International Journal of Applied Engineering Research ISSN 0973-4562 Volume 10, Number 18 (2015) pp 39696-39707

A Systematic Literature Review: Recent Trends and Open Issues in Software Refactoring

Sreeji K. S

Research Scholar, Department of Computer Science and Engineering, SRM University, SRM Nagar, Kattankalathur, Kanchipuram District-Tamilnadu. Sreejiks71@gmail.com

Dr. C. Lakshmi,

Head of the Department, Department of Software Engineering SRM University, SRM Nagar, Kattankalathur, Kanchipuram District-Tamilnadu. lakshmi.c@ktr.srmuniv.ac.in

Abstract
Software refactoring is a process of improving the internal structure of software artifacts through various steps of transformations without affecting the externally observed behavior. Refactoring aims to improve the quality of the software in several aspects like code understandability, maintainability and modularity. Extensive researches are taking place in this area for the last decade and several papers are available for review in various angles of software like code smell detection, refactoring algorithms, patterns and refactoring, program evolution and refactoring and code clone detection. The aim of this review paper is to structure and organize the major findings published since 2004 with more emphasis given to papers published for the last five years to understand the current trends in refactoring and also to formulate better research problems for further research.

Keywords: AntiPatterns, Code smells, Code clones, Metaprogramming, Software Refactoring, Web mashups.

Introduction

Software tend to evolve over time according to the changing user requirements, as a result the code becomes more and more complex [1][2] and deviates much from the original design. This may lead to poor quality software. Also, software evolution is time consuming, complex and incurs much of the software development cost. So much attention must be given to the maintenance of software which necessitates the development of flexible and maintainable tools to deal with reduction of software complexity. The area of software engineering that focuses and deals with this problem in software development process is referred to as software restructuring or in case of object oriented systems, refactoring. According to Arnold [3] software restructuring can be defined as the modification of software to make the software (1) easier to understand and to change or (2) less susceptible to error when future changes are made." The term refactoring was first discussed by William Opydke [6] in the year 1999 in his Ph. D dissertation as a variation of object oriented restructuring. According to Martin Fowler [4] "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet it improves its internal structure". The refactored code exhibits improvement in internal quality attributes such as understandability and maintainability and reusability. Thus the ultimate aim of software refactoring is to transform the design of the program in to better quality by resolving antipatterns, code smells, code clones and other anomalies. For the past fifteen years, researchers contributed a great deal of knowledge and emerging ideas in the area of software refactoring. Their valuable findings encompasses various activities of the software development process such as requirement analysis and modeling, design integration, testing and maintenance. The various research findings emphasized that though software refactoring brings much reduction in software complexity and maintenance, it is necessary to have a thorough understanding, awareness and skills to use of available tools to fully utilize the benefits of refactoring process during redesign of the software.

Background of Software Refactoring

The concept of software refactoring was first identified by William F. Opydke in his Ph. D. Dissertation, but it became more and more relevant after the publication of the book [4] Refactoring: Improving the design of Existing Code, written by Martin Fowler in the year 1999. The process of refactoring is used to restructure the software by applying a series of stepwise transformations without changing its observable behavior. The basic principle of object oriented refactoring is in reorganizing classes and methods along the class hierarchy such that future adaptations can be easily made resulting in more readable and maintainable code of good quality.

It is desirable to identify the parts of the source code that exhibits signs of 'bad smells'. This challenging task is the vital part of the refactoring process. According to Beck [5], 'Code smells are structures in the code that suggest the possibility of software refactoring". Fowler and Beck, also other researchers provide a list of bad smells and the corresponding refactoring strategies to make the refactoring process much easier.

Refactoring can be applied in various software development artifacts such as software design patterns, architectures, models, test suites etc.

i. The Activities Of Refactoring Process Are [8]:

1. Apply unit testing to the program

2. Identify where the software should be refactored by identifying code smells among the code
3. Select a refactoring strategy to remove the identified code smell.
4. Apply the strategy
5. Apply regression testing to the refactored code
6. Assess the effect of the refactoring on software quality characteristics such as complexity, effort, maintainability and readability.
7. Maintain the consistency between the program and other software artifacts.

ii. Benefits of Software Refactoring

There are two general categories of benefits to the activity of refactoring**:**

1. Maintainability. By applying refactoring strategies such as moving a method to appropriate class or removing unnecessary comments on the source code makes it more readable and understandable.

2. Extensibility. By applying good design patterns to the overall architecture of the software system it is easier to extend and adds more flexibility.

Software refactoring is a very useful and valuable technique but it is not a panacea to all the observable problems in software development. Software refactoring techniques can be classified based on how they make changes to the code and is shown in the following table.

Table I. Refactoring Techniques

Refactoring Technique	Description
Composing Methods	*Techniques that allow for breaking code apart into pieces e.g. Extract Method*
Moving Features between Objects	*Techniques that allow for improving location of code e.g. Move method*
Organizing data	*These techniques allow for working with data easier e.g. Self encapsulate field*
Those that allows for more abstraction	*e.g. replace conditional with polymorphism, Generalize type*
Making method calls simple	*These techniques allow making interfaces more straightforward and understandable. e.g. Rename method*
Move upwards in the hierarchy	*These techniques allows to pull up fields around a class hierarchy e.g. Pull up field*
Higher level refactoring	*This group consists of complex techniques that turns a procedural code into object oriented code. e.g. Extract Class Hierarchy.*

A Brief Of Related Work

Tom Mens (2004) conducted a detailed study of existing refactoring research such as the refactoring activities, formalisms, types of artifacts and effects of refactoring on software process using the concepts of software restructuring. Karim O. elish and M. Alshayeb [53] proposed a classification of refactoring methods based on their impact on testability attribute.

M. Kim et.al [14] conducted a field study of windows version history and reported many challenges to refactoring during cross branch integration from different teams. Also the paper identified the need for a better code understanding tool and validation tool that checks correctness of refactoring. Mefsin Abebe and Cheol-Jung Yoo [8] conducted a systematic literature review approach to classify the refactoring research literature and presented the contribution and gaps in each of the relevant area. They used a tool support for taking an inclusion and exclusion decision.

This review paper is different from the above mentioned one; since the main purpose is to conduct a review manually which includes more electronic database and more literatures. Apart from [8] this review gave a separate section on code clone detection and antipatterns since more relevant open issues are found in recent literatures regarding this area.

Review Methodology

The main objective of the study is to find out the relevant literature regarding software refactoring, major contributions and identify the recent trends in this emerging field.

i. Review Protocol

The review protocol includes the following steps:

1. Set up the research questions
2. Identify and locate the relevant literature based on the inclusion and exclusion criteria.
3. Select the relevant studies based on the quality assessment such as whether the study have a clear problem statement, specific section on limitations and open issues for future work.
4. Data is extracted, combined and summarized.
5. Write a review report.

ii. A Quick Tour On The Review Protocol

Following are the questions formulated to identify, classify and summarize the findings of the collected literature:

RQ: What are the recent trends, major contributions and gaps in the area of software refactoring research?

SQ: What are the general recent trends in refactoring activities?

SQ: What are the gaps in each of the relevant contributions?

Inclusion and exclusion is done by selecting the literature that is relevant to the area of software refactoring according to the following criteria:

- Papers written by researchers and professional developers
- Included only international conference and workshops, journals, tutorial, technical reports, thesis, dissertations and newsletters.
- Literatures published since 2004 and written in English language are included and others are excluded
- The papers are retrieved manually mainly from electronic databases such as IEEE Explore, Springer Link, ACM Digital Library.

These data sources are searched against the following search terms.

- Refactoring
- Refactoring and API evolution
- Bad smell and refactoring
- Design pattern, antipattern and refactoring
- Software metrics and Refactoring
- Agile Development and Refactoring
- Code clones and refactoring

The selected literatures are reviewed for data extraction. The extracted data is recorded manually and arranged in a chronological order. Once collected, combine and summarize the findings from each paper depending on the title, abstract, methods used, study limitations, open issues, conclusion and future work.
Finally, the summarized findings are reported as identified gaps and open issues.

Significant Contributions Of The Study

The attractive feature of this study can be viewed from three angles, which can help the aspirant researchers in the area of refactoring.

- Grouping of software refactoring literatures based on their aim, title, and the content
- Identify the significant contributions in each of the study based on the group.
- Identify the gaps in each of the group based on the methodology used, applicability, and open issues for further research.

A Narrative Of The Relevant Literatures In The Area Of Software Refactoring Research

The analysis results of the 73 papers are summarized in terms of the methodology used, its applicability, limitations and open issues.

i. Survey of Software Refactoring

This group consists of some of the surveys conducted in the field of refactoring, various approaches, software evolution, comparison of manual and automated refactorings and tool support.
Tom Tourwe and Tom Mens [10] specified that by applying the techniques of logic Meta programming one could detect bad smells and identifies refactoring opportunities. Their experiments were done on the SOUL research prototype, a medium sized application. But this approach detects only two kinds of bad smells such as obsolete parameter and inappropriate interfaces.
Tom Mens [8] conducted an extensive overview of the area of software refactoring, the activities of software refactoring process, the area in which it is applied, assessing the impact of software quality due to refactoring, the techniques and formalisms for program correctness and preservation of semantics using the concepts of software restructuring such as program slicing, formal concept analysis, program refinement and dynamic program analysis. The study also concentrated on the automatic tool support and process support towards refactoring. The study indicates important open issues in each of above categories that are yet to be solved.
Zhenchang [11] conducted a detailed study on the structural evolution of an integrated IDE like Eclipse and a plugin based framework and concentrated on what fraction of code modifications are refactorings and what are the most frequent types of refactorings. If the most frequent types of refactorings are identified the scope of refactoring can be narrowed down. They observed many mismatches between the programmer choice and automated refactoring as awareness, naming etc.
The studies by Tapa [12] made use of structural and semantic information of the source code for refactoring process. The researchers also advocate the use of semantic information such as comments in the source code.
S. Nagara, N. Chen et.al [13] conducted a comprehensive study of manual and automated refactorings. Their approach implemented a refactoring inference algorithm that checks for continuous changes. The study focused on the analysis of two version control system snapshots, they concluded that many of the refactorings are clustered in time and is incomplete or do not reach different versions. Their studies have the following results: 1) an average experienced developer performs automated refactoring. A novice user is less familiar with refactoring tools. 2) On average manual refactorings takes longer times than automated ones. Extract method refactoring is most time consuming both manually and automatically. Rename field refactoring is fast. The algorithm makes use of a snapshot analysis. They conclude that their algorithm can be used to infer intelligently changes occurring on the source code continuously.
M. Kim, Thomas Zimmermann et.al [14] conducted a field study of windows version history and found that the binary modules refactored have significant reduction in the number of inter module dependencies, post release defects than other regular changed modules. The study pointed out the major challenges associated with refactoring as cross branch integration from different teams. The studies suggested that even though the developers know the type of most common refactorings they perform it manually. The developers are unaware of the existence of certain tools support. Developers wanted to have a better code understanding tool and validation tool that checks correctness of refactoring.
E. M. Hill, C. Parrin et. al. [15] in their paper explains that high level refactorings are those that change the signatures of classes, methods or fields refactorings at this level include rename class, move static field and add parameters. They found that refactoring tools are seldom used because of lack of awareness, opportunity and trust in them. Also due to the limitation of refactoring tools within the programming environment would bring much benefit to the developers. Their study was based on JAVA and Eclipse environment. Their findings revealed another interesting fact that many of the refactorings are medium and low level categories. A few tools address higher level refactorings that change the signatures of classes, methods or fields. That means 24 to 60% of refactorings is yet found to be detected and this will produce much benefit to improve the maintenance of the software.

Pinto [16] conducted a study to understand the views of programmer regarding refactoring tools; they wanted to have more unimplemented features in the tools for refactoring in the future.

ii. Software Refactoring Tools

In most of the refactoring tools, the refactorings are performed by applying certain preconditions and transform the code automatically and manually by programmers. To alleviate this limitation, researchers [Emerson Murp, 17] proposed tools based on synthesis from examples.

Michael Mortenson, Sudipto Ghosh et.al [18] have developed a tool suite support of refactoring to legacy systems based on the principles of test driven development. The aim is to ensure that adding new aspects using mock stub systems and testing using regression tests does not introduce new faults when aspects are into a large legacy system.

Yasemin et.al [19] proposed a machine learning based model to predict classes to be refactored. This method is able to detect 82% of the candidate classes to be refactored with little effort.

Erica Mealy and Paul Strooper [20] conducted a framework study on six java refactoring tools using feature analysis method. Their paper revealed that existing tool support towards refactoring does not cover all of the aspects of the refactoring process such as usability, reliability, efficiency etc. and found to be immature in detecting code smells. They are arguing that, an integrated approach of identifying usability requirements and automated code smell detection will help the developers to maintain accurate software.

Refactoring is considered as a preventive maintenance activity. There is lack of tools for supporting decisions like when and where to apply refactorings. L. Zhao and J H Hayes [21] introduced the tool, JRIA (JAVA refactoring Inspection Assistant) a rank based software-measure driven refactoring decision support approach to assist managers. The approach used static software measures like size oriented, coupling, and dependency to rank the classes and packages that needs refactoring. The results have shown that maintainability prediction of JRIA is faster than human reviewers but applicable only to limited size of code.

Wafa Basit and Fukhar Lodhi et.al [22] proposed an extended set of refactoring guidelines and developed a model for building specification of extended refactoring guidelines. The guidelines address the semantic issues as the client code and test code evolves.

M. Vakilian [23] proposed that certain general characteristics such as supplier assessment, economic issues, easy of introduction, reliability, maintainability and compatibility is to be considered in developing refactoring tools.

Max Schafer et.al [24] addressed the problem of naming and accessibility of variables with respect to their program scope during refactoring. While performing refactoring current refactoring engines pay poor attention to preserve the program behavior with respect to access control preservation and name binding. They proposed a tool that transform the original java program to a representation that is look up free and access control free. They applied two types of refactoring that is extracting interface and pull up method on a collection of real world applications and assessed the effect of naming and accessibility adjustments arises on real code and compared the performance of other tools. Major tools do not address these issues of naming and accessibility bindings in refactoring and hence rejected. But the tool fails to assess the control flow and data flow properties preservation during refactoring.

iii. Bad Smell and Refactoring

Ganesh B et.al [25] briefed the 22 code smell that fowler identified and their results revealed that duplicate code smell has more emphasis in research and message chain has attained little focus.

J Perez [26] shown that the move method, move field, rename method, rename field are low level refactorings that provide service to more complex refactoring techniques.

Schumacher [27] analyzed the relationship among different kinds of bad smell and their impact on the resolution order.

Serguei Roubtsov, Alexander Serebrenik et.al [28] developed a classification of the dependency injections using java annotations, the associated modularity principle violations, and their impact on the deployment of software systems and the resolution of code smells.

Huaxin [29] documented the collection of refactorings as problem templates that identify suspect code design and suggested target design patterns as solutions.

Hui Liu [30] found out that there exists some indirect relationship among most of the commonly used refactorings and proposed an approach to automatically detect and optimally resolve bad smells. They evaluated their approach using two nontrivial open source applications, and the results suggest that a significant reduction in refactoring effort ranging from 17.64 to 20 percent can be achieved.

Almar Hamid, Muhammed Ilyas et.al [31] have made a comparative study on two different code smell detection tools such as JDeodorant and Insect only for Java source code. Both of them have used different approaches to identify code smells. The studies have shown that there is lack of mature tools for code smell detection and refactoring.

Dag I. K et.al [32] made an attempt to find out the relationship between code smells and maintenance effort. They used multiple linear regression analysis on twelve code smells and concluded that none of these code smells contributed significant effect on maintenance effort. The authors experience have shown that more focus on code size and the work practices limit the number of changes and thereby reduces the change induced smells. But other than these twelve code smells may still cause problems to maintenance effort.

Kathryn T. stolee and Sebastain Elbaum [33] made an attempt to automatically identify and refactor the code smells in pipe like web mashups. They have identified the candidate smells of mashup environment such as laziness smells, redundancy smells, environmental smells and population based smells. To perform refactorings they have used the concepts of graph transformation. The main aim is to unify duplicated code to simplify pipe structures and reduce their sizes. They have reached at a conclusion that the refactorings they proposed can reduce the number of smelly mashups to a certain level. They conclude their study in such a way that the study can be extended to end user programming environments like spreadsheets and web macros.

Hui Liu et.al [34] proposed a monitor based instant refactoring framework which helps the developers to detect code smells at the early stages of the software development. The framework consists of a 1) monitor which oversees changes made source code 2) smell detectors and refactoring tools provides smells and suggestions for refactoring. 3) a smell view presents the detected smells to the developer 4) The feedback controller adjust the feedback from refactoring tools. The framework produces improved software quality but only 8 types of code smells is capable to detect and the performance is not thoroughly evaluated.

Jiang Dexun, Ma Peijum et.al [35] have identified a new bad smell functionally over related classes but confused inheritance in some object oriented programs. Presence of this kind of bad smell reduces the understandability, reusability and ultimately the maintainability of software system. Their approach performs an analysis to form the number of large cluster group of entities with dependency relationships. The threshold value is computed using the metric which determines refactoring suggestions to be performed. This preset threshold computation is the limitation of this approach. Their studies revealed that the cost of this type of refactoring is lower but there is chance for reduced coupling and improved cohesion among modules. There is greater degree of encapsulation and inheritance which supports reusability but increased complexity reduces understandability.

Denys Poshy Vanyk et.al [36], presented a method book based on relational topic models. The method book used this model to store the friends of a method. Methodbook captures textual information from the source code to represent the relationship between methods. The approach detects feature envy bad smells in the source code. Their approach is briefed as follows. The method friendships are identified using relational topic model (RTM) which is a hierarchical probabilistic model of document attributes and link between documents. Methodbook keeps track of all comments, all types of identifiers, and literal strings present in a method. A cut of list is prepared then a term by document matrix is computed. The static analysis of the software matrix yields structural similarity between the methods and call based dependence between methods. Once the RTM similarity matrix is computed, rank friendships among methods, then find out best friends and identify the envied class. The studies with six software systems suggest that methodbook provides accurate suggestions for move method refactoring.

M. Kesantine, H. Saharoui et.al [37] presented code smell detection as a distributed optimization problem. Their idea is based on genetic programming for the detection of rules at the first level that yields a set of population. A second evolutionary algorithm is executed in parallel that generates detectors from well designed codes. A set of candidate solutions are evolved that reaches to a good solution. This approach is named by them as parallel-evolutionary algorithm (P-EA). The authors conducted an empirical study of their approach to two single population based approaches and two code smell detection techniques such as DÉCOR and JDEODORANT, and the results shown that, P-EA is more efficient and accurate in terms of precision and recall in the detection of eight different types of code smells.

Santiago et.al [38] presented a semi automated approach called SPIRIT (Smart Identification of Refactoring Opportunities) to suggest ranking of code smells based on a combination of three criteria such as 1) Past component modifications 2) important modifiability scenarios for the system and 3) relevance of the kind of smell. Their approach has been evaluated in two JAVA applications and suggested code smells is indeed useful to the developers.

iv. Software Artifacts and Refactoring

Demeyer et.al [39] presented a rule based inconsistency resolution which is reusable across different model refactorings that manages the flow of inconsistency resolution.

Detects design defects early in the design process yield more benefit to designers belonging to MDE process. Mika V et.al [40] conducted an empirical study of drivers for software refactoring decisions using Java code developed among a group of students. The study has more implications in determining which stings for refactoring, method size is a driver, also suggest code problem indicators such as poor algorithm.

Richard Mateos [41] studied the refactoring of use case models based on the information obtained using episode model and suggested 10 refactoring rules including validation of the behavior preserving property towards use case refactoring.

Dobrza [42] proposed a systematic approach to specification of UML model refactorings and bad smells in models, which forms the sound basis for model driven architecture. They exemplified their approach in Telelogic TAU, a use case tool.

M. Mohammed, M. Romdhani et. al [43], in their attempt to detect errors and defects early in the development process they proposed a tool M Refactor for model refactoring. They conducted domain analysis to identify anti-patterns and bad smells. For each design defects, identify the metric based heuristics, such as tight class cohesion, attribute per method etc. then model the refactoring places using UML model and restructures it after the user validation.

Mohammed et.al [44] performed a systematic literature review by a multistage selection process and analyzed the results based on different criteria such as the methods of model refactoring, the tool support towards model refactoring and the quality of the model after refactoring process. They argued that model refactoring is an active area of research in the future.

v. Agile Development and Refactoring

Thomas D [45] investigated the XP engineering activities: new design, refactoring and error fixing, and concluded that more the design is new less the effort required to refactor it and fix the errors.

Cledson R B de Souza et al. [46] studied the effects of refactoring in the coordination of software development activities. The results have shown that the core developers of the project are highly involved in active communication during the refactoring process. They suggest that refactoring process should be carefully planned to avoid too much stress while conducting refactoring activities.

Tony clear [47] and E. M Hill [15] in their studies revealed that when refactoring is applied in object oriented or agile oriented, as agile is a new buzzword in software development, software projects the quality and productivity increases.

vi. Design Pattern, Anti-pattern and Refactoring

Jing Wang [48] explained the relationship between design pattern, anti-pattern, code smell and refactoring, also how to use these techniques.

Monteiro [49] identified the causes of anti-patterns from different perspectives such as knowledge problems, artifacts problems and management problems. The proliferation of antipatterns can be prevented by promoting the awareness of anti-patterns to the software developers.

Yixin Luo, Allyson Hoss.et.al, [50] developed a knowledge engineering model that depicts the relationship between antipatterns and code smells and refactoring solutions. The ultimate aim of this research work was to improve software quality by identifying and removing code smells and anti patterns before coding begins.

Recent researches have shown that poor design choices such as anti-patterns and poor naming and commenting choices affect software understandability and overall software quality. Venera Arnaoudiva et.al [51] developed a detector prototype and a fast catalogue of linguistic anti-patterns as poor recurring design practices in the naming, documentation and choice of identifiers in a software system. The catalogue they proposed is applicable to methods and attributes. They developed detection algorithms for these linguistic antipatterns. The facts such as methods, types and attribute names are extracted from the source code using a meta language and produces a XML based parse tree. After that a part of speech analysis is performed using a natural language parser and find out the relations among source code elements and comments. The studies among two AgoUML releases, Coccoon and Eclipse reveal that there is more presence of linguistic anti-patterns. Out of the detected linguistic patterns the tool is able to validate only a subset of them for certain categories.

vii. Test Driven Development and Refactoring

Counsell [52] proposed a development environment called TTCN-3 (Testing and Test Control Notation Version 3) which suggest metrics and refactoring support to automatic restructuring of test suites.

Karim O [53] proposed a test driven development approach with motto test first which has three phases red, refactor and green. They used five refactoring methods such as Extract method, Extract class, consolidated conditional expression, Encapsulate field and Hide methods. All the refactoring methods except extract class method exhibited testability property.

Peng hua [54] and Roderickborg [55] conducted unit testing while performing refactoring to check that the changes made to the design preserves semantics after refactoring. Peng Hua used an approach based on object oriented quality model and validated the design pattern against the non functional requirements. Roderickborg proposed a tool for automated acceptance test maintenance using a refactoring approach. The studies have shown that this tool support reduces maintenance effort and makes the system less error prone.

Amog Katti, Sujatha Terdal [56] proposed an algorithm that can be used as a part of extraction refactoring which dynamically analyses the source code and produce static slices. Connection preserving transformation is applied prior to refactoring by applying structural testing of source code and the collected data is analyzed to find data dependencies and compute the slices.

Mark Harman in his keynote presentation at the first refactoring and testing workshop [57] revealed the concept of refactoring as usability transformation. A testability refactoring is a class of testability transformation in which the transformed version of the original program contains test data for the original program. The aim is to build a software that is easier to test and more maintainable. This paper presents several open issues related to testability refactoring such as search based testability refactoring and the concept of test carrying code.

Frens et.al [58] investigated and concluded that unit tests conducted during refactoring does not leads to quicker or quality code refactoring. But their study concentrated on small group controlled experiment, so further study is needed in this area.

viii. Software Refactoring and API Evolution

Danny Dig and Ralph Johnson [59] studied the impact of API changes in three major frameworks and one library. They found that much of the API changes are structural and behavior preserving transformations. They suggest that when components evolve they must be properly documented and proper migration tools must be available to integrate the new components into an application.

Danny Dig et al., [60] presented an algorithm which performs a fast syntactic analysis based on shingles encoding to detect refactoring candidates among two versions of the component followed by the semantic analysis using some strategies to compute the likelihood of refactoring based on references among the source code entities. This refactoring Crawler is robust and scalable but provides poor support for interfaces and fields.

When components in a framework upgrades, changes to its interface may invalidate existing component based applications and require adaptation. Ilie Savga and Michael Rudolf [61] developed a comeback tool that is capable of automatically constructing adaptation specification for the API refactorings such as type and method name changes, change in method signatures and inheritance relations. But the tool fails to adapt field refactorings such as renaming and moving public fields.

Mryung Kim et.al., [62] stated that even though it is believed that refactoring improves software quality, but there is no systematic study of the benefit regarding API level refactorings. Their study produced certain major results 1) After API level refactorings there is an increase in no. of bug fixes but 2) the time taken to fix the bug is shorter after refatorings and 3) revisions occur more frequently due to bug fixes. The study demands the necessity of new software engineering tools that are capable of detecting refactoring

mistakes. Also a quantitative assessment of the refactoring investment is to be made in future.

Miryung Kim, David Notkin et.al [63] proposed a rule based program differencing approach which automatically detects code changes as logical rules applied at API level and the code level. But the rule inference approach omits exception handling and access modifiers in methods. Also, do not address renaming of fields. The algorithm for program differencing using top down rule learning is inefficient because a large rule space is worst searched and discards most of them. The algorithm fails to handle complex refactorings such as changes to design patterns and renaming of fields at API levels.

ix. Software Metrics and Refactoring

Panita Meananeatra [64] proposed an approach to identify optimal refactoring sequences which when applied to programs yield a version with fewer numbers of bad smells with higher maintainability and changeability.

Thushar S. [65] presented a method to estimate the quality of the software design using an index called software design quality index. He concluded that more than a better refactoring tool, a better validation tool is essential to increase the productivity in software development.

Even though there is a belief that refactoring of software systems would improve their quality, there is little bit evidence regarding its quantitative benefits. S. H. Kannangara et.al [66] made an empirical evaluation on the impact of refactoring on code quality improvement. They conducted experiments to identify which refactoring has the highest impact on software quality. They have selected ten refactoring techniques having high impact on code quality and the external quality factors selected are maintainability and efficiency and excluded portability. The experiments were conducted among 60 students on their mini projects the refactoring benefits are assessed for each external quality attributes. Their studies shown that out of ten refactoring techniques they selected replace conditional with polymorphism ranked the highest impact on code quality improvement. The major drawback of their study is that the experiments were conducted among non experts. So it is better to conduct a survey in real industry to identify the refactoring techniques that have high impact on code quality rather than selecting them randomly.

Sultan Alschri and Luigi Benedizenti [67] applied an analytic hierarchy process in an agile environment to rank the refactoring techniques based on the internal quality attributes. Their aim is to exploit maximum benefit when applying refactoring to software systems by reducing the effort and time to a certain extent. They found that the extract method and extract class techniques exhibit internal quality attributes such as reduced complexity, coupling and code size with increased cohesion.

x. Code Clones and Refactoring

Ranier Koschke [68] have revealed certain techniques to detect code clones which may affect the maintainability of the software systems. He identified certain open issues like the integration of clone management tools to support automated refactoring.

Hoan Anh Ngyuen, Thung Thanh Nguyen et.al [69] introduced a clone management tool Jysync to support developers to identify clone relation among code fragments as software emerges and in making consistent changes when creating or modifying code clones. Their empirical study on large open source systems reveals that the tool accurately detect and update code clones using an AST based tree editing operations and analyze the changes of cloned code and performs synchronization and merging. These techniques are more focused on the preventive lines of clone management, that is to avoid new clones as well as compensative view that is to detect and perform consistency analysis of code clones. But there is an open issue for the corrective aspect of clone management that is to remove clones from a system.

Tsantalis and Giri Panamoottil Krishnan [70] in their seminal paper argued that there are still open areas of research in the field of clone management such as determining valid clone regions, refactoring of type-2 and type-3 clones as well as decomposing original clones into sub clones.

E. Kodhai and S. Kanmani [71] developed an enhanced clone manager for clone detection and modification of clones. The detected clones and clusters are documented in a text file. A metric based clone collection is built from the text file. A clone set with high value of number of code clones is a good candidate for applying refactoring after which improves the maintainability of the software system. As per experimental results, the enhanced tool is fast and is able to detect more refactoring opportunities only at the method level.

Franqui Meng et.al [72] proposed a refactoring model of legacy software in smart grid. The model is based on extracting code clones in the scanned source code by means of CCFinder tool. Clone functions which have similar syntax structure is identified and find out the extent of variations among them. Finally combined and other frequently invoked functions are encapsulated or packaged into a DLL file and reused in the development smart grid based new software system. But their studies shown one important fact that the amount valuable clones in legacy software are not too much and hence their model can be used as a subsidiary method in refactoring of large scale legacy software.

Manisankar Modal et.al [73] presented an empirical study about clone fragments that belong to the same clone class and co-change during evolution preserving their similarity. They found that these fragments are important candidates for refactoring. They defined these candidates as evolving, according to them a similarity preserving change pattern. Their studies revealed that merging of clones can greatly reduce the maintenance effort. They developed a prototype tool that detect similarity preserving change patterns clones and then mine important of them using association rule mining concept and also the tool is likely to be enhanced to predict future co-change candidates.

The Identified Gaps and Recent Open Issues in the Area of Software Refactoring Research

After the analysis of the 73 papers, those literatures in which there is a clear mention upon open issues and future work are summarized. The following are some of the identified gaps

and recent open issues in various areas of software refactoring research.

i. Survey of Software Refactoring

- Researchers found that if the most frequent types of refactorings are identified, the scope of refactoring can be narrowed down. They observed many mismatches between the programmer choice and automated refactoring such as awareness of tools, naming etc.
- Studies have found that an average experienced developer performs automated refactoring. A novice user is less familiar with refactoring tools. Studies have shown that manual refactoring takes longer times than automated ones. Also Extract method refactoring is most time consuming both manually and automatically. Rename field refactoring is fast.
- Some of the interesting findings produced by researchers are that the developers wanted to have a better code understanding and validation tool that checks correctness of refactoring.

ii. Software Refactoring Tools

- No refactoring tool is able to support all kinds of refactoring support as per the customer requirement. Therefore customers wanted to have a tool support with better customer support.
- An interesting fact is that many of the refactorings are medium and low level categories. A few tools address higher level refactorings that change the signatures of classes, methods or fields. That means 24 to 60% of refactorings is yet found to be detected and this will produce much benefit to improve the maintenance of the software.

iii. Bad Smell and Refactoring

- Even though a lot of studies are there to discuss the term code smell, but there is no formal definition of what is meant by code smell and how to deal with it.
- Studies note that there is no mature tools to deal with automatic code smell detection and resolution.
- Bad smell concept can be extended to web applications and end user environments like spread sheets and macros, suggest future area of refactoring research.

iv. Software Artifacts and Refactoring

- Refactoring can be applied to different software artifacts such as requirements analysis and design models; test suites etc. are future area of study.

v. Agile Development and Refactoring

- Refactoring advocates for agile development environment, so can this refactoring process is applicable to classical software development process is an open area of research.
- Refactoring improves productivity and maintainability in agile environment, but this needs strong evidence using a quantitative approach.

vi. Design Pattern, Anti-pattern and Refactoring

- Though there exists some studies which may exhibit the relationship between patterns, anti-patterns and refactoring, but the applicability in an industry environment needs further study.
- New researches in this field found out the presence of new anti-patterns like linguistic patterns which affects maintainability of the software needs further study.

vii. Test Driven Development and Refactoring

- Recent studies have shown that unit tests conducted during refactoring does not leads to quicker or quality code refactoring.
- A new area of research emerged in the field of refactoring as testability transformation with the concept of test carrying code.

viii. Software Refactoring and API Evolution

- Although refactoring improves software quality there is lack of tools to perform complex refactorings such as changes to design patterns, renaming fields and automatic consistent semantic preserving updating of renamed program entities at API level.
- Also a quantitative assessment of the benefit of API level refactorings is another open area of research.

ix. Software Metrics and Refactoring

Many works are there to deal with software metrics and refactoring, but the results are found to be inconsistent. Some of the researchers found that there is high impact for particular refactoring technique on code quality but this is to be thoroughly proved by a survey in real industry platform by considering all types of refactoring techniques rather than selecting them randomly.

x. Code Clones and Refactoring

Though code clones are important candidates for refactoring which affects maintainability of the software systems. There are open issues like identifying valid clone regions, removal of clones from the system, detect and remove clones from the package level, decomposing original clones into sub clones, and identification of co-change of clones prior to refactoring during evolution of software systems.

Validation of the Results

In this section, keep in mind the research questions try to discuss how the study findings address them. The collected literatures covered a wide variety of topics. For the past 5 years there is an increase in number of literatures in the area of software refactoring research especially in the group bad smell, design pattern, code clone detection and system evolution. But there are a few papers in survey of refactoring, programming languages and antipatterns. The identified gaps and recent open issues with respect to each group has collected during analysis itself by looking at the open issues and future work reported in the literature. This made the reporting of the result much easier.

Limitations of this Review

The literatures are collected manually and the relevant topics are included. The search terms used in this literature are few to formulate the query. Some terms like feature oriented refactoring, program transformation and refactoring, refactoring and metaprogramming, refactoring and reengineering will add much more results to this study.

Conclusion and Future Work

In software engineering code refactoring plays an important role in improving the quality of the product. The aim of this study is to reveal the recent trends and important contributions using a systematic literature review. The study utilized the literatures of high impact electronic databases. The title, abstract, methods of study, implementation aspects, conclusion and limitations are reviewed to reach at a summarization on where to focus and not to focus. The study can be extended by integrating interviews, questionnaires and practices in the software industry to improve further credibility of the findings.

REFERENCES

[1] M. Lehman, "Laws of Program Evolution-Rules and Tools for programming Management", proceedings Infotech State of the art conference, Why Software Project Fails, vol. 11, pp. 1-25, 1978.

[2] M. Lehman and J. Ramil, "Rules and Tools for Software Evolution Planning and Management", Annals of Software Engineering", vol. 1, no.1, pp. 15-44, 2001.

[3] R. S. Arnold, "Tutorial on Software Restructuring, "An Introduction to Software Restructuring", IEEE Press, 1986.

[4] M. Fowler, "Refactoring: Improving the Design of Existing Programs", Addison-Wesley, 1999.

[5] Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional 2004.

[6] W. F. Opydke, "Refactoring: A Program Restructuring Aid in Designing Object Oriented Application Frameworks", Ph. D. Thesis, University of Illinois at Urban-Champaign, 1992.

[7] T. Mens, "A Survey of Software Refactoring", IEEE, vol. 30, No.2, February 2004.

[8] Mefsin Abebe and Cheol-Jung Yoo, "Trends, Opportunities and Challenges of Software Refactoring: A systematic Literature Review", International Journal of Software Engineering and its Applications, vol.8, No.6, pp. 299-318, 2014.

[9] http://sourcemaking.com/refactoring/defining-refactoring.

[10] Tom Tourwe and Tom Mens, "Identifying Refactoring Opportunities using logic Meta Programming", Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR '03), IEEE Computer Society Press, pp. 91-100, 2003.

[11] Zhenchang Xing, Eleni Stroulia, "Refactoring Practice: How it is and How it should be Supported-An Eclipse Case Study. ICSM 2006, pp: 458-468, 2006.

[12] Ishwar Thapa, Harvey Siy, "Assessing the Impact of Refactoring assertions on the JHotDraw Project", SAC '10 Proceedings of the 2010 ACM Symposium on Applied Computing, pp: 2369-2370, ACM, 2010.

[13] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig, "A Comparative Study of Manual and Automated Refactorings", In ECOOP '13: Proceedings of the 27th European Conference on Object Oriented Programming, pp. 552-576, ACM, 2013.

[14] M. Kim, Thomas Zimmermann, Nachiappan Nagappan "A Field Study of Refactoring Challenges and Benefits", IEEE Transactions on Software Engineering, Vol. 40, No. 7, pp. 633-649, July 2014.

[15] Emerson Murphy-Hill, Chris Parnin, Andrew P. Black, "How We Refactor, and How We Know It", IEEE Transactions on Software Engineering, Vol.38, No.1, pp. 5-18, January/February 2012.

[16] Gustavo H Pinto and Fenando Kamei, "What Programmers say about Refactoring Tools? : an Empirical Investigation of Stack Overflow", In WRT'13: ACM workshop on Workshop on Refactoring Tools", pp 33-36, ACM, 2013.

[17] Emerson Murp, "Improving Usability of Refactoring Tools", In OOPSLA '06': companion to the 21st ACM SIGPLAN conference on Object Oriented Programming Systems, Languages, and Applications, pp 746-747, ACM, 2006.

[18] Michael Mortenson, Sudipto Ghosh, James M. Bieman, "Testing during Refactoring: Adding Aspects to Legacy Systems, 17th International symposium on Software Reliability Engineering (ISSRE '06), 2006.

[19] Yasemin Kosker, Burak Turhan and Ayse Bener, "An Expert System for Determining Candidate Software Classes for Refactoring", In Expert System with Applications: An International Journal, Vol. 36, Issue 6, August 2009, pp 10000-10003. 2009.

[20] E. Mealy and P. Strooper, "Evaluating Software Refactoring Tool Support", Proc. Australian Software Engineering Conference, pp. 331-340, 2006.

[21] L. Zao, J. H. Hayes, "Rank based Refactoring Decision Support: two studies", Innovation System Software Engineering, Springer, pp. 171-189, August 2011.

[22] Wafa Basit, Fakhar Lodhi, and M. U Bhatti, "Formal Specification of Extended Refactoring Guidellines", In Proceedings of QUORS 2012 pp. 260-265, 2012.

[23] M. Vakilian, N. Chen, S. Negara, B. A Rajkumar, B. P Bailey, and R. E. Johnson, "Use, Disuse, and Misuse of Automated Refactorings. In ICSE: 2012 34th International Conference on Software Engineering, pp 233-243, 2012.

[24] Max Schafer, Andreas Thies, F. Steimann and Frank Tip, "A Comprehensive Approach to Naming and

Accessability in Refactoring Java Programs", IEEE Transactions on Software Engineering, Vol. 38, No. 6, pp. 1233-1257, November/December 2012.

[25] Regulwar, Ganesh B.; Tugnayat, Raju M, " Bad Smelling Concept in Refactoring", In International Proceedings of Economics Development & Research, Vol. 45, pp. 56, 2012.

[26] J. PereZ and Y. Crespo, "Perspectives on Automated correction of code smells", in IWPSE-EVOL 2009, Amsterdam, pp 1-34, August 2009.

[27] J. Schumacher, N. Zazworka, F. Shull, C. Seaman and M. Shaw, "Building Empirical Support for Automated Code Smell Detection", In ESM'10: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Article No. 8, 2010.

[28] Serguei Routsov, Alexander Serebrenik and M. Van Den Brand, " Detecting Modularity smells in Dependencies Injected with Java Annotations", In 2010 14th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, pp. 244-242, 2010.

[29] Mu Huaxin and Jian Sherai, "Design Patterns in Software Development", In ISESS: 2011 IEEE 2nd International Conference on Software Engineering and Service Science, pp 322-325, 2011.

[30] Hui Liu, Zhiyi Ma, Weizhong Shao, and Zhendong Niu, " Schedule of Bad Smell Detection and Resolution: A New way to Save Effort", IEEE Transactions on Software Engineering, Vol. 38, No.1, pp. 220-238, January/February 2012.

[31] Hamid, M Illyas, M Hummayun and Asad Nawaz, "A Comparative Study on Code Smell Detection Tools", International Journal of Advanced Science and Technology, Vol. 60, pp. 25-32, 2013.

[32] Dag I. K. Sjøberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dyba, "Quantifying the Effect of Code Smells on Maintenance Effort", IEEE Transactions on Software Engineering, Vol. 39, No.8, pp. 1144-1156, August 2013.

[33] K. T. Stolee and Sebastian Elbaum, "Identification, Impact, and Refactoring of Smells in Pipe-Like Web Mashups", IEEE Transactions on Software Engineering, Vol. 39, No.12, pp. 1654-1679, December 2013.

[34] Hui Liu, Xue Guo and Weizhong Shao, " Monitor-based Instant Software Refactoring", IEEE Transactions on Software Engineering, Vol. 39, No. 8, pp. 1112-1126, August 2013.

[35] Jiang Dexun, Ma Peijun, Su Xiaohong and Wang Tiantian, "Functional Over-related Classes Bad smell Detection and Refactoring Suggestions", International Journal of Software Engineering and Applications, Vol. 5, No.2, pp. 29-48, March 2014.

[36] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia, "Methodbook: Recommending Move Method Refactorings via Relational Topic Models", IEEE Transactions on Software Engineering, Vol. 40, No. 7, pp. 671-694, July 2014.

[37] W. Kesantine, M. Kessantini, H. Sahroui, S. Bechikh and Ali Ouni, " A Cooperative Parallel Search-based Software Engineering Approach for Code Smells Detection", IEEE Transactions on Software Engineering, Vol. 40, No. 9, pp. 841-861, September 2014.

[38] Santiago A. Vidal, C. Marcos, J. Andres and Diaz Pace, "An Approach to Prioritize Code Smells for Refactoring", Springer Science and Business Media, November 2014.

[39] S. Demeyer, S. Ducasse and O. Nievstrasz, "Object Oriented Reengineering Patterns", Published by Square Bracket Associates, Switzerland, Reprinted first edition, 2009.

[40] Mika V. and C. Lassenium, "Drivers for Software Refactoring Decisions", In ISESE '06: International Symposium on Empirical Software Engineering", ACM, pp 297-306, 2006.

[41] Richard Mateos, "Software Development Patterns", IEEE Micro, Vol. 28, No. 5, pp. 72,71, IEEE 2008.

[42] L. Dobraznski and L. Kuzniarz, "An approach to Recovering of Executable UML Models", In SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing, pp 1273-1279, ACM 2006.

[43] M. Mohammed, M. Romdhani, and K. Ghedira, "M-REFACTOR: A New Approach and Tool for Model Refactoring", ARPN Journal of Systems and Software, Vo. 1, No.4, July 2011.

[44] Mohammed Misbhauddin and M. Alshayeb, "UML Model Refactoring: a Systematic Literature Review", Empirical Software Engineering, Vol. 26, Issue1, pp. 206-251, 2013.

[45] Thomas D., "Agile Programming Design to Accommodate Change", IEEE journals and Magazines, Vol. 22, Issue 3, pp. 14-16, 2005.

[46] Cledson R. B, M. P Rosa, C. S. Goto, Jean M. R. Costa, and P. J. F Treccani, " On the Effects of Refactoring in the Coordination of Software Development Activities", In ECSCW '09: Proceedings of the 11th European Conferenceon Computer Supported Cooperative Work, Springer Series in Computer Science, pp. 215-222, 2009.

[47] Tony Clear, "Disciplined Design Practices: A Role for Refactoring in Software Engineering?", In Newsletter ACM SIGCSE Bulleting Vol. 37, Issue 4, pp 15-16, 2005.

[48] Jing Wang, Y-T Song and L. Chung " From Software Architecture to Design Patterns", In SNPD/SAWN '05: Proceedings of the sixth International Conference on Software Engineering, Artificial Intelligegence, Networking and Parallel/Distrbuted Computing and First ACIS International Workshop on Self Assembling Wireless Network.

[49] M Monteiro and A. Aguiar, "Pattern for Refactoring to Aspects: An Incipient Pattern Language", In PLoP 2007, pages 1-19, 2007.

[50] Yixin Luo, A. Hoss and D. L Carver, "An Ontological Identification of Relationships between Anti-Patterns and Code Smells", IEEEAC Paper, IEEE Computer Society Press, pp. 1-10, 2010.

[51] Venera Arnaoudova, M. D Penta, G Antoniol, and Y. Gueheneuc, "A New Family of Software Ani-Patterns: Linguistic Anti-Patterns", In 2013 17th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, pp. 187-196, 2013.

[52] S. Counsell and R. M Hierons, "Refactoring Test Suites Versus Test Behaviour: A TTCN-3 Perspective", In Proceedings of SOQUA '07 Fourth international Workshop on SQA: in conjunction with the ESEC/FSE joint meeting, pp 31-38, ACM DL, 2007.

[53] Elish, Karim O., and Mohammad Alshayeb. (2009), "Investigating the Effect of Refactoring on Software Testing Effort" In Software Engineering Conference, APSEC'09, Asia-Pacific, pp. 29-34, IEEE Computer Society Press, 2009.

[54] Nien-Liu, H Such, Peng Hua and William chu, "A Test Case Refactoring Approach for Pattern-based Software Development", Journal of Systems and Software, Vol. 81, Issue 8, pp 1430-1439, 2008.

[55] Roderick Borg and M. Kropp, "Automated Acceptance Test Refactoring", in WRT '11: Proceedings of the 4th Workshop on Refactoring Tools, pp 15-21, ACM, 2011.

[56] Amogh Katti and Sujatha Terdal, " Program Slicing for Refactoring: slicer using dynamic Analyzer", International Journal of Computer Applications, Vol. 9, No.6, pp. 36-43, November 2010.

[57] Mark Harman, "Refactoring as Testability Transformation", Keynote Presentation in the Ist Refactoring and TESTing Workshop, Berlin, March 2011.

[58] Vonken Frens and Zaldman A.,"Refactoring with Unit Testing A Match made in Heaven", In WCRE '12: 19th Working Conference on Reverse Engineering, pp 29-38, IEEE, 2012.

[59] Dig, D., Johnson, R.: The Role of Refactorings in API Evolution. In: ICSM (2005), IEEE computer Society Press, pp. 389-398, 2005.

[60] Dig, D., Comertoglu, C., Marinov, D., Johnson, R.E.: Automated detection of refactorings in evolving components. In Proceedings of 20th European Conference on Object Oriented Programming (ECOOP), pp. 404-428, 2006.

[61] Illie Savga and Michael Rudolf, "Refactoring-Based Adaptation of Adaptation Specifications", Software Engineering Research, Management and Applications, SCI 150, Springer Series, pp. 189-203, 2008.

[62] M. Kim, D. Cai and S. Kim, "An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution", ICSE'11, ACM, 2011.

[63] M. Kim, D. Notkin, D. Grossman and Gary Wilson, " Identifying and Summarizing Systematic Code Changes via Rule Inference", IEEE Transactions on Software Engineering, Vol. 39, No. 1, pp. 45-62, January 2013.

[64] Panita Meananetra, "Identifying Refactoring Sequences for Improving Software Maintainability", ACM Press, pp. 406-409, September 2012.

[65] Tushar S., "Quantifying Quality of Software Design to Measure the Impact of Refactoring", In COMSACW '12: Proceedings of the 2012 IEEE 36th annual Computer Software and Application Conference Workshops, pages 266-271, ACM, 2012.

[66] S. H. Kannangara and W. M. J. I. Wijayanayake, "Impact of Refactoring on External Code Quality Improvement: an Empirical Evaluatiion", In Internatiional Conference on advances in ICT for Emerging Regions (ICTer), IEEE Computer Society Press, pp. 60-67, 2013.

[67] Sultan Alshehri and Luigi Benedicenti, "Ranking the Refactoring Techniques Based on the Internal Quality Attributes", International Journal of Software Engineering and Applications, Vol. 5, No. 1, pp. 9-30, January 2014.

[68] Ranier Koschke, "Frontiers of Software Clone Management", Frontiers of Software Maintenance 2008, pp. 119-128, 2008.

[69] H.A Nguyen, T. T Nguyen, N. H Pham, J. A Kofahi and T.N.Nguyen, "Clone Management for Evolving Software", IEEE Transactions on Software Engineering, Vol. 38, No. 5, pp. 1008-1026, September/October 2012.

[70] Nikolaos Tsantalis and Giri Panamoottil Krishnan, "Refactoring Clones: A New Prespecitve", IWSC 2013, IEEE Computer Society Press, 2013.

[71] E. Kodhai and S. Kanmani, "Method-level Code Clone Modification using Refactoring Techniques for Clone Maintenance", Advanced Computing: An International Journal (ACIJ), Vol. 4, No.2, pp. 7-26, March 2013.

[72] Fanqi Meng, Zhaoyang Qu and Xiaoli Guo, "Refactoring Model of Legacy Software in Smart Grid based on Cloned Codes Detection", International Journal Computer Science Issues, vol. 10, Issue1, No.3, pp. 296-303, January 2013.

[73] Manisankar Modal, Chanchal K. Roy and Kevin A. Schneider, "Automatic Ranking of Clones for Refactoring through Mining Association Rules", CSMR-WCRE 2014, IEEE Computer Society, Press, pp. 114-123, 2014.

Sreeji K. S was born in Thrissur, Kerala, India in 1970. She received her M. Tech degree in Computer Science and Engineering from SRM University, Chennai, India and pursuing PhD degree in the department of Computer Science and Engineering at SRM University under the guidance of Dr. C. Lakshmi, Professor and Head of the Department of Software Engineering, SRM University, Chennai, India. She is working as Assistant Professor in Malabar College of Engineering and Technology affiliated to Calicut University and A. P. J Abdul Kalam Kerala Technological University, Kerala, India.

Dr. C. Lakshmi received the Masters of Engineering in Computer Science and Engineering from Madras University, INDIA and received her Ph. D in Computer Science and Engineering from SRM University, Chennai, INDIA. She is the professor and Head of the Department of Software Engineering in SRM University. Her research interests include Digital image processing, Pattern Recognition, Web Services, E-learning, and Software Engineering. She is certified as adjunct Faculty for Architecture for software systems and Analysis of software Artifacts courses for the year 2014-2015 by Institute of software Research, Carnegie Mellon University, Pittsburgh, USA.