IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 1, 2018, accepted August 24, 2018, date of publication September 6, 2018, date of current version September 28, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2868990

Recommending Refactoring Solutions Based on Traceability and Code Metrics

ALLY S. NYAMAWE[1], HUI LIU[1], ZHENDONG NIU[1], WENTAO WANG[2], (Student Member, IEEE), AND NAN NIU[2], (Senior Member, IEEE)

[1]School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
[2]Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH 45221, USA

Corresponding authors: Hui Liu (liuhui08@bit.edu.cn) and Zhendong Niu (zniu@bit.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61472034, Grant 61772071, and Grant 61690205, in part by the National Key Research and Development Program of China under Grant 2016YFB1000801, and in part by the U.S. National Science Foundation under Grant 1350487.

ABSTRACT Software refactoring has been extensively used to rectify the design flaws and improve software quality without affecting its observable behaviors. For a given code smell, it is common that there exist multiple 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 metrics of source code. However, their impact on the traceability between source code and requirements is ignored 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 data set. The evaluation results suggest that on up to 71% of the cases, developers prefer our recommendation to the traditional recommendation based on code metrics.

INDEX TERMS Entropy, refactoring, requirements traceability, solution recommendation.

I. INTRODUCTION

Software systems often evolve to cope 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 [1], [2]. 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 [3].

Software refactoring has been extensively used to remedy software design flaws and to improve software quality, especially maintainability, reusability and extensibility [4]. The underlying principle of refactoring is reorganizing software elements such as classes, methods and variables to facilitate future extensions [1], [5]. 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 [3]; Second, according to the types of identified code smells, specific solutions (refactorings) are applied to remedy the smells [3].

Currently, several powerful refactoring tools have been developed to automate software refactoring process, e.g., JDeodorant [6], [7], iPlasma [8], Stench Blossom [9] and DECOR [10]. Moreover, most of the mainstream software development environments (SDEs) are equipped with refactoring capabilities, e.g., Eclipse and Microsoft Visual Studio. Such great tools make software refactoring simpler and less error-prone [11]. For a given code smell identified automatically, refactoring tools may suggest several refactoring solutions. However, the decision on which solution to be executed is left to developers. The decision to choose the best solution

is often challenging [12]. 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 [12]–[14]. Along with code metrics, other factors have been considered as well [15]. 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 [7], [12]. Basically, coupling refers to how parts of a design inter-depend each other, whereas cohesion refers to internal dependencies within parts of a design [16]. Usually coupling and cohesion are measured based on the fields or instance variables used by the methods within or between classes [12].

Such approaches ignore the fact that different refactoring solutions may have significantly different impact on the traceability between requirements and implementation (source code) [17]. 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*" [18]. 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 [19]–[22]. 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 [23]. 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 [23]. It is therefore of vital importance to ensure that traceability information is well maintained to achieve the aforementioned benefits. Moreover, Poshyvanyk and Marcus [24] 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 paper 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 [25], [26]. Entropy is extensively used in software engineering for several purposes such as measuring complexity [27], disorganization [25] of classes and software testing [28]. We therefore compute traceability entropy as the degree of disorganization of software components based on traceability information. Furthermore, to quantify the quality of source code design we use EP (Entity Placement) metric proposed by Tsantalis and Chatzigeorgiou [14]. 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 paper 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.

The rest of the paper is organized as follows. In Section II we review related work with our research. We describe the problem statement and provide a motivating example in Section III and IV respectively. Section V presents our recommendation approach. We validate the results and discuss threats to validity of our results in Section VI. We finally conclude our paper and state the future work in Section VII.

II. RELATED WORK

A. CODE SMELL DETECTION

Refactoring as the process of improving the internal structure of a software system without altering its observable behavior [3], has long been practiced by developers. Currently, refactoring is supported by most of the contemporary mainstream software development environments (SDEs) [29] and their several existing plugins, thereby providing the capability of (semi-) automatic application of refactoring. Refactoring is essential for removing design flaws in source code commonly known as code smells. Code smells are the potential indicator that the source code is claiming for improvement. A catalogue of code smells and their corresponding remedying solutions are well described in [3]. Software refactoring is part of software maintenance and evolution processes where software needs to be maintained to cope with users' changing requirements.

To improve the quality of the source code it is important to detect code smells and consequently refactor with the appropriate solution [30]. To detect code smells, refactoring tools or smell detectors are often used. So far, several powerful smell detectors (such as JDeodorant, iPlasma and inFusion) have been proposed and validated [30]–[32]. 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 [33]. Once the source code has been found to contain code smells, then applying proper refactoring operation is not always trivial. To address that, several refactoring recommendation techniques have been proposed. In software engineering, recommendation systems facilitate the provisioning of valuable information items for a particular task in certain context [34]. Particularly, refactoring recommendation targets at reducing developers' information overload by providing them with the most relevant solutions only.

B. REQUIREMENTS TRACEABILITY

Traceability, specifically requirements traceability has received a lot of attention in the literature [35]–[38]. Traceability has proven to be useful in various areas of software engineering, e.g. software refactoring [39] and software maintenance [36]. Traceability ensures that the developed product fulfills and covers the targeted requirements in both design and source code [40]. To fully utilize the promising benefits of traceability, generating [38], [41] and maintaining [42] traceability information is inevitable. For example, Eyl *et al*. [43] 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 [43]. Traceability has been proven to be useful in software engineering by numerous researchers. For example, Mäder and Egyed [36], empirically investigated the usefulness of traceability in software maintenance task of which it was found to improve maintenance quality significantly.

C. TRACEABILITY-BASED REFACTORING

Traceability for refactoring and vice versa has recently received attention in the literature to some extent. Individually traceability and refactoring have been witnessed in the literature for years now. Mahmoud and Niu [35] presented a refactoring based approach for maintaining traceability information. In their work, they argued that as the software system evolves its lexical structure corrupts which then distorts the traceability tracks of which they can be systematically reestablished through refactoring. Faiz *et al*. [44] also advocate the work done in [35] by assessing more other types of refactoring in improving traceability. Mahmoud and Niu [35] and Faiz *et al.* [44] 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.* [39] proposed a requirements driven 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 [39] with ours is on how to recommend refactoring solutions. We recommend refactoring solutions based on the requirements to source code traceability information, whereas Niu *et al.* [39] 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 [39] receives new requirements to be implemented as input and our approach only works on already implemented requirements. In addition to that, in [39] 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.

D. REFACTORING RECOMMMENDATION

In assisting developers executing optimal solution promptly, researchers have proposed numerous ways. For example, Liu *et al*. [4] developed a monitor-based tool to assist instant refactoring. The tool constantly runs at background to instantly analyze changes in source code committed by a software engineer. When changes with potential signs of introducing code smells are detected, the tool promptly invokes smell detection tools and alerts developers to take action accordingly. In line with that, Mkaouer *et al.* [45] proposed a recommendation tool which dynamically adapts and interactively suggests suitable refactorings to developers. The tool's recommendations were based on developers' feedback and introduced changes in source code. Furthermore, Ouni *et al*. [46] proposed a tool to recommend refactoring solutions to developers based on multi-objective search algorithm. The tool recommends the sequence of refactorings that ensures the balance between improving quality, removing code smells and introducing design patterns objectives. Lin *et al*. [15] proposed an interactive search-based recommendation tool to facilitate architectural refactoring. The tool calculates the discrepancies between the current implementation and the targeted design. Then iteratively the tool recommends refactoring steps towards realizing the desired design. Users' feedbacks on already recommended refactorings are used to improve subsequent refactoring recommendations.

To facilitate developers in making solutions selection decision, existing tools use ranking technique. Solutions are ranked based on particular criteria and are suggested to developers for selection. Chaparro *et al.* [12] proposed the tool which based on several code metrics to predict the impact of different refactorings. By using impact prediction functions, the tool allows developers to visualize the variation of code metrics and therefore assisting them in making design decisions before applying a particular refactoring.

Moreover, solutions are also ranked in the order of their effectiveness in improving code design such as coupling and cohesion [13]. For example, in JDeodorant (a widely used smell detector which is capable of providing refactoring choices) the suggested solutions are ranked based on ''entity placement'' value, i.e., a ratio of the overall system cohesion over its coupling [6], [7]. Another approach to select solution is based on the optimization of predefined objectives. Chisăliţă-Creţu [17] proposed multi-objective scenario-based approach to select a solution based on the cost and impact of the selected solution. Designers can decide whether to select the suggested solutions or not based on some other conceptual or design quality criteria [14]. According to [17], refactoring solutions can easily be organized and prioritized based on predefined goals. Recently, Kessentini *et al.* [47] proposed an approach to recommending refactorings based on the analysis of bug reports and history of change. Authors based on the premise that, a class which is frequently modified and appears in bug reports is more likely to demand refactoring. In addition to that, the previous applied refactorings were also used to deduce possible required refactorings for present release. Besides that, execution efforts [17], [48] and developers' feedback [15], [45] are taken into account as well.

E. 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 [16], [49]. One of the pertinent metrics currently in use in software engineering is entropy metrics. Etzkorn *et al.* [27] 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.* [25] 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.* [50] proposed the use of entropy in evaluating software quality degradation by assessing the number of links within and between abstraction models. Related work to [50] was also done by [51] and [52]. Maisikeli [51] proposed an approach to assess object oriented software maintainability and degradation by using the combination of entropy and other metrics.

III. PROBLEM STATEMENT

Most of the existing approaches recommend refactorings based on the improvement of code quality which is often quantified by code metrics [53]. Moreover, Ouni *et al.* [53] suggest that relying on design metrics only may be insufficient as there is a need to preserve the basis on why and how source code elements are grouped when applying refactorings. In addition to that, the existing approaches do not take into consideration the impact of such refactorings on the traceability between source code and requirements (use cases), 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.

IV. MOTIVATING EXAMPLE

Fig. 1 depicts a typical example drawn from iTrust, an open-source application for maintaining patients electronic health records [54]. This example illustrates the design fragment consisting of three classes; *GetVisitRemindersAction*, *TransactionDAO* and *VisitRemindersDAO*. According to the code smell detection techniques proposed in [9] and [7], 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 [55]. 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 [7].

Here we consider a suggested *Move Method* refactoring solution which is as well proposed in [56]. The solution moves the method *getVisitReminders*() from the class *GetVisitRemindersAction* to *TransactionDAO*.

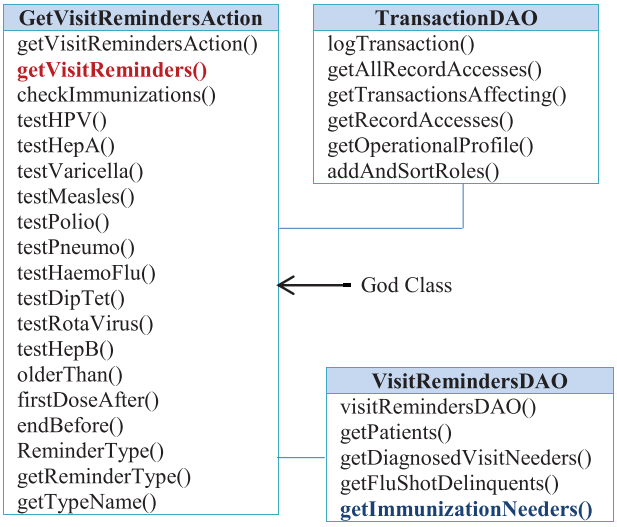


FIGURE 1. iTrust classes design snippet.

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 *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 [14]. 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 Ouni *et al.* [53] suggested that, to ensure quality improvement it is also important to consider additional objectives rather than solely relying on structural metrics.

V. RECOMMENDATION APPROACH

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

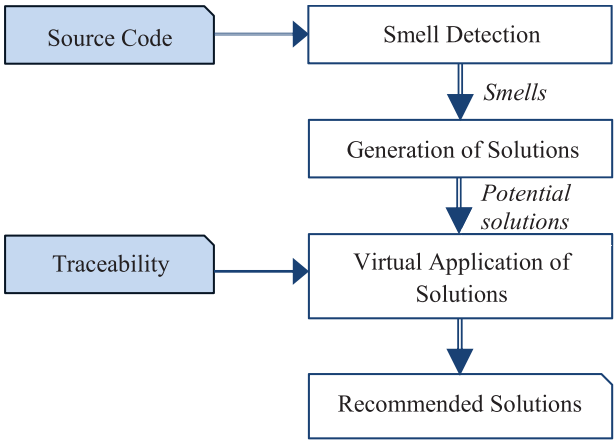


FIGURE 2. Traceability-enabled refactoring solutions recommendation framework.

A. OVERVIEW

An overview of the proposed approach is statically analyzed by Fig. 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.

B. 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 classes or methods that they are related to. Table 1 depicts a

TABLE 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

sample of TM. Traceability at the method granularity can be defined as:

- Let M be a set composed of methods $m_1, m_2,\ldots,m_k$.
- Let C be a set composed of classes $c_1, c_2,\ldots,c_h$. Each class from C is uniquely composed of one or more methods from M.
- Let U be a set composed of use cases, $u_1, u_2,\ldots,u_n$. 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 1 depicts a typical traceability matrix drawn from iTrust [54]. 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.

C. ENTROPY

Entropy was first proposed by Shannon [57] to measure the amount of information produced by the source. Shannon's definition of entropy determines number of bits required in identifying information distribution [25], [57]. Shannon [57] computed entropy as follows:

$$H_n(P) = -\sum_{k=1}^{n} p_k \log_2 p_k \tag{1}$$

where $p_k \geq 0 (k = 1, 2...., n)$ denotes the probability of occurrence for the k^{th} element and $\sum_{k=1}^{n} p_k = 1$ 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., $p_k = \frac{1}{n}$, for $k = 1, 2...., n$. But when one element e.g., j has a maximal probability of occurrence (i.e., $p_j = 1$), $H_n(P)$ 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* [50]. Entropy covers all the components of a software system and their traceability relationship [50]. 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.

1) CLASS TRACEABILITY ENTROPY

- Let c be a class composed of methods $m_1, m_2,\ldots, m_k$.
- 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:

$$P(u_i, c) = \frac{L(u_i, c)}{\sum L(U, c)} \tag{2}$$

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., $P(u_1, c) = P(u_2, c), \ldots = P(u_n, c)$, the class entropy is maximal. For every class, the sum of the ratios is always equal to 1, i.e.,

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

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

$$E(c) = -\sum_{i=1}^{n} P(u_i, c) \log_2 P(u_i, c) \tag{3}$$

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) 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 $c_1, c_2,\ldots, c_n$.

- 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:

$$R(u, c_i) = \frac{T(u, c_i)}{\sum T(u, S)} \tag{4}$$

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.; $R(u, c_1) = R(u, c_2), \ldots = R(u, c_n)$, the use case entropy is maximal. For every use case, the sum of the ratios is always equal to 1, i.e.;

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

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

$$E(u) = -\sum_{i=1}^{n} R(u, c_i) \log_2 R(u, c_i) \tag{5}$$

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.

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 $c_1, c_2, \ldots, c_n$.
- Let U be a set of use cases, $u_1, u_2, ..., u_j$ traced by one or more classes in S.

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

$$E(S, U) = \frac{\sum_{i=1}^{n} E(c_i) + \sum_{j=1}^{k} E(u_j)}{n + k} \tag{6}$$

where:

- $\sum_{i=1}^{n} E(c_i)$ is the sum of class traceability entropy of all classes in the system.
- $\sum_{j=1}^{k} E(u_j)$ 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.

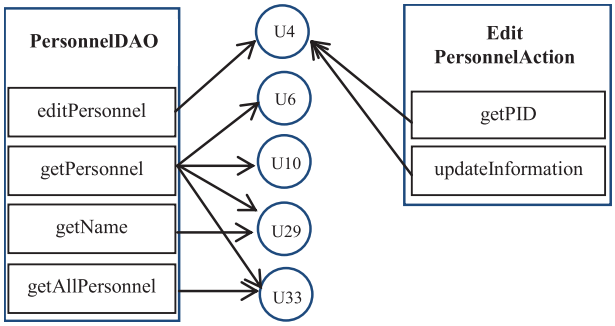


FIGURE 3. Traceability links before refactoring.

In Equation (6), 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.

4) COMPUTATIONAL EXAMPLE

Consider the traceability matrix depicted in Table 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. 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, therefore their entropy is zero. So we only need to compute entropy of use case $U4$:

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

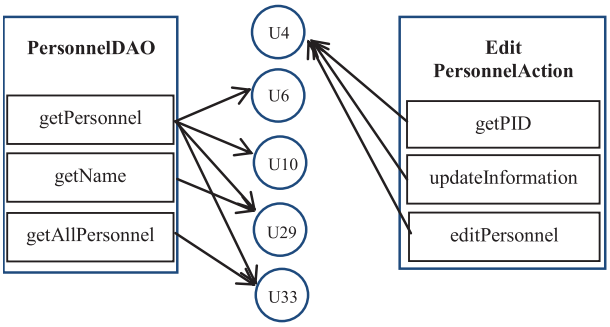


FIGURE 4. Traceability links after refactoring.

Therefore, traceability entropy before refactoring, $E(S, U)$, is computed by summarizing the traceability entropy of classes and use cases as shown in Equation (6). Thus, $E(S, U) = 0.451$.

To analyze the effect of the applied refactoring on traceability entropy, we compute entropy after refactoring as well. As depicted in Fig. 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 (6). 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.

D. 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 [14], 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 [58]. 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 [59]. 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 [2]. 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 [14] 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 [14]. Consequently, the proposed approach employs *EP* as well. *EP* is automatically computed by the refactoring tool (see Section V-E.1) and it is one of the parameters in our refactoring recommendation. The *EP* of a class C is therefore defined as:

$$EP_C = \frac{\frac{\sum_{e_i \in C} distance(e_i, C)}{|entities \in C|}}{\frac{\sum_{e_j \notin C} distance(e_j, C)}{|entities \notin C|}} \tag{7}$$

where e denotes an entity of the system and $distance(e_i, C)$ 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 [14].

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

$$EP_S = \sum_{C_i} \frac{|entities \in C_i|}{|allentities|} EP_{C_i} \tag{8}$$

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

E. 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 (9). 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.

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 [55] 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 [13]. 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" [55]. This method is often tightly coupled to the other class than its own [33]. 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 [7], [56]. 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 [37]. 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) 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.

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 V-C.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 V-D. 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 (6), 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:

$$ER(s) = \frac{EP_i - EP_s}{EP_i} + \frac{EN_i - EN_s}{EN_i} \tag{9}$$

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 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.

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 V-E.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 (9). The *ERList* holds

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**

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**

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.

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 V-E.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.

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.

VI. EVALUATION

A. RESEARCH QUESTIONS

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

- *RQ*1: Does the proposed approach outperform traditional code metrics-based approaches concerning the traceability between requirements and source code? If so, to what extent?
- *RQ*2: How often do developers prefer solutions recommended by our approach against those recommended by the traditional code metrics-based approaches?

Investigation of *RQ*1 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. *RQ*2 validates the proposed approach to assess its feasibility from the developers' point of view.

B. DATASET AND TOOL SUPPORT

We evaluated our approach on a well-known iTrust dataset [54]. 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 plugins, JDeodorant [6], [7], JMove [37], [56] and ARIES [60] 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 [30]–[32]. 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.

C. 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 *RQ*1, 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.

We further investigated the change in $ER(s)$ as shown in Equation (9) 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. 5 and Fig. 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. 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. 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 $ER(s)$.

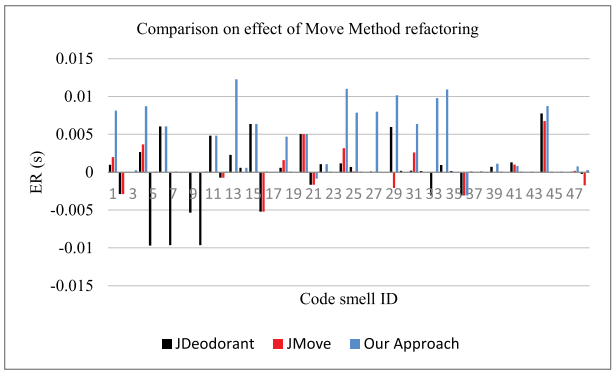


FIGURE 5. Comparison on effect of move method refactoring.

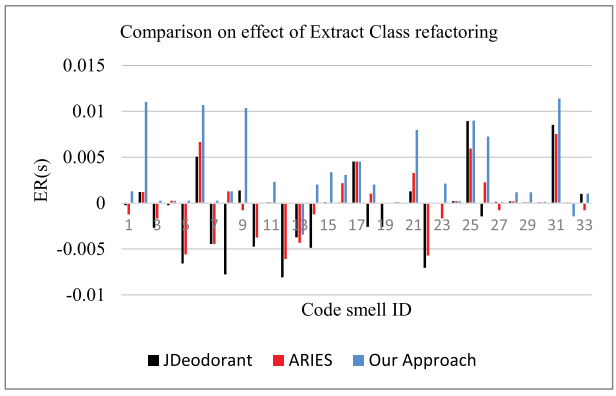


FIGURE 6. Comparison on effect of Extract Class refactoring.

D. 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 [39]. For example in [39] and [45], 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 *RQ*2.

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* ≥ 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:

$$P(ap) = \frac{Total\ number\ of\ preferred\ \mathbf{ap}\ solutions}{Total\ number\ of\ code\ smells} \quad (10)$$

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*:

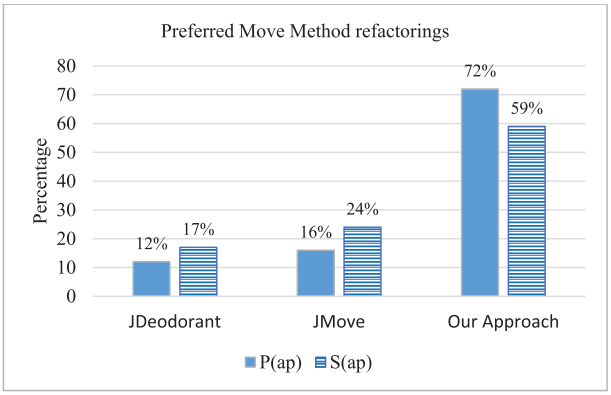


FIGURE 7. Preferred Move Method refactoring solutions.

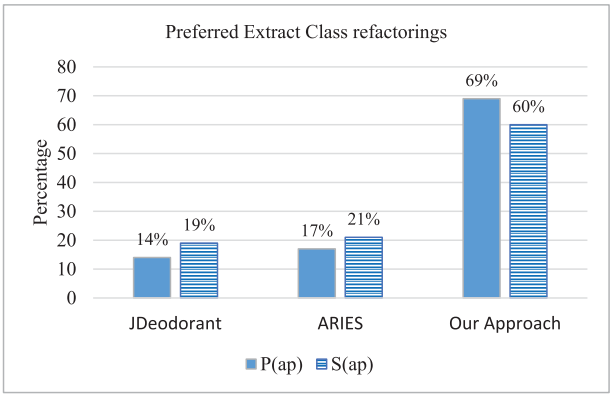


FIGURE 8. Preferred Extract Class refactoring solutions.

$$S(ap) = \frac{Total\ votes\ \mathbf{ap}\ received}{Total\ votes\ issued\ by\ developers} \quad (11)$$

Fig. 7 and Fig. 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. 7 and Fig. 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. 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. 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. 8. From this figure, we also observe that most (69%) of the refactoring solutions recommended by

```
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 9. Move Method refactoring.

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. 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

```
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 10. Extract Method refactoring.

relationship, to supplement our traceability-based solutions recommendation.

Second, our approach recommends refactoring based on assessing the effects of the solution 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. 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 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. 11 and Fig. 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. 11 and Fig. 12 it is evident that all developers have almost the same preference to the proposed approach.

E. 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

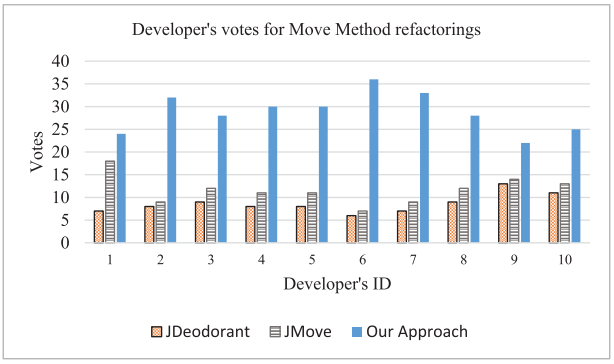


FIGURE 11. Developers' votes for the Move Method refactorings.

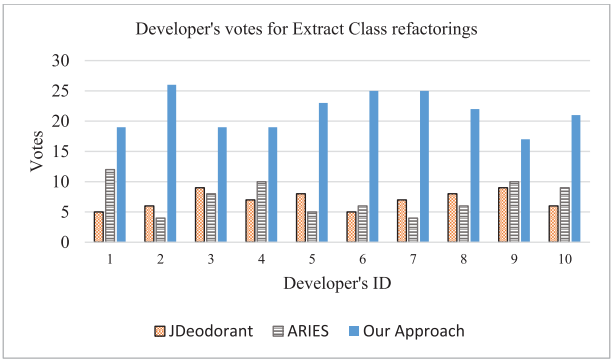


FIGURE 12. Developers' votes for the Extract Class refactorings.

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 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.

VII. CONCLUSION AND FUTURE WORK

In this paper, 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.

Future work in this direction is, first to validate our approach on other refactorings which are not included in this study, particularly those which involve splitting of methods. Second, we would like to further assess the effect of preserving and improving traceability on source code design metrics. This was not considered in this initial attempt 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. Finally, we want to devise a tool that will automatically and efficiently perform recommendation task.

ACKNOWLEDGMENTS

The authors would like to say thanks to the associate editor and the anonymous reviewers for their insightful comments and constructive suggestions.

REFERENCES

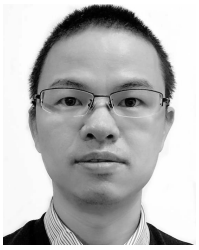
[1] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[2] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 671–694, Jul. 2014.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code* (Addison-wesley Object Technology Series). Reading, MA, USA: Addison-Wesley, 1999.
[4] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1112–1126, Aug. 2013.
[5] M. Abebe and C.-J. Yoo, "Trends, opportunities and challenges of software refactoring: A systematic literature review," *Int. J. Softw. Eng. Appl.*, vol. 8, no. 6, pp. 299–318, 2014.
[6] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "JDeodorant: Identification and removal of feature envy bad smells," in *Proc. IEEE Int. Conf. Softw. Maintenance (ICSM)*, Oct. 2007, pp. 519–520.
[7] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: Identification and application of extract class refactorings," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, Honolulu, HI, USA, May 2011, pp. 1037–1039.
[8] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel, "iPlasma: An integrated platform for quality assessment of object-oriented design," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance–Ind. Tool (ICSM)*, Budapest, Hungary, Sep. 2005, pp. 77–80.
[9] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proc. 5th ACM Symp. Softw. Vis.*, Salt Lake City, UT, USA, Oct. 2010, pp. 5–14.
[10] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan./Feb. 2010.
[11] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," *IEEE Softw.*, vol. 25, no. 5, pp. 38–44, Sep. 2008.
[12] O. Chaparro, G. Bavota, A. Marcus, and M. D. Penta, "On the impact of refactoring operations on code quality metrics," in *Proc. 30th IEEE Int. Conf. Softw. Maintenance Evol.*, Victoria, BC, Canada, Sep./Oct. 2014, pp. 456–460.
[13] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *J. Syst. Softw.*, vol. 85, no. 10, pp. 2241–2260, 2012.
[14] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 347–367, May 2009.
[15] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Seattle, WA, USA, Nov. 2016, pp. 535–546.
[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
[17] C. Chisăliţă-Creţu, "The multi-objective refactoring set selection problem-a solution representation analysis," in *Advances in Computer Science and Engineering*. Rijeka, Croatia: InTech, 2011.
[18] O. C. Z. Gotel and C. W. Finkelstein, "An analysis of the requirements traceability problem," in *Proc. 1st IEEE Int. Conf. Requirements Eng. (ICRE)*, Colorado Springs, CO, USA, Apr. 1994, pp. 94–101.
[19] P. Rempel and P. Mäder, "Continuous assessment of software traceability," in *Proc. 38th Int. Conf. Softw. Eng. Companion (ICSE)*, Austin, TX, USA, May 2016, pp. 747–748.
[20] P. Rempel and P. Mäder, "A quality model for the systematic assessment of requirements traceability," in *Proc. 23rd IEEE Int. Requirements Eng. Conf. (RE)*, Ottawa, ON, Canada, Aug. 2015, pp. 176–185.
[21] R. Tsuchiya, T. Kato, H. Washizaki, M. Kawakami, Y. Fukazawa, and K. Yoshimura, "Recovering traceability links between requirements and source code in the same series of software products," in *Proc. 17th Int. Softw. Product Line Conf. (SPLC)*, Tokyo, Japan, Aug. 2013, pp. 121–130.
[22] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol, "Requirements traceability for object oriented systems by partitioning source code," in *Proc. 18th Work. Conf. Reverse Eng. (WCRE)*, Limerick, Ireland, Oct. 2011, pp. 45–54.
[23] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, Oct. 2002.
[24] D. Poshyvanyk and A. Marcus, "Using traceability links to assess and maintain the quality of software documentation," in *Proc. Traceability Emerg. Forms Softw. Eng.*, vol. 7, 2007, pp. 27–30.
[25] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "How changes affect software entropy: An empirical study," *Empirical Softw. Eng.*, vol. 19, no. 1, pp. 1–38, 2014.
[26] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng. (ICSE)*, Vancouver, BC, Canada, May 2009, pp. 78–88.
[27] L. H. Etzkorn, S. Gholston, and W. E. Hughes, Jr., "A semantic entropy metric," *J. Softw. Maintenance Evol., Res. Pract.*, vol. 14, no. 4, pp. 293–310, 2002.
[28] L. Yang, Z. Dang, T. R. Fischer, M. S. Kim, and L. Tan, "Entropy and software systems: Towards an information-theoretic foundation of software testing," in *Proc. 18th ACM SIGSOFT Workshop Future Softw. Eng. Res. (FoSER) and Int. Symp. Found. Softw. Eng.*, Santa Fe, NM, USA, Nov. 2010, pp. 427–432.
[29] N. Niu, W. Wang, and A. Gupta, "Gray links in the use of requirements traceability," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, Seattle, WA, USA, Nov. 2016, pp. 384–395.
[30] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, "An experience report on using code smells detection tools," in *Proc. 4th IEEE Int. Conf. Softw. Test., Verification Validation Workshops (ICST)*, Berlin, Germany, Mar. 2011, pp. 450–457.
[31] T. Paiva, A. Damasceno, J. Padilha, E. Figueiredo, and C. Sant'Anna, "Experimental evaluation of code smell detection tools," in *Proc. 3rd Workshop Softw. Vis., Evol., Maintenance (VEM)*, 2015, pp. 17–24.
[32] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proc. 20th Int. Conf. Eval. Assessment Softw. Eng. (EASE)*, Limerick, Ireland, Jun. 2016, Art. no. 18.
[33] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *J. Object Technol.*, vol. 11, no. 2, pp. 1–5, 2012.
[34] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, Eds. *Recommendation Systems in Software Engineering*. Berlin, Germany: Springer-Verlag, 2014.
[35] A. Mahmoud and N. Niu, "Supporting requirements traceability through refactoring," in *Proc. 21st IEEE Int. Requirements Eng. Conf. (RE)*, Rio de Janeiro, Brazil, Jul. 2013, pp. 32–41.
[36] P. Mäder and A. Egyed, "Assessing the effect of requirements traceability for software maintenance," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Trento, Italy, Sep. 2012, pp. 171–180.
[37] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending move method refactorings using dependency sets," in *Proc. 20th Work. Conf. Reverse Eng. (WCRE)*, Koblenz, Germany, Oct. 2013, pp. 232–241.
[38] A. Delater and B. Paech, "Tracing requirements and source code during software development: An empirical study," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Baltimore, MD, USA, Oct. 2013, pp. 25–34.
[39] N. Niu, T. Bhowmik, H. Liu, and Z. Niu, "Traceability-enabled refactoring for managing just-in-time requirements," in *Proc. IEEE 22nd Int. Requirements Eng. Conf. (RE)*, Karlskrona, Sweden, Aug. 2014, pp. 133–142.
[40] Y. Shin, J. H. Hayes, and J. Cleland-Huang, "A framework for evaluating traceability benchmark metrics," DePaul Univ., Chicago, IL, USA, Tech. Rep. 21, 2012.
[41] N. Ali, Z. Sharafi, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study on the importance of source code entities for requirements traceability," *Empirical Softw. Eng.*, vol. 20, no. 2, pp. 442–478, 2015.
[42] J. Cleland-Huang, O. C. Z. Gotel, J. H. Hayes, P. Mäder, and A. Zisman, "Software traceability: Trends and future directions," in *Proc. Future Softw. Eng. (FOSE)*, Hyderabad, India, May 2014, pp. 55–69.
[43] M. Eyl, C. Reichmann, and K. Müller-Glaser, "Traceability in a fine grained software configuration management system," in *Proc. Int. Conf. Softw. Qual.* Cham, Switzeland: Springer, 2017, pp. 15–29.
[44] F. Faiz, R. Easmin, and A. Ul Gias, "Achieving better requirements to code traceability: Which refactoring should be done first?" in *Proc. 10th Int. Conf. Qual. Inf. Commun. Technol. (QUATIC)*, Sep. 2016, pp. 9–14.
[45] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó. Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, Vasteras, Sweden, Sep. 2014, pp. 331–336.
[46] A. Ouni, M. Kessentini, M. O. Cinnéide, H. Sahraoui, K. Deb, and K. Inoue, "MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells," *J. Softw., Evol. Process*, vol. 29, no. 5, p. e1843, 2017.
[47] M. Kessentini, T. J. Dea, and A. Ouni, "A context-based refactoring recommendation approach using simulated annealing: Two industrial case studies," in *Proc. Genetic Evol. Comput. Conf. (GECCO)*, Berlin, Germany, Jul. 2017, pp. 1303–1310.
[48] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in *Proc. 17th Eur. Conf. Softw. Maintenance Reengineering (CSMR)*, Genova, Italy, Mar. 2013, pp. 221–230.

[49] L. Cheikhi, R. E. Al-Qutaish, A. Idri, and A. Sellami, "Chidamber and kemerer object-oriented measures: Analysis of their design from the metrology perspective," *Int. J. Softw. Eng. Appl.*, vol. 8, no. 2, pp. 359–374, 2014.
[50] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio, "Evaluating software degradation through entropy," in *Proc. 7th IEEE Int. Softw. Metrics Symp.*, London, U.K., Apr. 2001, pp. 210–219.
[51] H. M. Olague, L. H. Etzkorn, and G. Cox, "An entropy-based approach to assessing object-oriented software maintainability and degradation—A method and case study," in *Proc. Int. Conf. Softw. Eng. Res. Pract. and Conf. Program. Lang. Compil. (SERP)*, Las Vegas, NV, USA, vol. 1, Jun. 2006, pp. 442–452.
[52] S. G. Maisikeli, "Evaluation and study of software degradation in the evolution of six versions of stable and matured open source software framework," in *Proc. 6th Int. Conf. Comput. Sci., Eng. Appl. (ICCSEA)*, Dubai, UAE, Sep. 2016, pp. 24–25.
[53] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, 2016, Art. no. 23.
[54] A. Meneely, B. Smith, and L. Williams, "iTrust electronic health care system: A case study," in *Software and Systems Traceability*, J. ClelandHuang, O. Gotel, and A. Zisman, Eds. London, U.K.: Springer-Verlag, 2012.
[55] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," in *Proc. 12th Work. Conf. Reverse Eng. (WCRE)*, Pittsburgh, PA, USA, Nov. 2005, pp. 155–164.
[56] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "JMove: A novel heuristic and tool to detect move method refactoring opportunities," *J. Syst. Softw.*, vol. 138, pp. 19–36, Apr. 2018.
[57] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Comput. Commun. Rev.*, vol. 5, no. 1, pp. 3–55, 2001.
[58] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proc. 8th Annu. Genetic Evol. Comput. Conf. (GECCO)*, Seattle, WA, USA, Jul. 2006, pp. 1909–1916.
[59] C. Bonja and E. Kidanmariam, "Metrics for class cohesion and similarity between methods," in *Proc. 44th Annu. Southeast Regional Conf.*, Melbourne, FL, USA, Mar. 2006, pp. 91–95.
[60] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba, "Supporting extract class refactoring in eclipse: The ARIES project," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Zürich, Switzerland, Jun. 2012, pp. 1419–1422.

ALLY S. NYAMAWE received the B.Sc. degree in computer science from the University of Dar es Salaam, Tanzania, in 2008, and the M.Sc. degree in computer science from the University of Dodoma, Tanzania, in 2011. He is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Beijing Institute of Technology, China. He is currently a Lecturer with the Department of Computer Science, University of Dodoma. His research interests include software refactoring, requirements engineering, requirements traceability, and computer programming.



HUI LIU received the B.S. degree in control science from Shandong University in 2001, the M.S. degree in computer science from Shanghai University in 2004, and the Ph.D. degree in computer science from Peking University in 2008. He was a Visiting Research Fellow with the Centre for Research on Evolution, Search and Testing, University College London, U.K. He is currently a Professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. He is particularly interested in software refactoring, software evolution, and software quality. He is also interested in developing practical tools to assist software engineers. He has served on the program committees and organizing committees of many prestigious conferences, such as ICSME and RE.

ZHENDONG NIU received the Ph.D. degree in computer science from the Beijing Institute of Technology, China, in 1995. He was a Post-Doctoral Researcher with the University of Pittsburgh from 1996 to 1998 and a Researcher/Adjunct Faculty Member with Carnegie Mellon University from 1999 to 2004. He has been an Adjunct Professor with the Computing and Information School, University of Pittsburgh, since2006. He is currently a Professor and the Deputy Dean of the School of Computer Science and Technology, Beijing Institute of Technology. He has published over 150 papers in journals and international conferences in his field. His research areas include digital libraries, e-learning techniques, neuroinformatics, information retrieval, and recommender systems. He serves as an Editorial Board Member for the *International Journal of Learning Technology*.

WENTAO WANG (S'15) received the B.Sc. degree in computer science from Shanghai Maritime University, Shanghai, China, in 2007, and the M.Eng. degree in software engineering from the Beijing Institute of Technology, Beijing, China, in 2010. He is currently pursuing the Ph.D. degree with the Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH, USA. His research interests include software engineering, requirements engineering, and cyber security.

NAN NIU (M'08–SM'13) received the B.Eng. degree from the Beijing Institute of Technology in 1999, the M.Sc. degree from the University of Alberta in 2004, and the Ph.D. degree from the University of Toronto in 2009, all in computer science. He is currently an Associate Professor with the Department of Electrical Engineering and Computer Science, University of Cincinnati, USA. His research interests include software requirements engineering, information seeking in software engineering, and human-centered computing. He received the U.S. National Science Foundation Faculty Early Career Development Award, the IEEE International Requirements Engineering Conference Best Research Paper Award in 2016, and the Most Influential Paper Award in 2018.