I.J. Information Technology and Computer Science, **2017, 1, 46-57**
Published Online January 2017 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2017.01.06

Analysis of Metric-Based Object-Oriented Code Refactoring Opportunities Identification Approaches

Bassey Isong, Nosipho Dladlu, Etim Duke
North-West University, Computer Science Department, Mafikeng, South Africa
E-mail: {bassey.isong, nosipho.dladlu, 25831127}@nwu.ac.za

Bassey Ele
University of Calabar, Computer Science Department, Calabar, Nigeria
E-mail: mydays2020@gmail.com

***Abstract*—**Refactoring is used to improve deteriorated software design, code and their maintainability. In object-oriented (OO) code, before refactoring is performed, its opportunities must be identified and several approaches exist this regard. Among the approaches is the software metric-based approach where quality software metrics are used. Therefore, this paper provide analysis of existing empirical studies that utilized software metrics to identify refactoring opportunities in OO software systems. We performed a comprehensive analysis on 16 studies to identify the state-of-the-practice. The focal point was on the workings, refactoring activities, the programming language and the impact on software quality. The results obtained shows approaches were not unique, each was designed either for a single refactoring activity or couple of them, move method and extract class dominated the refactorings activities, and most approaches were fully automated while few were semi-automated. Moreover, OO metrics played acritical role in both opportunities detection and factoring decisions. Based on the results, it would be beneficial if generic refactoring approach is developed that is capable of identifying needs for all refactoring activities.

***Index Terms*—**Object-oriented, Code, Metrics, Maintenance, Refactoring, Identification.

I. INTRODUCTION

Software quality is a fundamental element in the success or failure of any software organization and is of great concern in the software engineering (SE) field. With the recent increase in the size and complexity of software applications, assuring high quality in software products has been more and more difficult and a timewasting task [1][2]. Thus, to achieve high quality in software product, cost-effective and proactive techniques are of great importance. In the realm of object-oriented (OO) software maintenance, an indicator of good quality design is to adhere to low coupling and high cohesion in the design [3]. The principle of low coupling and high cohesion has been widely known and during software development they constitutes a primary target by software engineers. This is because, the realization of the their reverse form in software products has been linked to cases of costly rework, higher fault rates, lower developers' productivity and increasing design efforts [3][4]. Thus, to ensure that software systems remain useful throughout its life time, changes are inevitable [5][6]. Change is an inherent property of real-world software which is realized through evolution, maintenance and in conformance with the Lehman's first law of evolution [5][7], which advocate for continuous software changes. The drivers of software changes are faults fixing, adapt to new requirements or changed environment, improve performance and so on [6]. However, due to continuous modification and enhancement of the internal structure of the software system, the code easily becomes extremely complex and moves gradually away from its initial design [8]. In other words, the software structural design deviates from its initial design or deteriorates in quality, thereby revealing unanticipated values of cohesion and coupling. For instance, software components like classes grow swiftly as developers often time add new responsibilities to existing ones instead of new classes. Consequently, as the class responsibilities increases, so its complexity leading to the deterioration of its quality [9][10]. Studies have shown that some of the causes of design flaws in software systems are due to the application of inappropriate design solutions by developers leading to code not conforming to OO programming rules and market pressure resulting to strict deadline [11][12]. To this end, comprehending and maintaining the software systems becomes a difficult, if not impossible tasks. This constitute the reason why software maintenance is considered to be the costliest phases of software development since a considerable cost of development is highly expended on maintenance [13].

To put such situations under control, a cost-effective technique that lessen the complexity of the OO software systems by improving its internal structure and lower maintenance cost while preserving its quality is indispensable. In SE, this technique is knows as

Copyright © 2017 MECS

refactoring [14]. Software refactoring is a cost-effective technique to eliminate design problems often called "*bad smells*" in code [14][15]. It is a maintenance process where software systems are changed in a manner that only its internal structure is improved while the external behavior remains intact. In the perspective of OO software source code, refactoring is geared towards incrementally enhancing the internal complexity due to continuous changes in order to expedite future improvements [8][14][15].

Software refactoring has gained momentum today and in particular, has become an integral part of agile development process such as extreme programming (XP) [16]. To perform refactoring in software code, the first task is to pinpoint the refactoring candidates that manifest in the form of bad smells before applying the appropriate refactoring to remove them. Today, there exist several approaches to identify refactoring opportunities in OO software systems and the application of appropriate refactorings have been proposed, developed and utilized during software development [10][12][17][18][19][20]. However, these approaches are designed to identify opportunity for a particular refactoring candidate or couple of them which is achieved by either full automation or semi automation. In addition, there are no studies to the best of our knowledge which provides a comprehensive analysis of these approaches in order to give insights into their state-of-the-practice. Therefore, in this paper, we present analysis of the existing refactorings opportunities identification approaches in the context of quality metrics. We specifically performed analysis on 16 existing empirical studies in the literature and in particular, we aimed to answer the research questions: *How are the opportunities for refactorings identified in OO source code? Of what important is software metrics during refactorings opportunities identification? And do traditional metrics played the same role as OO design metrics in the identification of refactorings opportunities in OO source code?* Accordingly, the contributions of this paper is to give an insight into software refactorings, perform comprehensive analysis of the approaches to determine their mode of operations, refactorings activities, programming languages which are mostly refactored, and the impact of software metrics on code quality. Moreover, this paper provides a useful direction for future research.

The remaining parts of the paper is organized as follows: Section II is the related studies, III discusses refactoring opportunities identification process, Section IV is the analysis of the existing approaches, Section V is the paper discussion and VI is the conclusions.

II. REALATED STUDIES

This section presents related works in software refactoring in terms of refactoring process and research activities.

A. *Software Refactoring*

It is widely recognized that every real-world software system has to evolve during its lifetime in order to continue to remain useful. However, during software evolution or maintenance, as the software systems' internal structure is subjected to continuous enhancement, modification and adaptation, its code becomes complex and consequently drift away from the original design [5][21]. Furthermore, poor design decisions due to strict condition of deadline forces developers not to adhered to the principle of high cohesion and low coupling [11][12]. As a result, the software (packages, classes, methods or field) in turn becomes more complex and deteriorates in quality thereby making it difficult to understand and maintain [9]. In particular, this constitutes the reason why software maintenance is costly and several researches have shown that about 90% of the total development cost is consumed by maintenance [19][22].

Table 1. Refactoring Activities

Refactoring Activities	Description
Extract Subclass	A subclass that has subset of features that are used only in some instances in its superclass.
Move Method	Moving a method to a class in which most of its features are used by the method other than the class it resides in.
Extract Class	A new class being created from a large class that performs more than one task.
Move Class	Moving a class from package that is not suitable to a more suitable one.
Extract Method	Grouping code fragment to form a new more cohesive method from long methods
Pull Up Methods	Pulling up methods with identical results on child class to its parent class.
Extract Method	Removing and redefining code fragment as a new method from larger methods.
Form Template Method	Removing duplication by merging and pulling up to the superclass, similar steps of two methods in subclasses that perform similar steps while leaving their steps that are different in the subclasses.
Pull Up Constructor	Subclass methods created in a superclass from constructors in the subclasses that have similar or identical bodies.
Parameterize Method	A single method where a parameter is used for different values contained in methods body.

Therefore, to minimize the high cost attributed to maintenance, software refactoring has become the mainstream approach. It has been introduce in the perspective of OO programs to improve the complexities in code design. Refactoring is defined by Fowler et al [14] as "*the process of changing OO software systems in a manner that it does not alter the external code behavior yet improves its internal structure*". The essence of refactoring is to redistribute and rearrange OO software structure while keeping its semantics intact. In other words, refactoring is a technique that is effective in the removal of design defects called *bad smells* emanating from the violation of high cohesion and low coupling principles in order to improve code comprehension and maintainability [14][15]. A bad smell in source code is a classical sign of poor quality and has to be removed. There are several benefits that are linked to refactoring such as test effort reduction, design simplification, validation assistance, design change automation and new designs investigation [23]. In addition, it can be applied

Copyright © 2017 MECS

to other software artifacts other than source codes such as requirements specification, documentation, design documents, software architectures, test suites, etc. [8].

During software maintenance, to get rid of bad smells in code, Fowler [14] has identified and offered the explanation of 22 bad smells in OO source code and 72 refactorings strategies to enhance source code design with bad smells. Examples of code bad smells are duplicated code, large class, long method, lazy class, feature envy, long parameter list, shotgun surgery and so on [14]. The different refactorings operations to remove the bad smells are shown in Table 1. For more information on bad smells and their respective refactorings, referred to [14].

B. Refactoring Process

In OO software system, the main goal of refactoring is to reduce the internal structural complexity while preserving its external behavior [14]. However, performing refactoring is not a one-way process. The process involves series of steps or activities aimed at ensuring its appropriateness and software quality preservation. Mens and Tourwe [8] provides six distinct steps to be taken to perform refactoring. The steps are shown in Fig. 1.

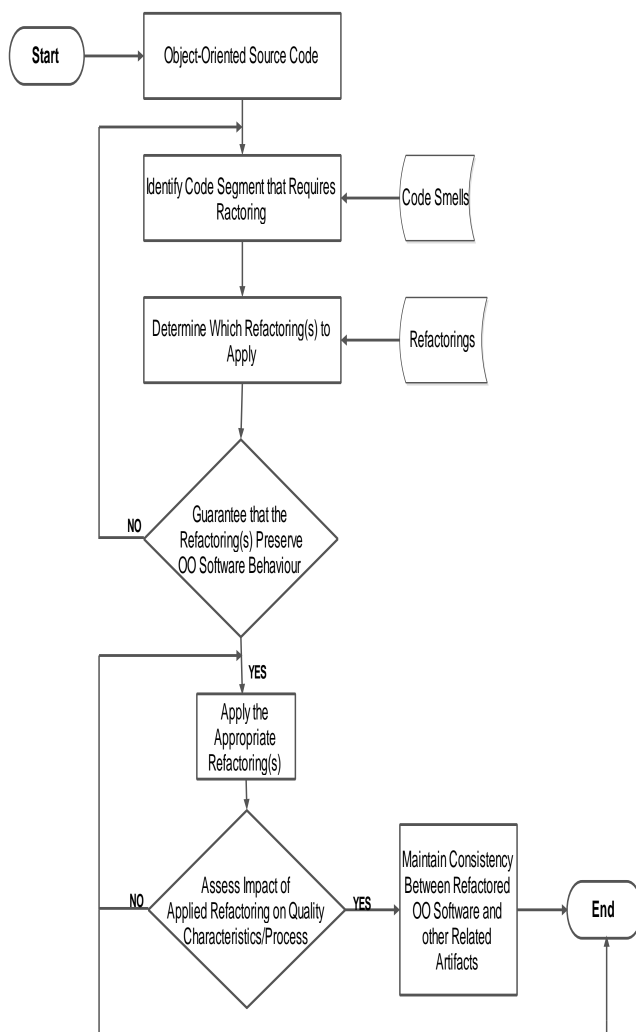

Fig.1. Refactoring process steps

In Fig. 1, the refactoring process starts by taking as input, the OO software system's source code to identify which segments of the code need to be refactored. This is simply identifying the bad smell in the code. It involve deciding on the suitable abstraction level which the refactoring should be applied as well as evaluating the cost/benefits of each identified refactoring [8]. The identification is followed by determining which appropriate refactorings are to be applied by examining the code bad smells identified. Fowler et al [14] had already offered solutions in which some are shown in Table 1. With the decisions on the appropriate refactorings to apply, next step is to assure that they preserves the external behavior of the software. In this case, for a given input value, the corresponding output values should be the same before and after refactoring has been conducted [8]. This step is very important as it ensures that other software activities that rely on the program are not invalidated. However, if it is found that the software behavior is not preserved, the refactoring process can be aborted or another appropriate refactorings chosen. With the preservation of the program behavior, the next step is the actual application of the chosen refactorings on the OO source code and is succeeded by the assessment of its effect on quality characteristics or the processes. To this end, quality characteristics such as maintainability, comprehensiveness and complexity are assessed. Others are the process characteristics such as the efforts, costs, productivity and so on [8]. The rationale is to ensure that the characteristics are improved by the refactorings applied. Finally, software engineers have to ensure that the refactored OO software is consistent with other software artifacts.

C. Refactoring Research Activities

There are few studies that exist in the form of surveys and literature reviews that have provided information on software refactoring with respect to the state-of-the-art and practices. Thus, this section highlights some of the research activities.

Mens and Tourwe [8] conducted a comprehensive survey of software refactoring research. In this study, they focused on the comparison and discussion of existing refactoring activities with their supporting techniques and formalisms, the different types of software artifacts that can be refactored, important issues on tool support as well as the effects refactoring has on the software process. Moreover, the study discussed on the steps to be taking to perform refactoring software artifacts where refactoring opportunities identification is one such steps. However, no single approach was considered or analyzed. In another study by Zhang et al [35], a systematic literature review was carried out. In their study, about 39 primary published studies based on code bad smells were considered. For each primary study, Zhang et al [35], focused on different the code smells under analysis, the underlying goals, methods applied in exploring the code bad smells, and the supporting indication that each was problematic during maintenance. In another study, Wangberg [36] performed a literature

Copyright © 2017 MECS

review where about 46 papers were examined in the perspective of both bad smells and refactoring. The different focal points of the paper were empirical, design, contribution, summaries, and so on. In addition, the study focused on several aspect of both design and non-OO systems including detecting where refactoring is needed, how it is performed and the analysis of their quality impacts. In a similar studies, Misbhauddin and Alshayeb [37] also performed a systematic literature review on existing software refactoring research. The study only focused on refactoring UML models and about 94 primary studies were considered. Their focus areas are the UML models, the applicable formalisms, and the software quality impact. In a study by Dallal [17], a systematic literature review was performed on code-based refactoring approaches involving about 47 primary studies that have provided empirical evidences on OO software refactoring identification opportunities. The study focused a number of criteria ranging from refactoring activities to the data set employed.

In the studies highlighted above [8][17][35][36][37], the study in [8] only considered refactoring opportunities identification as a step to perform refactoring in software artifact, [35] only considered the identification of code bad smells though related to refactoring opportunities identification but different in the problem solved. The study in [36] focused on refactoring opportunities identification and others for non-OO systems but no analysis and empirical evidences were provided for the proposed refactoring opportunity identification approach. Moreover, [37] focused on UML model refactoring which involves only design while [17] approach focused on refactoring opportunities identification of OO source code and provided information on the refactoring activities and the category of the different approaches. However, what was lacking is an in-depth analysis of each approach. Therefore in this paper, we extend the work by [17] to provide analysis of the state-of-the-practice of each approach under the quality metric-oriented approach.

III. Object-Oriented Refactoring Oportunities Identification

Till date, software refactoring has been seen as one of the cutting-edge software development practice to improve the internal structure of legacy software systems [15]. Nevertheless, before refactoring can be carried out, the refactoring candidates or the code segment where the bad smell resides has to be identified in order to decide on refactorings appropriateness. Identifying where refactoring is required is a crucial step in the refactoring process as shown in Fig. 1. However, it is not an easy task deciding the appropriate refactorings to apply and the point where to apply it in the code. This challenge emanates from the fact that wrong decisions can be destabilizing on the entire structure of the OO software system [8][14]. Thus, to identify segment of the OO code where bad smells threatens the quality, three processes are of importance [24]. This is shown in Fig. 2. Moreover, there are several approaches or techniques that have been proposed and utilized in the identification of refactoring opportunities [17][18][25]. These approaches are either completely automated or semi-automated [12][17][18]. Nonetheless, the goals are centered on reducing the high cost of maintenance due to increased complexity in order to increase source code comprehensibility, maintainability and extensibility [8][14]. Furthermore, identifying the refactorings opportunities in OO code manually has been deemed challenging, costly and timewasting task [17][18]. The existing approaches can be applied to OO code written in diverse languages and are grouped into six categories such as quality metric-oriented, precondition-oriented, cluster-oriented, graph-oriented, code slicing-oriented and dynamic analysis-oriented approaches [17]. Each is unique in its way of operations, some are based on code's structural information, semantic information while others are hybrid in nature [10][12][17]. Thus, this paper provides analysis of the state-of-the-practice on the quality metric-oriented approach.

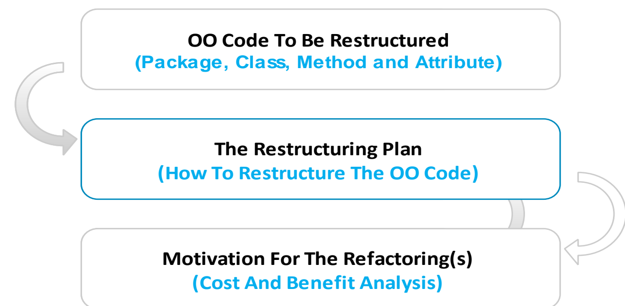

Fig.2. Refactoring opportunity identification steps

IV. Analysis of Metric-Based Object-Oriented Refactoring Opportunities Identification

The identification of refactorings opportunities is an active research area as reported in the systematic literature review performed by Al Dallal [17] and several work has been done. In this section, we performed analysis on 16 relevant studies whose approaches have been empirically evaluated. The analysis is performed by answering the research questions stated in this paper.

A. RQ1: How are the Opportunities for Refactorings Identified in OO Source Code?

Based on the 16 studies considered in this paper, the approaches that utilized software metrics employed the use of either structural, semantic or structural and semantic information to identify the bad smells in OO code in order to apply the appropriate refactorings to fix them [10][12][18]. Each approach has its unique operations in identifying the opportunities for either a single or multiple refactorings. However, some of the approaches shared identification techniques, though designed for different refactorings operations. The approaches that utilized code structural information employed prediction models whiles others only relied on

Copyright © 2017 MECS

both structural and sematic information. We provide analysis of each to explore their state-of-the-practice as follows:

Structural Information-based Approach: In this approach, three studies are known where two employed predictive model: Al Dalal [18] and Kosker et al [26] while the approach by Al Dallal and Briand [25] relied on the code structural information using either OO metrics, traditional metrics or hybrid of them to identify opportunities for refactorings.

"Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics": Al Dalal [18] built a model to predict opportunities for refactorings using empirically validated quality metrics and logistic regression (LR) as statistical technique. The study first explored empirically, the capabilities of some existing metrics: 25 size, cohesion, and coupling to predict refactoring opportunities individually in a given class using the univariate LR. The refactoring activities being explored was the extract subclass refactoring. Moreover, they applied multivariate LR to select optimal subset of metrics based on certain practical thresholds to construct models that predict the classes in the system that actually requires extract subclass refactoring or not. To this end, the model that best classify the classes is recommended and can be applied automatically to predict classes that requires extract subclass refactoring operations alongside suggestions for system improvement to developers during maintenance task [18]. According to [17][18], the automation was necessary to get rid of the inefficiency posed by the manual process. In general, extract subclass refactoring opportunity is identified by mutating source code a class with inheritance relations. The recommended predictive model was applied to 6 open source systems written in Java and the results obtained showed an overall improvement in the quality of classes, given the size, cohesion and coupling.

"An expert system for determining candidate software classes for refactoring": Kosker et al [26] proposed approach to predict refactorings opportunities based on expert system. The study employed complexity metrics of classes and Naive Bayes as prediction model based on machine learning to identify three refactoring operations such as extract superclass, extract interface and push members down. The study proposed and utilized complexity metrics to analyze complexity in OO code. It then constructed a prediction model using weighted Naive Bayes with InfoGain heuristic as the learner [26]. Moreover, class information were utilized and the modelling of the problem was considered as two-way classification problem with results that that suggest if a class is to be refactored or note. Based on the classifier results, classes that need refactoring can then be figured out by the engineers to identify the appropriate refactorings. The approach was empirical evaluated on a local GSM Operator Company system called Trcll1 project implemented in Java to assess its performances. The results obtained showed that the approach works better and by predicting which classes requires refactoring, effort to inspect code manually are reduced, assist in identifying complex and difficult code segments and as well, reduce the overall maintenance cost.

"A precise method-method interaction-based cohesion metric for object-oriented classes": Al Dallal and Briand [25] also proposed a cohesion metric based on pre-existing cohesion metrics of classes that relied mostly on method-method interactions (MMI) metrics to identify where refactorings are needed in code. The metric called Low-level design Similarity-based Class Cohesion (LSCC) [25], was used to quantify the degree of communication between methods in a class. It collects common attributes that exist between methods in a class and used them to quantify the degree of similarity. Moreover, a mathematical-based refactoring procedure for LSCC metric was proposed alongside their objectives. The refactoring activities detected by LSCC are the move method and extract class refactoring. The metric was validated both theoretical and empirically for the identification of refactoring opportunities originating from weakly cohesive classes. Theoretically, LSCC metric was validated for compliance with essential properties of the attribute it measured, while the empirical validation was to test its statistical relationships with external quality attributes [25]. By case study, [25] carried out empirical evaluation on 4 Java software systems, open source from diverse domains alongside 11 MMI cohesion metrics. The goal was to explore the association between LSCC, different cohesion metrics and class's faults. The results obtained indicates that LSCC is a better measure to guide Software Engineers in the refactoring of OO classes. In a nutshell, the indication is that cohesion metrics such as LSCC can better explain the quality of OO classes more precisely in terms of fault proneness. However, the limitation is that, LSCC is not able to differentiate class attributes and methods of diverse access level modifiers.

Structural and Semantic Information-based Approach: Approaches based on both structural and semantic information are as follows:

"Automating extract class refactoring: an improved method and its evaluation": Bavota et al [10] approach analyzes a class that need refactorings to detect groups of methods which are considered strongly and closely related for the creation of new class. The new class is expected to have higher cohesion and small increase in coupling value than the original class. For such class, the approach automatically apply extract class refactoring to fix the identified bad smell by suggesting appropriate way to split the initial class, while also finding appropriate classes number that can be created. To identify where refactoring is needed, the technique employed a two-steps clustering algorithm that depicts graphical representation. It starts by parsing the class to be refactored to construct a class *method-by-method matrix* to identify pair of methods likely to be in same class. Moreover, the information obtained at that stage is

used to identify the chain of methods after removing irrelevant structural or semantic associations between them from the graph based on certain threshold. Lastly, the identified chain are merged using a threshold called *minLength* to identify extracted classes considered to be trivial. To assess its performances, the approach was empirically evaluated using 5 Java open source software systems. In the first instance, it was artificially assessed using 50 Masters Students to evaluate the impact of the refactoring operations on OO software systems' cohesion and coupling. Secondly, 15 Masters Students refactored 11 classes that have been previously extract class refactored by the real developers in different versions of the systems. The results obtained showed the approach strongly increases the cohesion and slightly increases the coupling of the refactored class. Consequently, it can assist software engineers to carry out extract class refactoring efficiently.

"Using structural and semantic measures to improve software modularization": Bavota et al [19] approach used structural and semantic measures to re-modularize or split software package having low cohesion into software packages that is considered smaller and more cohesive. This approach shared the same identification technique with [10]. However, two class-level coupling metrics were utilized to achieve package re-modularization. The metrics are Information-Flow-based Coupling (ICP) and the Conceptual Coupling Between Classes (CCBC) to measure package cohesion which captured the classes' structural and semantic relationships respectively. These metrics were used to analyze the cohesion of software packages taking dependences and classes' responsibilities in the package into consideration. The information elicited are automatically use to define which classes should belong in a package as well as recommends how to divide the packages. Specifically, just like in [10], it takes a package to be refactored and built a *class-by-class matrix* in order to find class groups that are strongly related to form a package using certain threshold. In this case, if the extracted chains is one, no suggestion is made, otherwise, new packages are suggested having higher cohesion value than the initial one. Finally, the *minLength* threshold is used to identify trivial chains, calculate the coupling between the chains that are trivial and not as well as merging. The approach was empirically evaluated by conducting a case study using 5 open source Java systems and 4 students' software systems. The results indicated improvement of package decomposition with minimal increase in coupling as well as commendable re-modularizations solutions.

"Improving software modularization via automated analysis of latent topics and dependencies": Bavota, et al [12] approach is based on automation to enhance software package modularization via move class refactoring when taking the structure and the content of packages into consideration. It operates by employing source codes' latent topics analysis as well as structural dependences to suggest move class refactoring using Relational Topic Models (RTM) [12]. The topics removed from packages and classes identifies the refactoring activities for classes to be moved to a more suitable package alongside some justification for it. In addition, a tool called R3 was developed that automated the whole process. R3 modelled the analysis using structural and semantic information which helps in exploring the software package quality from both a conceptual and structural perspective [12]. In R3, semantic information are first extracted from classes and placed in a matrix called term-by-document utilized by the RTM to obtain the semantic associations between classes as well as expressing their topic distribution model. Static analysis is then used to obtain class dependencies and package composition using matrix called structural coupling and package decomposition respectively. The structural coupling matrix furnished the RTM with class dependencies information while the package decomposition matrix is used to consider developers' design decisions to ensure fine-grained re-modularization. With these original code design information, a suggestion for move class refactoring operation can be issued by the RTM technique as long as the quality of the original design is improved. To facilitate decision, R3 offers an assessment and reasons for the suggested refactorings in the form of a confidence level and qualitative data. To assess the performances of R3, two empirical studies were conducted on 9 software systems. The results showed about 30% reduction in coupling and more than 70% meaningful recommendations.

"Identifying method friendships to remove the feature envy bad smell (NIER track)": Oliveto et al [11] approach is called the MethodBook, specifically design to identify feature envy code smell and automatically applies move method refactoring through RTM method used in [12] to fix it. This approach employed the method utilized in Facebook where users' profiles are analyzed to recommend new friends or groups. In the perspective of OO software, MethodBook uses OO methods and classes to recommend the movement of a particular method to a class if the class host majority of friends of the method. It operates by first identifying methods friendships and then the envy class. MethodBook analyze the structural and conceptual associations between methods and employed RTM to determine groups of methods with many shared responsibilities that constitute friendship. The intuition is that, methods that share several responsibilities should be in the same class. Thus, if it found that friends of a method, M that reside in a class A_m belongs to another class B_f, then M is more related to methods in B_f than A_m in terms of responsibilities sharing. In that case, feature envy code smell is present and MethodBook fix this defect automatically by recommending refactoring operations of move method where M is relocated to B_f where large number of its friends resides. To assess the performance of MethodBook, Oliveto et al [11] performed a preliminary empirical evaluation on software system known as ArgoUML version 0.16, an open source. The results obtained showed significant refactoring solutions that assist software engineers avoid feature envy

Copyright © 2017 MECS

code smell during development and maintenance.

"Methodbook: recommending move method refactorings via relational topic models": Bavota et al [21] proposed a novel method to specifically detect bad smell in code feature envy and automatically fix them with move method refactoring. Their approach is also called the Methodbook and is based on the approach of R3 - RTM discussed in [11] and [12]. By employing RTM, Methodbook can identify class having the highest number of method friendships which is used to recommend refactoring operations of move method. It utilizes source codes' conceptual and structural information to identify feature envy instances using the associations between methods. With the Methodbook, the information collected from the code are placed in a term-by-document matrix and utilized RTM to captured the semantic relationships between methods and express a topic distribution model among them. In addition, the structural dependencies (interaction matrix and shared-data matrix) and the initial design information (original design matrix) are derived using static analysis by the Methodbook. These structural matrices and the information they hold (i.e. interaction among the methods and the design decisions made by the developers) are utilized by the RTM to recommend move method refactoring. However, the suggestion is only accepted if there is clear indication of design quality improvement, otherwise it is not accepted. To assess the performances, Methodbook was empirically evaluated in two case studies using 6 software systems in the first study and on 80 developers in the second study. The goals were assessing if design quality was improved by Methodbook as well as their refactoring recommendations. The results obtained showed precise and significant move method refactoring recommendations by Methodbook.

"Playing with refactoring: identifying extract class opportunities through game theory": Bavota et al [20] approach employs the theory of game to identify the need for extract class refactoring in OO source code. It was modeled as a non-cooperative game involving two players, with each player having the duty of creating a different class from the methods founds in the original class under refactoring. The assumption is that, the initial class has to be decomposed into two or more classes with each being more cohesive and less coupling than the initial class. Once the splitting is done, it is the software engineers' responsibility to analyze and ensure that cohesion and coupling are satisfied, otherwise, the approach is reapplied on the newly created classes. The approach starts with playing with methods where each player selects methods to extract, taking impacts on class cohesion and coupling into account. It then ends with payoff matrix computation using measures of Structural Similarity between Methods (SSM), Call-based Interaction between Methods (CIM) and Conceptual Similarity between Methods (CSM) to capture the structural and semantic of the classes. To assess the performances, Bavota et al [20] empirically evaluated the study using 2 Java software systems which is open source and the results indicated the usefulness and benefits of game theory.

"JDeodorant: identification and removal of feature envy bad smells": Tsantalis and Chatzigeorgiou [27] proposed an approach to identify opportunities for refactorings using a tool called JDeodorant. The tool was developed as eclipse plug-in designed to automatically detect feature envy in Java software systems, rank and fix them with move method refactoring. JDeodorant operates by first analyzing the relationships between source codes and modifies it based on the user's operation to detect bad smell known as feature envy bad smell. The code bad smell is detected if the distance of a method in a class from other classes is less than the distance from its own class. It then ranks the appropriate refactorings based on its effects on the design and lastly, applies move method refactoring as the appropriate refactoring. Moreover, to apply move methods refactoring, it first defines important set of preconditions that determine if the refactoring solution can preserve the design and its behavior and secondly, outlines an entity placement metric to assess a possible refactoring recommendation quality. An empirical studies was performed on two systems: Video store and LAN-simulation. The results obtained shows that, for the Video Store, JDeodorant was able to identify six out of six cases of feature envy code smells while seven out of eight of such cases were identified for LAN-simulation system.

"Aries: refactoring support environment based on code clone analysis": Higo et al [28] approach was designed to specifically detect the existence of code clone. To detect clone in software system, Higo et al [28] used a previous developed tool called CCFinder with processes involving lexical analysis, transformation, match detection and formatting. Moreover, to identify which fragment in the code clone that needs to be refactor, CCFinder searches for cohesive code fragment and invoke a method called CCShaper to detect clone pairs, provide syntax information and then extract code clone structural blocks based on the clone pairs and structural blocks information of location. Using this information, it then suggests appropriate refactorings such as pull up method or extract methods to get rid of the clone in the code. The process was facilitated using three metrics which are the Number of Referred Variables (NRV), Number of Assigned Variables (NAV) and the Dispersion of Class Hierarchy (DCH) [28]. In addition, the study developed a support tool in Java, called Arise. Arise has the capability to automatically detect code clone, characterize them using those metrics and finally suggest which code clones to remove alongside how to remove them. The usefulness of Aries was empirically evaluated on a system called Ant 1.6.0 and the results shows it can support software maintenance more effectively.

"A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system": Higo et al [29] proposed another technique to identify where refactoring is required in OO code based on metrics set that recommend how to refactor source code clones. It operates just the same way as the one in [28] in the detection of clone codes.

Copyright © 2017 MECS

However, a merging technique was introduce that runs through detection, extraction and measurement phases. The measurement phase utilizes the measures of DCH, NRV and NAV of [28] to recommend appropriate refactorings to apply. With the metrics, the approach can identify needs for possible refactorings such as super class, extract class, extract method, form template method, pull up method, move method, pull up constructor and parameterize Method. The code clones merging approach was also implemented in Aries which offers metrics which indicates certain refactoring operations but do not suggest the refactoring operations. It was also empirically evaluated using Ant 1.6.0 and the results obtained shows the technique can efficiently merge code clone.

"A new design defects classification: marrying detection and correction": Mahouachi et al [13] proposed a new approach to classify code defects using possible corrections. Unlike other approaches where defect detection and correction steps were addressed separately, this approach introduce a new classification rules that combines code defects identification and correction steps generated using genetic programming. With the combination, each defect group has almost the same refactoring operations that can be applied to fix it. Mahouachi et al [13] claimed this approach can be used to identify all types of refactorings operations. The approach inputs are set of code smells with the appropriate refactorings to correct them and takes set of quality metrics as well as a comprehensive list of refactoring operations to produce output in the form of rule set. The resulting rule is generated by combining metrics, cut-off threshold and the appropriate refactorings in a given rule expression. To create rules, the threshold value are assigned arbitrarily to each metric and refactoring operation which are joined within logical expressions with OR and AND. In this case, rules set that best identify defects set and the corresponding refactoring operations stands the solution to the defect identification and correction problem. Mahouachi et al [13] performed an empirical studies on 6 open-source Java systems and the results showed a high precision and recall correction scores on the different systems used.

"Recommending move method refactorings using dependency sets": Sales et al [30] proposed approach specifically detect the need for move method refactoring operations using methods' static dependencies set. The suggestion to move a method is given by computing the similarities between dependencies of source method and target methods. That is, the approach operates by identifying methods residing in an inappropriate classes and then recommends their movements to more appropriate classes. The approach utilizes methods' static dependencies set, M_A in a class A, the source class and M_B in another class B, the target class by computing their similarities. To this end, two similarity coefficients are computed such as dependencies similarity between M_A and other methods in A as well as between M_A and other methods, M_B in B. If the similarity between M_A and M_B is more than M_A and other methods in A, it thus indicate that, M_A has to be moved to class B. furthermore, the study implemented a tool called JMove. To evaluate its performances, an empirical studies was carried out using Qualitas Corpus version [30]. The results achieved showed high average precision and recall scores which are comparatively better than results from JDeodorant in terms of move method refactorings recommendations.

"Identifying Fragments to be extracted from long methods": Yang et al [31] approach was designed to automatically detect the presences of long method bad smells in a class and recommends extract method refactoring operations to fix it. The study proposed a prototype tool called AutoMeD to assist software engineers to get rid of long methods during software development and maintenance. The approach takes a class as input then detects long methods in the class, breaks the long methods into fragments according to their structures, compose the fragments to form larger compound fragments and gather variables from each fragment, move variable declarations to ease coupling among different fragments while preserving its behaviors. In addition, it then calculate the value of coupling among fragments and lastly, sort the fragments and suggest candidate fragments for the refactoring operations. The recommendation for extract method is highly based on coupling and other information of the fragments. To assess if AutoMeD helps to reduce refactoring cost or improve software quality, the study conducted an empirical studies on an open source project known as ThoutReader. The results obtained showed approximately 40% reduction in long methods refactoring cost.

"Predicting classes in need of refactoring: an application of static metrics": Zhao and Hayes [22] proposed a novel approach to predict classes that requires refactoring operations. The approach utilized complexity and size metrics on the classes under consideration by computing the weighted average metric values to rank the classes. In this case, classes with high ranks are given a high priority for refactoring without suggesting appropriate refactoring operations. This however, poses a limitation as the approach can't differentiate between different refactoring operations. The study implemented a maintainability decision support tool with components such as code repository analyzer (that parses OO code, find out their structural features, and gather metrics), maintainability prediction component, and refactoring planning component and is written in both C and Java. The study conducted an empirical studies to compare the predictions made by the tool and that of Java programmers. The results obtained showed that the refactoring decision support tool can better help software engineers that manual operations.

B. RQ2: Of What Importance is Software Metrics during Refactorings Opportunities Identification?

The primary goal of SE is to develop high quality software product. However, developing software products having structural characteristics of high coupling and low cohesion is an indication of poor quality [3]. This in turn, signifies bad smells in code or design which has to be removed.

Table 2. Refactoring Opportunities Identification Summary

Ref.	Refactoring Activity	PL	System Type	Tool	Refactoring Method	Empirical Evaluation
[18]	Extract Subclass	Java	OS	-	FA	Yes
[25]	Move Method, Extract Class	Java	OS	-	FA	Yes
[10]	Extract Class,	Java	OS	-	FA	Yes
[19]	Move Class	Java	OS,SP	-	FA	Yes
[12]	Move Class	Java	OS,SP,CP	R3	FA	Yes
[21]	Move Method	Java	OS,SP,CP	-	FA	Yes
[20]	Extract Class	Java	OS	-	FA	Yes
[27]	Move Method	Java	AP	JDeodorant	FA	Yes
[28]	Extract Method, Pull Up Methods	Java	OS	Arise	FA	Yes
[29]	Move Method, Extract Method, Form Template Method, Pull Up Constructor, Pull Up Method, Parameterize Method, Extract Class, Extract Superclass	Java	OS	Arise	FA	Yes
[13]	Any RA	Java	OS	-	FA	Yes
[11]	Move Method	Java	OS	-	FA	Yes
[30]	Move Method	Java	OS	JMove	FA	Yes
[31]	Extract Method	Java	OS	AutoMeD	FA	Yes
[22]	Extract Class	Java	SP	-	SA	Yes
[26]	Extract Superclass, Push Members Down, Extract Interface	Java	CP	-	SA	Yes

*PL=programming language, *OS=open source, *SP=student project, *CP=commercial project, *FA=full automation, *SA=semi-automation
*Ref=reference.

However, in order detect the presence of these smells, software metrics are vital. Software metrics offers the tool to assess, monitor, control and take useful decisions aimed at improving the quality of the software [32][34]. Existing software metrics are broadly classified into traditional metrics and OO metrics [34]. In particular, OO product metrics captures different structural features of OO software systems such as class complexity, coupling and cohesion [7][15]. Today, several OO metrics exist and empirically validated in the assessment of OO design and codes quality [32][33]. In the 16 studies analyzed in this paper, we have seen the impact of software metrics in the assessment and improvement of software quality. They played a critical role as they offer developers the opportunities to pinpoint problematic bad smells in code and decide on whether to apply refactorings or not. In particular, software measures relied on in the refactoring opportunities identification were mostly traditional metrics such as complexity, size, etc. and the OO metrics such as cohesion and coupling measures. In the perspective of OO measures, cohesion refers the degree of relatedness of members found in the class while coupling is the degree of interdependencies of a class and other classes [11][21][25]. These OO measures and size such as software lines of code (SLOC) were mostly used in the identification of bad smells in code. They actually measured the degree of difficulty faced by developers in performing maintenance tasks on the system. Studies that used these measures are [11][12][18][19][21][25][27][28][31] and their refactoring goal was to achieve high cohesion and low coupling.

C. RQ3: Do Traditional Metrics Played Same Role as OO Design Metrics in the Identification of Refactorings Opportunities in OO Source Code?

In the 16 studies considered, we found that software measures played a critical role in the identification of opportunities for refactoring. Several studies employed OO design measures, some traditional measure while some employed a combination of both metrics. However, in the improvement of OO software quality, OO metrics are specifically used since the traditional metrics are insufficient in capturing the structural attributes of OO software [32][34]. Thus, in terms of the role played by both, it is interesting to know that OO metrics played more role than the traditional counterpart. Though both metrics were crucial in the identification of refactoring opportunities (bad smells) in source code, only OO metrics assisted in the decision on which suitable refactorings to apply. For instance, in all the studies that utilized OO metrics or combination of OO and traditional metrics, the studies were able to identify where refactoring is needed in code and automatically suggest appropriate refactorings operations [10][11][12][18][19][20][21][25][27][28][29]. However, the studies that used only traditional measures such as size and complexity metrics only identified such opportunities without suggesting the appropriate refactoring operations to apply [22][26]. In Zhao and Hayes [22] and Kosker et al [26] approaches, once a class is identified as a refactoring candidate, it is the responsibility of the engineers to manually analyze the source code in order to decide on which refactorings

Copyright © 2017 MECS

operations to apply.

V. Discussion

Refactoring is an indisputably a technique that is employed to gradually improve the internal complexity of OO software systems to prepare them for further enhancements as the software aged. However, the process is not an easy task as code bad smells must be detected first, decide on appropriate refactorings to apply as well as to check if quality is improved or not. To identify segments in the source code that need refactorings, several approaches has been proposed and developed. In this study, we analyzed 16 studies that based their approaches on quality metrics. The summary of the analysis is shown in Table 2. In the studies considered for analysis, we found out that most refactoring approaches were designed to automatically identify refactoring opportunities and recommends appropriate refactorings to fix them, while few were based on semi-automation for the identification without suggesting appropriate refactorings [22][26]. In addition, some of the approaches can only detect needs for a single refactoring or couple of refactorings activities. (See Table 2). As shown in Table 2, move method was the highest refactoring operations' need identified by the different approaches followed by extract class, extract method and so on.

Furthermore, to detect where refactoring are required, the approaches employed source code structural and semantic information by utilizing software metrics. Both OO design metrics and traditional metrics were employed by software engineers in the identification of refactoring opportunities in all the studies considered. However, we found that both metrics did not played the same role. While both metrics helped in the identification of refactoring opportunities, only OO metrics actually aided in the decision of which appropriate refactorings should be applied. The goal of refactorings approaches where all based on achieving high cohesion and low coupling as well eliminating clone in code. In addition, all the studies empirically evaluated their proposed method and some went further to implement tools to assist software engineers in terms of refactorings during software maintenance. The results obtained from the evaluation indicates the approach is more effective in the detection of refactorings opportunities. Moreover, all the studies assessed the performances of their approach using software systems developed mostly in Java programming language and dominated by open source software. No other software systems developed in other programming was used in the evaluation.

VI. Conclusions

In the light of high maintenance cost due to increased software complexity and deteriorated quality, refactoring offers an approach to improve the maintainability, understandability, or other elements while keeping the external code behavior of the software system intact. However, the software system has to be analyzed to identify needs for refactorings before it is applied. Several approaches exist today in that capacity. Thus, this paper has analyzed several studies, 16 in number on refactorings opportunities identification. The summary of the findings is shown in Table 2. In particular, the study found that several approaches exist, though designed for either a single or multiple refactorings operations. Move method and extract class where the most refactorings operations the approaches were designed to cater for. Moreover, software metrics played a great role in bad smells detections and refactoring decisions. The ultimate goal of refactoring is geared towards achieving high quality software system via high cohesion and low coupling. In addition, the approaches have been empirically evaluated and their performance appears promising for software maintenance.

Based on the findings of this paper, the following recommendations are important for further research:

a) In order to further establish confidences on the performances of the refactorings opportunities identification approaches, more empirical evaluations should be carried out on applications developed on other OO programming languages, other than Java and open source applications.
b) A generic refactoring tool should be developed that is able to identify opportunities for all refactorings operations. This is important because the existing approaches only identify needs for one or multiple but not all refactorings operations.

In general, due to the mode of operations involve in identifying refactorings opportunities, we suggest that developers during software development should consistently evaluate the quality of their software using appropriate software measures. This is important to ensure that the software complexity is reduced which in turn reduces the high cost of software maintenance.

References

[1] Singh, Y. Kaur, A. and Malhotra, R. Empirical validation of object-oriented metrics for predicting fault proneness models. Software Quality Journal, vol.18 pp. 3–35, 2010
[2] Ruchika, Malhotra, Nakul Pritam and Yogesh Singh: On the Applicability of Evolutionary Computation for Software Defect Prediction. International Conference on Advances in Computing, Communications and Informatics (ICACCI, 2014). Pp. 2249 – 2257, 2014
[3] Marcus, A. Poshyvanyk, D., Ferenc, R. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. TSE, 34(2):287–300, 2008.
[4] Liu Y, Poshyvanyk D., Ferenc R., Gyimóthy T., Chrisochoides N. Modelling class cohesion as mixtures of latent topics. In: Proceedings of the 25th IEEE international conference on software maintenance. IEEE Press, Edmonton, pp 233–242, 2009
[5] Sommerville, I. Software Engineering. Nineth ed., Addison-Wesley, 2011.
[6] Bohner, S. A.: "Extending software change impact analysis into COTS components" Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop,

Greenbelt, USA, pp.175 -182, 2000
[7] Lehman MM, Ramil JF. Rules and tools for software evolution planning and management. Ann Softw Eng 11:15–44, 2001
[8] Mens T, Tourwé T. A survey of software refactoring. IEEE Transaction on Software Engineering, 30(2): 126–139, 2004.
[9] Olbrich S, Cruzes DS, Basili, V, Zazworka N. The evolution and impact of code smells: a case study of two open source systems. In: Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement, ESEM '09, pp 390–400, 2009.
[10] Bavota, G. De Lucia, A. Marcus, A. Oliveto, R. Automating extract class refactoring: an improved method and its evaluation, Empir. Softw. Eng. (2013) 1–48.
[11] R. Oliveto, R. Gethers, M., Bavota, G. Poshyvanyk, D. De Lucia, A. Identifying method friendships to remove the feature envy bad smell (NIER track), In: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 820–823
[12] Bavota, G. Gethers, M. Oliveto, R. Poshyvanyk, D. De Lucia, A. Improving software modularization via automated analysis of latent topics and dependencies, ACM Transact. Softw. Eng. Methodol. 23 (1) (2014).
[13] Mahouachi, R. Kessentini, M. Ghedira, K. A new design defects classification: marrying detection and correction, in: Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering, 2012, pp. 455–470
[14] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA, 1999
[15] W. Pan, B. Li, Y. Ma, J. Liu, Y. Qin, Class structure refactoring of object-oriented softwares using community detection in dependency networks, Frontiers Comput. Sci. China 3 (3)396–404, 2009.
[16] Alshayeb, M. Empirical investigation of refactoring effect on software quality. Information and Software Technology 51 (2009) 1319–1326
[17] Al Dallal, J. Identifying refactoring opportunities in object-oriented code: A systematic literature review. Information and Software Technology 58, 231–249, 2015
[18] Al Dallal, J. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics, J. Inform. Softw. Technol. Arch. 54 (10) pp.1125–1141, 2012.
[19] Bavota, G. De Lucia, A. Marcus, A. Oliveto, R. Using structural and semantic measures to improve software modularization, Empir. Softw. Eng. 18 (5) (2013) 901–932.
[20] Bavota, G. Oliveto, R. De Lucia, A. Antoniol, G. Gueheneuc, Y. Playing with refactoring: identifying extract class opportunities through game theory, in: IEEE International Conference on Software Maintenance (ICSM), 2010, pp. 1–5.
[21] Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia, Methodbook: recommending move method refactorings via relational topic models, IEEE Trans. Software Eng. (2013)
[22] Zhao, L. Hayes, J. Predicting classes in need of refactoring: an application of static metrics, in: Proceedings of the 2nd International PROMISE Workshop, Philadelphia, Pennsylvania USA, 2006
[23] Tokuda, L., Batory, D. Evolving object-oriented designs with refactorings. Automated Software Engineering 8, 89–120, 2001.
[24] H. Liu, Q. Liu, Y. Liu, and Z. Wang. Identifying renaming opportunities by expanding conducted rename refactorings. IEEE Transactions on Software Engineering, (99):1–1, 2015.
[25] Al Dallal, J., Briand, L.C. A Precise Method-Method Interaction-Based Cohesion Metric for Object-Oriented Classes, ACM Transact. Softw. Eng. Methodol. (TOSEM) TOSEM 21 (2) (2012). Article No. 8.
[26] Kosker, Y. Turhan, B. Bener, A. An expert system for determining candidate software classes for refactoring, Expert Systems with Applications 36 (6) (2009) 10000–10003
[27] Fokaefs, M. Tsantalis, N. Stroulia, E. Chatzigeorgiou, A. JDeodorant: Identification and removal of Feature Envy bad smells, in: Proceedings of IEEE International Conference on Software Maintenance, 2007, pp. 467–468.
[28] Higo, Y. Kamiya, T. Kusumoto, S. Inoue, K. Aries: Refactoring support environment based on code clone analysis, in: Proceedings of the 8th IASTED International Conference on Software Engineering and Applications, Article No. 436–084, 2004, pp. 222–229.
[29] Higo, Y. Kusumoto, S. Inoue, K. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system, J. Software Maintenance Evolution.: Res. Practice 20 (6) (2008) 435–461
[30] Sales, V. Terra, R. Miranda, L.F., Valente, M.T. Recommending move method refactorings using dependency sets, in: IEEE 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 232–241.
[31] Yang, L. Liu, H. Niu, Z. Identifying Fragments to be Extracted from Long Methods, in: Proceedings of the 16th Asia-Pacific Software Engineering Conference, 2009, pp. 43–49.
[32] Chidamber, S.R., Kemerer, C.F.A metrics suite for object oriented design. IEEETransactions on Software Engineering 20, 476–493, 1994
[33] Isong, B.E. and Ekabua, O.O. "A Systematic Review of the Empirical Validation of Object-oriented Metrics towards Fault-proneness Prediction". International Journal of Software Engineering and Knowledge Engineering (IJSEKE) WSPC. Vol. 23, No. 10. pp. 1513–1540, 2013.
[34] Fenton, N., Neil, M. Software metrics: successes, failures, and new directions. Journal of Systems and Software vol. 47, pp. 149-157, 1999
[35] Zhang, M., Hall, T., Baddoo, N. Code Bad Smells: a review of current knowledge, J. Software Maintenance Evolut.: Res. Practice 23 (3) (2011) 179–202.
[36] Wangberg, R.D. A literature review on code smells and refactoring, Master Thesis, Department of Informatics, University of Oslo, 2010.
[37] Misbhauddin, M. Alshayeb, M. UML model refactoring: a systematic literature review, Empir. Softw. Eng. (2013) 1–46.

Authors' Profiles

Isong Bassey received B.Sc. degree in Computer Science from the University of Calabar, Nigeria in 2004 and M.Sc. degrees in Computer Science and Software Engineering from Blekinge Institute of Technology, Sweden in 2008 and 2010 respectively. Moreover, he received a PhD in Computer Science in the North-West University, Mafikeng Campus, South Africa in 2014. Currently, he is a Lecturer in the

Department of Computer Sciences and a Faculty member of FAST Mafikeng Campus, North-West University. He is also a member of IEEE, IEEE Computer, Communication and Education Societies. His research interests include Software Engineering, Requirements Engineering, Software Maintenance, Cybersecurity, Software-Defined Networks, Cloud and Mobile Computing, ICT4D and Computer Science Education.

Nosipho Dladlu obtained her B.Sc. (Hons) and M.Sc. degrees in Computer Science from the North-West University, Mafikeng, South Africa in 2011 and 2014 respectively. Currently, she is a Lecturer in the Department of Computer Sciences and a Faculty member of FAST, North-West University, Mafikeng Campus. Her research interests include: Cloud Computing, Mobile Computing, Networks and HCI.

Bassey Ele (MCPN) obtained a Bachelor of Science degree (B.Sc.) in Computer Science from University of Calabar, Nigeria in 2001. Moreover, he obtained M.Sc. and Ph.D. in Computer Science from Ebonyi State University, Nigeria in 2010 and 2015 respectively. He is currently a Lecturer in the Department of Computer Science, University of Calabar, Nigeria. Also, he is a member of the Nigerian Computer Science (NCS) and Computer Professional Registration Council of Nigeria (CPN). His research interests include Expert Systems, Network Security and Cybersecurity and Software Engineering.

Etim Duke obtained his B.Sc. degree in Computer Science from the University of Calabar, Nigeria in 1999 and M.Sc. degree in Computer Science from the North-West University, Mafikeng, South Africa in 2016. He is currently a research student in the Department of Computer Science at the North-West University. His research interests include: Software engineering, Cybersecurity, Cloud Computing, and ICT4D.

How to cite this paper: Bassey Isong, Nosipho Dladlu, Etim Duke, Bassey Ele,"Analysis of Metric-Based Object-Oriented Code Refactoring Opportunities Identification Approaches", International Journal of Information Technology and Computer Science(IJITCS), Vol.9, No.1, pp.46-57, 2017. DOI: 10.5815/ijitcs.2017.01.06