softw. Engg. 21 (6) (2016) 2503–2545 (Dec. 2016). doi:10.1007/s10664-015-9414-4.
URL https://doi.org/10.1007/s10664-015-9414-4

[68] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (3) (2009) 347–367 (2009).

[69] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, Recommending refactoring operations in large software systems, in: M. P. Robillard, W. Maalej, R. J. Walker, T. Zimmermann (Eds.), Recommendation Systems in Software Engineering, Springer Berlin Heidelberg, 2014, pp. 387–419 (2014).

[70] M. O'Keeffe, M. Ó Cinnéide, A stochastic approach to automated design improvement, in: International Conference on Principles and practice of programming in Java, Computer Science Press, Inc., 2003, pp. 59–62 (2003).

[71] M. Harman, L. Tratt, Pareto optimal search based refactoring at the design level, in: 9th annual conference on Genetic and evolutionary computation, 2007, pp. 1106–1113 (2007).

[72] O. Seng, J. Stammel, D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, in:

International conference on Genetic and evolutionary computation, ACM, 2006, pp. 1909–1916 (2006).

[73] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni, Design defects detection and correction by example, in: International Conference on Program Comprehension (ICPC), IEEE, 2011, pp. 81–90 (2011).

[74] A. Ouni, M. Kessentini, H. Sahraoui, Search-based refactoring using recorded code changes, in: Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013), pp. 221–230.

[75] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, M. Ó Cinnéide, Recommendation system for software refactoring using innovization and interactive dynamic optimization, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE 2014), pp. 331–336.

[76] P. W. McBurney, S. Jiang, M. Kessentini, N. A. Kraft, A. Armaly, M. W. Mkaouer, C. McMillan, Towards prioritizing documentation effort, IEEE Transactions on Software Engineering 44 (9) (2018) 897–913 (2018).