需求驱动的软件重构推荐方法

Ally Selemani Nyamawe

2020 年 06 月

中图分类号：

UDC分类号：

需求驱动的软件重构推荐方法

作　者　姓　名	Ally Selemani Nyamawe
学　院　名　称	计算机学院
指　导　教　师	牛振东教授
答辩委员会主席	张路教授
申　请　学　位	工学博士
学　科　专　业	计算机科学与技术
学 位 授 予 单 位	计算机科学与技术
论 文 答 辩 日 期	2020 年 06 月

Requirements-Driven Recommendation of Software Refactoring

Candidate Name:	Ally Selemani Nyamawe
School or Department:	Computer Science and Technology
Faculty Mentors:	Prof. Zhendong Niu
Chair, Thesis Committee:	Prof. Zhang Lu
Degree Applied:	PhD of Computer Science and Technology
Major:	Computer Science and Technology
Degree by:	Beijing Institute of Technology
The Date of Defence:	June，2020

需求驱动的软件重构推荐方法

计算机科学与技术

摘要

软件需求的不断变化经常导致软件演化，从而增加软件系统内部结构的复杂性，降低软件整体质量。因此，需要通过软件重构以改善软件的内部结构。软件重构是在不改变软件外部行为特性的情况下通过改变软件内部结构提高代码质量的一种方法。然而，对于一个大型的软件系统，程序员往往难以决定该重构哪些代码以及如何重构这些代码。为此，人们提出了很多重构推荐方法。现有方法主要根据代码中的设计缺陷进行推荐，目标是消除相应的代码坏味（bad smells）。但最近的研究结果表明，引起重构的主要原因是软件需求的变化而不是代码坏味。为此，本文研究提出了需求驱动的软件重构方法，充分发掘需求变化引发的重构需求，进而及时、准确地进行重构推荐。本文主要贡献如下：

首先，我们提出一种基于软件需求追踪和代码度量的重构推荐方法。需求可追踪性在软件维护、程序理解、影响分析、需求追踪和软件重用等方面起着重要作用。为此，我们的方法充分利用了需求与代码之间的追踪关系，通过推荐一系列的重构操作来优化需求与代码的映射关系，提高软件的可追踪性。除追踪信息之外，我们的方法还利用软件代码度量指标（内聚和耦合性）衡量代码设计方面的改进效果。利用追踪质量和代码设计度量指标综合比较重构方案并推荐改进最大的方案。实验结果表明，与传统推荐方法相比，大部分情况下开发人员倾向于使用我们提出的方法。

其次，我们提出了一种基于学习的重构推荐方法，该方法可以帮助开发人员选出最优的重构应用于他们的软件系统，以使它们能够适应新的需求 (即，特征请求)。行业实践表明在实现新特征之前对软件进行重构，不仅可以提高软件的质量，还可以大大降低实现新需求的难度。该方法利用了过去的特征请求和以前应用的重构的历史去训练为新特征请求预测和推荐重构的机器分类器。这样的特征请求可以与其他应用程序集合相关联或训练中涉及到的那些，因此它确保了方法的普遍性。该方法在 43 个开源 Java 项目中进行了评估，结果表明该方法在预测和推荐重构的准确性方面显著提高了目前最先进水平。

最后，我们提出一种混合式的重构推荐方法。该方法充分利用了增量式需求、重构历史信息以及代码坏味。对于每条历史特征请求（feature request），现有工具通过 commit 信息挖掘已经应用的重构和解决的代码坏味问题。我们的机器学习分类器基于特征请求、代码坏味和重构信息等历史数据进行训练。分类器的输入为特征请求以

及实现该特征前的相关代码的坏味信息。分类器的输出是为实现该特征所实施的重构操作。基于训练好的分类器，可以针对新的特征请求以及当前的代码坏味情况推荐相应的重构操作。我们在 55 个开源 Java 项目上进行评估，结果表明所提方法大大提高推荐的准确率、查准率、查全率和 F1 值。

关键词：代码坏味；熵；特征请求；机器学习；重构推荐；需求追踪；软件重构

Abstract

Software requirements are ever-changing which often leads to software evolution. Such phenomenon increases the complexity of the internal structure of software systems, which consequently degrades their quality. As a result, software systems consistently need to be maintained to reduce complexity and improve their internal structure through refactoring. Software refactoring is a wide-spread practice of improving software quality by applying changes on internal structure without altering its external behaviors. However, selecting and applying optimal refactorings especially for large and non-trivial software systems is often challenging. Consequently, to facilitate developers in such task, a majority of effort has been devoted in evolving refactorings recommenders. Although the conventional refactorings recommenders have been largely accepted as useful solutions, there are still some challenges in attempt to provide accurate refactoring suggestions that are tailored to the practices and actual needs of software developers. That is because most of these approaches are limited to resolving code smells and do not consider other motivations of applying refactorings. However, recent studies suggest that, in practice, refactoring is mainly motivated by the changes in the requirements and rarely by resolution of code smells . To this end, this thesis proposes refactoring recommendation approaches that take requirements change as the primary driving factor of software refactorings. The major contributions of this thesis are summarized as follows.

First, we proposed an approach that is based on traceability and code metrics to recommend refactoring solutions to developers. Requirements traceability plays a great role in assisting software maintenance, program comprehension, impact analysis, requirements tracing and software reuse. To achieve these benefits, it is therefore of vital importance to ensure that the quality of traceability is well maintained throughout the course of software refactoring. In this approach, requirements traceability information is leveraged to identify how well source code entities (e.g., methods) can be grouped into classes based on how they trace to requirements (i.e., use cases). The rationale being strongly related use cases should trace to strongly related methods. Besides traceability information, this approach further incorporates source code design metrics (i.e., cohesion and coupling) to quantify the improvement in code design. The tradeoff between the quality of traceability and code design is computed to compare the refactoring solutions and recommend the one that leads to greatest improvement. The evaluation results suggest that in most cases developers prefer solutions recommended by this approach than those recommended by the traditional approaches.

Second, we proposed a learning-based refactorings recommendation approach that facilitates developers in selecting optimal refactorings to apply on their software systems to prepare them adapt to new requirements (i.e., feature requests). Industry practice shows that refactoring software before implementing new features can not only improve the quality of a software, but also greatly reduces the difficulty of implementing new requirements. The approach leverages the history of the past feature requests and previously applied refactorings to train the machine classifiers that would be used to predict and recommend refactorings for the new feature requests. Such feature requests can be associated with the set of other applications or those involved in the training, thus it ensures the generalizability of the approach. The approach is evaluated on 43 open source Java projects and the results suggest that the approach significantly improves the state-of-the-art in terms of accuracy of predicting and recommending refactorings.

Finally, we proposed a hybrid refactoring recommendation approach that considers code smells information besides the history of past feature requests and applied refactorings. For each past feature request, the state-of-the-art tools were leveraged to uncover the applied refactorings and the resolved code smells on the commits that were used to implement such request. Consequently, the machine classifiers were trained with the history data of feature requests, code smells and refactorings to predict and recommend refactorings for the new feature requests. Incorporating code smells information during training and prediction can potentially lead to accurate recommendation of the types of refactorings required. That is because, understanding the sort of code smells associating with a feature request can give an insight on what types of refactorings that are required. Experimental results on the dataset of 55 open source Java projects show a significant improvement in recommendation performance in terms of accuracy, precision, recall and F-measure metrics.

Key Words: Code smells; entropy; feature requests; machine learning; refactorings recommendation; requirements traceability; software refactoring.

Denotation

API	Application Programming Interface
AST	Abstract Syntax Tree
CDM	Call Dependence between Methods
CNN	Convolutional Neural Network
CSM	Conceptual Similarity between Methods
CVS	Control Version System
DT	Decision Tree
EP	Entity Placement
ERP	Enterprise Resource Planning
IDE	Integrated Development Environment
LR	Logistic Regression
MNB	Multinomial Naïve Bayes
MSR	Mining Software Repositories
NLOC	Number of Lines Of Code
NLP	Natural Language Processing
NLTK	Natural Language Processing Toolkit
RF	Random Forest
SDE	Software Development Environent
SSM	Structural Similarity between Methods
SVM	Support Vector Machine
TM	Traceability Matrix
TF-IDF	Term Frequency - Inverse Document Frequency

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Research background

Software systems usually undergo several changes in order to continue satisfying their intended objectives or fix shortcomings. Throughout software lifetime, new requirements emerge and existing ones are changed as the business and technology running such software systems evolve. Consequently, some parts of the software would need to be modified to adapt new requirements, resolve faults, or generally improve its performance. In a nutshell, software systems continuously evolve to accommodate the changes. Such phenomenon increases software systems complexity, which usually leads to the quality degradation of their internal structures[1]. According to Lehman‘s 7th law of software evolution[2], the quality of an evolving system will appear to decline as it evolves. Therefore, quality should be consistently monitored throughout the course of system evolution. Mens and Tourwé[3] suggest that, to adapt with the spiral of complexity, there is a need for techniques that reduce software complexity through incremental improvement of their internal quality. In object oriented software development paradigm, the research area that focuses on this problem is referred to as refactoring[3]. The term “refactoring” was originally coined by Opdyke in 1992[4], and as a field of study has recently gained a significant advancement and popularity in the software engineering research community. Fowler[5] defines refactoring as the process of improving the internal structure of a software system without changing its observable behavior. Software refactoring has been widely accepted as a means to improve source code comprehensibility, maintainability, extensibility and reusability[3,6,7].

Generally, software refactoring involves various activities including to identify where the software should be refactored and to determine what sort of refactorings should be applied in the identified places[3]. Although in practice the two activities usually go together, the latter is the main focus of this study. Identifying where to refactor or rather identifying refactoring opportunities is an important step that precedes the actual application of refactoring operation. One of the prominent ways of identifying refactoring opportunities is detection of code smells[3,8]. Code smells are the potential indicator that the source code is claiming for improvement. In other words, code smells are indicators of poor design and implementation choices[9] that suggest for possibility of refactoring. The existence and proliferation of code smells in software systems can potentially affect their further development, evolution and maintenance[10]. Besides that, code smells have shown to be one of the potential causative agents of software bugs[11]. Thus, if code smells are left unattended they could result to a serious repercussions on the quality of software systems. Consequently, numerous studies have invested the majority of effort in investigating code smells[7,12,13] and recommending refactoring solutions[14–16]. The catalog of well-known code smells and their possible refactoring solutions were initially introduced by Fowler[9] and subsequently extended in other studies[17–19]. To detect code smells, existing approaches leverage various techniques which can be broadly categorized into five groups: source code metrics, predefined rules, source code change history, machine learning and optimization algorithms[20]. However, the detection of code smells is beyond the scope of this study. In the rest of this thesis, to uncover smells in the source code, we will rely on some of the existing state-of-the-art code smells detectors[17,21–25].

Although manual refactoring is possible, it is inherently tedious, error-prone, and time consuming[26]. Consequently, several researches have proposed various solutions from the (semi-) automation point of view. Such approaches implement different heuristics and often provide tool support to recommend various types of refactorings such as: move method[1,14], extract class[27,28], and extract method[29–31]. Besides that, modern mainstream IDEs such as Visual Studio, Eclipse and IntelliJ IDEA also provide refactoring functionality and often compete on the basis of supporting refactoring[32]. Generally, refactoring recommendation aims at reducing developers' information overload by suggesting them the most relevant refactoring solutions and consequently facilitate their refactoring selection decision.

1.2 State-of-the-art

1.2.1 Refactoring Operations

Fowler[9] proposed a catalogue of 72 different refactoring operations and their possible mechanics. Besides, the catalogue also links code smells to refactorings. For each code smell, it is suggested to the developer what sort of actions to be taken to alleviate it. The refactoring operations span from lower levels of granularity of the program (e.g., attribute) to higher granularity levels (e.g., class). Examples of well-known refactoring operations include: $Move\ Method$[14] applied when classes collaborate too much and are too much coupled, $Etract\ Class$[16] applied to split a complicated class with too many responsibilities, $Rename\ Method$[33] applied to rename a method that it's signature does not correspond to what it does, $Etract\ Method$[29] often applied to split a method that is too long, and $InlineMethod$ applied to minimize indirection and methods delegations. Although there is a long list[9] of refactoring operations, empirical studies suggest that, in practice, some refactorings are more frequently applied than the others. For example, the study in[34] found that, Extract Method is the most popular high-level refactoring, whereas Push Down refactorings are among the least popular refactorings. The same study also suggests that, some refactorings have higher level of automation than the others. One of the surprising findings of the study is that, programmers need to apply Extract Method refactoring more often than Rename refactoring, but they actually apply Rename with tool support more often (58%) than performing Extract Method with tools (29%). Such trend can also be observed on the variation of the number of tools and techniques proposed for different refactoring operations. Additionally, such tools and techniques only support a subset of refactoring operations, for example, JDeodorant[23,24] (*extract class, move method*), JMove[22,35] (*move method*), ARIES[25] (*extract class*), and GEMS[29](*extract method*).

1.2.2 Refactoring Tools

In recent years, software engineering community has witnessed an increase in the number of refactoring tools and even the contemporary IDEs (e.g., Eclipse and VisualStudio) also provide refactoring functionality in their top menu. Generally, the extent of automation of refactoring tools varies according to which specific activity of refactoring is supported. For example, some tools are only for identifying where to refactor whereas others can even suggest what refactorings to apply. In semi-automatic refactoring, for instance, it is often left to a developer to identify where to refactor and select the most optimal refactoring operation, but the actual application of such refactoring is fully automatic. Automated refactoring has shown to significantly improve the efficiency of implementing and debugging changes than doing it manually[36]. The other key advantage of refactoring tools, as per developers' perspective, is their ability to preserve program behavior which remarkably reduces the need for testing and debugging, such two tasks are highly labor-intensive and time-demanding[3]. Besides semi-automatic approaches[22–25,29,35], some refactoring tools claim to be fully automated. For example, a *MultiRefactor* tool proposed by Mohan and Greer[37] is fully automated and supports 26 different refactoring types. Additionally, the tool contains a wide-range selection of metrics to quantify the impact of refactorings and six search-based optimization algorithms (both mono- and multi-objective) to improve the software. Another fully automated refactoring tool for refactoring methods and restructuring inheritance hierarchies is proposed in[38]. This thesis is particularly interested with tools related to smells detection and refactorings recovery. Next we briefly describe such tools.

1.2.3 Code Smells Detection

Fontana *et al*[39] consider code smells as the software structural characteristics that indicate a problem in design or code and they can complicate software maintenance and evolution. For example, one of the classical code smells is $Feature\ Envy$ (i.e., a method is more interested in another class than the one it is currently in); this smell can be resolved by moving a method to the class it mostly deserves (this operation is known as $Move\ Method$ refactoring). Detecting code smells and applying relevant refactorings especially in large and non-trivial software systems is often challenging[28]. Consequently, extensive research has been devoted to develop tools and techniques to automatically detect code smells, recommend refactorings, and apply them. Such tools as JDeodorant[24], iPlasma[40] and DECOR[17], make software refactoring efficient and less error-prone. To improve the quality of the source code it is important to detect code smells and consequently refactor with the appropriate solution[41]. To detect code smells, refactoring tools or smell detectors are often used. So far, several powerful smell detectors (such as JDeodorant, iPlasma, PMD, JSpIRIT, and inFusion) have been proposed and validated[10,41–43]. Smell detectors use various detecting techniques to uncover code smells. Mostly, code design metrics are used to identify design flaws in source code. Smell detectors are anticipated to allow users to understand the cause of the smell and to display smell information in a way that will not overload developers[39].

1.2.4 Refactorings Recovery

Recovery of refactorings is essential in understanding how object oriented systems have evolved by discovering the changes applied during their lifetime. Empirical data retrieved from analyzing past refactorings can be useful in evolving new refactoring tools, evaluating tools and approaches, and assessing programmer's refactoring preferences[44]. Usually, refactorings recovery approaches detect past refactorings through: mining the commit logs, refactoring activities logs, observing developers, and analyzing the code history[45]. The detailed analysis and comparison of such approaches are well presented by Murphy-Hill *et al.*[44] and Soares *et al.*[46]. In the context of this thesis, we leverage the refactorings recovery approaches that are based on analyzing code history. This technique takes a window of two versions of a program from software archives and detect where refactorings have been applied through manual comparison or automated comparison by tool support[44]. The two versions of a program are not necessarily to be adjacent. Moreover, refactorings extraction from software repositories is essential for many applications including identifying intent of changes (e.g., feature extension and bug fixes), detecting possible sources of errors and capturing and replaying of changes[47]. For example, the latter allows for identifying what refactorings to apply on the source code by analysing and adopting refactorings that have been applied on the related or similar program code[47]. Kim *et al.*[48] proposed `Ref-Finder` an Eclipse plugin which is based on logic programming engine to detect 63 refactoring types. The tool takes two versions of a program as input and employs AST analysis to extract facts about code elements syntactic structure. Consequently, the tool invokes logic queries to detect program differences according to the constraints of each refactoring type. Other well-known tools for recovery of refactorings include `RMINER`[49], `RefDiff`[50], `RefactoringCrawler`[51], `UMLDiff`[52], and `DIFFCAT`[53]. These approaches are designed to detect various types of refactorings including *rename*, *move method*, *extract class*, e.t.c. Also there exist specialized tools designed to detect specific refactorings such as rename refactorings. For example, such tools include `renaming detector`[54] and `REPENT`[55].

1.2.5 Limitations

As previously mentioned, the internal structure of software systems increasingly become complex and their quality degrades as they evolve to cope with changing requirements[1,2]. In this context, refactoring is usually applied to improve the internal structure and maintain the quality of an evolving system. However, deciding what refactorings to apply is often challenging especially for large and non-trivial software systems. Consequently, tools and approaches (i.e., refactoring recommenders) based on various techniques have been proposed to facilitate software refactoring. The majority of existing refactoring recommenders are based on computing source code metrics (e.g., coupling and cohesion) to rank and recommend refactoring solutions[56–59]. Such approaches allow developers to visualize the impact (in terms of code design metrics) of the recommended refactorings before applying them. Other approaches recommend refactorings based on software change history[60–62]. Generally, such approaches rely on the premise that, a piece of code that has undergone several changes in the past or has been modified recently is more likely to demand refactoring in the future. In addition, such approaches also leverage analysis of bug reports to identify buggy classes and recommend them for refactoring. Another common recommendation technique is the one that involves defining the specific objectives that a recommended refactoring has to realize[15,63]. The general idea of these approaches is to find an optimal sequence of refactorings that, for example, improves quality, minimizes code changes and preserves design semantics. Generally, such approaches are commonly referred to as search-based refactoring recommenders. Besides that, other approaches also take into account developers' feedback[64,65] and execution efforts[66] to drive refactoring recommendation.

Despite of the existing wide-range of refactoring recommenders, recent studies show that there is still a need for further improvements[26]. An empirical evidence reported in[67] suggests that, refactoring tools are not utilized as much as they could be, simply because they sometimes lack refactoing tactic mostly preferred by developers. For instance, the survey that involved 112 agile programmers showed that, despite of the availability of a refactoring tool for the refactoring to be performed, they opt to use the tool on average of 68% of the time, and the rest of the time they refactor manually. Authors further contended that, if that is the case for agile programmers who are often enthusiastic about refactoring, then refactoring tools usage is more likely to be lower for non-agile programmers[67]. Furthermore, the study conducted by Murphy-Hill et al.[26] which based on the correlation of the data from CVS commits with the refactoring tools usages datasets found that, close to 90% of refactorings are performed manually. Additionally, in a field study of refactoring practice conducted at Microsoft by Kim et al.[68] found that, on average, developers perform 86% of refactoring manually. Similar findings are also reported in[32] and[69]. In summary, the findings from these studies show that refactoring tools are underused and most refactorings are manually performed despite of having tool support. Among the common challenges reported in these studies[26,67–69] is that the tools do not fulfill developers' preferences and sometimes they do not match with how refactoring is practiced in the wild.

Understanding the practice of refactoring is therefore useful for toolsmiths to improve the existing generation of tools or evolve new tools that could help developers to effectively perform their routine maintenance tasks[32]. For example, In the study conducted by Murphy-Hill *et al.*[70] which involved a dataset that span $13,000$ developers and $240,000$ tool-supported refactoring operations to investigate how developers refactor revealed that, floss refactoring[67] is common. The study suggested that, during floss refactoring, developers apply refactoring as a means to realize a specific goal such as fixing a bug or adding a feature. In addition, developers combine refactoring with other types of code changes to maintain the code. On the other hand, developers commonly apply root-canal refactoring[67] to correct the deteriorated source code. This aspect was further confirmed by Soares *et al.*[71], that refactorings are mostly applied when fixing bugs or adding features, than in the sessions exclusively for evolving software design. Similar findings was further reported in the extensive research conducted in[26]. Silva *et al.*[34] monitored the repositories of 124 Java-based projects archived in GitHub to identify recently applied refactorings and consequently interviewed the developers to state the reasons to why they refactored the code. Authors applied a thematic analysis on the developers' responses and compiled a catalogue of 44 unique motivations for 12 common refactoring types. Generally, the study found that refactoring is mainly motivated by requirements changes and rarely by resolution of code smells. Such changes include implementation of new features and fixing bugs. For instance, for the $Extract\ Method$ refactoring, only 2 out of the 11 motivations related to resolving code smells. The study further found that, manual refactoring is still widespread. The statistics show that, around 55% of the developers applied manual refactoring. Authors concluded that, future refactoring recommenders should refocus from solutions based on resolving code smells to maintenance-task-based solutions. That might boost the adoption of tool supported refactoring. Understanding the rationale driving the application of refactorings can be useful in evolving refactoring recommenders that are tailored to the real needs of software developers[34]. Moreover, Palomba *et al.*[72] conducted an exploratory study that investigated the relationship between code changes and refactorings to explore the developers' reasons behind applying refactorings. Contrary to other studies[34,73] which investigated motivations behind applying refactorings based on interviewing developers, this study leveraged the analysis of software repositories. The settings of this study involved a dataset containing $12,922$ operations related to 28 different refactoring types that were applied on the change history of 3 open source projects. The study revealed that, developers mostly apply refactorings that target at improving comprehensibility and maintainability of code when fixing bugs. Furthermore, during implementation of new features, more complex refactorings are applied to improve code cohesion and conforming to the object oriented programming principles. Further analysis showed that, one of the main motivations for the developers to refactor code during implementation of new features is to alleviate the accumulated technical debts before implementing new code.

In summmary, several research[34,72,73] have invested in empirically investigating the developers' motivations behind applying refactorings, which generally found that refactoring is mostly motivated by the changes in the requirements. In addition, developers often apply refactorings to improve source code maintainability, comprehensibility, and prepare their systems to adapt to new requirements[6,29]. To the best of our knowledge, none of these approaches have attempted to automate refactoring recommendation based on past feature requests. In this context, we propose a set of refactoring recommendation approaches that focus on improving requirements traceability and facilitating developers during implementation of feature requests. Inclusion of such key motivations into refactoring recommenders will facilitate the approaches in making more precise and tailor made recommendations.

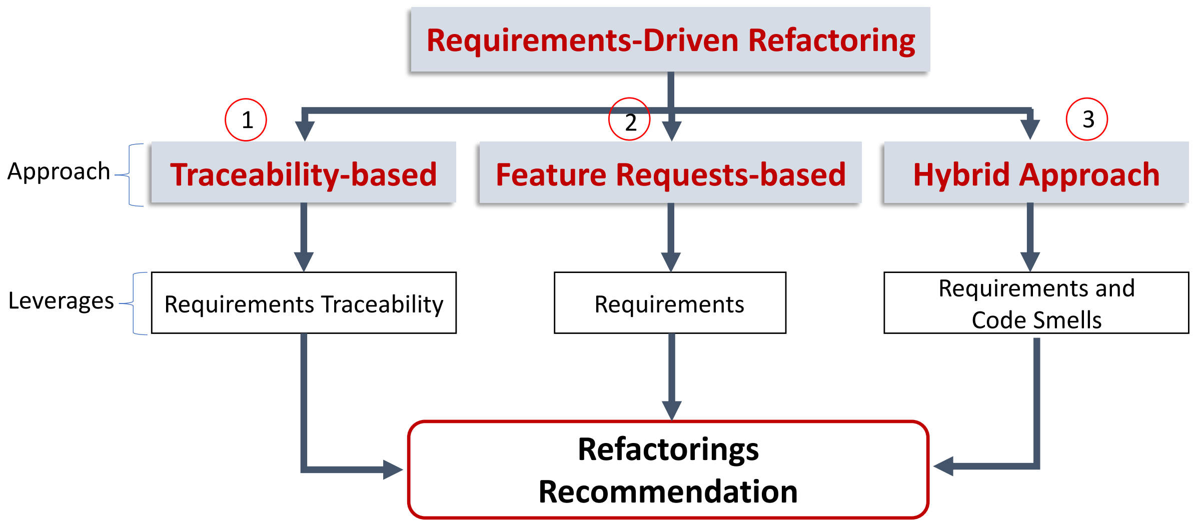

Requirements-Driven Refactoring
1
2
3
Approach
Traceability-based
Feature Requests-based
Hybrid Approach
Leverages
Requirements Traceability
Requirements
Requirements and Code Smells
Refactorings Recommendation

Figure 1.1 The Correlation of the Proposed Approaches.

1.3 Research Objectives

The main objective of this thesis is to leverage requirements information to evolve refactoring recommendation approaches that aim at facilitating developers in selecting optimal refactorings during maintenance task. This study contributes significantly to the refactoring recommendation paradigm, and further shades light on using requirements to drive refactoring. We therefore, specifically propose three requirements-driven software refactoring approaches. Notably, each of the proposed approaches can be used as a separate approach as well as in combination with the existing state-of-the-art approaches. The correlation of the proposed approaches is depicted in Fig.1.1. The key specific objectives of the thesis are summarized in the following:

1. Traceability-based software refactoring approach. The traceability between requirements (i.e., use cases) and source code is leveraged to drive refactoring recommendation. Traceability plays a significant role in software evolution and maintenance, for instance, it greatly facilitates program comprehension, generation of comprehensive

test cases, and as well as in ensuring that requirements are fully implemented. Hence, it is inevitable to ensure that traceability information is well maintained. Notably, different refactorings may have significant different impact on traceability, hence it is vital that traceability information is taken on board during refactoring recommendation. We leveraged existing state-of-the-art refactoring tools to detect code smells and suggest the candidate refactoring solutions. Consequently, for each solution, our approach computes it's impact on the traceability and source code design. A solution that will lead to the greatest improvements is recommended to the developer. On one hand, to quantify the quality of traceability we define entropy-based metrics to ensure that closely related methods (i.e., they trace to the same use cases) are well grouped into classes. This approach specifically involved the $Move\ Method$ and $Extract\ Class$ refactoring operations. On the other hand, to quantify the quality of source code design we employed the traditional cohesion and coupling metrics. Finally, the tradeoff between the traceability and design metrics is computed to rank refactorings and recommend to the developer accordingly. We evaluated this approach on a well-known iTrust (Java-based open source project) dataset since, to the best of our knowledge, it was the only application with traceability information publicly available. Our qualitative analysis involved software developers to investigate how likely they would prefer refactorings recommended by our approach. The evaluation results suggested that, our proposal significantly outperformed the state-of-the-art approaches and developers mostly preferred our recommended solutions.

2. Feature requests-based software refactoring approach. In practice, software developers receive new requirements often expressed as feature requests. To implement such features, developers often have to prepare their systems to adapt new requirements by applying refactorings. This approach focuses at facilitating developers on choosing what refactoring types to apply when implementing feature requests. We mined past feature requests from software repositories and linked them to their associated implementing source code commits from version control systems. Consequently, we employed state-of-the-art approaches to detect applied refactorings on such commits. Next, we train the machine classifiers with the history data of feature requests and

their related refactorings. The classifiers are then used to recommend refactoring types for the new feature requests. To improve the recommendation accuracy, the approach implements two classification tasks. First, a binary classifiction is applied to predict whether refactoring would be required or not. Such classifier is trained with the feature requests that need refactoring and those that do not. Second, a multi-label classification is applied on feature requests that need refactoring to recommend the required refactoring types. We selected multi-label classification since a given feature request may be associated with more than one refactoring types. We evaluated our approach on a set of 43 open source projects maintained by Apache. Statical analysis of our experiments suggests that, our proposal significantly improves the state-of-the-art.

3. A hybrid approach to recommending refactorings. The approach learns from the training set of past feature requests, associated code smells and the applied refactorings that are detected from the implementing source code commits. Code smells are leveraged to further enrich the features set that could consequently help to improve classification performance. Code smells have shown to be useful in identifying (e.g., based on smells catalogs) the special types of software refactorings needed to remedy them. Notably, the approach can learn from the training set of some applications and it can be used to predict and recommend refactorings for the feature requests associated with other applications or new feature requests of the training applications. In other words, our approach can learn some generic rules applicable to different applications, e.g., the presence of special code smells may suggest a specific type of refactoring actions, and some special words or phrases in feature request may suggest the necessity of refactorings. This approach is evaluated on 55 open source Java projects, and generally the results suggest that our proposal significantly outperforms existing approaches.

1.4 Thesis Structure

This thesis comprises six chapters which are organized as follows:

Chapter 1 introduces refactoring research background and further provides an overview of refactoring recommendation problem. Existing approaches on refactoring recommendation are summarized along with their potential challenges. Additionally, the chapter outlines the main and specific objectives of this thesis. Finally, the main contributions of the thesis are described and the chapter is concluded by outlining the structure of the thesis.

Chapter 2 presents a traceability-based software refactoring approach. The chapter shows how traceability information can be exploited to select refactorings that are potential in maintaining and improving the traceability between requirements and source code. Besides that, source code metrics are incorporated to find a tradeoff between the improvement in source code design and traceability. The chapter finally presents an experimental evaluation of the approach and discusses the results. This chapter covers the specific objective 1 of this thesis.

Chapter 3 describes a feature requests-based software refactoring approach. The chapter presents how the past history of the previously requested features and the applied refactorings can be exploited to learn the machine classifiers that subsequently will be used to predict and recommend refactorings for new feature requests. In this chapter we address the specific objective number 2 and present in detail the contribution number 2. The chapter is finally concluded with the experimental evaluation of the approach and the results discussion.

Chapter 4 presents a hybrid approach to recommending refactorings. The approach integrates the information mined from past feature requests and the related code smells to evolve a refactoring recommender. Specifically, the chapter identifies how to enrich the features set required to train the classifiers by incorporating code smells information that could consequently improve the recommendation accuracy. This chapter tackles the specific objective number 3. The chapter further evaluates the proposed approach and the comparison with the existing approaches is presented.

Chapter 5 concludes this thesis and presents possible directions for future research.

Chapter 2 Traceability-based Software Refactoring

Software refactoring has been extensively used to remedy design flaws and improve software quality without affecting its observable behaviors. For a given code smell, it is common that there exist several refactoring solutions. However, it is challenging for developers to select the best one from such potential solutions. Consequently, a number of approaches have been proposed to facilitate the selection. Such approaches compare and select among alternative refactoring solutions based on their impact on source code metrics. However, their impact on the traceability between source code and requirements is neglected although the importance of such traceability has been well recognized. To this end, we select among alternative refactoring solutions according to how they improve the traceability as well as source code design. To quantify the quality of traceability and source code design we leverage the use of entropy-based and traditional coupling and cohesion metrics respectively. We virtually apply alternative refactoring solutions and measure their effect on the traceability and source code design. The one leading to greatest improvement is recommended. The proposed approach has been evaluated on a well-known dataset. Evaluation results suggest that on up to 71% of the cases, developers prefer our recommendation to the traditional recommendation based on code metrics.

2.1 Introduction

Software systems often evolve to adapt with changing requirements. Throughout software lifetime, new requirements are added while existing ones are modified or dropped. This process continuously complicates the internal structure of the software, which consequently reduces software quality[3],[1]. As a result, software systems consistently need to be maintained to reduce complexity and to improve their internal structure through refactoring. Software refactoring focuses on improving software quality by applying changes on internal structure that do not alter the external behaviors of involved systems[9].

Software refactoring has been widely used to resolve software design flaws and to improve software quality, particularly reusability, maintainability, and extensibility[6]. The underlying principle of refactoring is reorganizing software elements such as classes, methods and variables to facilitate future extensions[3],[74]. A typical process of software refactoring is as follows: First, the pieces of code needing for improvement (commonly known as ”bad smell” or ”code smell”) are identified[9]; Second, according to the types of identified code smells, specific solutions (refactorings) are applied to remedy the smells[9].

Currently, numerous refactoring tools have been proposed to automate refactoring process, e.g., JDeodorant[23,24], iPlasma[40], Stench Blossom[75] and DECOR[17]. Moreover, most of the contemporary software development environments (SDEs) provide refactoring capabilities, e.g., Eclipse and Microsoft Visual Studio. Such great tools make software refactoring simpler and less error-prone[67]. Usually, refactoring tools may suggest multiple refactoring solutions for a given code smell identified automatically. However, the decision on which solution to be executed is left to developers. The decision to choose the best solution is often challenging[56]. To facilitate developers in making selection decision, existing tools use various techniques. The most common way is to compare and select alternative solutions based on their impact on defined code metrics[56,57,76]. Along with code metrics, other factors have been considered as well[65]. This approach takes the assumption that solutions leading to the best source code metrics are the best ones. Among the pertinent code metrics used for quantifying and ranking refactoring solutions are coupling and cohesion[23,56]. Basically, coupling refers to how parts of a design inter-depend each other, whereas cohesion refers to internal dependencies within parts of a design[77]. Usually coupling and cohesion are measured based on the fields or instance variables used by the methods within or between classes[56].

Such approaches ignore the fact that different refactoring solutions may have significantly different impact on the traceability between requirements and implementation (source code)[63]. Usually, a software system is composed of several artifacts at different levels of abstraction. Such artifacts can be related through traceability. In general, traceability is “*the ability to describe and follow the life of a requirement, in both a forwards and backwards direction*”[78]. Commonly, traceability information is maintained in the traceability matrix (TM) that is intended to show the correct links between high-level entities and low-level entities. The traceability is critical for software evolution and maintenance. To ensure correctness and completeness of the traceability information, researchers have proposed varied techniques from retrieval, assessment and maintenance of traceability links[79–82]. One of the most common traceability is the one between requirements and source code. It plays a critical role in assisting program comprehension, software maintenance, requirements tracing, impact analysis and software reuse[83]. First, traceability helps to determine if the software fulfills its intended objectives. Second, it facilitates the generation of comprehensive test cases covering all requirements. Third, while software requirements change, the traceability helps to identify parts of code that are affected by the change. Finally, it also facilitates code inspection[83]. It is therefore of vital importance to ensure that traceability information is well maintained to achieve the aforementioned benefits. Moreover, Poshyvanyk and Marcus[84] contend that, quality traceability information should accurately reflect the structure of the traced source code. For example, entities of the requirements e.g., use cases, which trace to strongly related entities of the source code e.g., methods, should also be strongly related. Such associations between the entities of the two types of artifacts, source code and requirements, are expressed by traceability links. However, existing approaches to recommending refactoring solutions consider refactorings' impact on source code metrics only and ignore their impact on traceability.

To this end, in this chapter we make full use of the traceability combining with the traditional source code design metrics in recommending refactoring solutions. A refactoring recommendation should take into account the issue of traceability to ensure that traceability links are well maintained throughout the course of software maintenance. To quantify the quality of traceability, we leverage the use of entropy of software systems[85,86]. Entropy is extensively used in software engineering for several purposes such as measuring complexity[87], disorganization[85] of classes and software testing[88]. Therefore, we compute traceability entropy as the degree of disorganization of software elements based on traceability information. Furthermore, to quantify the quality of source code design we use EP (Entity Placement) metric proposed by Tsantalis and Chatzigeorgiou[57]. EP is the ratio of the overall system cohesion over its coupling. The traceability entropy distinguishes itself from coupling and cohesion based on how the relationship between methods and classes is determined. To determine interdependencies of the design parts, coupling and cohesion consider fields usage and methods invocation, whereas traceability entropy leverages the traces between methods or classes and use cases.

The major contributions of this chapter include:

(1) A new approach to compare and select alternative refactoring solutions. To the best of our knowledge, the proposed approach is the first one that compares refactoring solutions by leveraging their impact on the traceability between requirements and source code.

(2) Evaluation of the proposed approach whose results suggest that the proposed approach outperforms the traditional code metrics based approaches. In 71% cases, developers prefer the solutions recommended by our approach against those recommended by traditional approaches.

2.2 Background

2.2.1 Requirements Traceability

Traceability has long been considered as one of the key quality attributes of a well-designed software system[78,89]. In practice, traceability is established through trace links which can be defined as a specified associations between certain pair of artifacts, one being the source artifact and the other one the target artifact[79,89]. This thesis is particularly interested with requirements traceability[78]. Requirements traceability has received a considerable attention in the literature[35,78,79,89–91], however, the studies that combine traceability and refactoring[92–94] are still scarce. Traceability has shown to benefit various areas of software engineering, e.g. software refactoring[94] and software maintenance[90]. Traceability ensures that the developed product covers and fulfills the intended requirements in both design and source code[95]. To fully utilize the promising benefits of traceability, generating[91,96] and maintaining[89] traceability information is inevitable. Traceability has been proven to be useful in software engineering by numerous researchers. For example, authors in[90], empirically investigated the usefulness of traceability in software maintenance activity of which it was found to improve maintenance quality significantly.

2.2.2 Traceability-based Refactoring

Traceability and refactoring have been proposed in the literature to support each other. For example, Eyl *et al.*[92] proposed the establishment of the so called fine granular traceability links. Their proposal targeted at establishing the traceability links between requirements and texts in the source code, which goes beyond classes and methods granularities. Moreover, they developed a tool which supports refactoring and ensures traceability links are not broken despite of the changes committed in the source code. Mahmoud and Niu[93] presented a refactoring based approach for maintaining traceability information. In their work, they argued that the lexical structure of a software system corrupts as it evolves, as a result the traceability tracks are distorted and that they can be systematically reestablished through refactoring. Faiz *et al.*[97] also advocate the work done in[93] by assessing more other types of refactoring in improving traceability. Authors in[93] and[97] proposed the use of refactoring to maintain textual information in source code which plays a vital role in traceability links retrieval. Their work is different from ours in the sense that, they employ refactoring to support requirements traceability information retrieval, whereas our work leverages traceability information to support refactoring.

Niu *et al.* in[94] proposed a requirements based approach to accurately locate where software should be refactored and what sort of refactorings should be applied. The traceability links between the requirements under development and the implementing source code were used to retrieve the to be refactored source code. Authors relied on the semantics of the requirements under implementation to recommend type of refactorings that can remedy the identified smells. The major difference of the approach in[94] with ours is on how to recommend refactoring solutions. We recommend refactoring solutions based on the requirements to source code traceability information, whereas authors in[94] employ requirements semantics to determine the threat that could hinder the implementation of the requirement and recommend refactorings to remove the threat. Moreover, the two approaches are running in different scenarios: the approach in[94] receives new requirements to be implemented as input and our approach only works on already implemented requirements. In addition to that, in[94] requirements traceability is only used for locating source code to be refactored and no assessment is done to determine how traceability is impacted by the applied refactoring.

2.2.3 Entropy Metrics

To ensure software quality, software development processes need to conform with predefined standards. Usually, to quantify the quality of development processes and the end products, metrics are commonly used. So far several metrics suits for object-orientation systems have been proposed and validated[77,98]. Entropy metric is one of the widely-used metrics in software engineering. Authors in[87] proposed a semantic class definition entropy which they used to measure the complexity of a class. Entropy was used to measure the amount of domain knowledge required to understand a class which consequently determines complexity.

Canfora *et al.*[85] proposed the use of change entropy of source code to investigate the relationship between the complexity of source code and disorganization. Authors investigated several factors that can be related to source code change entropy, particularly refactoring of which was found to affect entropy. In addition to that, Bianchi *et al.*[99] proposed the use of entropy in evaluating software quality degradation by assessing the number of links within and between abstraction models. Related work to[99] was also done by[100] and[101]. In[100] authors proposed an approach to assess object oriented software maintainability and degradation by using the combination of entropy and other metrics.

2.2.4 Code-Metrics-based Refactoring

One of the most common ways to recommend refactoring solutions is based on the computation of source code metrics[56,57]. Such approaches take the assumption that, refactorings that lead to the improved source code metrics (e.g., cohesion and coupling) are the best ones. For example, Bavota *et al.*[58] proposed an approach that recommends extract class refactorings based on the analysis of semantic and structural relationships between methods of a class to identify chains of strongly related methods. Consequently, the chains are used to create new classes with improved cohesion than the original class. Simon *et al.*[59] proposed a metrics based refactoring approach to facilitate developers in deciding where to apply which refactoring. They defined a metrical distance measure between the members of a class (i.e., attributes and methods) to identify how they are closely related. Consequently, a defined metric is used to measure cohesion of a class. For the two entities of a class, say x and y, the metric is formalized as:

$$distance(x, y) = 1 - \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|} \quad (2.1)$$

where $p(x)$ and $p(y)$ are the set of properties possessed by x and y respectively. Therefore, entities with low distances are considered more cohesive than those with higher distances. In this case, for instance, a move method refactoring is recommended if a method is closer to the entities in another class than those of the class it is currently in. Similar approaches in recommending refactorings based on code design metrics are also proposed in[22,24,76].

2.2.5 Code Smells Detection Tools

Recently, there is an increasing number of software refactoring tools available for detecting code smells. One of the widely-used smell detectors is *JDeodorant* proposed in[23,24,57]. Such tool not only detects code smells but also suggest and apply refactorings that can resolve them. For example, to detect *feature envy* smell[57], the tool applies the concept of distance between source code entities (e.g., attributes and methods). For instance, the distance between a given method and the class it belongs or another class is computed based on the number of entities referenced by such method. Therefore, the case is considered as *feature envy* smell if a method is closer to the other class than the class it is actually in. Another tool specialized in detecting move method refactoring opportunities coined as *JMove* was proposed by Valente *et al.*[22]. The tool identifies opportunities by comparing the similarity of the dependencies of a given method with the dependencies of the methods in candidate target classes. A method *m* in a class *C* is considered to be moved to a candidate target class C_t if the average similarity between the dependencies of *m* and other methods in C_t is greater than the average similarity between *m* and other methods in *C*. The similarity computation is based on Jaccard coefficient as in[57]. Bavota *et al*[1] proposed *Methodbook*, a tool to identify move method refactoring opportunities and suggests solutions. The tool leverages relational topic models (RTM), that is a probabilistic technique to model and represent documents, topics and their known relationship. Methods and classes are modelled as people and groups of people and their relationships are based on structural (i.e., methods calls) and conceptual (i.e., sharing similar comments and identifiers). Therefore, a tool uses RTM to analyze classes and suggests to move a given method to a class with the highest number of friends of such method. In case of *Large Class* smell for example, *ARIES* a smell detector proposed by Bavota *et al*[25] detects such smell and suggests to extract two or more classes from the large class. The tool uses an extract class refactoring technique that takes a large class as input and creates a method-by-method matrix of which each of it's entries represents the likelihood of two different methods to be in the same new class. The likelihood is calculated as a hybrid coupling metric between methods obtained by averaging three semantic and structural metrics (i.e., CDM, SSM, and CSM). Moreover, PMD[102] and Checkstyle[103], for instance, they detect *Large Class* smell by leveraging NLOC metric as a detection strategy. The tools allow a user to define a threshold for each of the exploited metrics. Other tools and their details (i.e., smells they detect and their implementation strategy) are summarized in[39]

In the preceding paragraphs we just highlighted some few selected tools and their implementation heuristics, however, an extensive list of code smells detection tools can be found in[43]. According to the study conducted by Fernandes *et al*[43], there are 84 code smells detection tools proposed in the literature, 29 of them are publicly available online. Additionally, altogether, such tools targeted to detect 61 different types of code smells and they based on more than six detection techniques. Besides that, the tools support a variety of programming languages including Java, C#, C, and C++.

2.3 Problem Statement

Most of the existing approaches recommend refactorings based on the improvement of code quality which is often quantified by code metrics[15]. Moreover, Ouni *et al.*[15] suggest that depending on design metrics only may not be enough as it is crucial to preserve the basis on why and how source code elements are grouped when applying refactorings. Besides, the existing approaches ignore the impact of such refactorings on the traceability between requirements (i.e., use cases) and source code, although the importance of such traceability has been well recognized. In object-oriented systems, the source code elements (e.g. Methods and Classes) are created to handle specific functionalities and they can be linked to use cases they implement through traceability. During the initial implementation of the system when objected-oriented principles are adhered to, the source code elements are well grouped based on the functionalities to realize the implementation of the use cases. However, in the course of system evolution and maintenance, the programs undergo several modifications (refactorings) which could consequently affect their traces to use cases as a result of improper code elements grouping. As a result, during refactorings recommendation, there is need to consider traceability aspects as well which can also infer how code elements relate and how well they can be grouped rather than relying on code metrics only.

Therefore, given the suggested refactoring solutions R_s for each code smell C_s, the refactoring recommendation problem can be formally defined as quantifying the impact of each solution from R_s on traceability and source code design. Consequently, the solutions are ranked based on how they improve the traceability and code design and recommend to developers the solution that will lead to greatest improvement.

2.4 Motivating Example

Fig. 2.1 depicts a typical example drawn from iTrust, an open-source application for maintaining patients electronic health records[104]. This example illustrates the design fragment consisting of three classes; $GetVisitRemindersAction$, $TransactionDAO$ and $VisitRemindersDAO$. According to the code smell detection techniques proposed in[75] and[23], the class $GetVisitRemindersAction$ is detected as a $God\ Class$. A $God\ Class$ is a large and complex class which tends to perform too much work[105]. Usually, this type of a smell is alleviated by extracting the data and functionalities of the $God\ Class$ to other collaborating classes or new classes through a $Move\ Method$ or $Extract\ Class$ refactoring respectively[23].

Here we consider a suggested $Move\ Method$ refactoring solution which is as well proposed in[22]. The solution moves the method $getVisitReminders()$ from the class $GetVisitRemindersAction$ to $TransactionDAO$. This refactoring is able to remove the detected code smell and reduce the number of the functionalities of the $GetVisitRemindersAction$ class. However, from the perspective of the proposed approach, there exist other target classes including $VisitRemindersDAO$ which can receive what is extracted from $GetVisitRemindersAction$ class as well. According to the traceability information, the functionalities (use cases) which are performed by the method

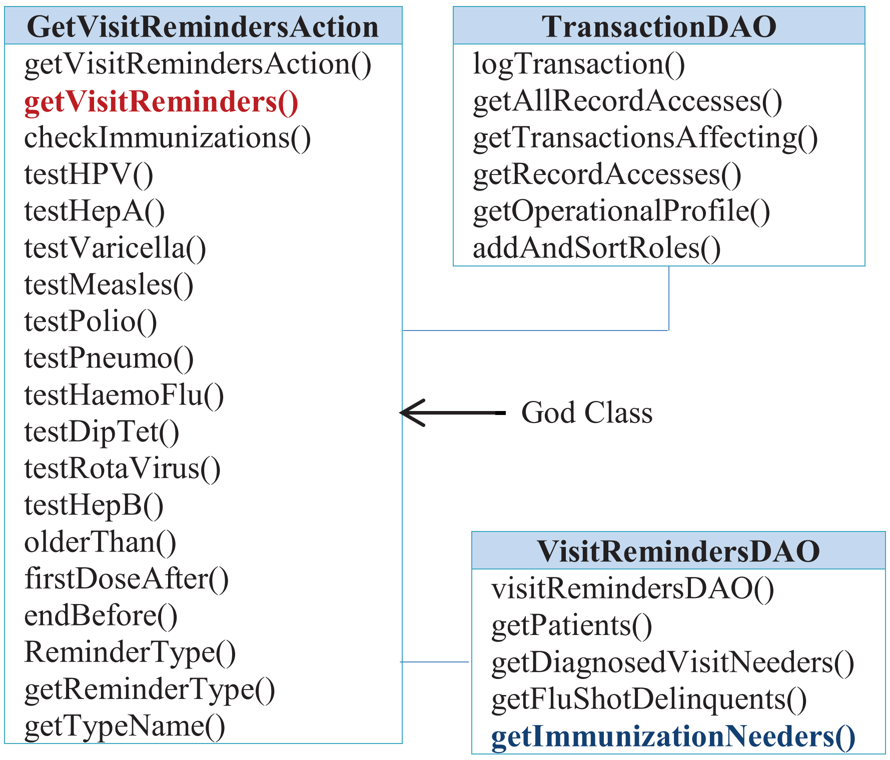


Figure 2.1 iTrust Classes Design Snippet.

$getVisitReminders()$ are more similar to that of the methods in $VisitRemindersDAO$ than $TransactionDAO$ classes. Consequently, the proposed approach recommends a solution that moves $getVisitReminders()$ method to the $VisitRemindersDAO$ class as it improves traceability. To asses the impact of these two refactorings on the source code design quality, an entity placement (EP) metric is computed as proposed by Tsantalis and Chatzigeorgiou[57]. The suggested solution by the proposed approach achieves better EP than the other refactoring solution. Moreover, we noted that, in the other case when the same class $GetVisitRemindersAction$ was refactored to remove a $Feature\ Envy$ code smell the same techniques suggested to move the method $getImmunizationNeeders()$ to the envied class $VisitRemindersDAO$. Furthermore, based on the textual similarity between these two classes it is evident that they are closely related. Consequently, $VisitRemindersDAO$ is more likely to be an optimal target class than $TransactionDAO$. This example is inspired by[15], where authors suggested that, to ensure quality improvement it is also important to consider additional objectives rather than solely relying on structural metrics.

2.5 The Proposed Refactoring Recommendation Approach

In this section, we first give an overview of the approach, followed by details of each step of the approach.

2.5.1 Overview

An overview of the proposed approach is statically analysed by Fig. 2.2. The approach works as follows. First, the source code under refactoring is scanned by a refactoring tool to uncover code smells (refactoring opportunities). The refactoring tool also suggests a number of potential refactoring solutions for each code smell. Second, the traceability information associating with the scanned source code is accessed to determine how the source code entities are linked to use cases. Third, suggested solutions are virtually applied to measure their impact on the traceability and source code design by using entropy and code metrics respectively.

Finally, appropriate recommendations are made based on the improvement of entropy and code metrics. The key steps of the approach are explained in detail in the following subsections. Since the traceability information that we have used in our experiment is expressed at the method granularity (use case to method), we considered code smells whose refactoring solutions directly affect methods entirely. This is because we can easily assess the effect of each refactoring on the traceability. The code smells which are therefore considered and can be supported by the used refactoring tools are $God\ Class$ and $Feature\ Envy$. The considered corresponding remedying solutions to such smells which support the movement of methods and can be automatically applied by the employed refactoring tools are $Extract\ Class$ and $Move\ Method$ respectively.

2.5.2 Traceability

Requirements traceability as its name suggests, is an ability to trace requirements to other artifacts such as source code. Requirements are implemented by source code entities. Information showing the mapping between requirements and source code entities is often put in a matrix called traceability matrix (called TM for short). In TM, use cases are mapped to

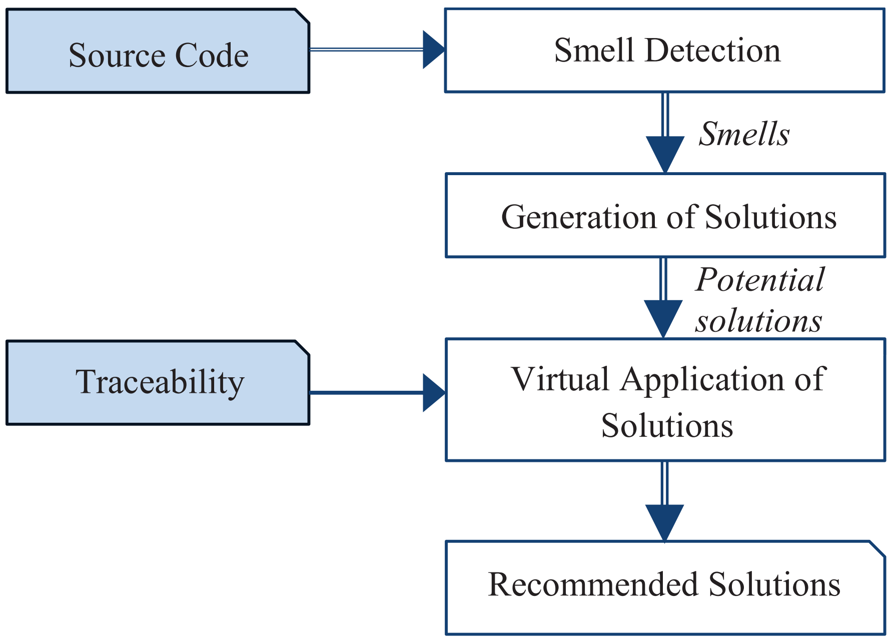


Figure 2.2 Traceability-enabled Refactoring Recommendation Framework

classes or methods that they are related to. Table I depicts a sample of TM. Traceability at the method granularity can be defined as:

- Let M be a set composed of methods m_1, m_2,...,m_k.
- Let C be a set composed of classes c_1, c_2,...,c_h. Each class from C is uniquely composed of one or more methods from M.
- Let U be a set composed of use cases, u_1, u_2,...,u_n. Each use case from U is mapped to one or more methods which belong to one or more classes in C. This implies that, a use case in U can be mapped to one or more classes in C.
- Let $L(U, M)$ be a matrix mapping U to M. $L_{i,j} = 1$ if method m_j implements use case u_i. Otherwise $L_{i,j} = 0$, where, $u_i \in U$ and $m_j \in M$.

Table 2.1 depicts a typical traceability matrix drawn from iTrust[104]. The matrix shows the mappings between methods from two classes ($EditPersonnelAction$ and $PersonnelDAO$) to five use cases ($U4$, $U6$, $U10$, $U29$ and $U33$). The values 1 and 0 indicate whether a method traces to a given use case or not, respectively. A method can trace to the same use case only once but it can trace to more than one use case at a time.

Table 2.1 Method to Use Case Traceability Matrix

Method name	U4	U6	U10	U29	U33
EditPersonnelAction. getPid	1	0	0	0	0
EditPersonnelAction. updateInform	1	0	0	0	0
PersonnelDAO. editPersonnel	1	0	0	0	0
PersonnelDAO. getPersonnel	0	1	1	1	1
PersonnelDAO. getAllPersonnel	0	0	0	0	1
PersonnelDAO. getName	0	0	0	1	0

2.5.3 Entropy

Entropy was first proposed by Shannon[106] to measure the amount of information produced by the source. Shannon's definition of entropy determines number of bits required in identifying information distribution[85,106]. Shannon[106] computed entropy as follows:

$$H_{\mathrm{n}}(P) = -\sum_{k=1}^{n} p_{\mathrm{k}} \log_2 p_{\mathrm{k}} \tag{2.2}$$

where $p_{\mathrm{k}} \geq 0(k = 1, 2...., n)$ denotes the probability of occurrence for the k^{th} element and $\sum_{k=1}^{n} p_{\mathrm{k}} = 1$ and P is the distribution of information.

For a distribution P, its entropy is maximized when all elements have the same probability of occurrence, i.e., $p_{\mathrm{k}} = \frac{1}{n}$, for $k = 1, 2...., n$. But when one element e.g., j has a maximal probability of occurrence (i.e., p_j=1), $H_{\mathrm{n}}(P)$ is minimized. The lower the entropy is, the smaller uncertainty to identify the distribution of information from the source. Consequently, lower entropy is often preferred.

Entropy is extensively used in software engineering as *a metric to assess the degree of disorder in software system structure*[99]. Entropy covers all the components of a software system and their traceability relationship[99]. In the following subsections, we slightly adapt the definition to measure the quality of traceability by assessing the degree of randomness of mapping between source code entities and use cases.

2.5.3.1 Class Traceability Entropy

- Let c be a class composed of methods m_1, m_2,..., m_k.

- Let U be a set of use cases, u_1, u_2,...., u_n that are connected to one or more methods in c.
- Let $L(U, c)$ be all the traceability links which connect each use case in U to one or more methods in c.
- Let $P(u_i, c)$ be the ratio of traceability links connecting use case u_i and class c contributed to $L(U, c)$, with $i = 1, 2, ...n$.

So, each use case from U is directly connected to one or more methods in c via some traceability links from $L(U, c)$. Thus, $P(u_i, c)$ is given as:

$$P(u_{\mathrm{i}}, c) = \frac{L(u_{\mathrm{i}}, c)}{\sum L(U, c)} \tag{2.3}$$

where $L(u_i, c)$ is the number of traceability links starting from use case u_i to class c. $P(u_i, c)$ determines at what extent a use case is traced by a class. If use cases in U share equal ratio, i.e., $P(u_1, c)$= $P(u_2, c)$, ... = $P(u_n, c)$, the class entropy is maximal. For every class, the sum of the ratios is always equal to 1, i.e.,

$$\sum_{i=1}^{n} P(u_{\mathrm{i}}, c) = 1$$

We finally define entropy of class c, $E(c)$, as:

$$E(c) = -\sum_{i=1}^{n} P(u_{\mathrm{i}}, c) \log_2 P(u_{\mathrm{i}}, c) \tag{2.4}$$

Entropy of a class is minimal, i.e., $E(c) = 0$, when a class traces to one use case only. The smaller the entropy is, the better a class is organized. Methods which are highly related are likely to trace to the same use case(s). When these methods are grouped in the same class will result to a well-organized class which is composed of strongly related methods. Therefore, a class implements less number of distinct use cases which consequently reduces entropy. In other words, cohesive methods are grouped in the same class. However, some methods trace to more than one use case and therefore this yields to some reasonable amount of coupling between classes. The entropy of the class becomes high when its methods trace to different use cases. This in turn increases cross mappings of the traceability links between classes and use cases.

2.5.3.2 Use Case Traceability Entropy

We further quantify the effect of the solution by assessing the entropy of use cases traced by classes. The definition is also straightforward as that of a class:

- Let S be a system composed of classes c_1, c_2,..., c_n.
- Let u be a use case connected to one or more classes in S.
- Let $T(u, S)$ be all the traceability links that connect use case u to classes in S.
- Let $R(u, c_i)$ be the ratio of traceability links connecting use case u and class c_i contributed to $T(u, S)$, with $i = 1, 2, ...n$.

So, a use case u is directly connected to a class from S with some links from $T(u, S)$. Thus $R(u, c_i)$ is given as:

$$R(u, c_\mathrm{i}) = \frac{T(u, c_\mathrm{i})}{\sum T(u, S)} \tag{2.5}$$

where $T(u, c_i)$ is the number of traceability links starting from use case u to class c_i. $R(u, c_i)$ determines at what extent a use case is traced by a class. If the classes in S share equal ratio, i.e.; $R(u, c_1)$= $R(u, c_2)$,... = $R(u, c_n)$, the use case entropy is maximal. For every use case, the sum of the ratios is always equal to 1, i.e.;

$$\sum_{i=1}^{n} R(u, c_\mathrm{i}) = 1$$

We finally define entropy of use case u, $E(u)$, as:

$$E(u) = -\sum_{i=1}^{n} R(u, c_\mathrm{i}) \log_2 R(u, c_\mathrm{i}) \tag{2.6}$$

Entropy of a use case is minimal, i.e., $E(u) = 0$, when a use case is traced by one class only. The smaller entropy implies a use case is implemented by methods which are well grouped into classes. Thus, a use case will tend to trace to fewer number of different classes. Consequently use case entropy is reduced. The entropy becomes higher when a use case is implemented by different methods in different classes, which implies the greater difficulty to follow the traceability links from use case to source code.

2.5.3.3 System Traceability Entropy

To compute traceability entropy of the whole system, we summarize the traceability entropy of classes and use cases.

- Let S be a system composed of classes $c_{\mathbf{1}}$, $c_{\mathbf{2}}$,..., $c_{\mathbf{n}}$.
- Let U be a set of use cases, $u_{\mathbf{1}}$, $u_{\mathbf{2}}$,...., $u_{\mathbf{j}}$ traced by one or more classes in S.

Then entropy of the whole system given as $E(S, U)$ is defined as:

$$E(S, U) = \frac{\sum_{i=1}^{n} E(c_{\mathrm{i}}) + \sum_{j=1}^{k} E(u_{\mathrm{j}})}{n + k} \tag{2.7}$$

where:

- $\sum_{i=1}^{n} E(c_{\mathrm{i}})$ is the sum of class traceability entropy of all classes in the system.
- $\sum_{j=1}^{k} E(u_{\mathrm{j}})$ is the sum of use case traceability entropy of all use cases traced by the system.
- n and k are the total number of classes and use cases in the system respectively.

In Equation (2.7), the sum of the classes and use cases entropy is divided by the total number of classes and use cases, n and k respectively. That is because some refactorings like $Extract\ Class$ may increase the number of classes. Consequently, the sum of the classes entropy or use cases entropy might increase as well. We therefore take the average (dividing by n and k) to consider possible changes in n.

Less entropy of the system implies better organization of methods into classes and less cross mapping between use cases and classes. This leads to a well-structured and easy-to-follow traceability between source code and use cases.

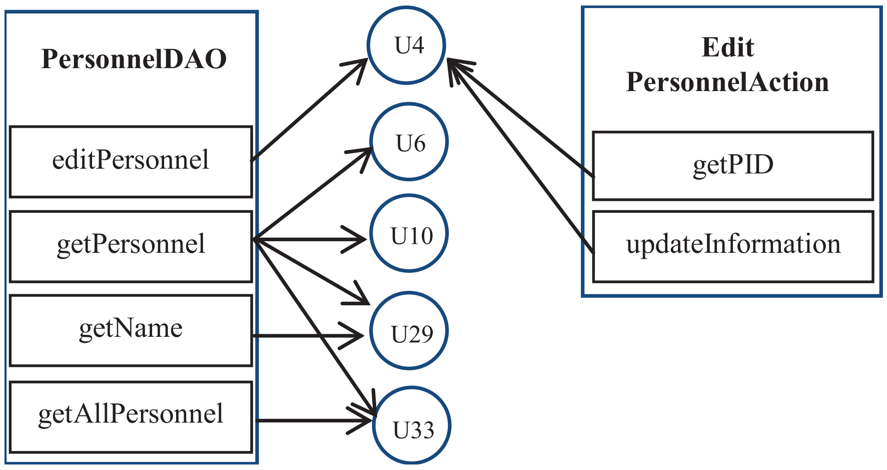


Figure 2.3 Traceability Links Before Refactoring

2.5.3.4 Computational Example

Consider the traceability matrix depicted in Table 2.1, which is the typical example drawn from iTrust. The matrix is composed of two classes of which altogether implement five use cases. The traceability links between methods and use cases before refactoring are as depicted in Fig. 2.3. Based on the matrix, to compute traceability entropy of class $EditPersonnelAction$ and $PersonnelDAO$ before refactoring, say $E(EditPersonnelAction)$ and $E(PersonnelDAO)$ respectively, we consider the links contributed by each use case traced by the classes:

$$E(EditPersonnelAction) = \frac{2}{2} \times \log_2 \frac{2}{2} = 0$$

$$E(PersonnelDAO) = \frac{1}{7} \times \log_2 \frac{1}{7} + \frac{1}{7} \times \log_2 \frac{1}{7} + \frac{1}{7} \times \log_2 \frac{1}{7} + \frac{2}{7} \times \log_2 \frac{2}{7} + \frac{2}{7} \times \log_2 \frac{2}{7} = 2.236$$

Suppose a refactoring that moves a method $editPersonnel$ from class $PersonnelDAO$ to $EditPersonnelAction$ class needs to be executed. Since the use cases will also be affected by the refactoring operation then their entropy should be computed as well. Thus we can determine change in traceability entropy before and after refactoring. But the use cases $U6$, $U10$, $U29$ and $U33$ trace to one class only, see Fig. 2.3, therefore their entropy is zero. So we only need to compute entropy of use case $U4$:

$$E(U4) = \frac{1}{3} \times \log_2 \frac{1}{3} + \frac{2}{3} \times \log_2 \frac{2}{3} = 0.918$$

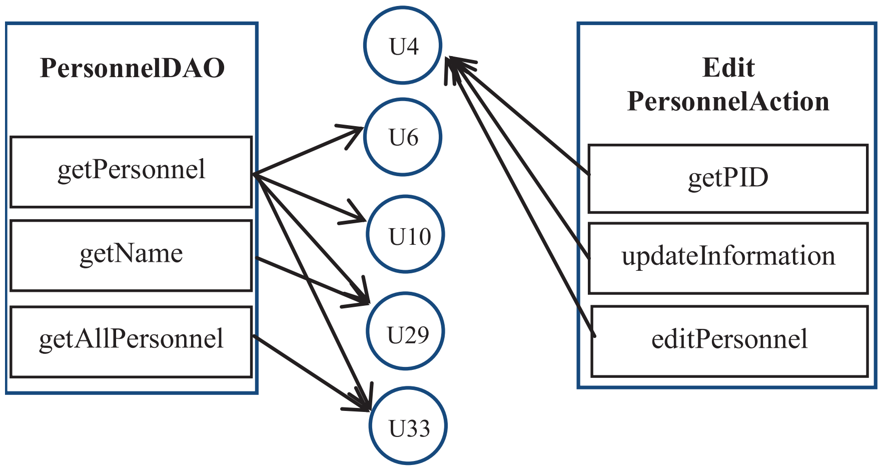


Figure 2.4 Traceability Links After Refactoring

Therefore, traceability entropy before refactoring, $E(S,U)$, is computed by summarizing the traceability entropy of classes and use cases as shown in Equation (2.7). Thus, $E(S,U) = 0.451$.

To analyse the effect of the applied refactoring on traceability entropy, we compute entropy after refactoring as well. As depicted in Fig. 2.4, after refactoring, all use cases are traced by only one class, therefore their entropy is zero. Moreover, class $EditPersonnelAction$ is still tracing to one use case only, consequently, its traceability entropy is still equal to zero. We thus compute traceability entropy of $PersonnelDAO$ class after refactoring, $E'(PersonnelDAO)$:

$$E'(PersonnelDAO) = \frac{1}{6} \times \log_2 \frac{1}{6} + \frac{1}{6} \times \log_2 \frac{1}{6} + \frac{2}{6} \times \log_2 \frac{2}{6} + \frac{2}{6} \times \log_2 \frac{2}{2} = 1.918$$

Finally, traceability entropy of the system after refactoring, $E'(S,U)$, is computed by summarizing the traceability entropy of classes and use cases after refactoring, as indicated in (2.7). Thus, $E'(S,U) = 0.274$.

This example has shown how traceability entropy of the classes and use cases are affected by the refactoring. In the example, when methods which implement similar use case are grouped together, entropy of a class is reduced. Similarly, entropy of the use case is reduced when a use case is traced by fewer classes.

2.5.4 Coupling and Cohesion Metrics

Considering that methods are one of the entities of the class in an object-oriented system, thus moving methods between classes of the system has direct impact to the design. Since coupling and cohesion of the class mainly depend on how relevant methods are placed in classes[57], they are therefore useful in determining how well the system is designed. Coupling and cohesion are among the most commonly used design quality metrics for object oriented systems[107]. Cohesion is generally defined as the degree of similarity between entities within a class. The more the entities of a class are similar to each other the more a class is considered to be cohesive[108]. Often the cohesion of a class is measured based on how its methods relate to each other. The similarity of methods is determined by the degree of sharing of instance variables within a class. Moreover, coupling has been defined as the degree of interrelatedness between classes[1]. Coupling generally indicates how the entities of one class are closely related to the entities of another class. A well designed class shall therefore exhibit the widely accepted rule of high cohesion and low coupling. To find the trade-off between cohesion and coupling, Tsantalis and Chatzigeorgiou[57] proposed $Entity\ Placement$ (EP) metric which is the ratio of the distances of the entities of a class (cohesion) to the distances of the entities of another class from that class (coupling). EP has been proved to be effective in measuring how well components are distributed[57]. Consequently, the proposed approach employs EP as well. EP is automatically computed by the refactoring tool (see Section 2.5.5.1) and it is one of the parameters in our refactoring recommendation. The EP of a class C is therefore defined as:

$$EP_{\mathrm{C}} = \frac{\frac{\sum_{e_{\mathrm{i}} \in C} distance(e_{\mathrm{i}}, C)}{|entities \in C|}}{\frac{\sum_{e_{\mathrm{j}} \notin C} distance(e_{\mathrm{j}}, C)}{|entities \notin C|}} \tag{2.8}$$

where e denotes an entity of the system and $distance(e_{\mathrm{i}}, C)$ indicates the distance from entity i to class C. The computation of a distance from an entity to a class is explained in detail in[57].

The EP of a system S which is the weighted metric for all classes in a system is further defined as:

$$EP_{\mathrm{S}} = \sum_{C_{\mathrm{i}}} \frac{|entities \in C_{\mathrm{i}}|}{|allentities|} EP_{\mathrm{C_i}} \tag{2.9}$$

The value of EP closes to zero implies proper allocation of entities into classes.

2.5.5 Refactoring Recommendation

Our recommendation approach first detects the two classical smells: $Feature\ Envy$ and $God\ Class$. For the $Feature\ Envy$ smell, a method m located in incorrect class C_s is detected and then the approach suggests moving such method to the class it mostly deserves. A refactoring to move a method m to each of the suggested target classes C_t is virtually applied and then we compute two metrics: (a) the resultant traceability entropy after moving m to C_t; and (b) the entity placement metric after moving m to C_t. Based on these two metrics the effect of refactoring is then computed as shown in Equation (2.10). Therefore, C_t will be recommended as the most suitable class to receive a method m if its refactoring will lower the value of the two metrics.

For the $God\ Class$ smell, an optimal extract class refactoring is evaluated by analyzing the sets of methods m_s to be moved to form a new class C_n. An extract class refactoring that moves the methods of each set is virtually applied. Then the two metrics described in the previous section are computed. Therefore, methods m_s will be recommended for extraction to class C_n if it will reduce the value of the metrics.

Generally, the key steps before recommending solution are smell detection, analyzing traceability information and traceability entropy computation and recommendation. Next we elaborate these key steps.

2.5.5.1 Code Smell Detection

The first step in recommending refactoring is to detect code smells. As an initial attempt, the proposed approach focuses on two common smells: $God\ Class$ and $Feature\ Envy$. Trifu and Marinescu[105] define $God\ Class$ as “large, complex and non-cohesive class which accesses many foreign data and tends to perform too much work while delegating very little”. Often, $God\ Class$ is the result of the violation of the principle that a class should implement one concept only[76]. Furthermore, $Feature\ Envy$ is "a design flaw that applies to a method that seems more interested in the data of other classes than that of the class it is currently located"[105]. This method is often tightly coupled to the other class than its own[39]. Our approach employs the state-of-the-art refactoring tools (JDeodorant, JMove and ARIES) which have the capabilities of detecting code smells and suggesting solutions. Source code unit under analysis is scanned by the refactoring tools to uncover code smells. For each identified smell, the tools suggest refactoring solutions and rank them based on how they improve code design quality. We collect such potential solutions as our candidate solutions. Notable, such refactoring solutions, if applied, can preserve the external behaviors of the subject software applications[23][22]. The employed refactoring tools, e.g., JDeodorant, have employed rigorous prediction checking algorithms to guarantee that the proposed potential solutions can preserve the behavior of the system[35]. The number of possible refactoring solutions depends on the nature of the code smell and the possible ways of transforming the code to remove the identified smell. Refactoring solutions are therefore presented as the transformation of source code and their relevant code metrics values.

2.5.5.2 Analyzing Traceability Information

The second step is to access the traceability information of the source code detected with code smells in the first step. Traceability information which is often presented in traceability matrix (TM) shows the traceability links connecting source code unit and use cases. At this stage we aim at establishing the number of traceability links connecting each method of the class under refactoring to use cases. This information is used in computing class traceability entropy. In addition to that, the number of traceability links connecting each use case (that will be affected by the suggested solutions in step one) to methods are counted as well. This information is used in computing use case traceability entropy. Moreover, the applied refactoring solutions such as $Move\ Method$ and $Extract\ Class$ lead to movement of methods between classes and addition of new classes respectively. These changes affect the number of traceability links between classes and use cases. Therefore, after each refactoring the traceability information is updated to accommodate the changes and ensure accurate traceability entropy computation in the subsequent refactorings.

2.5.5.3 Entropy Computation and Recommendation

The solutions suggested for each identified code smell are then virtually applied to determine their effects on traceability based on traceability entropy. Since the applied refactorings involve movement of methods between classes, this affects the number of traceability links between classes and use cases. Consequently traceability entropy is also affected. Traceability entropy for each solution is then computed as shown in the example in Section 2.5.3.4. To recommend solution, traditional object-oriented design metrics are also taken into consideration. We specifically considered coupling and cohesion metrics which are computed by the employed refactoring tool (JDeodorant) as EP (Entity Placement) values. EP is a ratio of the overall system cohesion over its coupling, see Section 2.5.4. The value of EP is used to determine the effect of each refactoring to the code design without actually applying it. So prior to execution of a particular refactoring the initial values for traceability entropy as shown in Equation (2.7), here denoted as EN, and EP of the system are computed. These values are noted as $preRefactoring$. Then virtual application of each of the suggested refactoring is applied to deduce the values of EN and EP after refactoring noted as $postRefactoring$. To compute the sum of the percentage of change between $preRefactoring$ and $postRefactoring$ for both EN and EP of the system after applying refactoring solution s we devise a metric, $Effect\ of\ Refactoring$, notated as $ER(s)$ which is defined as:

$$ER(s) = \frac{EP_{\mathrm{i}} - EP_{\mathrm{s}}}{EP_{\mathrm{i}}} + \frac{EN_{\mathrm{i}} - EN_{\mathrm{s}}}{EN_{\mathrm{i}}} \tag{2.10}$$

where EP_{i} and EN_{i} are the initial ($preRefactoring$) values of EP and entropy of the system respectively. EP_{s} and EN_{s} are the resultant EP and entropy ($postRefactoring$) values of a system if the refactoring solution s will be applied. The values of EP and EN are used in Formula 2.10 in order rank refactoring solutions based on the tradeoff of improvement of source code design and traceability.

The value of $ER(s)$ which is finally used to rank the refactoring solutions can either be positive or negative. Positive values indicate decrease in the resultant values of EP and entropy. Negative values indicate an increase in the resultant values of EP and entropy. The solution that will lead to the higher value of $ER(s)$ is recommended to the developer.

2.5.5.4 Recommendation Algorithm

Algorithm 1 depicts the proposed $Move\ Method$ recommendation algorithm. An envy method m of the system S which is implemented in the class C_s is first verified if it appears in the traceability matrix TM. This is because the number of traceability links between method m and the use cases will be used later to compute the traceability entropy which consequently used in ranking the candidate refactorings. The list of classes C_T which are envied by a method m are then detected using a smell detection technique presented in Section 2.5.5.1. Then for each class C_i from the list of candidate target classes C_T, the effect of refactoring metric (ER) is computed (line 9) if a method m will be moved to C_i. A function $computeER(m, C_i)$ computes the effect of refactoring based on traceability and code metrics as shown in Equation (2.10). The $ERList$ holds the computed values of ER for each candidate move method refactoring solution. In line 12, a class with the greatest ER value in $ERList$ is recommended as an optimal target class C_t to receive a method m. Finally, the recommendation to move a method m to a class C_t is given to the developer.

Algorithm 1 Move Method Recommendation

```
Input: Target system $S$, Traceability matrix $TM$
Output: Recommended refactorings
1: $Recommendations \leftarrow \emptyset$
2: for all $method\ m \in S$ do
3:     if $m$ $appearsIn$ $TM$ then
4:         $C_s = getSourceClass(m)$
5:         $C_T = getEnviedClasses(m)$
6:         $ERList \leftarrow \emptyset$
7:         for all $C_i \in C_T$ do
8:             $ER_i \leftarrow 0$
9:             $ER_i = computeER(m, C_i)$
10:            $ERList = ERList + ER_i$
11:        end for
12:        $C_t = recommendClass(m, ERList)$
13:    end if
14:    $Recommendations = Recommendations + moveMethod(m, C_t)$
15: end for
```

To recommend an $Extract\ Class$ refactoring solution, a slightly different algorithm is proposed as presented in Algorithm 2. A class c which has been identified as the $God\ Class$ is first verified to check if its methods m appear in the traceability matrix TM. The set M_E of possible methods that can be extracted from class c is suggested by the smell detection technique presented in Section 2.5.5.1. Then, for each set of candidate methods m_i to be extracted to a new class C_n, the effect of refactoring ER is computed by the function $computeER(m_i, C_n)$. In line 11, a set of methods m_s which attains the greatest value of ER in $ERList$ is recommended to be extracted to a new class C_n.

Algorithm 2 Extract Class Recommendation

Input: Target system S, Traceability matrix TM
Output: Recommended refactorings

```
1: $Recommendations \leftarrow \emptyset$
2: for all $class\ c \in S$ do
3:     if $methods\ m \in c$ and $m$ $appearsIn$ $TM$ then
4:         $M_E = getExtractableMethods(c)$
5:         $ERList \leftarrow \emptyset$
6:         for all $m_i \in M_E$ do
7:             $ER_i \leftarrow 0$
8:             $ER_i = computeER(m_i, C_n)$
9:             $ERList = ERList + ER_i$
10:        end for
11:        $m_s = recommendMethods(m, ERList)$
12:    end if
13:    $Recommendations = Recommendations + extractClass(m_s, C_n)$
14: end for
```

Consequently, the recommendation to extract m_s from class c to class C_n is suggested to the developer. To reduce the time complexity and the manual overhead of the proposed approach, we stored the traceability information in a database to facilitate easy querying and manipulation of the information. Moreover, we devised a tool to efficiently compute the traceability entropy.

2.6 Evaluation

2.6.1 Research Questions

In this section we present the evaluation of the proposed approach. The evaluation investigates the following research questions.

- $RQ1$: Does the proposed approach outperform traditional code metrics-based approaches concerning the traceability between requirements and source code? If so, to what extent?
- $RQ2$: How often do developers prefer solutions recommended by our approach against those recommended by the traditional code metrics-based approaches?

Investigation of $RQ1$ should reveal to what extent the solutions recommended by the proposed approach can improve traceability compared against the solutions recommended by the traditional metrics-based approaches. $RQ2$ validates the proposed approach to assess its feasibility from the developers' point of view.

2.6.2 Dataset and Tool Support

We evaluated our approach on a well-known iTrust dataset[104]. iTrust is an open-source application for maintaining patients electronic health records created and used as students' educational project at North Carolina State University. The application comes with packaged source code and traceability matrix which shows the traceability links between use cases and source code at the method granularity. iTrust is specifically chosen because it maintains traceability links which trace down to Java methods. The used version of iTrust contains 365 classes, and the traceability matrix contains 50 use cases and the total of 444 use case to methods traceability links. The total of 103 code smells were identified for refactoring. Out of which, 18 code smells $(17.5\% = 18/103)$ were not considered for evaluation since their solutions recommended by both approaches (JDeodorant, JMove, ARIES and our approach) were the same. We therefore evaluated the proposed approach on 85 code smells $(82.5\% = 85/103)$ whose solutions recommended by the approaches were different.

To automatically detect code smells and apply refactorings we use three Eclipse plug-ins, JDeodorant[23,24], JMove[22,35] and ARIES[25] as the refactoring tools of our choice. We note that, JDodorant can detect and recommend refactorings for both $Feature\ Envy$ and $God\ Class$ code smells, whereas JMove and ARIES can detect and recommend refactorings only for $Feature\ Envy$ and $God\ Class$ respectively. Also the evaluation compares our approach against these techniques in recommending solutions because they are the state-of-the-art refactoring tools that use structural metrics for refactoring recommendation. Particularly, JDeodorant is among the powerful and pertinent refactoring tools currently in use and has been empirically validated in several studies[41–43]. JDeodorant is actively developed, well maintained and available. Furthermore, we specifically choose these tools because they are capable of detecting smells, suggesting refactoring solutions and apply selected solutions automatically. Unlike some other refactoring tools which are just smell detectors and are not capable of suggesting solutions or apply them automatically. We note that, these tools do not use traceability information in their recommendation, however we employ them as baselines as they use code metrics for recommendation like our approach. To the best our knowledge the proposed approach is the first attempt to combine traceability and code metrics to recommend refactorings. In addition, traceability information was stored in the database with tuples for all use cases and the methods they trace. Moreover, we developed a tool to facilitate efficient computation of traceability entropy.

2.6.3 Results and Analysis

We conducted our evaluation on 85 (49 for $Feature\ Envy$ and 36 for $God\ Class$) identified code smells in iTrust. Each smell was refactored by both solutions recommended by our approach and the baseline approaches. After refactoring each smell, the system traceability entropy was computed.

To answer research question $RQ1$, we investigated the change in traceability entropy of the system after resolving each code smell. Each individual solution was applied and assessed how it affected entropy. The result shows that solutions recommended by our approach outperform those suggested by the baseline approaches. On 69 out of 85 code smells $(81\% = 69/85)$, the solutions recommended by our approach lead to reduced system entropy. We also note that on 22 out of 85 code smells $(25\% = 22/85)$, the solutions suggested by JDeodorant lead to reduced system entropy. We further note that, 31% and 29% of the solutions recommended by JMove and ARIES respectively reduced system entropy. As a result, on average the solutions by JDeodorant, ARIES and JMove resulted to an output system with entropy increased by 3.6%, 3.1% and 2.7% respectively, whereas solutions recommended by our approach outputted the system with entropy reduced by 6%. The results generally show that our approach significantly reduced system entropy compared to the baseline approaches.

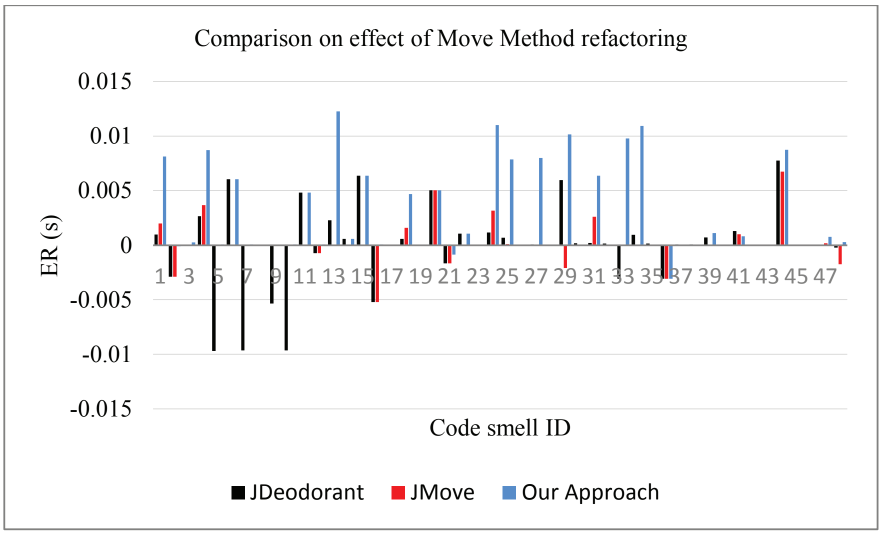


Figure 2.5 Comparison on Effect of Move Method Refactoring

We further investigated the change in $ER(s)$ as shown in Equation (2.10) which computes the percentage of change in both EP and entropy. The higher the value of $ER(s)$ indicates reduced in entropy and better design of a system in terms of coupling and cohesion which is summarized by EP. The results of $ER(s)$ of the system after applying Move Method and Extract Class refactorings on code smells are as depicted in Fig. 2.5 and Fig. 2.6 respectively. The results generally show that to a large extent the solutions recommended by our approach outperform those suggested by the baseline approaches in resulting to higher values of $ER(s)$. For the Move Method refactorings as shown in Fig. 2.5, the results indicate that, on 29 out of 49 code smells ($59.3\% = 29/49$), the solutions recommended by our approach lead to increase in $ER(s)$. We also note that on 12 out of 49 code smells ($24.5\% = 12/49$), the solutions suggested by JDeodorant lead to increase in $ER(s)$. On the other hand, 35% of the solutions recommended by JMove resulted to increase in $ER(s)$. Furthermore, as depicted in Fig. 2.6, we compared our approach against JDeodorant and ARIES on Extract Class refactoring. The results indicate that, on 23 out of 36 code smells ($64\% = 23/36$), the solutions recommended by our approach lead to increase in $ER(s)$, whereas the total number of 8 and 13 solutions, equivalent to 22% and 36% of the solutions recommended by JDeodorant and ARIES respectively resulted to increase in $ER(s)$.

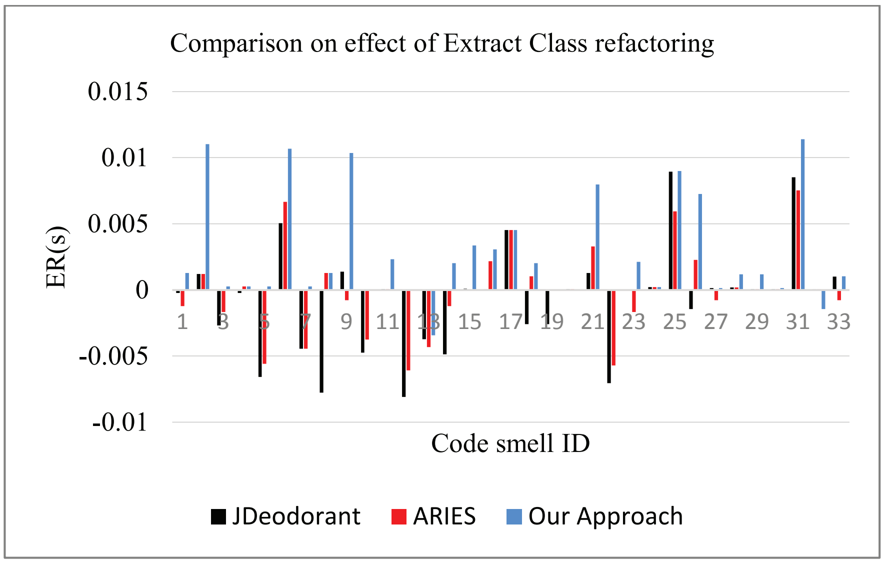


Figure 2.6 Comparison on Effect of Extract Class Refactoring

2.6.4 Developers' Selection

Generally, two different ways are often used to evaluate refactoring recommendations. The first way is to employ human evaluators for example application developers or domain experts. The second involves implementing recommendations as a prototype or a tool and get tested or compared with other tools[94]. For example in[94] and[64], with other techniques authors also adopted the qualitative evaluation approach to collect opinions from human subjects who are familiar with application development and refactorings to evaluate the refactorings recommended by their proposed approaches. To this end, we also validate the solutions recommended by the proposed approach from the developers' perspective to assess at what extent such solutions are preferred. This subsection aims to answer research question $RQ2$.

Ten developers were requested to manually evaluate the recommended solutions and select those which they would prefer. The recruited developers included 7 graduate and 3 undergraduate university students. Their working experience on Java development ranges from 1 to 5 years. In particular, we provided the developers with necessary details required for refactoring, including source code with specific sources of code smells uniquely marked, type of the code smells and the available alternative solutions. The provided list of alternative solutions was supplemented with details including source code design considerations. The alternative refactoring solutions for each code smell were presented to developers for selection. Those solutions were the top recommendations from the baseline approaches and the other from our approach. However, this was not explicitly known to developers. It is worth noting that, the cases that were considered for evaluation were those whose top recommendations were different. We performed two experiments, the first one involved the recommended $Move\ Method$ refactorings by our approach, JDeodorant and JMove, whereas the second involved the $Extract\ Class$ refactorings recommended by our approach, JDeodorant and ARIES. For each of the proposed solution the developers had to analytically assess the code smells and consequently suggest the selection. Our goal here is to validate whether the recommended refactoring solutions that have shown to improve traceability entropy are also preferred by the developers.

To analyze how often the particular solutions are preferred by the majority of the developers, i.e., in how many code smells the approach is preferred by the majority of the developers, we defined a measure:

- A $Preferred\ Solution$ is a solution that is selected by the majority of developers. Since we had 10 developers, a solution is considered to be selected by the majority if its $votes \geq 6$. That means a solution is selected by at least six developers.

- Let $P(ap)$ be a measure of how often the recommended solutions by approach ap are preferred solutions:

$$P(ap) = \frac{Total\,number\,of\,preferred\,\boldsymbol{ap}\,solutions}{Total\,number\,of\,code\,smells} \tag{2.11}$$

To compute how often individual developers prefer the particular solutions:

- Let $S(ap)$ be a measure of how often individual developer preferred the solution recommended by approach ap:

$$S(ap) = \frac{Total\,votes\,\boldsymbol{ap}\,received}{Total\,votes\,issued\,by\,developers} \tag{2.12}$$

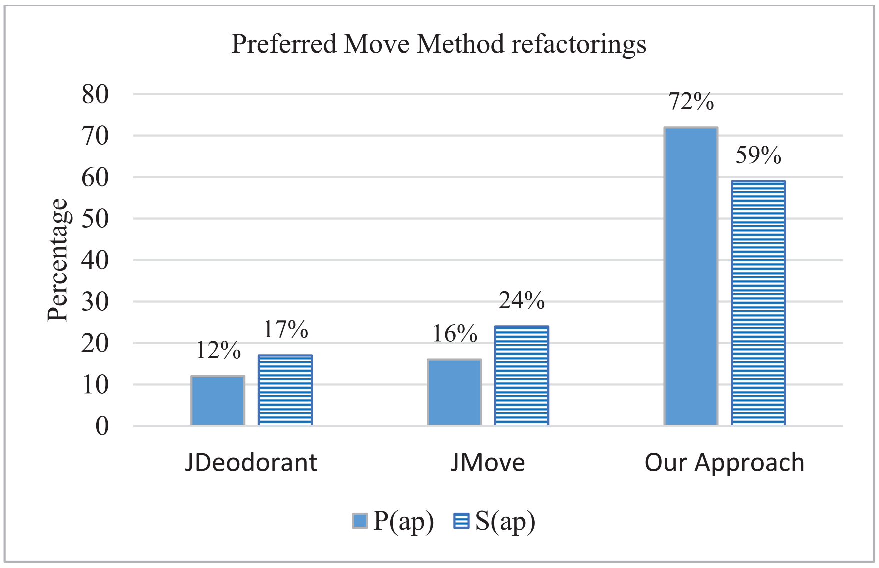


Figure 2.7 Preferred Move Method Refactoring Solutions

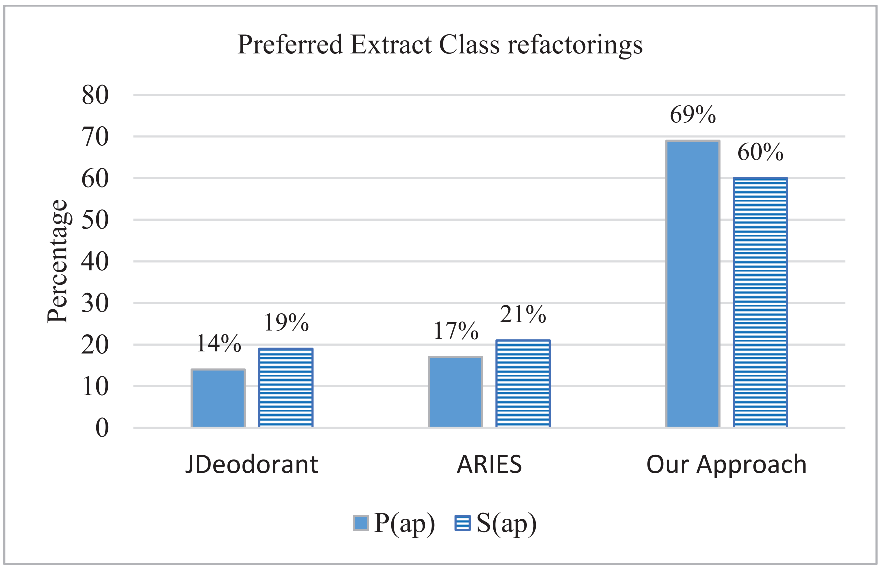


Figure 2.8 Preferred Extract Class Refactoring Solutions

Fig. 2.7 and Fig. 2.8 respectively depict the percentage of the recommended $Move$ $Method$ and $Extract$ $Class$ refactoring solutions that were preferred by the developers. As depicted in Fig. 2.7 and Fig. 2.8, generally the results indicate that, in both experiments the developers mostly preferred the solutions recommended by our approach than that of the baseline approaches. Next we present the results analysis of each experiment in detail.

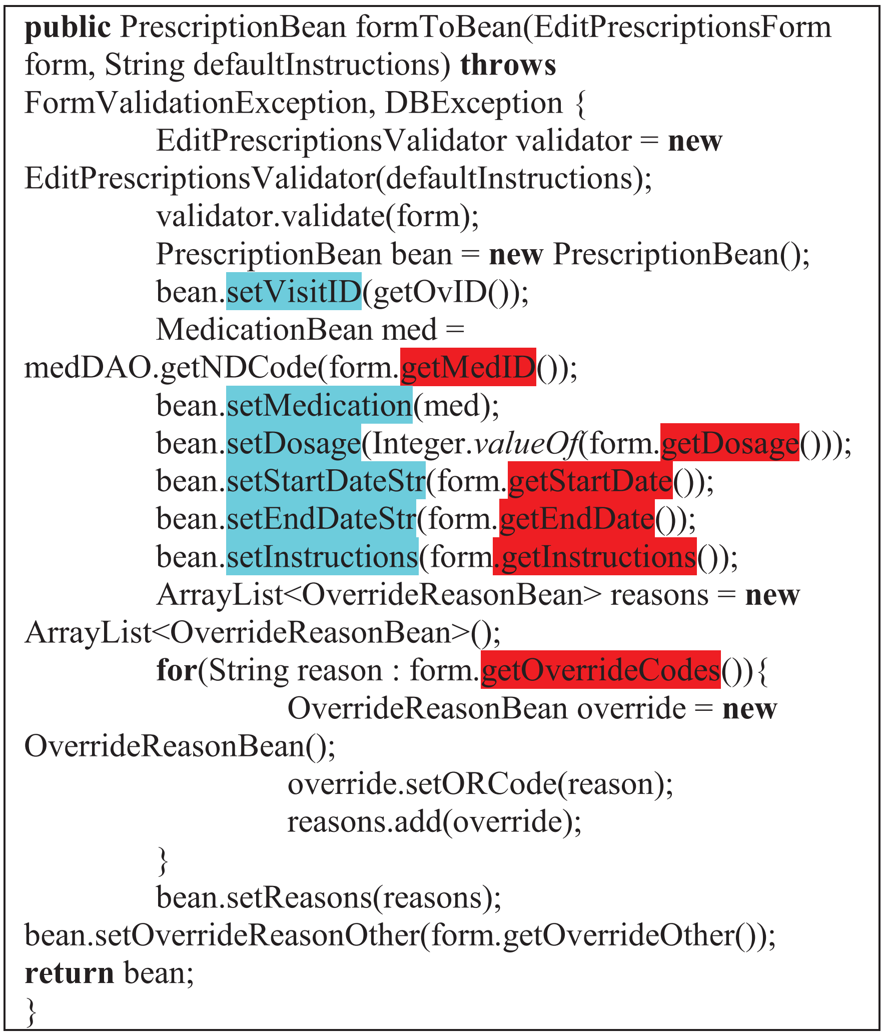
As shown in Fig. 2.7 ($p(ap)$), around 72% of the $Move$ $Method$ refactoring solutions recommended by our approach were preferred by the developers. In other words, on 35 out of the 49 ($72\% = 35/49$) code smells, solutions recommended by the proposed approach were preferred, whereas 12% and 16% of the solutions recommended by *JDeodorant* and *JMove* were preferred, respectively. From Fig. 2.7 ($S(ap)$), we also observe that in total the proposed approach has received most (59%) of the votes whereas *JDeodorant* and *JMove* received 17% and 24%, respectively. From the preceding analysis, we conclude that developers frequently preferred the solutions recommended by our approach.

Evaluation results on $Extract\ Class$ refactorings are presented in Fig. 2.8. From this figure, we also observe that most (69%) of the refactoring solutions recommended by our approach were preferred, and it received most (60%) of the votes from developers. Notably, for all code smells whose solutions were analyzed by the developers in the two experiments, there was not any case where all developers preferred the same solution for the same code smell. This implies that the selection of solutions is highly subjective.

Although most of the refactoring solutions recommended by the proposed approach were preferred by the developers, we also note that, on 25 out of the 85 ($29\% = 25/85$) code smells, the solutions recommended by our approach were not preferred. This implies that, in those cases developers preferred the solutions recommended by the baseline approaches instead of our approach. The cases where some of the solutions recommended by our approach were not preferred by the developers can be explained based on the following two major reasons.

First, the developers may make selection decision based on some other information than traceability and code metrics. For example as depicted in Fig. 2.9, the method $formToBean$ uses a number of functions from other two classes, $EditPrescriptionsForm$ and $PrescriptionBean$, which causes a code smell called $Feature\ Envy$. The highlighted parts of the envy method are the source of the smell. This smell can be resolved by moving the envy method to one of the classes where it wants to be. In this case, our approach recommends moving the envy method $formToBean$ to class $PrescriptionBean$, which would lead to reduced entropy. However, developers preferred to move the envy method to another class $EditPrescriptionsForm$ that is semantically related to the envy method. To improve our recommendation, in future we will consider other factors, e.g., conceptual relationship, to supplement our traceability-based solutions recommendation.

Second, our approach recommends refactoring based on assessing the effects of the so-



```
public PrescriptionBean formToBean(EditPrescriptionsForm
form, String defaultInstructions) throws
FormValidationException, DBException {
        EditPrescriptionsValidator validator = new
EditPrescriptionsValidator(defaultInstructions);
        validator.validate(form);
        PrescriptionBean bean = new PrescriptionBean();
        bean.setVisitID(getOvID());
        MedicationBean med =
medDAO.getNDCode(form.getMedID());
        bean.setMedication(med);
        bean.setDosage(Integer.valueOf(form.getDosage()));
        bean.setStartDateStr(form.getStartDate());
        bean.setEndDateStr(form.getEndDate());
        bean.setInstructions(form.getInstructions());
        ArrayList<OverrideReasonBean> reasons = new
ArrayList<OverrideReasonBean>();
        for(String reason : form.getOverrideCodes()){
                OverrideReasonBean override = new
OverrideReasonBean();
                override.setORCode(reason);
                reasons.add(override);
        }
        bean.setReasons(reasons);
bean.setOverrideReasonOther(form.getOverrideOther());
return bean;
}
```

Figure 2.9 Move Method Refactoring

lution on traceability. Traceability information of iTrust provides the links to method level. Thus only refactoring that affects a whole method was possible to be analyzed to assess their effects on traceability and consequently compute the entropy. We were not able to recommend other types of refactorings because in this initial attempt we do not re-compute traceability information. Particularly, refactorings which involve splitting of methods. For example, the case depicted in Fig. 2.10, the highlighted parts of the method (add) are methods called from another class, $personnelDAO$, which cause $Feature\ Envy$ code smell. In this case we recommended the solution which entirely moves the envy method to the envied class $personnelDAO$. However, developers selected extract method refactoring solution which usually extracts parts of the method causing feature envy smell and move them as the new method to the envied class. But in future we will consider such refactorings which involve

```
public long add(PersonnelBean p) throws
FormValidationException, ITrustException {
        new AddPersonnelValidator().validate(p);
        long newMID =
personnelDAO.addEmptyPersonnel(Role.HCP);
        p.setMID(newMID);
        personnelDAO.editPersonnel(p);
        String pwd = authDAO.addUser(newMID,
Role.HCP, RandomPassword.getRandomPassword());
        p.setPassword(pwd);
        return newMID;
}
```

Figure 2.10 Extract Method Refactoring

extracting methods, renaming identifiers and addition of new components by allowing re-computation of traceability matrix. In summary, in some few cases developers made decision based on other information or refactoring possibilities which are not incorporated in our refactoring recommendation. Furthermore, recommending refactoring solutions by only considering some few refactorings due to limitation of traceability granularity may leave out other potential alternative refactoring solutions.

Finally, we further assessed the votes of the individual developers for the $Move\ Method$ and $Extract\ Class$ refactorings, the results are summarized in Fig. 2.11 and Fig. 2.12 respectively. The votes indicate the number of refactoring solutions recommended by a particular approach that were preferred by a given developer. As depicted in Fig. 2.11 and Fig. 2.12 it is evident that all developers have almost the same preference to the proposed approach.

2.6.5 Limitation and Threats to Validity

One of the major limitations of the study is that refactorings which affect source code units below method level were not considered. For example, refactorings which involve splitting of methods and renaming identifiers. That is because iTrust maintains the traceability information which goes up to method granularity. In industry, traceability information can be automatically generated by tools. However, such tool generated traceability information still requires developers to manually validate or slightly modify them. But with the advancement in efficient and accurate generation of traceability links, in future we will try to validate our

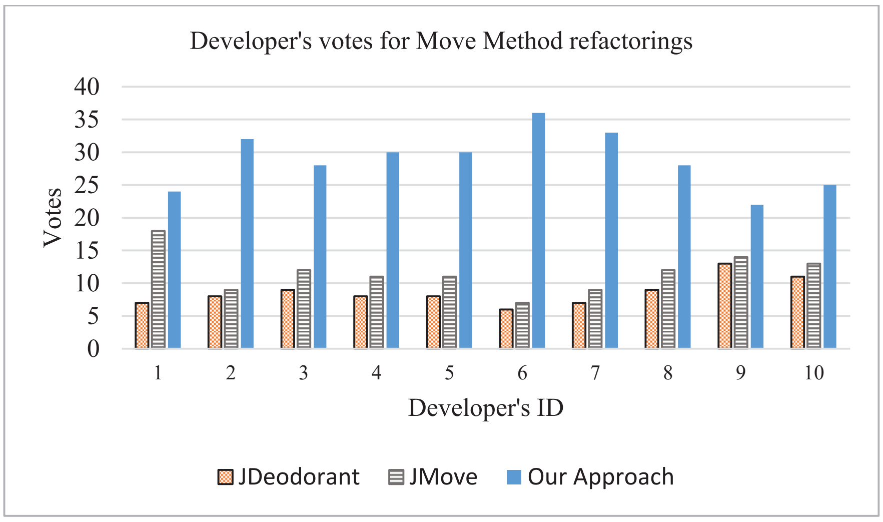


Figure 2.11 Developers' Votes for the Move Method Refactorings

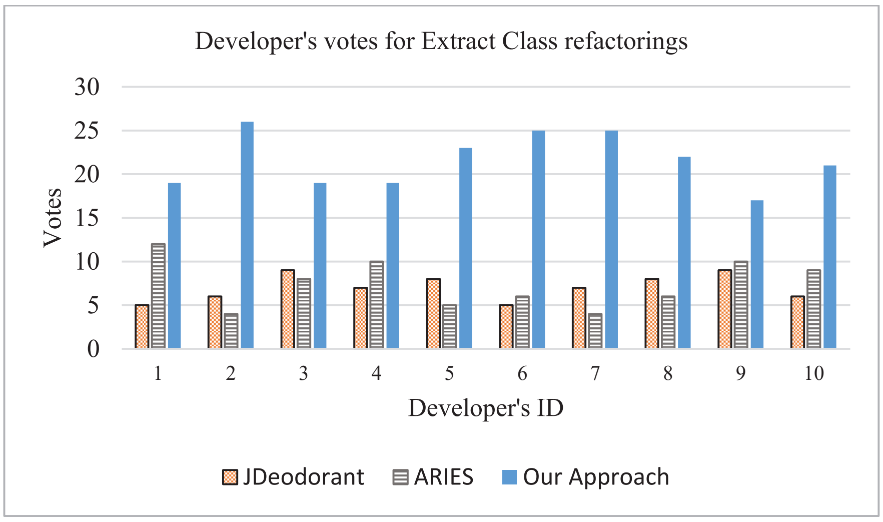


Figure 2.12 Developers' Votes for the Extract Class Refactorings

approach on such applications with automatically generated traceability information.

The first threat to external validity is only one subject system is used for evaluation. To the best of our knowledge, among the open-source datasets publicly available only iTrust maintains traceability information at method level. Evaluating with only one subject system limits the generalizability of our findings. To reduce the threat, we selected the modules that their refactorings had several suggestions so as to challenge the recommendation. The second threat is that we validated our approach on two types of smells only. This is because other types of smells would require traceability information that goes beyond method granularity. The threat can be addressed by incorporating the traceability information that goes below method granularity so as to include code smells and refactorings that affect only some parts of the method. The third threat could stem from the fact that, we qualitatively evaluated our findings by asking for developers' opinions. Though that way is scientifically acceptable due to our aim here focusing on developers' selections, still our results could be threatened due to the fact that we only evaluated with developers who are not original developers of iTrust. To slightly mitigate this threat, developers were not informed the purpose of the evaluation. Moreover, the solutions were presented to them without explicitly indicating which approach suggested which particular solution.

2.7 Conclusion

In this chapter, we propose an approach to recommend refactoring solutions based on requirements traceability. Our approach aims at facilitating developers in refactoring solutions selection from the multiple refactorings which are suggested by refactoring tools. We leverage the use of entropy metric to assess the degree of randomness of classes and methods in traceability matrix. Consequently, our approach recommends solutions that improve traceability entropy and code design. In this study, we make use of traceability to assess how best refactoring can be done and still maintains the well-structured traceability links. We validate our approach on a well-known dataset. The results obtained from the two conducted experiments which involved $Move\ Method$ and $Extract\ Class$ refactorings suggest that the solutions recommended by the proposed approach were mostly preferred by the developers than those of the baseline approaches.The results show that on average 71% of all solutions recommended by our approach to resolve the identified code smells were preferred by the developers. The results further show that, the refactoring solutions recommended by our approach were able to reduce traceability entropy by 6%, whereas the solutions suggested by JDeodorant, ARIES and JMove led to traceability entropy increase by 3.6%, 3.1% and 2.7% respectively. Our research contributes to the traceability utilization endeavor particularly in facilitating software refactoring. The possible future research in this direction include, first to validate the proposed approach on other refactorings which are not included in this study, particularly those which involve splitting of methods. Second, it would be interesting to further assess the effect of preserving and improving traceability on source code design metrics.

This was not considered in this study because we do not re-compute traceability information. As a result we were not able to assess how traceability-based refactorings impact the code design metrics.

Chapter 3 Feature Requests-based Software Refactoring

During software evolution, developers often receive new requirements expressed as feature requests. To implement the requested features, developers have to perform necessary modifications (refactorings) to prepare for new adaptation that accommodates the new requirements. Software refactoring is a well-known technique that has been extensively used to improve software quality such as maintainability and extensibility. However, it is often challenging to determine which kind of refactorings should be applied. Consequently, several approaches based on various heuristics have been proposed to recommend refactorings. However, there is still lack of automated support to recommend refactorings given a feature request. To this end, in this chapter, we propose a novel approach that recommends refactorings based on the history of the previously requested features and applied refactorings. First, we exploit the state-of-the-art refactoring detection tools to identify the previous refactorings applied to implement the past feature requests. Second, we train a machine classifier with the history data of the feature requests and refactorings applied on the commits that implemented the corresponding feature requests. The machine classifier is then used to predict refactorings for new feature requests. We evaluate the proposed approach on the dataset of 43 open source Java projects and the results suggest that the proposed approach can accurately recommend refactorings (average precision 73%).

3.1 Introduction

Software systems continuously change and evolve to adapt new requirements and accommodate new components. Change of requirements is inevitable as the business and stakeholder demands continuously evolve. As a result, software systems constantly need to be maintained in order to continue satisfying their intended objectives. During software evolution, developers often receive new requirements expressed as feature requests. To implement the requested features, developers often perform necessary modifications (refactorings) to prepare their systems to accommodate the new requirements[94]. Software refactoring is a well-known technique that has been extensively used to improve software quality by applying changes on internal structure that do not alter its external behaviors[5].

Soares *et al.*[71] state that refactorings are most commonly applied for a particular reason such as implementing a feature or bug fixing, than in the dedicated refactoring sessions aiming at evolving the software design. The recently conducted empirical study to determine the motivation of applying refactoring suggests that refactoring is mostly motivated by requirements changes and very less by code smells resolution[34]. Moreover, Silva *et al.*[34] suggest that there is a need for the refactoring recommendation systems to recommend suitable solutions to facilitate maintenance tasks especially the implementation of the feature or bug fix requests.

Usually, in order to improve the next releases of software, developers of most software systems, particularly open source projects, allow users to report the issues (i.e., new feature and bug fix requests). The most common and dominant means to track and manage feature requests is the use of issue tracking systems e.g JIRA[109], Bugzilla[110], and GitHub Issue Tracker[111]. Through issue tracker, a feature request can be discussed, assigned to a developer, and keep track of its status[112]. Given the requirements expressed in such feature requests, developers often need to locate the source code that should be modified to allow the implementation of the requested feature. As a result, several techniques have been proposed to leverage feature requests to allow locating (e.g., based on requirements traceability and text similarity) and recommending software entities (e.g., API methods) that can be used to implement the feature[94,113,114]. However, to the best of our knowledge, there is still lack of automated support to recommend refactorings during the implementation of feature requests.

To this end, in this chapter we propose a machine-learning-based approach that recommends refactorings based on the history of the previously requested features and applied refactorings. The proposed approach learns from the training dataset associated with a set of applications and can be used to suggest refactorings for feature requests associated with other applications (or new feature requests associated with the training applications). Our approach involves two classification tasks: first a binary classification that suggests whether refactoring is needed or not for a given feature request, and then a multi-label classification that suggests the type of refactoring. Although the proposed approach suggests refactoring types only and does not point to the classes involved in the refactoring, the proposed approach is helpful to implement feature request. To carry out software refactoring, we should know both ‘what’ (refactoring classes) and ‘where’ (where the refactoring should be applied). Consequently, the baseline approach proposed by Niu et al.[94] suggests both ‘where’ and ‘what’. However, the proposed approach suggests ‘what’ only. One of its potential practical usefulness is to replace/improve the second part of the baseline approach[94] (that suggests refactoring classes) because the evaluation results suggest that it is more accurate than the baseline approach in suggesting refactoring classes. The two approaches working together could suggest both ‘what’ and ‘where’ to developers. The proposed approach may also be integrated with other approaches/tools that could suggest ‘where’. It is interesting in future to investigate how change impact analysis approaches[115] may help to identify which specific source code entity (e.g., class or method) should be refactored. Another potential practical usefulness of the proposed approach is to help developers pick up proper refactoring tools. Existing study[116] suggests that it is often up to developers to pick up the proper tools to identify different classes of refactoring opportunities, e.g., *GEMS*[29] for *extract method* refactoring opportunities, and *JMove*[22] for *move method* refactoring opportunities. Suggesting refactoring classes (by the proposed approach) may significantly facilitate the selection. The third potential practical usefulness of the proposed approach is to prioritize the implementation of different features. Because different categories of refactorings may interfere with each other[116], knowing the required refactoring types of different features help in deciding the order of refactoring types and hence deciding the order of implementing features.

The past feature requests and their associated commits (from which refactorings are detected) are retrieved from the corresponding issue trackers and software repositories respectively. Normally, each feature request can be linked to the corresponding source code commits to identify the types of refactorings applied on such commits. The previously applied refactorings are recovered from the corresponding software repositories by using the state-of-the-art refactoring detection tools, e.g., `Ref-Finder`[48], `RefactoringCrawler`[51], `RefDiff`[50], and `RMINER`[49].

The proposed approach is evaluated on the dataset of 43 open source Java projects altogether consisting of $13,550$ commits from GitHub repository and $13,367$ feature requests from JIRA issue tracker. The evaluation results suggest that, the proposed approach can accurately recommend refactorings and attain an average precision of 73%.

The contributions of this chapter are two folds:

(1) A new automated approach to recommend refactoring solutions given a feature request based on the history of the previous feature requests and applied refactorings from a set of applications. Such applications could be different from the one where refactorings are recommended.

(2) Evaluation results of the proposed approach on the refactorings and feature requests history data suggest that the proposed approach can accurately recommend refactorings given a new feature request.

3.2 Background

Ouni *et al.*[15] proposed a multi-criteria refactoring recommender which suggests an optimal sequence of refactorings that, among other criteria, targets at maximizing consistency with the refactorings applied previously. In their work they contended that, the history of code changes is essential and can increase confidence in recommending new refactorings. To ensure consistency with past refactorings, they defined the following fitness function:

$$Sim\_refactoring\_history(RO) = \sum_{j=1}^{n} e_j \tag{3.1}$$

where n is the number of previously applied refactorings, and e_j is a refactoring weight that measures the similarity between the recommended refactoring operation (RO) and the past refactoring operation j. A detailed survey of search-based refactorings recommendation approaches is recently conducted in[117].

Kessentini *et al.*[60] proposed an approach that recommends refactorings based on the analysis of bug reports and history of change. The assumption of this approach is that, a class which is recently modified or listed in previous bug reports is more likely to demand refactoring. Additionally, the previously applied refactorings were also considered to deduce possible potential refactorings for current release. Tsantalis and Chatzigeorgiou[61] proposed a tool (Eclipse plugin) that exploits past source code changes to rank refactoring suggestions. The philosophy of this approach is that, a piece of code that has undergone several changes in the past, is more likely to demand refactoring in the future. Consequently, a refactoring involving such code should receive a higher priority. Similar approaches to recommend refactorings based on development history are proposed in[62,118]. Generally, these approaches[15,60–62,118] suggest that past source code change history is useful in recommending new refactorings. In addition, historical data is essential in producing quality code to evolve software systems[119]. However, such approaches do not consider feature requests in their recommendation.

The proposed approach is inspired by the earlier work proposed by Niu *et al.*[94]. The authors in[94] proposed a traceability-based refactoring recommendation approach to ensure that the requested requirements are fully implemented. Their approach leverages requirements traceability between the requirements under development and the implementing source code to accurately locate where the software should be refactored. To determine what types of refactorings should be applied, the authors developed a new scheme that examines the requirements semantics as they relate to the implementation. Requirements semantics involves manual analysis of requirements action themes which refer to the intended action (e.g., Add, Enhance, Remove) to be taken to implement a feature such as enhancing a quality attribute. Consequently, semantic characterization is leveraged to detect code smells that may hinder the fulfillment of such actions. In summary, this approach proposed a novel scheme that maps requirements action themes and code smells to refactorings. This approach was qualitatively evaluated based on asking opinions from the developers of the involved subject application to rank the recommended refactorings. Generally, the approach scored 3.8 in the 5 point scale which suggests that the recommended refactorings were somewhat appropriate. The key difference of the approach in[94] with our proposed approach is on how to recommend refactoring solutions. Although both approaches rely on feature requests (i.e., requirements) to recommend refactorings, the proposed approach leverages previous feature requests and their associated applied refactorings to predict refactorings for the implementation of the current feature request. On the other hand, Niu *et al.*[94] only work with current feature request as input and recommend refactorings that ensure full implementation of the requested feature. Additionally, the approach in[94] is based on manual analysis of requirements which is often tedious and error-prone, whereas our work implements an automated recommendation approach.

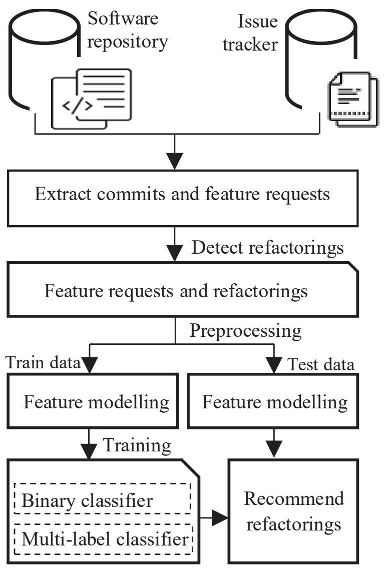


Figure 3.1 Refactoring Recommendation Approach

In line with facilitating developers during maintenance task especially when implementing feature requests, Thung *et al.*[113] proposed an approach to recommend API methods given a feature request. Their approach takes as input the textual description of a new feature request and recommends methods from API library that a developer can use to implement a feature. The proposed approach learns from the training dataset of the past resolved or closed feature requests and changes applied to a software system recorded in issue trackers and software repositories respectively. Then, the past similar feature requests are retrieved along with the relevant methods used to implement them. The approach then learns a ranking function and consequently recommends the potential and relevant library methods to the developer. This approach is different from ours in the sense that, based on feature request they recommend API methods to the developer, whereas the proposed approach recommends refactoring solutions.

3.3 The Proposed Feature Request-based Refactoring Recommendation Approach

The framework of the proposed approach is analyzed in Fig. 3.1. First, we extract the feature requests from the issue tracker and their corresponding commits from a software repository. Second, by using the state-of-the-art refactoring detection tools we recover refactorings applied in the retrieved commits. Next, we apply the preprocessing on the retrieved

Table 3.1 Data Fields of a Feature Request

Attribute	**Description**
ID	a number which uniquely identifies a feature request
Summary	the summary or title of a request
Description	the detailed description of a request
Status	the current status of a request
Resolution	the implementation status of a request

feature requests. Finally, we train the machine-learning-based classifier which is then used to predict refactorings for a new feature request. In the following we elaborate each of these steps in detail.

3.3.1 Feature Requests and Commits Extraction

Software users are usually allowed to request for the new feature or enhancement of the existing feature by submitting a feature request. A feature request often requires some new source code to implement the requirements that cannot be satisfied by the current code-base. Generally, a feature request contains several data fields including: unique request ID, summary, description, resolution, etc. This study is concerned with the data fields which are listed in Table 3.1.

The feature requests for each of the subject applications that have been addressed to completion (i.e., marked as "Closed" or "Resolved") are retrieved from the issue tracker. The details (ID, Summary, and Description) of the retrieved feature requests which are generally the free-form texts are stored for further processing. Such feature requests were then used to formulate our dataset. The dataset was split into training and testing set. The testing set of feature requests is the one that was used to evaluate the performance of the approach in predicting and recommending refactorings. All feature requests had to undergo the NLP steps (e.g., stop word removal, lemmatization and vector space modelling) prior to training and testing the classifiers. Therefore, once the classifiers are trained with existing feature requests (say original FRs) then it can be readily used to predict and recommend refactorings for the new FRs (after preprocessing and vector space modelling). Next, the repository of each subject application (comprising of several commits) is cloned from Git repository to a local computer by using Eclipse. To speed up the process of detecting refactorings the repositories of the subject applications were first cloned to a local machine rather than being cloned during the refactoring detection process. Next, by using Git bash commands[120] we retrieve all commits that contain in their commit messages the specified feature request identifiers of the feature requests we retrieved earlier. Finally, at this stage for each feature request fr from the set of all feature requests FR is formalized as:

$$fr =< frID, summ, desc, commitID > \tag{3.2}$$

where $frID$ represents a unique feature request identifier, $summ$ is the summary or title of a feature request, $desc$ is the detailed description of a feature request and $commitID$ is the unique identifier of a commit used to implement a feature request. The identified commits are then inputted in the next step to detect the applied refactorings. Note that, all commits associated with a given feature request are retrieved in order to identify all types of refactorings applied for such feature request. That means a one-to-many association between feature requests and commits is taken into consideration. In this study we only focus on the cases where the links between the feature requests and commits are known explicitly. In the future the state-of-the-art techniques can be leveraged to generate missing links. For example, recently, Rath *et al.*[121] proposed an approach that trains a classifier to uncover the missing issue tags in commit messages and generate the missing links.

3.3.2 Detection of Refactorings

To detect refactorings applied in the commits we leveraged the state-of-the-art refactoring detection tools (`RefDiff` and `RMINER`). `RefDiff` is an automated approach introduced by Silva and Valente[50] to detect refactorings applied between two source code revisions archived in Git repository. The tool uses the combination of heuristics based on static analysis and code similarity to detect 13 common refactoring types. Moreover, `RMINER` is a novel technique recently proposed by Tsantatlis *et al.*[49] to mine refactorings from software repositories. `RMINER` runs an AST-based statement matching algorithm to detect 15 representative refactoring types. These tools are selected because they can easily be used as Eclipse plugins and they are effective in detecting applied refactorings by comparing subsequent versions of the program. Such tools were recently applied on manually validated refactorings oracle and reported the precision and recall of 98% and 87% for `RMINER` and 76% and 86% for `RefDiff` respectively[49]. Note that, the results from the two detectors were combined to form a union set. Consequently, the set was manually checked to identify the overlapping cases and eliminate duplication. To identify refactorings, the text file with the list of all commits identified in the previous step is inputted to the refactoring detection tool. For each $commitID$ the tool detects refactorings applied and then outputs the *txt* file with the list of $commitID$ and the associated refactorings. If the inputted $commitID$ is not returned in the output file, then such commit is considered not to have any refactorings. At the end of this step for each feature request fr from the set of all feature requests FR is such that:

$$fr =< frID, summ, desc, commitID, ref > \tag{3.3}$$

where ref represents the set of refactorings detected in a commit. Note that, if no refactorings are detected then the value of ref is set to null. Therefore, we identify if a given feature request fr_i demands refactorings or not based on the following condition:

$$fr_i = \begin{cases} \text{no refactoring}, & \text{if } ref = \emptyset \\ \text{need refactoring}, & \text{if } ref \neq \emptyset \end{cases} \tag{3.4}$$

3.3.3 Text Pre-Processing

To clean and prepare the feature requests for classification we employed text pre-processing. Such process is essential in improving the classification performance[122]. Text preprocessing is purposely used to transform the feature requests (which are written in natural languages) into a form suitable for textual analysis by using Python Natural Language Processing Toolkit (NLTK)[123]. The texts which are considered here are those from the summary and description fields of the feature requests. The applied NLP techniques include tokenization, stop word removal, and lemmatization. First, tokenization involves breaking up a document into a lists of individual words (i.e., tokens). In this step some characters such as numbers and punctuation are excluded as they do not contain any useful information. Second, stop word removal is applied, the common and frequently used words such as “a”, “an”, “the”, “in”, and “is” are eliminated as they do not carry any useful information and just introduce noise to NLP activities. Finally, lemmatization is applied to convert the words as they appear in the document back into their common base form. This base form is usually referred to as *Lemma*. This process reduces the number of tokens and hence the complexity of NLP activities. In this study we use Porter's stemming[124] which implements suffix stripping algorithm for lemmatization. Porter's stemmer has been extensively used in various software engineering studies[114,125].

3.3.4 Feature Modelling

The classification of feature requests which are free-form texts written in natural language involves some key natural language processing steps which include vector space representation[126,127]. Therefore, we selected the vector space model for representation of word features extracted from feature requests into numerical representation which is suitable for machine learning. Vector space model is among the mostly used model mainly because of its conceptual simplicity[128] and it has been widely used in texts clustering, categorization and classification[127,129,130]. Consequently, in this step, the preprocessed feature requests are converted into a feature vector space model[131] which represents the bag of words extracted from feature requests as a vector of weights. The weight of a word represents its importance in a document. To quantify the importance of each word forming up a document in a corpus the term frequency (TF) and inverse document frequency (IDF) are often used. We therefore use TF-IDF to represent features in a feature vector. Suppose in our corpus D we have a term t and a document fr, then Term Frequency $TF(t, fr)$ defines the number of times the term t appears in a document fr, whereas Document Frequency $DF(t, D)$ defines the number of documents in the corpus that contain the term t. Here, a corpus refers to the collection of all feature requests, whereas a document and term refer to a single feature request (fr) and a word (i.e., a token) respectively. Note that, Inverse Document Frequency (IDF) is the reciprocal of the Document Frequency (DF). Therefore, TF-IDF computes the weight w of a term t in a document fr from corpus D as follows:

$$IDF(t, D) = \frac{1}{DF(t, D)} \tag{3.5}$$

$$w_{\mathrm{t,fr,D}} = TF(t, fr) \times IDF(t, D) \tag{3.6}$$

The higher the value of the weight, the more important the term is and has higher discriminating power between documents.

3.3.5 Training and Recommendation

The feature vectors obtained in the previous step are then subject to the classifiers for training and prediction (i.e., recommending refactorings). The proposed approach leverages the Logistic Regression (LR), Multinomial Naïve Bayes (MNB), Support Vector Machine (SVM), and Random Forest (RF) classifiers. The classifiers have been implemented by a well-known machine learning library based on python called scikit-learn[132]. We have specifically selected such machine learning algorithms because they are widely used and have shown to be effective in text classification[130,133]. We generally model our text classification problem such that, we have a description $fr \in FR$ of a feature request, where FR is the feature requests space; and a fixed set of classes $R = \{r_1, r_2, ..., r_{\mathrm{m}}\}$. In our case here, classes are also referred to as labels or refactorings. Suppose we have a training set T of labeled feature requests (fr, r), where $(fr, r) \in FR \times R$. Our goal is to train a classifier or a classification function f that maps feature requests to classes (i.e., recommending refactorings): $f : FR \to R$.

Our approach involves two classification tasks, i.e., binary and multi-label classification. The main reason for that is, predicting the need for refactoring and recommending refactoring types can be formulated independently as binary and multi-label classification problems respectively. Besides that, not all feature requests may require refactoring and therefore it is not logical to predict refactoring types for such cases. Consequently, separating the two tasks could boost the training and recommendation performance of the classifiers. In the following we describe these two tasks in details.

- **Binary classification**, at this first stage the classifier is trained to determine whether refactoring is needed or not. The input to the classifier is the feature requests that required refactoring and those which did not require refactoring. Then, the classifier categorizes a feature request fr into a class c as function f such that:

$$c = f(fr), \;\; c \in \{0, 1\}, \;\; fr \in FR \tag{3.7}$$

where c represents the classification result: 0 implies a feature request fr does not require refactoring whereas 1 implies that refactoring is needed.

Consequently, the feature requests which are identified to need refactoring will save as input to the next stage to identify the types of refactorings required.

- **Multi-label classification**, after identifying the feature requests that require refactoring, the multi-label classification is performed to predict the specific types of refactorings which are required. Multi-label classifiers are leveraged because a given feature request may involve more than one type of refactoring. To train the classifier, past feature requests (i.e., those identified to need refactoring) and the applied refactorings will save as input. Consequently, the classifier categorizes a feature request fr into a class c as function f such that:

$$c = f(fr), \;\; c \subseteq R, \;\; fr \in FR \tag{3.8}$$

  where c is the set of one or more refactorings.

Therefore, the classifiers are generally trained to determine whether refactoring is required, if yes then they should predict (i.e., recommend) the types of refactorings required for the new feature requests.

3.4 Evaluation

In this section we present the evaluation of the proposed approach which we refer to as *FR-Refactor* (Feature-request-based refactoring). To evaluate the performance of *FR-Refactor* in predicting the need for refactoring and recommending required refactoring types, we compared it with the state-of-the-art approach proposed by Niu *et al.*[94]. In the following, we first highlight the research questions that this study is addressing. We then describe the dataset used in our experiments. Next, the process and metrics used as the basis of our evaluation are described. Finally, we present and analyse the experimental results and conclude the section by highlighting the threats to validity of our results.

3.4.1 Research Questions

The evaluation investigates the following research questions:

- **RQ1**: How accurate are different machine learning classifiers in predicting the need for refactoring?
- **RQ2**: How accurate are different machine learning classifiers in recommending required refactorings?
- **RQ3**: How accurate is *FR-Refactor* in predicting the need for refactoring compared to the state-of-the-art baseline approach?
- **RQ4**: How accurate is *FR-Refactor* in recommending required refactorings compared to the state-of-the-art baseline approach?
- **RQ5**: Can the proposed approach still obtain good results on applications that are different from those involved in the training?

The research questions **RQ1** and **RQ2** respectively investigate the performance of different machine learning classifiers in predicting the need for refactoring and identifying which refactoring types are required. **RQ3** concerns the performance of *FR-Refactor* in predicting if a given feature request would demand refactoring or not. The research question **RQ4** evaluates the accuracy of *FR-Refactor* in recommending refactoring types. The accuracy of recommendation is essential to determine at what extent the approach suggests useful refactoring types (true positives) to the developers rather than just overloading developers with irrelevant refactoring types (false positives). Finally, **RQ5** investigates the performance of the proposed approach when it is applied to predict and recommend refactoring types to new applications that were not involved in the training.

To answer **RQ3** and **RQ4** we compare *FR-Refactor* with the traceability-enabled approach proposed by Niu *et al.*[94] which is based on manual analysis of requirements semantics to recommend refactoring types. Authors performed such analysis by analyzing the description of each feature request to identify terms similar or related to the action (e.g., Add, Enhance, Remove) that has to be taken to implement a feature. Given the identified action themes along with the code smells information, the required refactoring types can be identified based on the novel scheme proposed by Niu *et al.*[94]. We note that, the baseline approach[94] was implemented as a separate approach but was evaluated on the same dataset applied for evaluating our proposed approach. To avoid missing out the feature requests with relevant themes, we also considered the themes (i.e., verbs) in their different forms. For example, “Add”, “Addition” and “Adding” were considered to have the same intention such as adding a functionality. The other key consideration that we took into account is to identify synonyms of each of the key themes. Therefore, a feature request containing a key verb or it’s synonym were considered to be in the same action theme category. This baseline approach is selected because, to the best of our knowledge, it is the only existing approach which is based on requirements to drive refactoring.

3.4.2 Dataset

We note that there exist a few publicly available and manually validated datasets of refactorings mined from software repositories[49,50,134]. However, these oracles contain few representative refactorings and feature requests. For example, the evaluation oracles used in[50] and[49] consist of 448 and $3,188$ known refactoring operations respectively. In addition to that, not all refactorings in such oracles are associated with the implementation of feature requests. Therefore, to attain the reasonable amount of feature requests and refactorings to effectively train our classifiers, we exploit `RefDiff` and `RMINER` which were recently validated manually and used in creating an oracle of refactorings proposed by Tsantatlis *et al.*[49]. The tools are publicly available and effective in detecting applied refactorings by comparing subsequent versions of the program.

Table 3.2 highlights the distribution of the feature requests and commits of the subject applications used in building our dataset. First, we extracted the feature requests (Request ID, Summary, and Description fields) of the subject applications from JIRA issue tracker which have been addressed to completion (i.e., marked as “Closed” or “Resolved”). JIRA explicitly links the feature requests to their corresponding commits in a repository through request ID. Second, we retrieved the relevant commits from the repository that was used to implement the retrieved feature requests. We only selected Java open source projects from GitHub repos-

Table 3.2 The Distribution of Feature Requests (FR) and Commits (COM) in Our Dataset

Project	**FR**	**COM**	**Project**	**FR**	**COM**	**Project**	**FR**	**COM**
Accumulo	474	479	Gora	31	34	Spring-Datamongo	261	264
Archiva	193	198	Groovy	385	389	Spring-Integration	517	523
Aries	314	319	Hbase	1389	1396	Spring-ROO	797	800
Atlas	70	74	Impala	178	182	Spring-Security	87	90
Axis2-Java	204	208	Jclouds	122	127	Stanbol	154	157
Beam	295	298	Jena	202	209	Storm	302	305
Bookkeeper	132	132	Kafka	307	314	Struts 2	214	219
Calcite	74	77	Lens	172	176	Synapse	79	82
Carbondata	226	230	Ode	102	106	Systemml	34	36
Cayenne	390	393	Oodt	26	26	Tajo	444	451
Curator	24	26	PDFBox	482	489	Tapestry-5	381	385
Drill	446	451	Pivot	185	189	Velocity	56	58
Flink	555	562	Sentry	87	89	Wicket	237	242
Geronimo	38	44	Sling	2369	2387	Zookeeper	80	80
Giraph	252	254						
TOTAL:	Projects: 43, Feature Requests: $13,367$, Commits: $13,550$							

itory whose commits explicitly specify in their log messages the issue (i.e feature request identifier) which is addressed. Thus our selection contains a reliable link between a feature request in an issue tracker and the corresponding commit in a software repository. Third, the retrieved commits were subject to the refactoring detection tools to recover the applied refactorings. Finally, we created a dataset of 43 open source Java projects altogether consisting of $13,550$ commits from GitHub repository and $13,367$ feature requests from JIRA issue tracker. Out of $13,367$ feature requests, a total of $7,943$ ($59\% = 7,943/13,367$) feature requests are associated with one or more refactorings, whereas the remaining 41% ($= 5,424/13,367$) of the feature requests do not have any refactoring. The feature requests and their associated refactorings used are available online at http://doi.org/10.5281/zenodo.3335978. Such subject applications were selected because they cover a wide-range of domains, publicly available, developed by different developers, and have long evolution history. Consequently, it is likely that they will have varieties of feature requests and refactorings. Further, Table 3.3 highlights the distribution of refactoring types in our dataset.

Table 3.3 The Distribution of Refactorings in the Dataset

Refactorings	**Number**	**Refactorings**	**Number**
Extract Interface	141	Move Method	1441
Extract Method	4225	Pull Up Attribute	186
Extract Superclass	132	Pull Up Method	255
Inline Method	784	Push Down Attribute	102
Move And Rename Class	268	Push Down Method	112
Move Attribute	957	Rename Class	801
Move Class	784	Rename Method	2940
Total			13,128

3.4.3 Process and Metrics

The evaluation of the proposed approach follows two key steps. First, the classifiers are trained to predict whether the given feature requests would require refactoring or not. Second, for the feature requests identified to require refactoring, the classifiers are trained to predict the required refactoring types. To answer the research questions (RQ1–RQ4), we conduct 10-fold cross validation where the dataset is randomly partitioned into ten subsets. On each fold of the evaluation, one subset is employed as testing data whereas others are taken as training data. Furthermore, we cast the problem of predicting the need for refactoring as a binary classification problem. To evaluate the binary classifiers we leveraged the traditional accuracy, precision, recall, and F-measure. Moreover, since in the second step each feature request can be associated with one or more refactoring types, we cast our refactoring recommendation problem as the multi-label classification problem. In multi-label classification each example can be associated with several labels simultaneously, hence its performance evaluation is much more complicated than in the traditional single-label classification[135]. To evaluate the performance of the multi-label classifiers we use the common and widely-used metrics including hamming loss, hamming score, and subset accuracy[135–137], as defined in the following.

Suppose $FR = \{fr_1, fr_2, ..., fr_m\}$ denotes the feature requests space, and $R = \{r_1, r_2, ..., r_n\}$ denotes the refactoring space with possible n different types of refactorings. The task of multi-label learning is to train a classifier with an evaluation dataset $D = \{(fr_i, r_i) | 1 \leq i \leq m\}$, where $r_i \subseteq R$ is the set of refactoring types associated with a feature request fr_{i}. For any unseen feature request $fr_{\mathrm{i}} \in FR$, the classifier H predicts $H(fr_{\mathrm{i}}) \subseteq R$ denoted as z_{i} as the set of possible refactoring types for a feature request fr_{i}. It follows that:

- **Hamming loss**, computes the fraction of labels (refactoring types) incorrectly predicted, i.e., a relevant refactoring is missed or an irrelevant refactoring is predicted. Hamming loss is formally defined as:

$$HammingLoss(H) = \frac{1}{|D|}\sum_{i=1}^{|D|}\frac{|z_{\mathrm{i}}\Delta r_{\mathrm{i}}|}{|R|} \tag{3.9}$$

where Δ represents the symmetric difference between the two sets (i.e., the set of predicted refactoring types and the set of true refactoring types for the feature request fr_{i}). Note that, as the value of the metric closes to 0 the better the classifier's performance.

- **Hamming score**, symmetrically computes how close the set of the predicted refactoring types (z_{i}) is to the true set of refactoring types (r_{i}) for the given feature request fr_{i}. Note that, the larger the value of the metric (with optimal value of 1) the better the classifier's performance. Hamming score can be formally defined as:

$$HammingScore(H) = \frac{|r_{\mathrm{i}} \cap z_{\mathrm{i}}|}{|r_{\mathrm{i}} \cup z_{\mathrm{i}}|} \tag{3.10}$$

- **Subset accuracy**, measures the fraction of examples classified correctly, i.e., the predicted set of refactoring types (z_{i}) is similar to the true set of refactoring types (r_{i}) for the given feature request fr_{i}. Subset accuracy can be intuitively considered as the traditional accuracy metric[135]. It is formally defined as:

$$SubsetAccuracy(H) = \frac{1}{|D|}\sum_{i=1}^{|D|}[\![z_{\mathrm{i}} = r_{\mathrm{i}}]\!] \tag{3.11}$$

where $[\![z_{\mathrm{i}} = r_{\mathrm{i}}]\!]$ returns 1 if the two sets are identical or 0 otherwise. Note that, the larger the value of the metric (with optimal value of 1) the better the classifier's performance.

Table 3.4 Classifiers' Performance for Predicting the Need for Refactoring (%)

Classifier	**Accuracy**	**Precision**	**Recall**	**F-measure**
SVM	64.13	**69.49**	71.22	70.32
MNB	**65.40**	66.36	**85.32**	**74.63**
LR	**65.35**	69.30	75.35	72.16
RF	63.59	69.16	70.59	69.79

3.4.4 Results and Analysis

3.4.4.1 **RQ1:** Performance of Different Classifiers in Identifying the Need for Refactoring

Table 3.4 highlights the effectiveness of the employed classifiers in predicting whether refactoring is required to implement a given feature request. Note that, the best recorded results for each metric is shown in bold. From the table, it is observed that the precision of predicting whether refactoring would be required ranges between 66.36% and 69.49%. Therefore, on average of up to 68.58% the need for refactoring can be accurately predicted. Furthermore, the results generally suggest that, on average, MNB and LR classifiers outweigh all other classifiers. MNB classifier achieved an F-measure of 74.63% and an accuracy of 65.40%, whereas, LR classifier achieved an F-measure of 72.16% and can accurately predict the need for refactoring (average precision 69.30%). In addition to that, Fig. 3.2 compares the performance of the classifiers in terms of accuracy, precision, recall, and F-measure in ten-fold cross validation. As depicted in Fig. 3.2, it is evident that there is no significant difference in classifiers' individual performance across different folds.

Note that, our binary classification problem involves classifying texts which assigns feature request to different classes (i.e., refactoring or non-refactoring) based on the words features present in the document. In this case Naïve Bayes classifier performed better than other classifiers. MNB has shown to be effective in binary text classification in various studies including enhancement requests approval prediction[130] and spam emails detection[138]. We note that, our binary classification problem also involves classifying texts which assigns feature request to different classes (i.e., refactoring or non-refactoring) based on the words features present in the document. MNB classifier is the widely used generative classifier that can easily accommodate any domain-specific knowledge and also performs better with

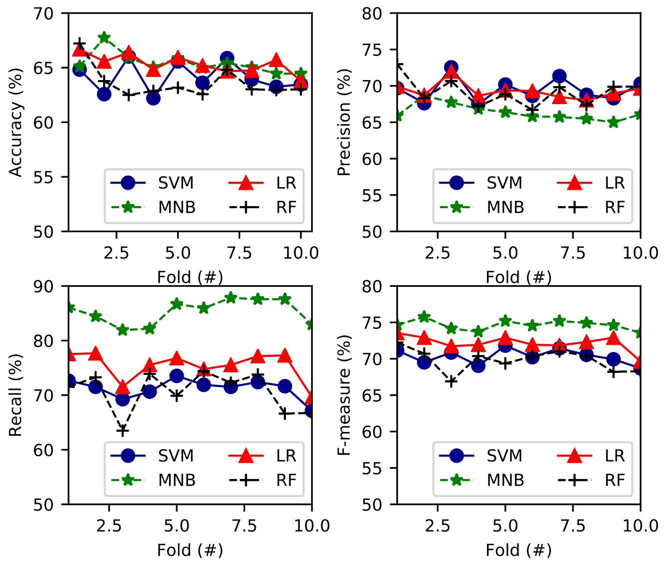


Figure 3.2 Performance of the Classifiers in Predicting Refactoring

Table 3.5 Classifiers' Performance for Refactorings Recommendation (%)

Classifier	Subset accuracy	Hamming score	Hamming loss
SVM	**70.75**	**70.85**	**0.027**
MNB	14.98	15.05	0.062
LR	43.86	43.86	0.043

hierarchical classification scenario[133]. Because the evaluation results suggest that MNB works best in predicting whether refactoring is required to implement a given feature request (noted as binary classification), MNB would be used in the rest of the chapter for the binary classification.

3.4.4.2 **RQ2:** Performance of Different Classifiers in Recommending Refactorings

The refactorings recommendation is cast here as a multi-label classification problem, hence Table 3.5 depicts the results in terms of subset accuracy, hamming score, and hamming loss which are widely used metrics for evaluating multi-label classifiers. We only selected three representative classifiers (SVM, MNB, and LR) because they can adapt popular learning techniques and directly work on multi-label data without transforming multi-label learning problem into other classification problems such as single-label or binary classification[135,139]. Note that, the best recorded results for each metric is highlighted in bold. As shown in Table 3.5, the results indicate that SVM classifier outperforms MNB and LR by the difference of

Table 3.6 Results for Individual Refactorings Recommendation

Refactoring type	**Precision**	**Recall**	**F-measure**
Extract Interface	0.63	0.58	0.61
Extract Method	0.66	0.74	0.70
Extract Superclass	0.85	0.43	0.58
Inline Method	0.83	0.38	0.52
Move And Rename Class	0.89	0.54	0.67
Move Attribute	0.86	0.34	0.49
Move Class	0.79	0.42	0.55
Move Method	0.82	0.42	0.56
Pull Up Attribute	0.69	0.60	0.64
Pull Up Method	0.67	0.42	0.52
Push Down Attribute	0.84	0.38	0.52
Push Down Method	0.83	0.33	0.47
Rename Class	0.72	0.39	0.50
Rename Method	0.65	0.51	0.58
Micro avg	0.71	0.52	0.60
Macro avg	0.77	0.46	0.56
Weighted avg	0.73	0.52	0.59

$56\%(= 70.75\% - 14.98\%)$ and $27\%(= 70.75\% - 43.86\%)$ in terms of subset accuracy respectively. The results suggest that the needed refactorings can be accurately recommended on up to 71% accuracy. Moreover, compared to other classifiers, SVM achieved the lowest value (i.e., 0.027) on hamming loss which identifies to what extent the classifier predicts the irrelevant refactorings and omits relevant refactorings. This metric is normalized between 0 and 1. The value of the metric closes to 0 indicates better performance of the classification. Furthermore, Table 3.6 presents the performance of SVM classifier in recommending individual refactorings. The results generally suggest that, the classifier achieves an average precision of 73% in recommending refactorings. However, in some few cases (e.g., Push Down Method) the classifier achieved a recall and F-measure below 50%. This can be justified by the fact that some of the refactorings, including Push Down Method, are very few in the dataset which can consequently affect their prediction. From these findings, we therefore conclude that the proposed approach is accurate in recommending refactorings.

We note that, SVM performed better than other classifiers in refactorings recommendation. SVM often performs better due to the following reasons. First, high dimensional

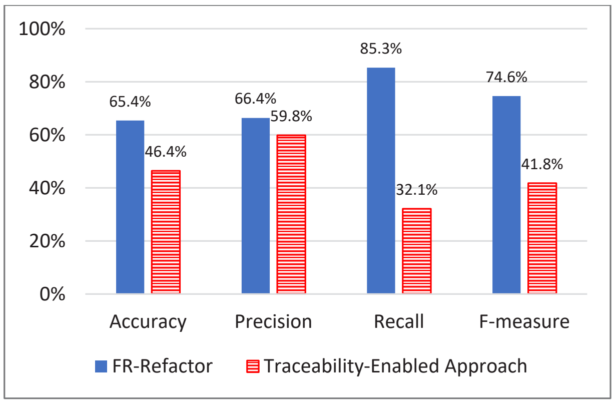


Figure 3.3 Performance in Predicting the Necessity of Refactoring

input space texts produce a lot of features which consequently lead to a very large feature spaces. SVM employs overfitting protection and has the ability to learn which can be independent from the dimensionality of the feature space. Second, few irrelevant features, in text classification irrelevant features are very few and therefore a classifier should be able to combine several features (i.e., dense concept). Third, sparsity of document vectors, usually each document contains a document vector with a lot of entries which are zeros. SVM based classifiers have shown to be effective in handling problems with sparse instances and dense concepts. Based on these facts, SVM is shown to be effective for text classification and well recognized to be accurate[136].

Because the evaluation results suggest that SVM works best in suggesting refactoring classes (noted as multi-label classification), SVM would be used in the rest of the chapter for the multi-label classification.

3.4.4.3 **RQ3:** FR-Refactor vs State-of-the-art Baseline Approach in Predicting the Need for Refactoring

To evaluate the performance of *FR-Refactor* in predicting the need for refactoring, we compared it (based on MNB) with the state-of-the-art approach proposed by Niu *et al.*[94]. As depicted in Fig. 3.3, *FR-Refactor* (based on MNB classifier) significantly outperforms the state-of-the-art approach. We note that, *FR-Refactor* improves F-measure by $32.8\%(74.6\% - 41.8\%)$, and attains better performance in terms of recall (85.3%) because of its ability to learn from past feature requests and predict accordingly, compared to the state-of-the-art approach which attains lower recall (32.1%). In addition to that, *FR-Refactor* improves

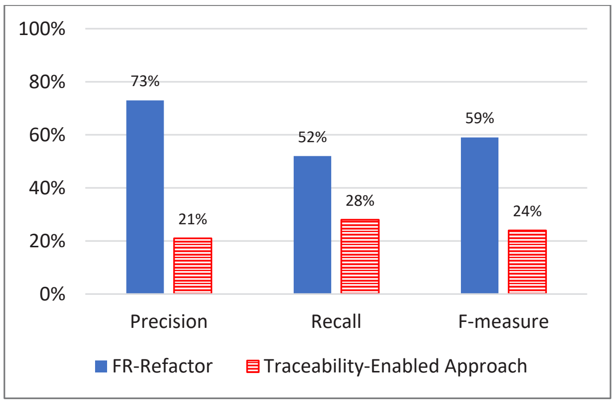


Figure 3.4 Performance in Refactorings Recommendation

accuracy and precision by 19% and 6.6% respectively. The results lead us to the conclusion that, *FR-Refactor* can accurately predict the need for refactoring. Whereas the baseline approach may fail to explicitly identify the requirements semantics and consequently predict the need for refactoring, *FR-Refactor* leverages past history to predict the need for refactoring of a new feature request. For example, the baseline approach failed to identify if a feature request `CAY-1350`: *"Implement memorized sorting of modeler columns"* will need refactoring. However, *FR-Refactor* predicted the need for refactoring for such feature request. This feature request suggests for additional functionality that would allow users to sort items in the table and their preferences should be memorized. Analysis on the history data revealed that, the feature request `CAY-1350` for the additional functionality is similar to past features like (`CAY-1251`: *"Memorize user-selected column widths in preferences"*) and the later feature request (`CAY-1350`) is an improvement request related to (`CAY-1251`). Moreover, the two feature requests were implemented in two different commits but the same type of refactoring (i.e., Extract Method) was applied.

3.4.4.4 **RQ4:** FR-Refactor vs State-of-the-art Baseline Approach in Recommending Refactorings

Fig. 3.4 compares the performance of *FR-Refactor* (based on SVM classifier) against the state-of-the-art approach[94] in recommending refactorings. Generally, the results suggest that, on average *FR-Refactor* outperforms the baseline approach in accurately recommending refactorings. We observe that *FR-Refactor* can accurately recommend relevant refactorings with an average precision of 73% which is equivalent to an improvement of precision by $52\%(73\% - 21\%)$ compared against the baseline approach. Furthermore, *FR-Refactor* significantly improves recall and F-measure by $24\%(52\% - 28\%)$ and $35\%(59\% - 24\%)$ respectively.

We note that, *FR-Refactor* achieves better results than the baseline approach. The baseline approach only relies on the predefined requirements semantics and also ignores the possibility that a given feature request may also belong to different categories of requirements semantics. Consequently, the approach may miss out some relevant refactorings. For example, consider the following part of a feature request.

"Simplify SDK API interfaces. Current SDK API interfaces are not simpler and don't follow builder pattern. If new features are added, it will become more complex". This feature request suggests for enhancing the quality attributes that will lead to the reduction of interface complexity. One way of getting rid of complexity is to allow for separation of concerns and reduce unneeded code. *FR-Refactor* recommended the following refactorings. First, a *Rename Method* and *Extract Method* refactorings. The later refactoring is recommended to decompose long and complex methods. Second, an *Inline Method* refactoring for the calls instances of unneeded methods. Ideally, *Inline Method* refactoring involves replacing calls to the method with its content and delete the method itself. On the other hand, the baseline approach (based on its scheme) only recommended *Substitute Algorithm* refactoring to alleviate *Long Method* flaw which is considered to cause complexity. Furthermore, consider the following part of another feature request. *"Improve broadcast table cache. Currently, broadcast implementation keep a tuples on scan operator and It creates a duplicated table cache in memory"*. This feature request highlights the problem of duplicated feature. The proposed refactorings are *Extract Method*, *Move Method*, and *Rename Class*. The *Extract Method* refactoring is applied to remove code duplication, whereas *Move Method* refactoring to move a method to class which is more functionally related to it. In addition, *Rename Class* is applied to rename the class from which a method was removed to properly reflect its responsibility. In this case, the baseline approach failed to explicitly uncover the requirement's action theme of such feature request and hence was unable to identify the required refactoring types.

3.4.4.5 **RQ5:** Cross Project Evaluation

In the preceding evaluation, we take all dataset from different applications as a whole, and conduct 10-fold evaluation on the resulting dataset. To answer RQ5, in this section we conduct cross project evaluation. We use 40 subject applications for the evaluation, and divide them into 10 groups (each contains 4 applications). On the 10 groups of data, we conduct 10-fold evaluation where testing projects are different from training projects.

Evaluation results suggest that the proposed approach works well even if the testing projects are different from training projects. Its precision, recall and F-measure on binary classification are 65%, 73%, and 68.8%, respectively. Its precision, recall, and F-measure on multi-label classification are 77%, 50%, and 55.4%, respectively. Comparing such results against those in Section 3.4.4.3 and Section 4.4.4.4, we conclude that the proposed approach still obtains good results when testing projects are different from training projects.

3.4.5 Threats to Validity

The major threat relates to the correctness of the recovered refactorings. That is because the leveraged refactorings detection tools are not 100% accurate. The inaccuracy of the refactorings oracle may be accelerated by the fact that the refactoring detection tools may be unable to detect all of the past applied refactorings. Besides that, the tools may suggest incorrect refactorings (false positives) and may leave out some true refactorings (false negatives). As a result, that may lead to inaccuracy of the refactorings oracle. To minimize the threat the dataset were checked for possible errors, however, there could be some errors slipped in unnoticed. That would be due to the lack of the systems knowledge as the process did not involve original developers. Finding original developers is challenging considering the number of the involved applications and some of them have long development history. Threats to external validity is related to the generalizability of the proposed approach. To address this threat, in this study we have considered several feature requests from varied 43 Java open source projects and 14 common refactoring types.

3.5 Conclusion

During software evolution, developers often receive feature requests that demand for the implementation of the new feature or extension of an existing feature. To implement the requested features, developers usually apply refactorings to make their systems adapt to the new requirements. However, deciding what refactorings to apply is often challenging and there is still lack of automated support to recommend refactorings given a feature request. In this chapter we propose a machine-learning-based approach to recommend refactorings based on the history of previous feature requests and applied refactorings. The proposed approach learns from the training dataset associated with a set of applications and can be used to suggest refactorings for feature requests associated with other applications or that associated with the training applications. The proposed approach is evaluated on the dataset of 43 open source Java projects altogether consisting of $13,550$ commits from GitHub repository and $13,367$ feature requests from JIRA issue tracker and their associated refactorings recovered by using the state-of-the-art refactoring detection tools. The experimental results suggest that, the proposed approach can accurately recommend refactorings and attains an average precision of 73%. The possible future research direction includes the following. First, it would be interesting to investigate how to locate where recommended refactorings should be conducted by mining the feature requests and analyzing the related source code. Finally, it would be interesting to investigate how to improve the proposed approach by leveraging word embedding and deep learning techniques.

Chapter 4 A Hybrid Approach to Recommending Refactorings

Software requirements are ever-changing which often lead to software evolution. Consequently, throughout software lifetime, developers receive new requirements often expressed as feature requests. To implement the requested features, developers sometimes apply refactorings to make their systems adapt to the new requirements. However, deciding what refactorings to apply is often challenging and there is still lack of automated support to recommend refactorings given a feature request. To this end, we propose a hybrid machine learning-based approach that recommends refactorings based on the history of the previously requested features, applied refactorings, and code smells information. First, the state-of-the-art refactoring detection tools are leveraged to identify the previous refactorings applied to implement the past feature requests. Second, a machine classifier is trained with the history data of the feature requests, code smells, and refactorings applied on the respective commits. Consequently, the machine classifier is used to predict refactorings for new feature requests. The proposed approach is evaluated on the dataset of 55 open source Java projects and the results suggest that it can accurately recommend refactorings (accuracy is up to 83.19%).

4.1 Introduction

Requirements change is inevitable as the business, technologies, and stakeholder demands continuously evolve[140]. The adaptation to ever-changing software requirements is one of the key factors for the evolution of software systems. During software evolution, developers often receive new requirements expressed as feature requests which demand for the implementation of a new functionality or enhancement of an existing feature. To implement the requested feature, first, developers usually need to locate the source code that should be modified. Next, they often apply refactorings on the located source code to make their systems adapt to the new requirements [94,141]. However, deciding what refactorings to apply is often challenging. Currently, most of the existing refactorings recommendation approaches focus at resolving design flaws in source code commonly known as code smells[6,29,142]. However, the recently conducted case study to investigate the motivation behind refactoring found that refactoring is mainly motivated by the changes in the requirements and very less by code smell resolution[34]. Furthermore, the empirical analysis of refactorings from software repositories found that refactorings are most commonly applied by developers for a specific goal such as implementing a feature or bug fixing, than in the refactoring sessions dedicated for evolving the software design[26,71]. This finding was further confirmed by Palomba *et al.*[72] in the exploratory study on the relationship between changes and refactoring. Silva *et al.*[34] advocate the need for refactorings recommender systems that focus on facilitating maintenance tasks (i.e., implementing feature requests or fixing bugs). However, to the best of our knowledge, there is still lack of automated support to recommend refactorings during the implementation of feature requests.

To this end, in this chapter we propose a learning-based approach that recommends refactoring types based on the history data of the previously requested features, applied refactorings, and code smells information. The proposed approach learns from the training dataset associated with a set of applications and can be used to suggest refactoring types for feature requests associated with other applications (or new feature requests associated with the training applications). Our approach involves two classification tasks: first a binary classification that suggests whether refactoring is needed or not for a given feature request, and then a multi-label classification that suggests the type of refactoring. Notably, the proposed approach suggests refactoring types only and it does not point to the locations in the codebase for the recommended refactoring. However, it could be integrated with other approaches/tools that could suggest such locations, and thus makes more complete refactoring suggestions. The proposed approach also helps developers pick up proper refactoring tools by suggesting refactoring types. The past feature requests and their associated commits are retrieved from the corresponding issue trackers and software repositories respectively. Usually, each feature request can be linked to the respective commits that addressed the request through a unique feature request identifier which is often added in the commits' messages. The proposed approach is evaluated on the dataset of 55 open source Java projects altogether consisting of $18,899$ feature requests from JIRA issue tracker. The evaluation results suggest that, the proposed approach can accurately recommend refactoring types and attain an accuracy of up to 83.19%.

The contributions of this chapter include the following:

- The approach presented in Chapter 3 is extended to leverage code smells in source code besides feature requests. With such additional information, the proposed approach is improved significantly. The precision and recall for predicting the need for refactoring are improved from 66.36% to 80.75%, and from 85.32% to 93.76%, respectively. In addition, the accuracy of refactorings recommendation has improved from 70.75% to 83.19%.
- We evaluate the proposed approach with larger dataset. The number of involved feature requests has been increased significantly by 41%.
- We investigate additional research question(RQ3) about the impact of code smells on refactorings recommendation.
- We implement our approach on three additional classifiers (i.e., Random Forest, Decision Tree, and Convolutional Neural Network) that were not implemented in 3.

4.2 Background

Several approaches have been proposed to employ machine learning techniques to detect refactoring opportunities and recommend refactorings. For example, Ratzinger *et al.*[141] leveraged change history mining to extract features that can be used to predict the need for refactoring in the next two months by using machine learning classifiers. Besides extracting features from evolution data, they also identified the changes applied as either refactoring or not based on the commit messages. However, their prediction models do not distinguish different refactorings types (e.g., rename class, extract method, etc.). In contrast, our approach explicitly suggests the classes of refactorings required. Liu *et al.*[143] proposed an approach that leverages deep learning techniques to automatically generate labeled training set consisting of methods with or without feature envy. Consequently, such training set is used to train the neural network classifier to predict whether a given method envies another class. Besides that, the classifier also predicts the potential target class where the envy method should be moved to. The structural and textual information were used to decide whether a given method should be moved to another class. Structural information computes how close a method is to the target classes, whereas textual information reveals the semantic relationship between methods and classes. Recently, Pantiuchina *et al.*[144] developed a refactoring recommender that targets at preventing the introduction of code smells into the codebase. Their approach relies on machine learning techniques to train the classifier that predicts classes which are likely to be affected by a particular smell in near future. The approach takes a change history of a system with its latest version as an input and deduces the historical trend of 14 predefined class quality metrics. Consequently, they compute regression slope line fitting the values of the metrics for each class that would be used to infer whether the quality is degrading or not. For example, a high positive slope for the WMC (Weighted Methods per Class) metrics indicates that the complexity of a given class is strongly increasing.

In addition, Xu *et al.*[29] proposed a machine learning based approach that learns a probabilistic model to recommend $Extract\ Method$ refactorings. The proposed approach extracts structural and functional features from software repositories which encode the concepts of complexity, coupling, and cohesion. Based on these features the approach learns to extract appropriate code fragments from a source of a given method. The proposed approach called `GEMS` is developed as an Eclipse plug-in for Java programs. Xu *et al.*[29] contend that, usually human involvement is required in identifying true refactorings which often leads to the use of small-sized datasets for efficiency. However, to allow working on large datasets and ensure correct recommendations, the deployment of machine learning based approaches is inevitable. Yue *et al.*[145] proposed a learning-based approach that recommends code clones (i.e., duplicated code) for refactoring by training the machine learning models with features extracted from refactored and non-refactored clones mined from software repositories. The authors defined 5 categories of key features that characterize clones' content, evolution history, co-evolution relationship, as well as spatial locations of clone peers and syntactic difference between clones. The findings suggest that, history-based features are effective in recommending refactorings than the features extracted from the present version of a software. This leads to a very interesting observation that, when applying refactoring, developers mostly consider the past history than the existing version. These approaches, however, differ from our proposed approach as they do not consider feature requests in their recommendation.

4.3 Approach

4.3.1 Overview

As depicted in Fig. 4.1, the proposed approach follows the following six key steps to predict the need for refactoring and recommend the required refactorings. First, we extract the feature requests from the issue tracker (JIRA) and their respective commits from a software repository (GitHub). Usually, the two artifacts (feature requests and commits) can be linked through a unique feature request identifier. Second, we recover the previously applied refactorings on the retrieved commits. Third, by using the state-of-the-art tool we identify the code smells associated with the source code in each of the retrieved commits. The output from the previous two steps is the file containing feature requests with their associating refactorings and code smells. Fourth, we apply the text preprocessing on the contents of the file to prepare the textual data suitable for the next steps. Fifth, feature modelling is applied to convert the textual data into a numerical representation (feature vectors) for training the classifiers. Finally, we train the machine-learning-based classifier which gives a prediction model for predicting and recommending refactorings for new feature requests. To avoid redundancy, this chapter will not discuss the steps for feature requests and commits extraction, refactorings detection, text preprocessing, and feature modelling, but rather we refer the reader to Sections 3.3.1, 3.3.2, 3.3.3, and 3.3.4 respectively. Therefore, in the current chapter we focus our discussion on a newly introduced step i.e., detection of code smells and the training and recommendation step which has been slightly modified to include code smells as input. Next, we elaborate these two steps in detail.

4.3.2 Detection of Code Smells

To uncover code smells we leveraged an automated code review tool (i.e., Codacy)[146] which applies static analysis to identify design issues in software repositories. The tool is powered by PMD[147] to implement the set of rules which are the basis for identification of design issues in the codebase. Therefore, to determine how code smells associate with a feature implementation, we first establish a link between a feature request and the commit through feature request identifier. This aspect is well described in Section 3.3.1. Next, by

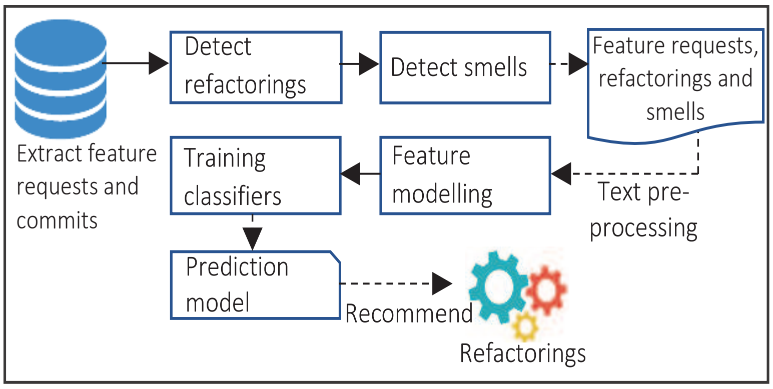


Figure 4.1 The Framework of the Proposed Approach

using Codacy, we detect code smells that were introduced by such particular commit. We employed this tool because it supports a variety of languages and it is publicly available for reviewing open source projects. Besides that, the tool can be easily integrated with GitHub repositories and provide quality analysis on the source code files which are to be modified to implement the requested feature. It follows that, for the past feature requests, we identified the list of smells that were fixed in each commit that implemented such feature requests. Thus, a feature request that has been identified as needing refactoring in Formula 3.3 will further be presented as:

$$fr =< frID, summ, desc, commitID, ref, smells > \tag{4.1}$$

where $smells$ is the set of code smells related to a commit $commitID$.

The information of code smells is leveraged to further enrich the feature set that could improve the discrimination of feature requests and consequently improve the prediction performance. Therefore, the feature requests' summary $summ$, description $desc$, and code smells $smells$ will serve as the primary source of feature set that will be used to train our classifiers that target to predict refactorings ref. The usage of feature requests as presented in Formula 4.1 is described in detail in Section 4.3.3. Furthermore, it is worth noting that, Formula 3.2, 3.3, and 4.1 do not imply that, a given a feature request can only be related to only one commit. But rather, they just show a tuple that contains a feature request and related commit. However, one feature request could have several tuples relating it to different commits.

4.3.3 Training and Recommendation

The proposed approach leverages six widely-used machine learning classifiers: Logistic Regression (LR), Multinomial Naïve Bayes (MNB), Support Vector Machine (SVM), Decision Tree (DT), Random Forest (RF) and Convolutional Neural Network (CNN). The classifiers have been implemented by a well-known machine learning library based on Python called scikit-learn[132]. We have specifically selected such machine learning algorithms for various reasons including the nature of our classification problem and effectiveness of the individual classifier. Usually, text classification is attributed with high dimensional input space which often leads to a lot of features[148,149]. For example, SVM classifiers are effective in managing high dimensional feature space and removing irrelevant features[149]. Besides that, SVM classifiers are effective in handling sparsity problem and well recognized as accurate text classifiers[136]. Furthermore, Jiang *et al.*[150] suggest that, Naïve Bayes classifiers are widely-used to address the classification problem in the domains with large number of attributes such as text classification. Naïve Bayes classifiers are relatively effective, fast and easy to implement[150]. Given the assumption of attribute conditional independence of Naïve Bayes classifiers, the parameters for each attribute can be estimated easily and separately which consequently simplifies learning[150]. In summary, these classifiers can handle the problem of sparseness and high dimensionality. Such properties of the classifiers apply to our dataset as well which consequently make them optimal for our approach. Generally, the employed classifiers have been widely used and shown to be effective in text classification[130,133].

We generally model our text classification problem such that, we have a feature request fr from the feature requests space FR such that $fr \in FR$. Each feature request fr may belong to one or more classes from the set of fixed classes R, where $R = \{r_1, r_2, ..., r_{\mathrm{m}}\}$. In our case here, classes are also referred to as labels or refactoring types. Suppose we have a training set T of labeled feature requests (fr, r), where $(fr, r) \in FR \times R$. Our goal is to train a classifier (or a classification function) f that maps feature requests to classes (i.e., recommended refactoring types):

$$f : FR \rightarrow R \tag{4.2}$$

Our approach involves two classification tasks, i.e., binary and multi-label classification. As we mentioned earlier, the main reason for that is, predicting the need for refactoring and recommending refactoring types can be formulated independently as binary and multi-label classification problems respectively. Besides that, not all feature requests may require refactoring and therefore it is not logical to predict refactoring types for such cases. Consequently, separating the two tasks could boost the training and recommendation performance of the classifiers. In the following we describe these two tasks in details.

- **Binary classification**, at this first stage the classifier is trained to determine whether refactoring is needed or not. The input to the classifier is all feature requests (noted as $allReq$) and the goal is to classify them into two categories: those that deserve refactoring (noted as $desReq$) and those that do not deserve refactoring (noted as $otherReq$). The training set contains the texts (i.e., summary and description) of the feature requests, code smells information, along with their labels. At this stage, the labels i.e., ref (see Formula 4.1) are presented as 1 if refactoring is needed or 0 otherwise. The test data contains feature requests for which we need to predict if they would need refactoring or not. The classifier categorizes a feature request fr into a class c as function f such that:

  $$c = f(fr), \quad c \in \{0, 1\}, \quad fr \in FR \tag{4.3}$$

  where c represents the classification result: 0 implies a feature request fr does not require refactoring whereas 1 implies that refactoring is needed.

  Consequently, the feature requests which are identified to need refactoring will serve as input to the next stage to identify the types of refactorings required.

- **Multi-label classification**, after identifying the feature requests that require refactoring (noted as $desReq$), the multi-label classification is performed to predict the specific types of refactorings which are required. Multi-label classifiers are leveraged because a given feature request may involve more than one type of refactoring. To train the classifier, past feature requests (i.e., those identified to need refactoring), code smells information, and the applied refactorings will serve as input (see Formula 4.1). The classifier will then be tested with data containing feature requests and their associating code smells to recommend the types of refactorings that would be needed. The classifier is therefore trained to categorize a feature request fr into a class c as function f such that:

$$c = f(fr), \ \ c \subseteq R, \ \ fr \in FR \tag{4.4}$$

where c is the set of one or more refactoring types.

In summary, the classifiers are generally trained to determine whether refactoring is required or not, if yes then they should predict (i.e., recommend) the types of refactoring required for the unseen feature requests.

4.4 Evaluation

In this chapter we present the evaluation of our approach that we still refer to as *FR-Refactor* (Feature-Request-based Refactoring). As pointed earlier, *FR-Refactor* is compared against the state-of-the-art approach proposed by Niu *et al.*[94] to assess it's efficacy in predicting the need for refactoring and recommending required refactoring types.

4.4.1 Research Questions

The evaluation investigates the following research questions:

- **RQ1**: How accurate are different machine learning classifiers in predicting the need for refactoring?
- **RQ2**: How accurate are different machine learning classifiers in recommending required refactoring types?
- **RQ3**: How does the code smells information influence the performance of the classifiers?
- **RQ4**: How accurate is *FR-Refactor* in predicting the need for refactoring compared to the state-of-the-art baseline approach?
- **RQ5**: How accurate is *FR-Refactor* in recommending required refactorings compared to the state-of-the-art baseline approach?

Table 4.1 The Distribution of Feature Requests (FR) in Our Dataset

Project	**FR**	**Project**	**FR**	**Project**	**FR**	**Project**	**FR**
Accumulo	474	Flink	555	Nifi	430	Spring-ROO	797
ActiveMQ	753	Geronimo	38	Nutch	348	Spr-Security	87
Ambari	391	Giraph	252	Ode	102	Stanbol	154
Archiva	193	Gora	31	Oodt	26	Storm	302
Aries	314	Groovy	385	Oozie	318	Struts 2	214
Atlas	70	Hbase	1389	Opennlp	392	Synapse	79
Axis2-Java	204	Impala	178	PDFBox	482	Syncope	599
Beam	295	Jclouds	122	Pig	577	Systemml	34
Bookkeeper	132	Jena	202	Pivot	185	Tajo	444
Calcite	74	Kafka	307	Ranger	278	Tapestry-5	381
Carbondata	226	Kylin	680	Sentry	87	Velocity	56
Cayenne	390	Lens	172	Sling	2369	Wicket	237
Curator	24	Maven	359	Spring-Datamongo	261	Zookeeper	80
Drill	446	Myfaces	407	Spring-Integration	517		
TOTAL:			Projects: 55, Feature Requests: 18,899				

- **RQ6**: Can the proposed approach still obtain good results on applications that are different from those involved in the training?

Note that, these research questions, except **RQ3**, were already explained in detail in Section 3.4.1. However, for clarity we repeat them here but in this case they are investigated with respect to including code smells in predicting and recommending refactorings. Specifically, in this chapter, **RQ3** aims at exploring how the addition of code smells information influences the classification performance of the proposed approach.

4.4.2 Dataset

To evaluate the proposed approach, we expanded the dataset created in Section 3.4.2 by adding more feature requests (41% addition) and their associated refactorings. We note that, such dataset did not include code smells information. The dataset is therefore expanded to significantly increase the number of examples with code smells and generally to further improve the classification performance. Notably, machine learning classifiers often achieve high performance with larger datasets[151]. The code smells associated with each commit were identified by an automated code review tool (i.e., Codacy)[146] which is powered by

PMD[147] and applies static analysis to detect design issues in software repositories. The tool is specifically selected because it is publicly available for open source projects and can be easily integrated with GitHub repository.

Table 4.1 highlights the distribution of the feature requests of the subject applications used in creating our dataset. The dataset contains 55 open source Java projects consisting of $18,899$ feature requests from JIRA issue tracker. Out of $18,899$ feature requests, a total of $9,915$ ($52\% = 9,915/18,899$) feature requests are associated with one or more refactorings, whereas the remaining 48% ($= 8,984/18,899$) of the feature requests do not have any refactorings. The dataset is publicly available on GitHub (https://github.com/nyamawe/FR-Refactor). As pointed earlier, the subject applications forming up our dataset were selected because they cover a wide range of domains, publicly available, developed by different developers, and have long evolution history. Consequently, it is more likely that they will have a variety of feature requests, code smells, and refactorings.

Furthermore, Table 4.2 summarizes the distribution of refactoring types in our dataset. Additionally, Fig. 4.2 presents the distribution of the number of refactoring types (i.e., all labels excluding no-refactoring category) per each data point (i.e., feature request). From the figure we observed that, in our dataset, most ($63\% = 6,221/9,915$) of the feature requests are associated with only one refactoring type, whereas only 37% ($= 3,694/9,915$) are associated with more than one refactoring type. However, such amount is significant which is why we casted the refactoring recommendation problem as multi-label classification problem. We further noted that, there was not feature request that had more than 10 refactoring types.

4.4.3 Process and Metrics

The evaluation of the proposed approach follows the two key steps which we described in detail in Section 3.4.3. On one hand, to evaluate the multi-label classifiers which are employed for recommending refactoring types we use the widely-used metrics namely: $Subset\ accuracy$ (Formula 3.11), $Hamming\ score$ (Formula 3.10), and $Hamming\ loss$ (Formula 3.9). Such metrics are well described in Section 3.4.3. To evaluate the binary classifiers we leveraged the traditional accuracy, precision, recall, and F-measure metrics which are for-

Table 4.2 The Distribution of Refactoring Types in our Dataset

Refactoring types	**Number**
Extract Interface	192
Extract Method	5,310
Extract Superclass	176
Inline Method	997
Move And Rename Class	357
Move Attribute	1,244
Move Class	975
Move Method	1,884
Pull Up Attribute	248
Pull Up Method	339
Push Down Attribute	136
Push Down Method	159
Rename Class	1,316
Rename Method	3,664
Total	16,997

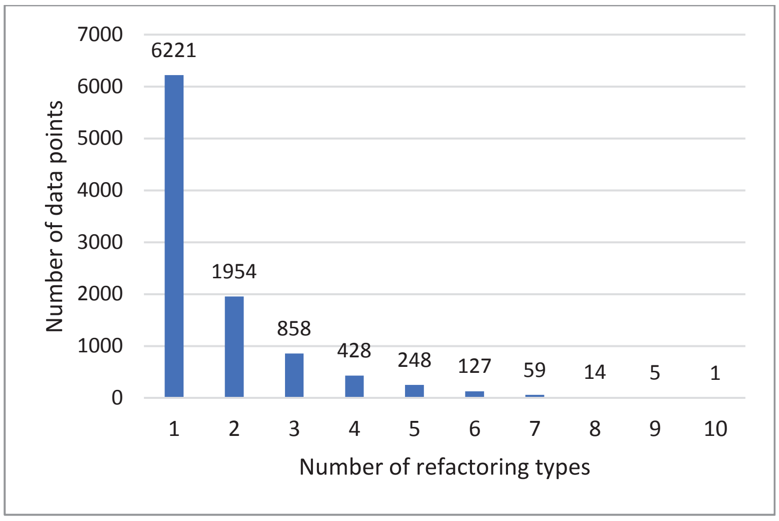


Figure 4.2 Number of Refactoring Types per Data Points

malized as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.5}$$

$$Precision = \frac{TP}{TP + FP} \tag{4.6}$$

$$Recall = \frac{TP}{TP + FN} \tag{4.7}$$

$$F - measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.8}$$

where TP is the number of feature requests that are correctly predicted as they require refactoring, TN is the number of feature requests that are correctly predicted as they do not require refactoring, FP is the number of feature requests that are incorrectly predicted as they require refactoring, and FN is the number of feature requests that are incorrectly predicted as they do not require refactoring.

4.4.4 Results and Analysis

4.4.4.1 **RQ1:** Performance of Different Classifiers in Identifying the Need for Refactoring

To investigate which classifier will be effective in predicting the need for refactoring when implementing a given feature request, we evaluated the performance of six machine learning algorithms: SVM, MNB, LR, RF, DT, and CNN. Table 4.3 presents the effectiveness of each classifier in terms of accuracy, precision, recall, and F-measure. Note that, the best recorded result for each metric is highlighted in bold. From the table, it is observed that the accuracy of predicting whether refactoring would be required ranges from 70.44% to 76.01%. The results further suggest that, on average precision (see column 2) of up to 78.12% the need for refactoring can be accurately predicted. Furthermore, the results generally indicate that, on average, MNB and LR classifiers outperform all other classifiers. MNB classifier achieved an F-measure of 86.77% and an accuracy of 76.01%, whereas, LR classifier achieved an F-measure of 82.75% and can accurately predict the need for refactoring with up to 79.37% precision. Furthermore, from the table we can observe that, a convolutional neural network (CNN) classifier has performed slightly lower than the rest of the classifiers. One possible reason for that is, deep learning classifiers generally require significantly large datasets to achieve a competitive performance. We, therefore, conclude the preceding analysis that Multinomial Naïve Bayes (MNB) classifier turned out to be the best classifier in this case. Since the evaluation results suggest that MNB works best in predicting whether refactoring is required to implement a given feature request (noted as binary classification), MNB would be used in the rest of the chapter for the binary classification.

The possible potential challenges that may have hindered higher prediction accuracy in-

Table 4.3 Classifiers' Performance for Predicting the Need for Refactoring (%)

Classifier	**Accuracy**	**Precision**	**Recall**	**F-measure**
SVM	74.08	79.58	86.14	82.73
MNB	**76.01**	**80.75**	**93.76**	**86.77**
LR	74.00	79.37	86.43	82.75
RF	73.78	79.51	82.79	81.12
DT	73.21	75.58	82.35	78.82
CNN	70.44	73.90	83.06	78.21

clude: 1) for the same feature request, developers may propose different solutions (implementation strategies), which has significant influence on the refactoring activities. Consequently, predicting the refactoring while implementation strategies are not specified is natively challenging. 2) Refactoring activities are not indispensable even if they are helpful for the given programming task. Consequently, many factors, e.g., the preference/experiences of the developers, the schedule of the team, and the employed QA measures may help in accurate prediction of refactorings. However, it is challenging to get such information in advance, and thus the prediction of refactoring is challenging. Although precision is relatively low (76.5%), recall is higher (89.37%). Consequently, based on our prediction, developers can apply refactoring tools for suggested feature requests only and thus save significant cost.

4.4.4.2 **RQ2:** Performance of Different Classifiers in Recommending Refactoring Types

As mentioned earlier, the refactoring types recommendation is cast here as a multi-label classification problem, hence Table 4.4 summarizes the results in terms of subset accuracy, hamming score, and hamming loss which are widely used metrics for evaluating multi-label classifiers. The best recorded result for each metric is presented in bold. Note that, subset accuracy (also called $classification\ accuracy$) returns the percentage of instances where the set of labels (i.e., refactoring types) predicted by the classifier is exactly the same with their corresponding truth set[139]. Generally, as shown in Table 4.4, SVM turned out to be the best performing classifier with the recorded subset accuracy and hamming score of 83.19% and 83.03% respectively. Therefore, this implies that, the needed refactoring types can be accurately recommended with up to 83.19% accuracy. Furthermore, from Table 4.4, we make

Table 4.4 Classifiers' Performance for Refactoring Types Recommendation(%)

Classifier	**Subset accuracy**	**Hamming score**	**Hamming loss**
SVM	**83.19**	**83.03**	**0.031**
MNB	40.53	39.92	0.059
LR	81.88	81.26	0.033
RF	63.72	63.84	0.041
DT	60.31	59.99	0.040
CNN	75.44	75.81	0.047

the following observations:

- In the case of multi-label classification, SVM classifier has significantly outperformed MNB, RF, and DT classifiers by the difference of 42.66, 19.47, and 22.88 percentage point on subset accuracy respectively. On the other hand, SVM has only outperformed LR marginally in both metrics.

- Compared to other classifiers, SVM achieved the lowest value (i.e., 0.031) on hamming loss which identifies to what extent the classifier predicts the irrelevant refactoring types and omits relevant refactoring types. This metric is normalized between 0 and 1. As the value of the metric closes to 0, it indicates better performance of the classification.

Since the preceding results analysis suggests that SVM works best in suggesting refactoring classes (noted as multi-label classification), SVM would be used in the rest of the chapter for the multi-label classification.

Moreover, we investigated the performance of SVM in recommending individual refactoring types. As depicted in Table 4.5, generally the results suggest that the classifier achieves an average precision of 76% in recommending refactoring types. We further observed that, the classifier achieved the best results (72%) in terms of F-measure for the Extract Method refactoring. The possible reason for this best performance comparing to the F-measure results recorded for other refactorings could be the amount of such refactorings in the dataset. If we refer to Table 4.2, Extract Method refactoring is the one with the most (31%) representative examples in the dataset. This is due to the fact that, in practice, certain refactoring types are more prevalent than the others. For example, the recently conducted empirical study by Silva

Table 4.5 Results for Individual Refactoring Types Recommendation

Refactoring type	**Precision**	**Recall**	**F-measure**
Extract Interface	0.63	0.61	0.62
Extract Method	0.68	0.77	0.72
Extract Superclass	0.85	0.41	0.55
Inline Method	0.84	0.39	0.53
Move And Rename Class	0.89	0.54	0.67
Move Attribute	0.86	0.37	0.52
Move Class	0.78	0.43	0.55
Move Method	0.85	0.47	0.67
Pull Up Attribute	0.69	0.60	0.64
Pull Up Method	0.67	0.40	0.50
Push Down Attribute	0.81	0.36	0.50
Push Down Method	0.83	0.35	0.49
Rename Class	0.75	0.42	0.54
Rename Method	0.69	0.55	0.61
Weighted avg	0.76	0.54	0.61

et al.[34] on the refactorings performed by developers on 539 commits of 185 java projects retrieved from GitHub found that, Extract Method is the most popular high-level refactoring. On the other hand, in some few cases (e.g., Push Down Method) the classifier achieved a recall and F-measure below 50%. This can be justified by the fact that some of the refactoring types, including Push Down Method and Push Down Attribute have very few instances in the dataset (see Table 4.2) which can consequently affect their prediction. The same study by Silva *et al.*[34] also found that Push Down refactorings were among the least popular refactorings. It is therefore challenging to create a dataset with sufficient number of refactorings and yet balance the number of representative examples for each refactoring type. In view of that, we implemented the classifier which is based on the distribution of refactorings in our dataset. The classifier predicts a given feature request fr will deserve a refactoring type r at a chance of $fr(r)$ such that:

$$fr(r) = \frac{|FR(r)|}{|allFR|} \tag{4.9}$$

where, $FR(r)$ are all feature requests that are associated with refactoring r in the training set, and $allFR$ are all feature requests in the training set.

On average, the classifier recorded the precision, recall and F-measure of 79%, 63%,

Table 4.6 Classifiers' Performance for Predicting the Need for Refactoring (without Code Smells) (%)

Classifier	**Accuracy**	**Precision**	**Recall**	**F-measure**
SVM	71.96	77.09	80.63	78.82
MNB	**73.10**	76.51	**89.37**	**82.44**
LR	72.17	77.74	80.70	79.19
RF	69.49	**77.82**	78.77	78.29
DT	68.23	76.68	78.01	77.34
CNN	64.36	69.92	78.45	73.94

and 70% respectively for recommending individual refactoring types. Such performance is slightly higher than that reported earlier in Table 4.5 where a classifier attained an average precision, recall and F-measure of 76%, 54%, and 61% respectively. We therefore conclude the preceding analysis that, the proposed approach is accurate in recommending refactoring types.

It is worth noting that, in general binary classification should be more accurate than multi-label classification if they are applied to the same dataset. However, in our case, they are applied to different datasets, and thus there is no obvious relation between their performances. As mentioned earlier in Section 4.3.3, we first apply the binary classifier to all of the feature requests (noted as $allReq$), and such feature requests are classified into two categories: those that deserve refactoring (noted as $desReq$) and those that do not deserve refactoring (noted as $otherReq$). The multi-label classifier is then applied to the $desReq$ only, and $otherReq$ is not involved. The difference in their testing data is the major reason for the confusing phenomenon, i.e., the multi-label classification achieves higher accuracy than the binary classification.

4.4.4.3 **RQ3:** The Influence of Code Smells Information

This evaluation explores the influence of code smells information on the classifiers' performance in predicting refactoring and recommending the required refactoring types. In contrast to other binary and multi-label classification experiments reported in this study, in this case we trained our classifiers with only feature requests and refactoring types. That means we excluded code smells information. Consequently, the classifiers were tested to predict

Table 4.7 Classifiers' Performance for Refactoring Types Recommendation (without Code Smells) (%)

Classifier	**Subset accuracy**	**Hamming score**	**Hamming loss**
SVM	69.21	70.46	0.026
MNB	24.73	24.79	0.059
LR	69.21	69.93	0.026
RF	27.55	27.63	0.036
DT	22.14	22.16	0.039
CNN	66.53	66.97	0.042

refactoring and recommend refactoring types for new feature requests. The results for predicting refactoring and recommending refactoring types are respectively reported in Table 4.6 and Table 4.7. Note that, the results reported in these tables are based on the previous approach proposed in[152] which do not consider code smells in it's recommendation. Consequently, to analyze the influence of code smells on the classifiers' performance we compare the results obtained by the previous approach[152] reported in Table 4.6 and Table 4.7 (noted as without code smells) with that obtained by our proposed approach reported earlier in Table 4.3 and Table 4.4 (noted as with code smells) respectively.

To investigate the influence of code smells information on classifiers' performance in predicting the need for refactoring we compare the results reported in Table 4.3 and Table 4.6. We note that, this comparison includes two more additional machine learning classifiers (i.e., Decision Tree and Neural Network) that were not implemented in[152] for binary classification. To clearly visualize the change in performance, we graphically depict such comparison in Fig. 4.3 in terms of accuracy. From the figure we observe that, code smells information slightly increased the performance of the classifiers. The noticeable increase in performance of around 5 percentage point is recorded for RF, DT, and CNN classifiers. Such a slight change can be justified by the fact that, it is not necessarily that only feature requests that associate with code smells will demand refactoring.

Furthermore, to visualize the influence of code smells on refactoring recommendation we compare the results reported in Table 4.4 and Table 4.7. Also note that, this comparison includes three more additional machine learning classifiers (i.e., Random Forest, Decision Tree and Neural Network) which were not implemented in[152] for multi-label classification. For clarity, the graphical representation of that comparison is depicted in Fig. 4.4 in terms

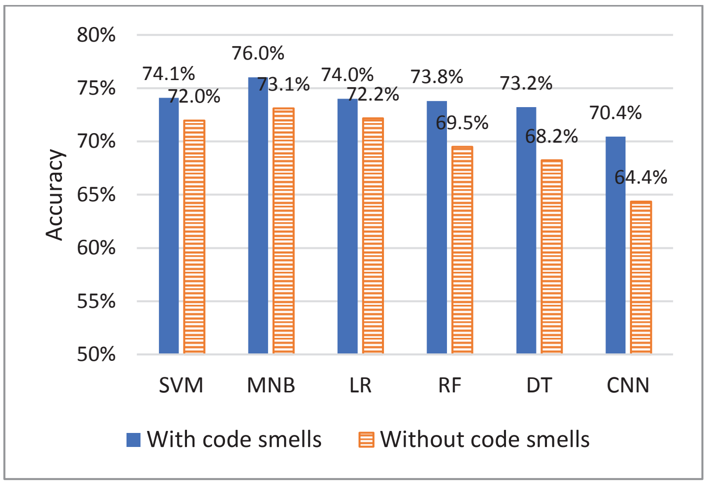


Figure 4.3 The Influence of Code Smells Information on Refactoring Prediction Performance

of subset accuracy. From the figure we observe that, code smells information positively influenced the performance of the classifiers. For example, SVM classifier recorded a performance increase of 14 $(= 83.2\% - 69.2\%)$ percentage point when including code smells information as input to the classifier along with feature requests and refactoring types during training. Such trend can also be observed in the rest of the classifiers where MNB, LR, RF, DT, and CNN respectively recorded the change of 15.8, 12.7, 36.1, 38.2, and 8.9 percentage point by including code smells information. Notably, only 58% of the feature requests and refactorings in our dataset were associated with code smells. This implies therefore that, the refactorings recommendation is somewhat not biased. Therefore, the possible reason for such improvement is that, code smells have been proven useful in accessing (e.g., based on smells catalogs) the special kinds of software refactorings required to alleviate them[153]. Consequently, understanding the specific smell type along with a given feature request can lead to accurate prediction of a required refactoring type. That is because, often a given code smell could have multiple refactoring solutions[23,94]. We, therefore, conclude the preceding analysis that inclusion of code smells information boosts the accuracy of refactoring types recommendation.

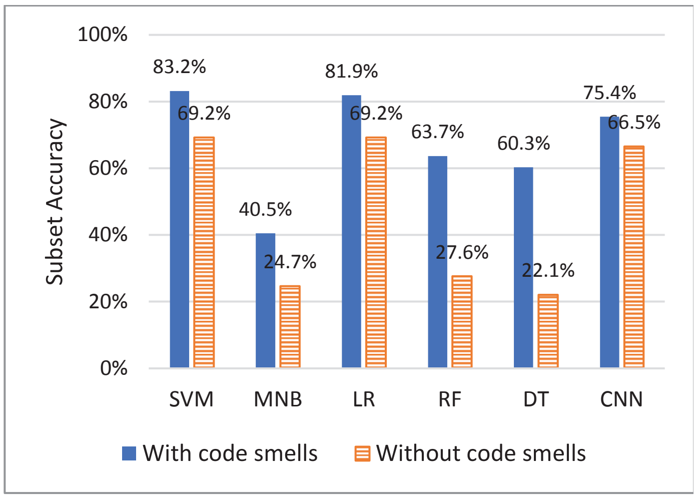


Figure 4.4 The influence of Code Smells Information on Refactoring Types Recommendation Performance

4.4.4.4 **RQ4:** FR-Refactor vs State-of-the-art Approach in Predicting the Need for Refactoring

To investigate the performance of *FR-Refactor* in predicting the need for refactoring, we compared it (based on MNB classifier) with the state-of-the-art traceability-enabled approach proposed by Niu *et al.*[94]. The results of the comparison are depicted in Fig. 4.5. Generally, results suggest that *FR-Refactor* (based on MNB classifier) significantly outperforms the state-of-the-art approach. From the figure we observe that, *FR-Refactor* significantly achieved an increase in F-measure of 43.3 $(82.4\% - 39.1\%)$ percentage point. Moreover, *FR-Refactor* achieves better performance in terms of recall (89.4%) because of its ability to learn from past feature requests and predict accordingly, compared to the state-of-the-art approach which attains lower recall (28.5%). In addition to that, *FR-Refactor* improves accuracy and precision by 27.6 $(= 73.1\% - 45.5\%)$ percentage point and 12.1 $(= 76.5\% - 64.4\%)$ percentage point respectively. The results lead us to the conclusion that, *FR-Refactor* can accurately predict the need for refactoring.

4.4.4.5 **RQ5:** FR-Refactor vs State-of-the-art Approach in Recommending Refactoring Types

To answer research question **RQ5** we compared the performance of *FR-Refactor* (based on SVM classifier) and the state-of-the-art traceability-enabled approach[94] in recommend-

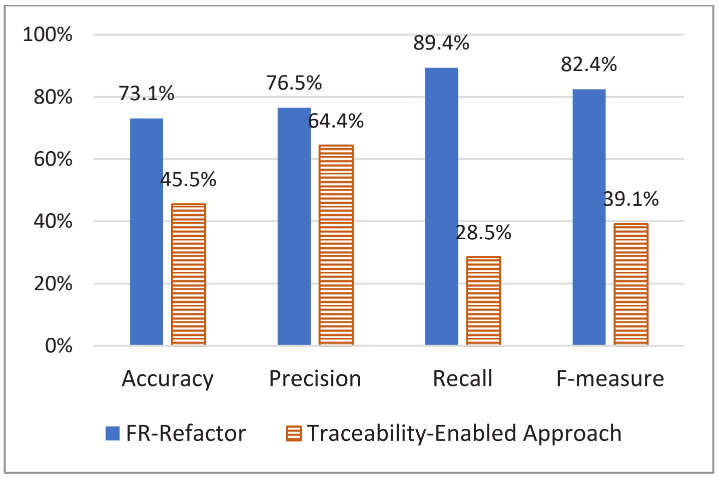


Figure 4.5 Performance in Predicting the Necessity of Refactoring

ing refactoring types. Evaluation results are presented in Fig. 4.6. From the figure we make the following observations:

- First, *FR-Refactor* can accurately recommend relevant refactoring types for most of the feature requests. Its average precision is up to 76%. Compared to the state-of-the-art baseline approach, it improves precision significantly by 56 $(= 76\% - 20\%)$ percentage point.

- Second, *FR-Refactor* significantly outperforms the state-of-the-art in terms of F-measure. It achieved an average F-measure of 61%. This is equivalent to an improvement of 36 $(= 61\% - 25\%)$ percentage point compared to the baseline approach which recorded an F-measure of 25%.

- Third, the recall (54%) of *FR-Refactor* is significantly higher than that of the baseline approach (34%).

Based on the preceding analysis, we conclude that *FR-Refactor* achieves better results than the baseline approach. The possible reasons for the underperformance of the baseline approach are well described in Section 3.4.4.4.

4.4.4.6 **RQ6:** Cross Project Evaluation

This evaluation aims to explore the effectiveness of the proposed approach in predicting and recommending refactoring solutions for the feature requests that their applications

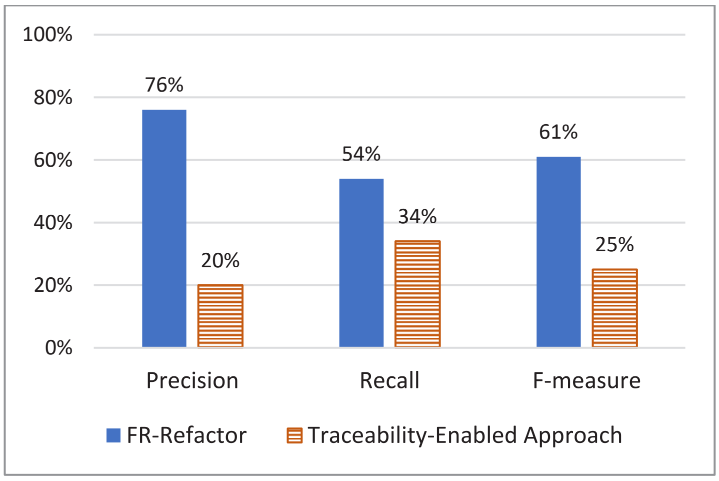


Figure 4.6 Performance in Refactoring Types Recommendation

were not involved in the training of classifiers. Basically, there might exist a case where some applications have not accumulated enough history, hence it is crucial to understand if the history of other applications can be useful to recommend refactorings for such applications with insufficient history. Note that, in the preceding experiments we take all the dataset from different subject applications as a whole, and conduct 10-fold evaluation on the resulting dataset. To answer **RQ6**, in this section we conduct a cross project evaluation. We use 50 subject applications for the evaluation, and divide them into 10 groups (each contains 5 applications). On the 10 groups of data, we conduct 10-fold evaluation where testing projects are different from training projects. Evaluation results suggest that the proposed approach performs well even if the testing applications are different from training applications. Its precision, recall, and F-measure on binary classification are 72.86%, 81.92%, and 77.12%, respectively. Whereas, its precision, recall, and F-measure on multi-label classification are 81.54%, 59.37%, and 68.71%, respectively. Comparing such results against those reported in Section 4.4.4.4 and Section 4.4.4.5, we conclude that the proposed approach still obtains good results when testing projects are different from training projects.

One possible reason for the success is that the classifiers can learn some generic rules applicable to different applications, e.g., the presence of special code smells may suggest a specific type of refactoring actions, and some special words or phrases in feature requests may suggest the necessity of refactorings. To investigate the latter, we slightly adopted the comprehensive list of patterns which are potential in inferring refactoring-related activities proposed recently by AlOmar *et al*[154]. We note that, such patterns were drawn from commit messages and shown to have significant correlation with refactoring-related changes applied on their associated commits. Consequently, from the list of patterns we identified 24 unique and representative keywords which we used to evaluate the feature requests. For each keyword w we computed how likely (denoted as $RefactoringRate(w)$) the feature requests containing such keyword can be associated with refactoring. $RefactoringRate(w)$ can be formalized as follows:

$$RefactoringRate(w) = \frac{|refactoredFR(w)|}{|allFR(w)|} \tag{4.10}$$

where, $refactoredFR(w)$ are feature requests that contain the keyword w and deserve refactoring, and $allFR(w)$ are all feature requests that contain the keyword w.

For clarity, we only list the top ten keywords with the highest $RefactoringRate$ in Table 4.8. Other leveraged keywords not listed include: $Replace$, $Improve$, $Extract$, $Reduce$, $Move$, $Change$, $Import$, $Enhance$, $Modify$, $Remove$, $Better$, $Avoid$, Add, and Fix. The $RefactoringRate$ of these keywords ranges from 54% to 65%. We noted that, the feature requests with such keywords tend to have more significant refactoring rate than those without. Generally, the keywords clearly indicate the intention of a feature request such as improving or optimizing a quality attribute. Additionally, we observed that, certain feature requests explicitly mention the word *refactor* to infer the need for improving a specified functionality. For example, the description of the feature request with identifier $CARBONDATA-1838$ reads as "*Refactor and optimize 'SortRowStepUtil' to make it efficient and more readable*". Furthermore, it is worth noting that, feature requests are often stated in solution-space terms contrary to the classical requirements[94,155]. Consequently, that facilitates the mapping between feature requests and refactorings. The classifiers can therefore learn the combination of such terms as features from the feature requests and use them as the basis for prediction.

4.4.5 Threats to Validity

First, the threat to external validity is concerned with the generalizability of the proposed approach. To address this threat, in this study we have considered several feature requests from different 55 Java open source projects and 14 common refactoring types. To further

Table 4.8 Top-10 Keywords with Highest Refactoring Rate

Keyword	Refactoring rate (%) in our Dataset
Refactor	83
Restructure	80
Rewrite	78
Rename	76
Introduce	75
Simplify	72
Extend	70
Optimize	67
Split	66
Cleanup	66

reduce this threat, we also conducted a cross-project validation where the projects involved in the training set were different from those in the testing set. This is the standard practice to evaluate how the proposed approach would perform on an independent dataset. Second, the internal threat may stem from the classification models that we leveraged in our approach. The classifiers have been implemented by the well-known Python-based library for machine learning called scikit-learn. To reduce the threat, the implementation of the models and the classification results were carefully examined, however there could be some errors slipped in unnoticed. Finally, a threat to construct validity is related to the implementation of the approach. The major threat relates to the correctness of the recovered refactorings and code smells that constituted our dataset. That is because the leveraged detection tools are not 100% on both recall and precision. The inaccuracy of the dataset could be accelerated by the fact that the detection tools may be unable to identify all of the past applied refactorings and the respective smells. Moreover, the tools may suggest irrelevant refactorings or smells (false positives) and may miss out some true refactorings or smells (false negatives). Consequently, that may threaten the accuracy of our dataset. However, in the future, the more improved tools after applying necessary fix as suggested in[156] can be leveraged to boost the accuracy of refactorings detection. In addition, another threat may stem from the possibility that a commit could have some additional refactorings that are not related to the implementation of feature requests. To reduce such threats we checked the dataset for possible errors, but still there could be some unnoticed errors. That would be due to the lack of the systems knowledge as the process did not involve original developers. Finding original developers is challenging considering the number of the subject applications we used and some of them have long evolution history.

4.5 Conclusion

Empirical investigations on developers' motivations behind applying refactorings on their software systems suggest that, refactoring is mostly motivated by the changes in the requirements. However, most of the existing refactorings recommenders solely focus on identifying refactorings opportunities for the sake of resolving code smells and ignore other refactoring motivations such as adding or extending features in software systems. To implement the requested features, developers sometimes apply refactorings to prepare for new adaptation that accommodates the new requirements. However, it is often challenging to determine which types of refactorings should be applied. Consequently, in this chapter we propose a learning-based approach that recommends refactoring types based on the past history of feature requests, code smells information, and the applied refactorings on the respective commits. The proposed approach learns from the training dataset associated with a set of applications and can be used to suggest refactoring types for feature requests associated with other applications or that associated with the training applications. To demonstrate the efficacy of our approach, we conducted experiments on the dataset of 55 open source Java projects consisting of $18,899$ feature requests retrieved from JIRA issue tracker and their associated refactorings recovered by using the state-of-the-art refactoring detection tools. Experimental results suggest that our approach significantly improves over the state-of-the-art approach. Evaluation analysis on two tasks (i.e., predicting the need for refactoring and recommending refactoring types) indicates that our approach attains an accuracy of up to 76.01% and 83.19% respectively. The possible future research direction includes the following. First, it would be interesting to investigate how to accurately locate where recommended refactoring types should be applied by exploiting the advancement of requirements traceability. Finally, our results encourage further investigation that would improve our approach, for example, leveraging more advanced techniques such as deep learning algorithms.

Chapter 5 Conclusion and Future Work

The complexity and quality degradation of software systems increase as the systems evolve to cope with changing requirements. Consequently, software refactoring is often applied to reduce the complexity and maintain quality. Despite the significant growth of software refactoring as a field of study, the existing challenges still motivate for more researches in investigating and proposing new refactoring techniques and tools to cater for realistic developers' demand. Recent studies suggest that, refactoring tools are underused by developers, they often refactor manually despite of the availability of tool support. It is suggested that, refactoring tools sometimes lack refactoing tactic mostly preferred by developers. Therefore, it is essential to understand how refactoring is performed in the wild. In view of that, some researches have invested effort in investigating the developers' motivations behind applying refactoring. Empirical evidences suggest that, refactoring is mainly motivated by changes in the requirements and much less by code smells resolution. However, most of the existing refactoring recommendation approaches mainly focus on resolving code smells and fulfilling other design objectives. Evolving accurate and applicable refactoring recommenders requires critical consideration of actual needs and practices of software developers. Therefore, the main objective of this thesis was to develop refactoring recommendation approaches that along resolving code smells also incorporate other considerations such as improving requirements traceability and facilitating developers in selecting what refactorings to apply when implementing feature requests. Maintaining traceability information is essential given the key roles it plays in assisting code inspection, generation of comprehensive test cases, and ensuring that requirements are fully implemented, e.t.c. On the other hand, during implementation of feature requests, developers often apply refactorings to prepare their systems adapt to new requirements, however, deciding what refactorings to apply is very challenging. In this context, this thesis proposed a set of approaches that aimed at recommending refactoring solutions that correspond to developers' refactoring preferences. The proposed approaches were evaluated to investigate their efficacy based on the available public datasets and well-known open source applications. In the following we summarize the key contributions of this thesis.

5.1 Thesis Summary

In this thesis, we make three key contributions that result from the proposed approaches. First, in Chapter 2, we exploited traceability information and code metrics to recommend refactoring solutions. The traceability between requirements (i.e., use cases) and the implementing source code is used to infer how code elements should be grouped. The basic idea is that, source code elements e.g., methods, which trace to similar use cases should also be strongly related. Since, the employed traceability information goes up to *method* granularity level, we therefore considered only refactorings that involve movement of methods, i.e., *Move Method* and *Extract Class*. This approach targeted at maintaining the traceability information which otherwise if not taken into account it can be affected by some refactorings. To quantify the quality of traceability we defined entropy-based metrics that are computed based on how methods trace to use cases and viceversa. Along with the traceability improvement objective, the approach also considers source code design improvement which is quantified by the widely-applicable cohesion and coupling metrics. Consequently, refactoring solutions are recommended based on the trade-off between improving traceability and code design. To ensure behavior preservation of the involved systems, we leveraged the existing state-of-the-art refactoring tools to detect code smells and suggest candidate refactoring solutions. The approach was evaluated both qualitatively and quantitatively.

Second, in Chapter 3, a learning-based approach is proposed to recommend refactoring types based on the past history of resolved feature requests and applied refactorings. We implemented our approach on four different machine learning classifiers. The classifiers were trained to perform two classification tasks: Binary and Multi-label classification. In binary classification, the approach predicts whether implementing a given feature request would demand refactoring or not. Consequently, for each feature request predicted to need refactoring, the multi-label classification is applied to predict the specific types of refactorings that would be required. Multi-label classification is selected because a given feature request can be associated with more than one type of refactorings. Ideally, the proposed approach offers automated solutions to whether and how to perform refactoring for a feature request to be implemented. The first refactoring-or-not classification supports the developer's decisions before implementing a feature request. In practice, such recommendations can help prioritize the features (requirements) to be delivered in a sprint, e.g., the codebase may be readier for developers to implement those features that do not require refactoring. For the features that refactoring is preferred, our approach automatically recommends which refactoring should be performed. Having this information can be valuable in practice because different refactorings may interfere with each other. Deciding the ordering of refactoring and hence the ordering of implementing features can avoid developers' re-work. The approach is evaluated on a wide-range of Java-based open source applications and the results suggest that the approach significantly improves the state-of-the-art.

Finally, in Chapter 4, the approach proposed earlier in Chapter 3 is significantly extended to incorporate code smells information. Code smells are leveraged because they are potential indicator of the need for refactoring, and they are useful in determining what refactorings to apply. Consequently, that remarkably boosts the prediction and recommendation performance. In this case, five widely-used classical machine learning classifiers and one neural network classifier were leveraged to implement the proposed approach. Besides that, the dataset were significantly expanded to include more feature requests examples and the associated code smells and refactorings. In addition, further analysis is conducted to investigate how special words or phrases in feature requests may infer the necessity of refactoring.

5.2 Future Research Directions

Unwillingness of refactoring tools adoption among developers calls for more researches that will help revolutionizing refactoring tools to boost their applicability. Besides that, future research would also need to establish solid empirical evidences on the applicability of tools' recommended refactorings in real-world settings with developers to boost their confidence on using tools. Moreover, we believe that, more advanced approaches (than that reported in this thesis) that are based on refactoring motivations due to changing requirements would be of much interest to the refactoring practitioners. For example, such approaches can exploit the advancements in requirements traceability to precisely locate where software should be refactored. Finally, another possibility is the need for the deeper analysis of feature requests to uncover what special characteristics or features they possess that can be potential in predicting and recommending certain types of refactorings.

Bibliography

[1] Bavota G, Oliveto R, Gethers M, et al. Methodbook: Recommending Move Method Refactorings via Relational Topic Models [J/OL]. IEEE Trans. Software Eng., 2014, 40 (7): 671–694. https://doi.org/10.1109/TSE.2013.60.

[2] Lehman M M, Ramil J F. Rules and Tools for Software Evolution Planning and Management [J/OL]. Ann. Software Eng., 2001, 11 (1): 15–44. https://doi.org/10.1023/A:1012535017876.

[3] Mens T, Tourwé T. A Survey of Software Refactoring [J/OL]. IEEE Trans. Software Eng., 2004, 30 (2): 126–139. https://doi.org/10.1109/TSE.2004.1265817.

[4] Opdyke W. Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks [J]. PhD thesis, Univ. of Illinois at Urbana-Champaign, year = 1992,.

[5] Fowler M. Refactoring - Improving the Design of Existing Code [M/OL]. Addison-Wesley, 1999. http://martinfowler.com/books/refactoring.html.

[6] Liu H, Guo X, Shao W. Monitor-Based Instant Software Refactoring [J/OL]. IEEE Trans. Software Eng., 2013, 39 (8): 1112–1126. https://doi.org/10.1109/TSE.2013.4.

[7] Chatzigeorgiou A, Manakos A. Investigating the evolution of code smells in object-oriented systems [J/OL]. ISSE, 2014, 10 (1): 3–18. https://doi.org/10.1007/s11334-013-0205-z.

[8] Tourwe T, Mens T. Identifying refactoring opportunities using logic meta programming [C]. In Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings., March 2003: 91–100.

[9] Fowler M. Refactoring - Improving the Design of Existing Code [M/OL]. Addison-Wesley, 1999. http://martinfowler.com/books/refactoring.html.

[10] Paiva T, Damasceno A, Figueiredo E, et al. On the evaluation of code smells and detection tools [J/OL]. J. Software Eng. R&D, 2017, 5: 7. https://doi.org/10.1186/s40411-017-0041-1.

[11] Cairo A S, de Figueiredo Carneiro G, Monteiro M P. The Impact of Code Smells on Software Bugs: A Systematic Literature Review [J/OL]. Information, 2018, 9 (11): 273. https://doi.org/10.3390/info9110273.

[12] Palomba F, Bavota G, Penta M D, et al. Detecting bad smells in source code using change history information [C/OL]. In 2013 28th IEEE/ACM International Conference on Automated Soft-

ware Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, 2013: 268–278. https://doi.org/10.1109/ASE.2013.6693086.

[13] Bertran I M, Garcia A, von Staa A. An exploratory study of code smells in evolving aspect-oriented systems [C/OL]. In Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011, Porto de Galinhas, Brazil, March 21-25, 2011, 2011: 203–214. https://doi.org/10.1145/1960275.1960300.

[14] Liu H, Wu Y, Liu W, et al. Domino Effect: Move More Methods Once a Method is Moved [C/OL]. In IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1, 2016: 1–12. https://doi.org/10.1109/SANER.2016.14.

[15] Ouni A, Kessentini M, Sahraoui H A, et al. Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study [J/OL]. ACM Trans. Softw. Eng. Methodol., 2016, 25 (3): 23:1–23:53. http://doi.acm.org/10.1145/2932631.

[16] Bavota G, Lucia A D, Marcus A, et al. A two-step technique for extract class refactoring [C/OL]. In ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010, 2010: 151–154. https://doi.org/10.1145/1858996.1859024.

[17] Moha N, Guéhéneuc Y, Duchien L, et al. DECOR: A Method for the Specification and Detection of Code and Design Smells [J/OL]. IEEE Trans. Software Eng., 2010, 36 (1): 20–36. https://doi.org/10.1109/TSE.2009.50.

[18] Marinescu R. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws [C/OL]. In 20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA, 2004: 350–359. https://doi.org/10.1109/ICSM.2004.1357820.

[19] Emden E V, Moonen L. Java Quality Assurance by Detecting Code Smells [C/OL]. In 9th Working Conference on Reverse Engineering (WCRE 2002), 28 October - 1 November 2002, Richmond, VA, USA, 2002: 97. https://doi.org/10.1109/WCRE.2002.1173068.

[20] Sharma T, Spinellis D. A survey on software smells [J/OL]. Journal of Systems and Software, 2018, 138: 158–173. https://doi.org/10.1016/j.jss.2017.12.034.

[21] Palomba F, Bavota G, Penta M D, et al. Mining Version Histories for Detecting Code Smells [J/OL]. IEEE Trans. Software Eng., 2015, 41 (5): 462–489. https://doi.org/10.1109/TSE.2014.2372760.

[22] Terra R, Valente M T, Miranda S, et al. JMove: A novel heuristic and tool to detect move method refactoring opportunities [J/OL]. Journal of Systems and Software, 2018, 138: 19–36. https:

//doi.org/10.1016/j.jss.2017.11.073.

[23] Fokaefs M, Tsantalis N, Stroulia E, et al. JDeodorant: identification and application of extract class refactorings [C/OL]. In Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, 2011: 1037–1039. http://doi.acm.org/10.1145/1985793.1985989.

[24] Fokaefs M, Tsantalis N, Chatzigeorgiou A. Jdeodorant: Identification and removal of feature envy bad smells [C]. In ICSM 2007. IEEE International Conference on Software Maintenance, 2007, 2007: 519–520.

[25] Bavota G, Lucia A D, Marcus A, et al. Supporting extract class refactoring in Eclipse: The ARIES project [C/OL]. In 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, 2012: 1419–1422. https://doi.org/10.1109/ICSE.2012.6227233.

[26] Murphy-Hill E R, Parnin C, Black A P. How We Refactor, and How We Know It [J/OL]. IEEE Trans. Software Eng., 2012, 38 (1): 5–18. https://doi.org/10.1109/TSE.2011.41.

[27] Fokaefs M, Tsantalis N, Chatzigeorgiou A, et al. Decomposing object-oriented class modules using an agglomerative clustering technique [C/OL]. In 25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada, 2009: 93–101. https://doi.org/10.1109/ICSM.2009.5306332.

[28] Bavota G, Lucia A D, Marcus A, et al. Automating extract class refactoring: an improved method and its evaluation [J/OL]. Empirical Software Engineering, 2014, 19 (6): 1617–1664. https://doi.org/10.1007/s10664-013-9256-x.

[29] Xu S, Sivaraman A, Khoo S, et al. GEMS: An Extract Method Refactoring Recommender [C/OL]. In 28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017, 2017: 24–34. https://doi.org/10.1109/ISSRE.2017.35.

[30] Silva D, Terra R, Valente M T. Recommending automated extract method refactorings [C/OL]. In 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014, 2014: 146–156. https://doi.org/10.1145/2597008.2597141.

[31] Silva D, Terra R, Valente M T. JExtract: An Eclipse Plug-in for Recommending Automated Extract Method Refactorings [J/OL]. CoRR, 2015, abs/1506.06086. http://arxiv.org/abs/1506.06086.

[32] Negara S, Chen N, Vakilian M, et al. A Comparative Study of Manual and Automated Refactorings [C] // Castagna G. In ECOOP 2013 – Object-Oriented Programming, Berlin, Heidelberg, 2013:

552–576.

[33] Liu H, Liu Q, Liu Y, et al. Identifying Renaming Opportunities by Expanding Conducted Rename Refactorings [J/OL]. IEEE Trans. Software Eng., 2015, 41 (9): 887–900. https://doi.org/10.1109/TSE.2015.2427831.

[34] Silva D, Tsantalis N, Valente M T. Why we refactor? confessions of GitHub contributors [C/OL]. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, 2016: 858–870. https://doi.org/10.1145/2950290.2950305.

[35] Sales V, Terra R, Miranda L F, et al. Recommending Move Method refactorings using dependency sets [C/OL]. In 20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013, 2013: 232–241. https://doi.org/10.1109/WCRE.2013.6671298.

[36] Tokuda L, Batory D S. Evolving Object-Oriented Designs with Refactorings [J/OL]. Autom. Softw. Eng., 2001, 8 (1): 89–120. https://doi.org/10.1023/A:1008715808855.

[37] Mohan M, Greer D. MultiRefactor: Automated Refactoring to Improve Software Quality [C/OL] // Felderer M, Fernández D M, Turhan B, et al. In Product-Focused Software Process Improvement - 18th International Conference, PROFES 2017, Innsbruck, Austria, November 29 - December 1, 2017, Proceedings, 2017: 556–572. https://doi.org/10.1007/978-3-319-69926-4_46.

[38] Moore I. Automatic Inheritance Hierarchy Restructuring and Method Refactoring [C/OL] // Anderson L, Coplien J. In Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, USA, October 6-10, 1996, 1996: 235–250. https://doi.org/10.1145/236337.236361.

[39] Fontana F A, Braione P, Zanoni M. Automatic detection of bad smells in code: An experimental assessment [J/OL]. Journal of Object Technology, 2012, 11 (2): 5: 1–38. https://doi.org/10.5381/jot.2012.11.2.a5.

[40] Marinescu C, Marinescu R, Mihancea P F, et al. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design [C]. In Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary, 2005: 77–80.

[41] Fontana F A, Mariani E, Morniroli A, et al. An Experience Report on Using Code Smells Detection Tools [C/OL]. In Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings, 2011: 450–457.

https://doi.org/10.1109/ICSTW.2011.12.

[42] Paiva T, Damasceno A, Padilha J, et al. Experimental Evaluation of Code Smell Detection Tools [C]. In 3rd workshop on software Visualization, Evolution, and Maintenance (VEM), 2015: 24–25.

[43] Fernandes E, Oliveira J, Vale G, et al. A review-based comparative study of bad smell detection tools [C/OL]. In Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, EASE 2016, Limerick, Ireland, June 01 - 03, 2016, 2016: 18:1–18:12. http://doi.acm.org/10.1145/2915970.2915984.

[44] Murphy-Hill E R, Black A P, Dig D, et al. Gathering refactoring data: a comparison of four methods [C/OL]. In Second ACM Workshop on Refactoring Tools, WRT 2008, in conjunction with OOPSLA 2008, Nashville, TN, USA, October 19, 2008, 2008: 7. https://doi.org/10.1145/1636642.1636649.

[45] Orrù M, Marchesi M. Assessment of Approaches for the Analysis of Refactoring Activity on Software Repositories An Empirical Study [C/OL]. In Proceedings of the Scientific Workshop Proceedings of XP2016, Edinburgh, Scotland, UK, May 24, 2016, 2016: 22. http://doi.acm.org/10.1145/2962695.2962717.

[46] Soares G, Gheyi R, Murphy-Hill E R, et al. Comparing approaches to analyze refactoring activity on software repositories [J/OL]. J. Syst. Softw., 2013, 86 (4): 1006–1022. https://doi.org/10.1016/j.jss.2012.10.040.

[47] Weißgerber P, Diehl S. Identifying Refactorings from Source-Code Changes [C/OL]. In 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan, 2006: 231–240. https://doi.org/10.1109/ASE.2006.41.

[48] Kim M, Gee M, Loh A, et al. Ref-Finder: a refactoring reconstruction tool based on logic query templates [C/OL]. In Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010, 2010: 371–372. http://doi.acm.org/10.1145/1882291.1882353.

[49] Tsantalis N, Mansouri M, Eshkevari L M, et al. Accurate and efficient refactoring detection in commit history [C/OL]. In Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, 2018: 483–494. http://doi.acm.org/10.1145/3180155.3180206.

[50] Silva D, Valente M T. RefDiff: detecting refactorings in version histories [C/OL]. In Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017, 2017: 269–279. https://doi.org/10.1109/MSR.2017.14.

[51] Dig D, Comertoglu C, Marinov D, et al. Automated Detection of Refactorings in Evolving Components [C/OL]. In ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings, 2006: 404–428. https://doi.org/10.1007/11785477_24.

[52] Xing Z, Stroulia E. Refactoring Detection based on UMLDiff Change-Facts Queries [C/OL]. In 13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy, 2006: 263–274. https://doi.org/10.1109/WCRE.2006.48.

[53] Kawrykow D, Robillard M P. Non-essential changes in version histories [C/OL] // Taylor R N, Gall H C, Medvidovic N. In Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, 2011: 351–360. https://doi.org/10.1145/1985793.1985842.

[54] Malpohl G, Hunt J J, Tichy W F. Renaming Detection [J/OL]. Autom. Softw. Eng., 2003, 10 (2): 183–202. https://doi.org/10.1023/A:1022968013020.

[55] Arnaoudova V, Eshkevari L M, Penta M D, et al. REPENT: Analyzing the Nature of Identifier Renamings [J/OL]. IEEE Trans. Software Eng., 2014, 40 (5): 502–532. https://doi.org/10.1109/TSE.2014.2312942.

[56] Chaparro O, Bavota G, Marcus A, et al. On the Impact of Refactoring Operations on Code Quality Metrics [C/OL]. In 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, 2014: 456–460. https://doi.org/10.1109/ICSME.2014.73.

[57] Tsantalis N, Chatzigeorgiou A. Identification of Move Method Refactoring Opportunities [J/OL]. IEEE Trans. Software Eng., 2009, 35 (3): 347–367. https://doi.org/10.1109/TSE.2009.1.

[58] Bavota G, Lucia A D, Marcus A, et al. Automating extract class refactoring: an improved method and its evaluation [J/OL]. Empirical Software Engineering, 2014, 19 (6): 1617–1664. https://doi.org/10.1007/s10664-013-9256-x.

[59] Simon F, Steinbrückner F, Lewerentz C. Metrics Based Refactoring [C/OL]. In Fifth Conference on Software Maintenance and Reengineering, CSMR 2001, Lisbon, Portugal, March 14-16, 2001, 2001: 30–38. https://doi.org/10.1109/.2001.914965.

[60] Kessentini M, Dea T J, Ouni A. A context-based refactoring recommendation approach using simulated annealing: two industrial case studies [C/OL]. In Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017, 2017: 1303–1310. http://doi.acm.org/10.1145/3071178.3071334.

[61] Tsantalis N, Chatzigeorgiou A. Ranking Refactoring Suggestions Based on Historical Volatility [C/OL]. In 15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany, 2011: 25–34. https://doi.org/10.1109/CSMR.2011.7.

[62] Ouni A, Kessentini M, Sahraoui H A, et al. The use of development history in software refactoring using a multi-objective evolutionary algorithm [C/OL]. In Genetic and Evolutionary Computation Conference, GECCO '13, Amsterdam, The Netherlands, July 6-10, 2013, 2013: 1461–1468. https://doi.org/10.1145/2463372.2463554.

[63] Chisăliţă-Creţu C. The Multi-Objective Refactoring Set Selection Problem-A Solution Representation Analysis [M] // Chisăliţă-Creţu C. Advances in Computer Science and Engineering. InTech, 2011, 2011:.

[64] Mkaouer M W, Kessentini M, Bechikh S, et al. Recommendation system for software refactoring using innovization and interactive dynamic optimization [C/OL]. In ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, 2014: 331–336. http://doi.acm.org/10.1145/2642937.2642965.

[65] Lin Y, Peng X, Cai Y, et al. Interactive and guided architectural refactoring with search-based recommendation [C/OL]. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, 2016: 535–546. http://doi.acm.org/10.1145/2950290.2950317.

[66] Ouni A, Kessentini M, Sahraoui H A. Search-Based Refactoring Using Recorded Code Changes [C/OL]. In 17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013, 2013: 221–230. https://doi.org/10.1109/CSMR.2013.31.

[67] Murphy-Hill E R, Black A P. Refactoring Tools: Fitness for Purpose [J/OL]. IEEE Software, 2008, 25 (5): 38–44. https://doi.org/10.1109/MS.2008.123.

[68] Kim M, Zimmermann T, Nagappan N. An Empirical Study of Refactoring Challenges and Benefits at Microsoft [J/OL]. IEEE Trans. Software Eng., 2014, 40 (7): 633–649. https://doi.org/10.1109/TSE.2014.2318734.

[69] Vakilian M, Chen N, Negara S, et al. Use, disuse, and misuse of automated refactorings [C/OL]. In 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, 2012: 233–243. https://doi.org/10.1109/ICSE.2012.6227190.

[70] Murphy-Hill E R, Parnin C, Black A P. How we refactor, and how we know it [C/OL]. In 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada,

Proceedings, 2009: 287–297. https://doi.org/10.1109/ICSE.2009.5070529.

[71] Soares G, Catao B, Varjao C, et al. Analyzing Refactorings on Software Repositories [C/OL]. In 25th Brazilian Symposium on Software Engineering, SBES 2011, Sao Paulo, Brazil, September 28-30, 2011, 2011: 164–173. https://doi.org/10.1109/SBES.2011.21.

[72] Palomba F, Zaidman A, Oliveto R, et al. An exploratory study on the relationship between changes and refactoring [C/OL]. In Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017, 2017: 176–185. https://doi.org/10.1109/ICPC.2017.38.

[73] Kim M, Zimmermann T, Nagappan N. A field study of refactoring challenges and benefits [C/OL]. In 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012, 2012: 50. https://doi.org/10.1145/2393596.2393655.

[74] Abebe M, Yoo C-J. Trends, opportunities and challenges of software refactoring: A systematic literature review [J]. International Journal of Software Engineering and Its Applications, 2014, 8 (6): 299–318.

[75] Murphy-Hill E R, Black A P. An interactive ambient visualization for code smells [C/OL]. In Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, UT, USA, October 25-26, 2010, 2010: 5–14. http://doi.acm.org/10.1145/1879211.1879216.

[76] Fokaefs M, Tsantalis N, Stroulia E, et al. Identification and application of Extract Class refactorings in object-oriented systems [J/OL]. Journal of Systems and Software, 2012, 85 (10): 2241–2260. https://doi.org/10.1016/j.jss.2012.04.013.

[77] Chidamber S R, Kemerer C F. A Metrics Suite for Object Oriented Design [J/OL]. IEEE Trans. Software Eng., 1994, 20 (6): 476–493. https://doi.org/10.1109/32.295895.

[78] Gotel O C Z, Finkelstein A. An analysis of the requirements traceability problem [C/OL]. In Proceedings of the First IEEE International Conference on Requirements Engineering, ICRE '94, Colorado Springs, Colorado, USA, April 18-21, 1994, 1994: 94–101. https://doi.org/10.1109/ICRE.1994.292398.

[79] Rempel P, Mäder P. Continuous assessment of software traceability [C/OL]. In Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume, 2016: 747–748. http://doi.acm.org/10.1145/2889160.2892657.

[80] Rempel P, Mäder P. A quality model for the systematic assessment of requirements traceability [C/OL]. In 23rd IEEE International Requirements Engineering Conference, RE 2015, Ottawa,

ON, Canada, August 24-28, 2015, 2015: 176–185. https://doi.org/10.1109/RE.2015.7320420.

[81] Tsuchiya R, Kato T, Washizaki H, et al. Recovering traceability links between requirements and source code in the same series of software products [C/OL]. In 17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013, 2013: 121–130. http://doi.acm.org/10.1145/2491627.2491633.

[82] Ali N, Guéhéneuc Y, Antoniol G. Requirements Traceability for Object Oriented Systems by Partitioning Source Code [C/OL]. In 18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011, 2011: 45–54. https://doi.org/10.1109/WCRE.2011.16.

[83] Antoniol G, Canfora G, Casazza G, et al. Recovering Traceability Links between Code and Documentation [J/OL]. IEEE Trans. Software Eng., 2002, 28 (10): 970–983. https://doi.org/10.1109/TSE.2002.1041053.

[84] Poshyvanyk D, Marcus A. Using traceability links to assess and maintain the quality of software documentation [J]. Proc. of Traceability in Emerging Forms of Software Engineering, 2007, 7: 27–30.

[85] Canfora G, Cerulo L, Cimitile M, et al. How changes affect software entropy: an empirical study [J/OL]. Empirical Software Engineering, 2014, 19 (1): 1–38. https://doi.org/10.1007/s10664-012-9214-z.

[86] Hassan A E. Predicting faults using the complexity of code changes [C/OL]. In 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, 2009: 78–88. https://doi.org/10.1109/ICSE.2009.5070510.

[87] Etzkorn L H, Gholston S, Jr W E H. A semantic entropy metric [J/OL]. Journal of Software Maintenance, 2002, 14 (4): 293–310. https://doi.org/10.1002/smr.255.

[88] Yang L, Dang Z, Fischer T R, et al. Entropy and software systems: towards an information-theoretic foundation of software testing [C/OL]. In Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010, 2010: 427–432. http://doi.acm.org/10.1145/1882362.1882449.

[89] Cleland-Huang J, Gotel O, Hayes J H, et al. Software traceability: trends and future directions [C/OL]. In Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014, 2014: 55–69. http://doi.acm.org/10.1145/2593882.2593891.

[90] Mäder P, Egyed A. Assessing the effect of requirements traceability for software maintenance [C/OL]. In 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012, 2012: 171–180. https://doi.org/10.1109/ICSM.2012.6405269.

[91] Delater A, Paech B. Tracing Requirements and Source Code during Software Development: An Empirical Study [C/OL]. In 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013, 2013: 25–34. https://doi.org/10.1109/ESEM.2013.16.

[92] Eyl M, Reichmann C, Müller-Glaser K. Traceability in a Fine Grained Software Configuration Management System [C]. In International Conference on Software Quality, 2017: 15–29.

[93] Mahmoud A, Niu N. Supporting requirements traceability through refactoring [C/OL]. In 21st IEEE International Requirements Engineering Conference, RE 2013, Rio de Janeiro-RJ, Brazil, July 15-19, 2013, 2013: 32–41. https://doi.org/10.1109/RE.2013.6636703.

[94] Niu N, Bhowmik T, Liu H, et al. Traceability-enabled refactoring for managing just-in-time requirements [C/OL]. In IEEE 22nd International Requirements Engineering Conference, RE 2014, Karlskrona, Sweden, August 25-29, 2014, 2014: 133–142. https://doi.org/10.1109/RE.2014.6912255.

[95] Shin Y, Hayes J H, Cleland-Huang J. A framework for evaluating traceability benchmark metrics [J]. Technical Reports. Paper 21, 2012.

[96] Ali N, Sharafi Z, Guéhéneuc Y, et al. An empirical study on the importance of source code entities for requirements traceability [J/OL]. Empirical Software Engineering, 2015, 20 (2): 442–478. https://doi.org/10.1007/s10664-014-9315-y.

[97] Faiz F, Easmin R, Gias A U. Achieving better requirements to code traceability: Which refactoring should be done first? [C]. In Quality of Information and Communications Technology (QUATIC), 2016 10th International Conference on the, 2016: 9–14.

[98] Cheikhi L, Al-Qutaish R E, Idri A, et al. Chidamber and kemerer object-oriented measures: Analysis of their design from the metrology perspective [J]. International Journal of Software Engineering & Its Applications, 2014, 8 (2).

[99] Bianchi A, Caivano D, Lanubile F, et al. Evaluating Software Degradation through Entropy [C/OL]. In 7th IEEE International Software Metrics Symposium (METRICS 2001), 4-6 April 2001, London, England, 2001: 210. https://doi.org/10.1109/METRIC.2001.915530.

[100] Olague H M, Etzkorn L H, Cox G W. An Entropy-Based Approach to Assessing Object-Oriented Software Maintainability and Degradation - A Method and Case Study [C]. In Proceedings of the

International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 1, 2006: 442–452.

[101] Maisikeli S G. Evaluation and Study of Software Degradation in the Evolution of Six Versions of Stable and Matured Open Source Software Framework [C]. In Sixth International Conference on Computer Science, Engineering & Applications (ICCSEA 2016), Dubai, UAE, September, 2016: 24–25.

[102] PMD. https://pmd.github.io/ [J], Accessed October 2019.

[103] Checkstyle. https://checkstyle.sourceforge.io/ [J], Accessed October 2019.

[104] Andrew M, Ben S, Laurie W. iTrust Electronic Health Care System: A Case Study [J/OL], 2012, ch. Software and Systems Traceability. https://doi.org/10.1007/978-1-4471-2239-5.

[105] Trifu A, Marinescu R. Diagnosing Design Problems in Object Oriented Systems [C/OL]. In 12th Working Conference on Reverse Engineering, WCRE 2005, Pittsburgh, PA, USA, November 7-11, 2005, 2005: 155–164. https://doi.org/10.1109/WCRE.2005.15.

[106] Shannon C E. A mathematical theory of communication [J/OL]. Mobile Computing and Communications Review, 2001, 5 (1): 3–55. http://doi.acm.org/10.1145/584091.584093.

[107] Seng O, Stammel J, Burkhart D. Search-based determination of refactorings for improving the class structure of object-oriented systems [C/OL]. In Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006, 2006: 1909–1916. http://doi.acm.org/10.1145/1143997.1144315.

[108] Bonja C, Kidanmariam E. Metrics for class cohesion and similarity between methods [C/OL]. In Proceedings of the 44st Annual Southeast Regional Conference, 2006, Melbourne, Florida, USA, March 10-12, 2006, 2006: 91–95. http://doi.acm.org/10.1145/1185448.1185469.

[109] JIRA https://wwwatlassiancom/software/jira, Retrived on 28th November 2018.

[110] Bugzilla https://wwwbugzillaorg/, Retrived on 28th November 2018.

[111] GitHub https://githubcom/features, Retrived on 28th November 2018.

[112] Heck P, Zaidman A. An analysis of requirements evolution in open source projects: recommendations for issue trackers [C/OL]. In 13th International Workshop on Principles of Software Evolution, IWPSE 2013, Proceedings, August 19-20, 2013, Saint Petersburg, Russia, 2013: 43–52. http://doi.acm.org/10.1145/2501543.2501550.

[113] Thung F, Wang S, Lo D, et al. Automatic recommendation of API methods from feature requests [C/OL]. In 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, 2013: 290–300. https:

//doi.org/10.1109/ASE.2013.6693088.

[114] Palomba F, Salza P, Ciurumelea A, et al. Recommending and localizing change requests for mobile apps based on user reviews [C/OL] // Uchitel S, Orso A, Robillard M P. In Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, 2017: 106–117. https://doi.org/10.1109/ICSE.2017.18.

[115] Angerer F, Grimmer A, Prähofer H, et al. Change impact analysis for maintenance and evolution of variable software systems [J/OL]. Autom. Softw. Eng., 2019, 26 (2): 417–461. https://doi.org/10.1007/s10515-019-00253-7.

[116] Liu H, Ma Z, Shao W, et al. Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort [J/OL]. IEEE Trans. Software Eng., 2012, 38 (1): 220–235. https://doi.org/10.1109/TSE.2011.9.

[117] Mohan M, Greer D. A survey of search-based refactoring for software maintenance [J/OL]. J. Software Eng. R&D, 2018, 6: 3. https://doi.org/10.1186/s40411-018-0046-4.

[118] Ouni A, Kessentini M, Sahraoui H A, et al. Improving multi-objective code-smells correction using development history [J/OL]. Journal of Systems and Software, 2015, 105: 18–39. https://doi.org/10.1016/j.jss.2015.03.040.

[119] Mei H, Zhang L. Can big data bring a breakthrough for software automation? [J/OL]. SCIENCE CHINA Information Sciences, 2018, 61 (5): 056101:1–056101:3. https://doi.org/10.1007/s11432-017-9355-3.

[120] GIT bash commands [C]. In https://www.atlassian.com/git, Retrived on 28th November 2018.

[121] Rath M, Rendall J, Guo J L C, et al. Traceability in the wild: automatically augmenting incomplete trace links [C/OL] // Chaudron M, Crnkovic I, Chechik M, et al. In Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, 2018: 834–845. https://doi.org/10.1145/3180155.3180207.

[122] Uysal A K, Günal S. The impact of preprocessing on text classification [J/OL]. Information Processing and Management, 2014, 50 (1): 104–112. https://doi.org/10.1016/j.ipm.2013.08.006.

[123] Loper E, Bird S. NLTK: The Natural Language Toolkit [C]. In ACL Workshop Effective Tools Methodol. Teach. Natural Lang. Process. Comput. Linguistics (ETMTNLP), 2002: 63–70.

[124] Porter M F. An algorithm for suffix stripping [J/OL]. Program, 2006, 40 (3): 211–218. https://doi.org/10.1108/00330330610681286.

[125] Mahmoud A, Niu N. Supporting requirements to code traceability through refactoring [J/OL]. Requirements Engineering, 2014, 19 (3): 309–329. https://doi.org/10.1007/

s00766-013-0197-0.

[126] Runeson P, Alexandersson M, Nyholm O. Detection of Duplicate Defect Reports Using Natural Language Processing [C/OL]. In 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, 2007: 499–510. https://doi.org/10.1109/ICSE.2007.32.

[127] Sun C, Lo D, Wang X, et al. A discriminative model approach for accurate duplicate bug report retrieval [C/OL] // Kramer J, Bishop J, Devanbu P T, et al. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, 2010: 45–54. https://doi.org/10.1145/1806799.1806811.

[128] Manning C D, Schütze H. Foundations of statistical natural language processing [M]. MIT Press, 2001.

[129] Thung F, Kochhar P S, Lo D. DupFinder: integrated tool support for duplicate bug report detection [C/OL] // Crnkovic I, Chechik M, Grünbacher P. In ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, 2014: 871–874. https://doi.org/10.1145/2642937.2648627.

[130] Nizamani Z A, Liu H, Chen D M, et al. Automatic approval prediction for software enhancement requests [J/OL]. Automated Software Engineering, 2018, 25 (2): 347–381. https://doi.org/10.1007/s10515-017-0229-y.

[131] Manning C D, Raghavan P, Schütze H. Introduction to information retrieval [M]. Cambridge University Press, 2008.

[132] Scikit-learn https://scikit-learnorg/stable/, Retrived on 28th November 2018.

[133] Aggarwal C C, Zhai C. A Survey of Text Classification Algorithms [M] // Aggarwal C C, Zhai C. Mining Text Data. Springer, 2012: 2012: 163–222.

[134] Hegedüs P, Kádár I, Ferenc R, et al. Empirical evaluation of software maintainability based on a manually validated refactoring dataset [J/OL]. Information & Software Technology, 2018, 95: 313–327. https://doi.org/10.1016/j.infsof.2017.11.012.

[135] Zhang M, Zhou Z. A Review on Multi-Label Learning Algorithms [J/OL]. IEEE Transactions on Knowledge and Data Engineering, 2014, 26 (8): 1819–1837. https://doi.org/10.1109/TKDE.2013.39.

[136] Godbole S, Sarawagi S. Discriminative Methods for Multi-labeled Classification [C/OL] // Dai H, Srikant R, Zhang C. In Advances in Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference, PAKDD 2004, Sydney, Australia, May 26-28, 2004, Proceedings, 2004: 22–30. https://doi.org/10.1007/978-3-540-24775-3_5.

[137] Schapire R E, Singer Y. BoosTexter: A Boosting-based System for Text Categorization [J/OL]. Machine Learning, 2000, 39 (2/3): 135–168. https://doi.org/10.1023/A:1007649029923.

[138] Feng W, Sun J, Zhang L, et al. A support vector machine based naive Bayes algorithm for spam filtering [C/OL]. In 35th IEEE International Performance Computing and Communications Conference, IPCCC 2016, Las Vegas, NV, USA, December 9-11, 2016, 2016: 1–8. https://doi.org/10.1109/PCCC.2016.7820655.

[139] Gibaja E, Ventura S. A Tutorial on Multilabel Learning [J/OL]. ACM Computing Surveys, 2015, 47 (3): 52:1–52:38. https://doi.org/10.1145/2716262.

[140] Jayatilleke S, Lai R, Reed K. A method of requirements change analysis [J/OL]. Requir. Eng., 2018, 23 (4): 493–508. https://doi.org/10.1007/s00766-017-0277-7.

[141] Ratzinger J, Sigmund T, Vorburger P, et al. Mining Software Evolution to Predict Refactoring [C/OL]. In Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, ESEM 2007, September 20-21, 2007, Madrid, Spain, 2007: 354–363. https://doi.org/10.1109/ESEM.2007.9.

[142] Ouni A, Kessentini M, Cinnéide M Ó, et al. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells [J/OL]. Journal of Software: Evolution and Process, 2017, 29 (5). https://doi.org/10.1002/smr.1843.

[143] Liu H, Xu Z, Zou Y. Deep learning based feature envy detection [C/OL]. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018, 2018: 385–396. http://doi.acm.org/10.1145/3238147.3238166.

[144] Pantiuchina J, Bavota G, Tufano M, et al. Towards just-in-time refactoring recommenders [C/OL]. In Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018, 2018: 312–315. https://doi.org/10.1145/3196321.3196365.

[145] Yue R, Gao Z, Meng N, et al. Automatic Clone Recommendation for Refactoring Based on the Present and the Past [C/OL]. In 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018, 2018: 115–126. https://doi.org/10.1109/ICSME.2018.00021.

[146] Codacy. https://github.com/marketplace/codacy [J], Accessed October 2019.

[147] PMDCodacy. https://github.com/codacy/codacy-pmdjava [J], Accessed October 2019.

[148] Chen J, Huang H, Tian S, et al. Feature selection for text classification with Naïve Bayes [J/OL]. Expert Syst. Appl., 2009, 36 (3): 5432–5435. https://doi.org/10.1016/j.eswa.2008.

06.054.

[149] Khan A, Baharudin B, Lee L H, et al. A Review of Machine Learning Algorithms for Text-Documents Classification [C]. 2010.

[150] Jiang L, Cai Z, Zhang H, et al. Naive Bayes text classifiers: a locally weighted learning approach [J/OL]. J. Exp. Theor. Artif. Intell., 2013, 25 (2): 273–286. https://doi.org/10.1080/0952813X.2012.721010.

[151] Therrien R, Doyle S. Role of training data variability on classifier performance and generalizability [C/OL] // Tomaszewski J E, Gurcan M N. In Medical Imaging 2018: Digital Pathology, Houston, Texas, United States, 10-15 February 2018, 2018: 1058109. https://doi.org/10.1117/12.2293919.

[152] Nyamawe A S, Liu H, Niu N, et al. Automated Recommendation of Software Refactorings Based on Feature Requests [C/OL] // Damian D E, Perini A, Lee S. In 27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019, 2019: 187–198. https://doi.org/10.1109/RE.2019.00029.

[153] Vidal S A, Marcos C A, Pace J A D. An approach to prioritize code smells for refactoring [J/OL]. Autom. Softw. Eng., 2014, 23 (3): 501–532. https://doi.org/10.1007/s10515-014-0175-x.

[154] AlOmar E A, Mkaouer M W, Ouni A. Can refactoring be self-affirmed?: an exploratory study on how developers document their refactoring activities in commit messages [C/OL] // Tsantalis N, Cai Y, Demeyer S. In Proceedings of the 3rd International Workshop on Refactoring, IWOR@ICSE 2019, Montreal, QC, Canada, May 28, 2019, 2019: 51–58. https://doi.org/10.1109/IWoR.2019.00017.

[155] Alspaugh T A, Scacchi W. Ongoing software development without classical requirements [C/OL]. In 21st IEEE International Requirements Engineering Conference, RE 2013, Rio de Janeiro-RJ, Brazil, July 15-19, 2013, 2013: 165–174. https://doi.org/10.1109/RE.2013.6636716.

[156] Tan L, Bockisch C. A Survey of Refactoring Detection Tools [C/OL] // Krusche S, Schneider K, Kuhrmann M, et al. In Proceedings of the Workshops of the Software Engineering Conference 2019, Stuttgart, Germany, February 19, 2019, 2019: 100–105. http://ceur-ws.org/Vol-2308/emls2019paper02.pdf.

Publications During PhD Study

[1] **Ally S. Nyamawe**, Hui Liu, Nan Niu, Qasim Umer, Zhendong Niu. (2019) Automated Recommendation of Software Refactorings based on Feature Requests [C]. 2019 IEEE 27th Requirements Engineering Conference, (RE): 187-198, （EI）. *CCF B*.

[2] **Ally S. Nyamawe**, Hui Liu, Zhendong Niu, Wentao Wang, Nan Niu. (2018) Recommending Refactoring Solutions Based on Traceability and Code Metrics [J]. IEEE Access, 6, 49460-49475.（SCIE）. *IF = 4.098*.

[3] **Ally S. Nyamawe**, Hui Liu, Nan Niu, Qasim Umer, Zhendong Niu. (2020) Feature Requests-based Recommendation of Software Refactorings [J]. 2020 Empirical Software Engineering, (EMSE) (Minor Revision)（SCI）. *IF = 4.457, CCF B*.

[4] Guangjie Li, Hui Liu, **Ally S. Nyamawe**.(2020) A Survey on Renamings of Software Entities [J]. 2020 ACM Comput. Surv. 53, 2, Article 41 (April 2020), 38 pages（SCI）. *IF = 6.131*.

Acknowledgments

Throughout my PhD studies and the writing of this thesis, I have received a great deal of assistance and support from various institutions and individuals. I would first like to thank Allah (the most Merciful) for giving me good health, strength, knowledge and an opportunity to pursue this PhD study to its full completion. Without His blessings, this achievement would otherwise be impossible. I have a great pleasure to acknowledge my sincere gratitude to my PhD supervisor Prof. Zhendong Niu for his heartfelt support, guidance, and motivation in my quest for knowledge. Thank you for the wonderful opportunity to work under your guidance. You supported me greatly, and your passionate contributions have made this thesis possible. Besides my supervisor, I take pride to acknowledge the insightful guidance of Prof. Hui Liu for always leaving his door open whenever I was in need of help pertaining to my research or writing. Thank you so much for your invaluable expertise in formulating this research and steering it to its successful completion. I would also like to thank my research external guide Prof. Nan Niu from the University of Cincinnati, for his tireless support, motivation, and insightful suggestions in all the time of my PhD research. Many thanks!

I would like to say thank you to all the members of my defense committee. Your invaluable constructive comments and criticism have made this thesis worthwhile. Each of you has imparted to me an extensive guidance on how to act professionally in pursuit of scientific research and life in general. Special thanks to the Chairman of the committee, Prof. Zhang Lu. A very special gratitude goes out to all the coursework teachers and professors from the School of Computer Science and Technology for laying a solid foundation of my research and providing me with a comprehensive understanding of the field. Further, I acknowledge the management of the School of Computer Science and Technology for the provision of all the necessary facilities and tools to facilitate the undertaking and successful completion of my research. A lot of thanks to my fellow PhD labmates for their constructive criticisms, stimulating discussions, friendship and for all the fun we have had in the past five years. I am sincerely grateful to them for sharing their honest and illuminating opinions on a number of issues that targeted in making my research even better. I am also using this opportunity to further express my sincere appreciations to all friends and everyone who supported me throughout my PhD academic journey, sorry for not mentioning you individually. However, it would be inappropriate if I omit to mention Eng. Emmanuel Mbosso, who has been a good friend and motivative at all times. This work would not have been possible without the financial support from China Scholarship Council who funded my PhD studies. A lot of thanks for your generous support. I am grateful to the staff of the International Students Center of BIT for making my stay in China such conducive and pleasant. I am also especially indebted to my employer, The University of Dodoma, for granting me a study leave to pursue my PhD studies and for all the supports you extended to me throughout all these years of my study.

No body has been more important to me in the pursuit of my academic career than my parents. I would like to express my very profound gratitude to my father Mr. Selemani Nyamawe and my lovely mother Asha Mustapha, for their love, unfailing support, and moral guidance. You will always be my ultimate role models. My acknowledgement would be incomplete without thanking my lovely wives: Rukaya and Fatma, for their unprecedented love, caring and emotional support. Specifically, I would like to appreciate your understanding and patience for all the years of my PhD studies far away from home. You have been extremely supportive of me and you have made countless sacrifices just to help me successfully complete my studies. Thank you so much and may Allah bless you always. I also extend my deepest appreciation to my sons: Suleyman, Abdulhakeem, and Abdulhaseeb, in deed they are the vital source of motivation and inspiration to finish my PhD degree with success. Without such a team behind me, I doubt that I would be who I am today.

Finally, I would like to extend my sincere gratitude to my siblings: Khadija, Mustapha and Ramadhan for their endless support and encouragement throughout my academic journey and life in general. Last but by no means least, I would like to thank my brother in law, Shabani Maliwa for his kindness and specifically for encouraging me in the first place to enroll for PhD studies, I once again thank you so much.

Author Biography

Ally S. Nyamawe received his BSc degree in Computer Science from the University of Dar es Salaam, Tanzania in 2008. Beginning his career as a software developer, Mr. Nyamawe's portfolio included ERP Programmer and Technical Consultant at the Technobrain (T) Limited in Dar es Salaam, Tanzania, and the chief software developer and co-founder of INO Informatics and General Engineering. In August 2008 he joined the St. John's University of Tanzania as a tutorial assistant and programmer where he was tasked for assisting tutorials and developing computer applications for the University. In August 2009, Mr. Nyamawe joined the University of Dodoma (UDOM) as tutorial assistant at the then Department of Computer Science, and subsequently he enrolled for Masters in Computer Science degree at UDOM of which he graduated in 2011. He was then promoted to a rank of assistant lecturer in 2011 and eventually a Lecturer in 2015. He was appointed acting head of department of computer science from January 2013 to July 2013, and subsequently a head of department of computer science from August 2013 to August 2015. Prior to these appointments, he was the coordinator of the Center for Innovation, Research and Development in ICT of the College of Informatics and Virtual Education at the University of Dodoma from 2012 to April 2013. He served as part time lecturer at the Local Government Training Institute, Dodoma, Tanzania from 2012 to 2013. He is currently a PhD student at the School of Computer Science and Technology, Beijing Institute of Technology, China. During his PhD studies he published two journal articles and one conference paper as first author, and he co-authored more than eight scientific papers. His research interests include software refactoring, requirements traceability, requirements engineering, and computer programming.