International Journal of Software Engineering and Its Applications
Vol.8, No.6 (2014), pp.299-318
http://dx.doi.org/10.14257/ijseia.2014.8.6.24

Trends, Opportunities and Challenges of Software Refactoring: A Systematic Literature Review

Mesfin Abebe and Cheol-Jung Yoo

Chonbuk National University, Republic of Korea
567 Baekje-daero, Deokjin-gu, Jeonju-si,
Jeollabuk-do, Republic of Korea
mesfinabha@gmail.com, cjyoo@jbnu.ac.kr

Abstract

Software refactoring is a technique that transforms the various types of software artifacts to improve the software internal structure without affecting the external behavior. Refactoring is commonly applied to improve the software quality after a significant amount of features are added. Researchers in the area have studied the different angles of refactoring and developed the right evidence, knowledge and skill. And they published their research findings through journals and conference papers to provide an easy access to everyone. Eventually, the knowledge accumulated in these literatures is huge, so that it needs structuring and organizing. The main purpose of this study is to extend a previously conducted study by covering more literatures and applying a systematic literature review method to increase the accuracy and validity of the study. We study a collection of literature from different electronic databases, published since 1999 to understand and extract the software refactoring knowledge through classification and summarization. The classification and summarization can reveal the research pattern, common concerns and statistics of the published papers in the last fifteen years. The extracted information should help the researchers to formulate better research topics that can solve the crucial problems in software refactoring and save the researchers effort and time.

Keywords: *Software refactoring, Code smell, Design patterns, Refactoring tool*

1. Introduction

In the real-world, it is obvious for software system to evolve over time and adapt to the constantly changing work environment. Due to this, software enhancement, modification and adaptation to new requirements become more complex [1, 2]. In addition to this difficulty, software evolution is time-consuming, tedious, and often comprises up to 75% of the costs of the software development [3]. As the Lehman's laws of software evolution state [4, 5], a functionality increment of a system always brings a corresponding decrease in the quality and an increase in the internal complexity. Thus, software developers need flexible, maintainable, and extensible software development techniques to create smooth integration and extension of new features. In the past, practitioners proposed several techniques to solve the problem and one of these techniques is software refactoring.

The branch of software engineering that study and address the complexity problem in software development process is referred to as software restructuring [6, 7] or software refactoring in the case of object-oriented software [8, 9]. Though, there is no one universally accepted definition of software refactoring, it is just the process of changing the internal structure of a software system without altering its external behavior [10]. In so doing, the refactored code can have an optimized object oriented features such as: encapsulation, polymorphism, inheritance etc. that can improve the quality of the code in terms of maintainability, reusability and modifiability. In most cases, the key purpose of software refactoring is to transform the program into a better quality by fixing the quality defects like code smells, anti-patterns and anomalies [11].

In the last fifteen years, researchers contributed large size of knowledge and concepts in the area of software refactoring. This contribution covers different angles and phases of software development activities for instance requirement analysis, software design, implementation, integration, testing and maintenance [8]. However, the term software refactoring is highly associated and regularly used with coding activity, which is commonly known as *code refactoring*. Finally, it is necessary to acquire the right knowledge, skills, techniques, and tools to fully benefit from software refactoring.

In this paper, we examine software refactoring research papers published since 1999 to classify and summarize the various efforts and findings. To this end, we collected literatures from the following nine electronics databases: *IEEE Xplore*, *ScienceDirect*, *Springer Link*, *ACM DL*, *Web of Science*, *CiteSeer*, *Wiley Online Library*, *Scopus Elsevier*, *EI Compendex*. The collected papers are analyzed and classified based on their title, abstract and the main section of the papers. In doing so, we determined how far each group studied and also identified the areas which need further study. The reminder of the paper is organized as follows: Section 2; describes the research background. Section 3; explains the research method used in the study. Section 4; provides the contribution of the study. Section 5; discusses the result of the study. Section 6; describe the justification and limitation of the study. Finally, we preview future trends and conclude in Section 6.

2. Research Background: Software Refactoring

In this section, we first describe the field of software refactoring, the steps in refactoring process, the different groups of refactoring techniques, and the benefits and problems of software refactoring. We then present the previous works on software refactoring, justify the importance of this study, and state the research questions.

2.1. The Field of Software Refactoring

The term software refactoring was coined for the first time by William F. Opdyke in his Ph.D. dissertation [12], however it becomes more practical and commonly used after the publication of the book *Refactoring: Improve the Design of Existing Code,* which is written by Martine Fowler in the year 1999. Since then, refactoring has been used to improve software qualities for commercial and open source software systems for example; Microsoft reserves 20% of the development effort for code rewriting [13]. The basic principle in refactoring process is reorganizing classes, variables and methods across the class hierarchy to facilitate future adaptations and extensions, so that the source code can have better structure, readability and understandability. Sometimes the term software restructuring used with software evolution (non-refactoring code modification) that improve the quality of the software such as extensibility, modularity, reusability, complexity, maintainability and efficiency [12], even though there is a huge difference between the two.

Knowing where to apply the refactoring is quite challenging and not easy to identify the location of the bad design known as “*Bad Smells*” or “*Code smells*”. The detection of the bad smells is the most well-known approach to identify the part that requires software refactoring. According to Beck, bad smells are “structures in the code that suggest the possibility of software refactoring". The code smell doesn't tell us what to do to solve the design problem but it narrow down the numbers of potential refactoring techniques to deal with it. Fowler & Beck (Fowler & Beck 2000) and others researchers provide a list of bad smells with their corresponding remedy refactoring techniques to make the refactoring decision easier [10]. Generally, there are about 25 code smells and more than 75 software refactoring techniques for use.

Although, code refactoring is the common one, refactoring can also be applied with various software development artifacts like: requirement specifications, software architectures, design models, test suite, database and HTML document [15, 16]. Commonly, the software refactoring process includes the following steps and activities.

1. Apply unit test to the program.
2. Identify which part of the code needs the refactoring using code smells.
3. Select a refactoring technique to remove the identified code smell.
4. Apply the selected refactoring technique.
5. Apply regression testing to the refactored code.
6. Assess the effect of the refactoring using software quality characters or the process.
7. Maintain consistency between the program and the other artifacts.

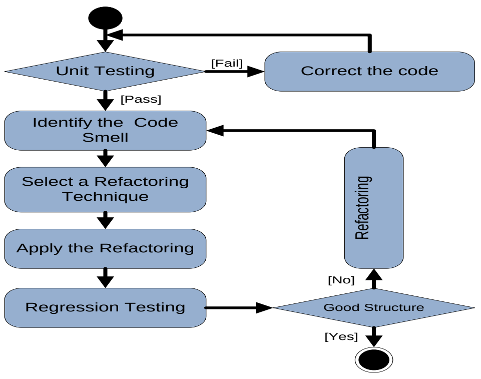

Figure 1. The Steps in Refactoring Process

In addition to improving the internal structure of the code, software refactoring can also provide the following benefits as it is shown in Table 1:

Table 1. Benefits of Software Refactoring

Benefit Type	*Description*
Remove duplication of code	Programs that have duplicate logic are hard to modify, since it needs be do the same modification to many places.
Improve the design	The design of a program will decay as developers changed the code to realize short term goals unless it is regularly refactor.
Makes the code easier to understand	Even if codes are written to instruct computers, there is always someone who will try to read it in a few months' time to make some change, so that it should be understandable to this future developer.
Helps to program faster	A good design is important to maintain speed in software development. Refactoring can help us to develop software more rapidly by preventing the design from decaying and improve it.

Software refactoring is a valuable technique in many cases, but it is not a silver bullet solution to every problem in software development. A common concern with software refactoring is the effect it has on the performance of the program since; the change may cause the program to run more slowly [15]. However, it also makes the software more amenable to performance tuning. Software refactoring techniques can be classified based on what they change in the code as follows:

Table 2. Grouping of Software Refactoring Techniques

Refactoring Type	*Description*
Composing methods	Refactoring techniques those deal with design problems related with methods e.g. Extract method, Inline method etc.
Moving features between objects	Refactoring techniques which can help in decision making of where to put the responsibility of an object e.g. Move method.
Organizing data	These are refactoring techniques that make working with data easier e.g. Self-encapsulate filed, Replace data value with object.
Simplifying conditional expression:	Refactoring techniques that are used to simplify the conditional expressions by breaking the condition into pieces e.g. Decompose condition, Remove control flag.
Making method calls simple	Refactoring techniques that make interfaces more straightforward and easy to understand e.g. Rename method.
Dealing with generalization	Those refactoring techniques which deal mostly with the methods around a hierarchy of inheritance e.g. Pull up filed.
Big refactoring	All the above refactoring techniques concentrated on small size refactoring, but this group includes techniques applied in large scale e.g. Extract hierarchy.

2.2. Summary of Related Work

Tom Mens (2004) provided an extensive overview of existing research area in the field of software refactoring such as: refactoring activities, techniques and formalism, types of software artifacts and effect of refactoring on software process [17]. Karim O. Elish and Mohammad Alshayeb proposed a classification of refactoring methods based on their measurable effect on software quality attributes [18]. A detail overview of the field of software restructuring and refactoring was provided by Bart Du Bois, Pieter Van Gorp and *et al*. They discuss the current research on refactoring using the research questions: what technique to use, how we can compose refactoring in a scalable way, dependency between refactorings, how can apply refactoring at high level of abstraction, how refactoring can relate to other software engineering techniques and how can we compare refactoring tools and techniques [19].

Panita Meananeatra [20] formulated four criteria for optimal software refactoring: the number of removed bad smells, maintainability, the size of refactoring sequence and the number of modified program elements. Tom Mens and Arie Van Deursen [21] identified emerging trends in refactoring research, and open questions. They also identified the need for formalisms, processes, methods and tools that address refactoring in a more consistent, generic, scalable and flexible way. Mesfin Abebe and Cheol-Jung Yoo [22] use a traditional literature review approach to classify and summarize the studies in software refactoring with the intention of determining the significant contribution and gaps in the software refactoring studies.

This study different from the above studies, since the main purpose is to implement a systematic literature review which follows a defined protocol to increase the validity and rationality of the study, so that the output can be high in quality and evidence based. At the same time we used more electronic database and large numbers of literatures to comprise all the possible candidate studies and to cover more works. In addition, we slightly modify the classification of the literatures and present the contribution and gaps of software refactoring with respect to each classification as explained in detail in Section 5.2 and 5.3.

2.3. Objective of the Study and Research Questions

The main objective of the study is to examine the existing software refactoring literatures and present the most relevant contributions and gaps. The following research question (RQ) and sub questions (SQ) are formulated to identify, analyze, classify and summarize the findings of the accumulated software refactoring literatures:

- RQ: What are the trends, opportunities, challenges and gaps in software refactoring research activities?
- SQ: What are the general studies areas (classification) in software refactoring research activities?
- SQ: Which part of the research area is exhaustively studied and what are the significant contributions of each study area?
- SQ: Which part of the research area does not received sufficient attention of study as well as what are the gaps in each study area?

3. Research Methodology

In this section, we established the methods used in the systematic literature review and described each stage in detail.

3.1. Protocol Development

This protocol is formulated to guide the systematic literature review based on the book Systematic Reviews: CRD's guidance for undertaking reviews in the health case [23] and using the best practices from systematic literatures review in Software Engineering [24, 25]. The protocol includes the research questions, search strategy, inclusion, exclusion and quality criteria's, data extraction, and data synthesis methods as it is described in the following sections. The protocol, we follow consists of the following three phases and a number of sub activities under each phase:

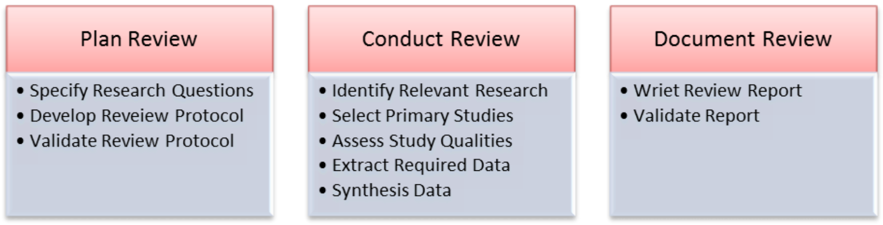

Figure 2. The Systematic Literature Review Protocol

3.2. Inclusion and Exclusion Criteria

Any software refactoring literature is eligible to be part of the study, if it meets all of the following inclusion criteria, but not any of the exclusion criteria listed below. The following are the inclusion criteria:

- Literatures related to only software refactoring study.

- Papers written by students, researchers and professional software developers.
- Only conference and journal papers.
- Literatures published since 1999 and retrieved from one of the nine electronic databases in *Table 3*.
- Literatures that are written in English language.

Literature that meets any of the following exclusion criteria shall be excluded from the study, even though it meets the above inclusion criteria's. The following are the exclusion criteria:

- Literatures which are not related to software refactoring topic.
- Literatures which are written in non-English language.
- Duplicated papers.
- Literature such as: book, thesis, dissertation or tutorial.
- Literatures which are written before the year 1999.

3.3. Data Source and Search Strategy

In the attempt to perform an exhaustive search, we use Breeton *et al.*, suggested list of the seven electronics database sources as relevant data source to software engineers study with a small modification. Table 3 shows the nine electronic databases and the number of retrieved literatures under each electronic database.

Table 3. List of Electronic Database and Number of Papers

No	*Electronics Database*	*No. of papers*	*Percentage*
1	IEEE Xplore	300	22.09%
2	ScienceDirect	129	9.50%
3	Springer Link	381	28.06%
4	ACM Digital Library	156	11.49%
5	Web of Science	155	11.41%
6	CiteSeer	41	3.02%
7	Wiley Online Library	76	5.60%
8	Scopus Elsevier	53	3.90%
9	EL Compendex	67	4.93%
	Total	***1358***	***100.00%***

Figure 3 describes the relevant data source selection process and the numbers of literatures identified at each steps of the process. The nine electronic databases were searched using a search query, which contains the terms: software refactoring, code smell, design patterns and software restructuring. To narrow down the number of hit with the search query, each database search was limited with the following criteria's: Journal or Conference, English language, papers published since 1999 and papers match only with their title, abstract or keywords with the query term.

In the first step of the data source selection process, the electronic databases were searched against the following search terms and 1358 papers are retired as it is shown in Table 3.

- Refactoring
- Code Restructuring
- "Code smell" AND Refactoring
- "Design pattern" AND Refactoring
- Refactoring OR "Software Refactoring" OR "Code Smell" OR "Design pattern".

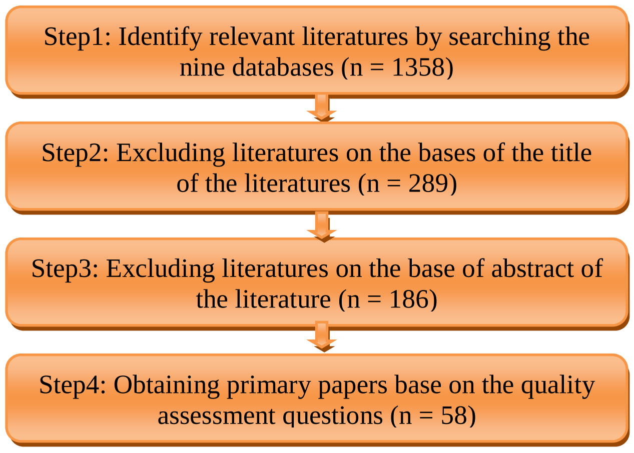
Step1: Identify relevant literatures by searching the nine databases (n = 1358)
Step2: Excluding literatures on the bases of the title of the literatures (n = 289)
Step3: Excluding literatures on the base of abstract of the literature (n = 186)
Step4: Obtaining primary papers base on the quality assessment questions (n = 58)

Figure 3. Steps in the Data Source Selection Process

3.4. Inclusion and Exclusion Decision

All the retrieved literatures (n = 1358) title and abstract were imported to *JabRef 2.10* reference manager for detail analysis. We created nine separate databases to store the search result from the nine electronic databases using the JabRef software. Then, we created one additional database to merge all the nine electronic databases and identify the duplicate literatures, which appeared in two or more electronic databases. The total numbers of the literatures decreased to 807, after the merging and removal of the duplicated papers. In the second step, we use the title of the papers to determine their relevance for the study. Though, titles are not always descriptive to indicate clearly what an article is about, we excluded literatures which are not related to software refactoring based on the paper title. As a result of this filtering based on the title of the papers, we excluded 518 papers.

In the third step, the abstract of the remaining papers is used to find out those papers which are still eligible for the study. Finally, at the fourth step we include the studies that meet the inclusion criteria and the quality assessment questions. Whenever a disagreement occurred among the group member to make decision on the inclusion of a paper, it was resolved through discussion before we proceeded to the next step. Finally, after all the four steps are completed and the disagreements are resolved, we left with *58* papers for the detailed analysis and classification.

3.5. Quality Assessment

The outcome quality of a systematic literature review has a direct correlation to the quality of the data sources. To assess the quality of the data source, we include the following three screening criteria, which are very useful to evaluate the rigor, credibility and relevance of the study. Each paper is evaluated against the following quality assessment questions to identify the papers used for the summarization part of the study:

- Is the study covered significant enough knowledge of software refactoring?
- Is there a clear statement of objective of the research?
- Is there a dedicated section that shows clearly the contribution, gap and proposed future work in software refactoring?

Taking into account the above three criteria, we believe that any study which passed these criteria has valuable information for the systematic literature review. Eventually, the 186 papers evaluated against the quality assessment questions. At this step another 128 papers are excluded and only 58 studies are remained that are used to support the data extraction and synthesis process. The list of these papers is shown in *Appendix A*.

3.6. Data Extraction

In this section, we extracted data from the 58 papers using an ad-hoc data extraction form shown in Appendix B. The data extraction form help us to store the extracted data for further data processing and to identify how each paper addressed the research questions. It is designed and developed based on the review questions and the subsequent analysis process of information extract. The aim, setting, research methods descriptions, findings, conclusions, proposed future works and so on were recorder using the data extraction form. All the data extraction process was conducted by one research and the extracted data was evaluated by another researcher. We also carefully analyzed the extracted data and generate a common concept to remove the inconsistent in language usage among the different papers.

3.7. Data Synthesis

At this section we collected, combined and summarized the extracted data from each paper for the final steps of the systematic literature review, which is the classification and summarization of the software refactoring research activity. This study uses Meta-analysis to contrast and combine findings from different papers to identify similarity among papers and established a classification scheme. The title, abstract and conclusion section of all the 186 papers examined to extract the central idea of the literatures in order to classify them into different groups as it is shown in Table 4 and 5. This preorganization of the literatures into different groups create a playground for the summarization activity, so that we are able to deal with a specific area of refactoring at a time. For the summarization part of the study, we use a qualitative data analysis approach to understand and interpret the contributions and gaps of the 58 papers that are identified at the quality assessment step. The data extracted from the 58 papers was used to formulate the significant contribution and gaps of the software refactoring researches as it is discussed in detailed in Section 5.2 and 5.3.

4. Contribution of the Study

The main contribution of this study is the following three benefits, which can guide the researchers to focus into a specific direction to save their time, effort and resources.

- Classify the software refactoring studies into logical groups' based on their content.
- Identify the significant contributions in the software refactoring study with respect to each group.
- Determine the software refactoring study areas that need further study and suggest the available research areas for future work.

5. Results of the Study

This section discusses the result of the study staring with the classification then the summarization, which itself consists of the significant contribution and the gaps in software refactoring research activities. The identified 186 papers cover a range of software refactoring topics and research questions. These papers use different research methods and conducted by various professionals from universities, research centers and software industries. We classified the studies into ten logical groups as it is discussed in detail in Section 5.1.

5.1. Classification of Software Refactoring Research

The title, abstract and conclusion sections of the 186 papers were analyzed and classified into the following ten logical groups: For the sake of simplicity, a paper can lied in only one group, even though it matched with more than one group. The decision was done by the two researches by putting the papers to the group which has high similarity.

Table 4. Classification of Software Refactoring Researches

Classification	*Description*
Survey of software refactoring activities	This group includes literatures, which study the research activities in software refactoring to asses thoughts, opinion and former research findings to summarize them from a certain angle.
Software refactoring tools	It consists of software refactoring research papers that are written on software refactoring tools or those related to refactoring techniques.
Bad smell and Refactoring	This group includes research papers that discuses bad smells (code smell) and software refactoring with their relationship.
Software artifacts and Refactoring	This group addresses research papers, which study software artifacts such as requirement specification, design document, test case, and their relationship with software refactoring.
Agile development and Refactoring	This group consists of research papers which focus on agile development process, software refactoring and their relationship.
Design pattern and Refactoring	It holds studies related to design pattern, micro pattern, anti-pattern and their relationship with software refactoring towards the development of good quality software.
Test driven development and Refactoring	This group considers research papers associated with test driven development (TDD) approach and software refactoring which is an important pashas in TDD.
Software refactoring and System Evolution	This group study research papers that discuss software refactoring, system evolution and their similarity and difference in their activities.
Software metrics and Refactoring	This group contains studies focus on internal and external software quality attributes and how they are affected after software refactoring techniques are applied.
Programming Language and Software Refactoring	This group includes studies related to specific types of programing language to show how software refactoring can be applied with the context of that particular language or technique.

Table 5. The Classification of the Literatures

No	*Classification*	*No. of papers*	Percentage
1	Survey of software refactoring activities	13	6.99%
2	Software refactoring tools	29	15.59%
3	Bad smell and Refactoring	22	11.82%
4	Software artifacts and Refactoring	18	9.68%
5	Agile development and Refactoring	25	13.44%
6	Design pattern and Refactoring	21	11.29%
7	Test driven development and Refactoring	17	9.14%
8	Software refactoring and system evolution	13	6.99%
9	Software metrics and refactoring	19	10.22%
10	Programming Language and refactoring	9	4.84%
	Total	***186***	***100.00%***

Table 5, summarizes the classification of the literatures in respect to each group with the number of papers. As it is indicated in the table, the different groups have different research coverage, which can be inferred easily from the number of papers in each group.

5.2. Significant Contribution in Software Refactoring Researches

The following are the analysis result of the 58 papers. We read the entire section of the papers to analyze and summarize the following significant contributions from the studies.

Survey of Software Refactoring: The term Database, HTML document and refactoring mentioned together frequently to indicate the possibility of applying refactoring with this software artifacts. To this end, researchers have created code refactoring techniques, Database Refactoring Framework and a general framework of how to apply refactoring in HTML document. (2)

- The qualitative analysis of win 7 version history modules by a team of experienced researchers indicate a significant reduction in the number of inter-module dependencies and post-release defects, which indicate the visible benefit of software refactoring[1].(3)
- Though, it is not an exhaustive list, researchers identified the driving force of system refactoring namely: cost, profits, suppliers, information, technical or features. (4)
- Identification of refactoring techniques to apply is associated to a particular application domain e.g. Web based system. Researchers discovered that the input-output behaviors preservation of the refactoring activity depends on the software types e.g. Real-time system depends on temporal constraint, embedded system depends on memory and power constraint and safety critical system depends on safety factors. (18)
- A field survey conducted on programmers reveal the factors that affect the use of automated refactorings likes: invocation method, awareness, naming, trust and predictability, the major mismatches between the programmer expectations and automated refactoring. (19)
- Some studies determine the combinational use of the structural and semantic information of the source code for software refactoring. Since many of the existing literatures mainly exploiting structural relationships of the code, e.g. method call. These researchers also suggest the use of the semantic information embedded in the source code, e.g. the terms in the comments are also important. (46)
- A survey conducted to evaluate the programmer views on the software refactoring tools, indicate that programmers do not often relay on refactoring tools, but they desire a number of unimplemented features to be incorporate in the tools. (50)
- A detailed overview of existing research in the field of software restructuring and refactoring was conducted to list out the open questions, which indicate the future research directions. (47)
- How can we improve programmers experience, knowledge and skill of refactoring is a challenge which need researches attention. (57)

Software Refactoring Tools: Some Authors suggests refactoring based on migration tools should be used to update application, as they observed that over 80% of these changes are refactorings. (5)

- A wide variety of formalisms and techniques are used to deal with refactoring activities. The use of assertions (preconditions, post conditions, and invariants) and

[1] The reference numbers in the curly brace refer to *Appendix A*.

graph transformation explained in detailed. How formalisms can help to guarantee program correctness and preservation is also described in detail. (10)

- To build software refactoring tool or to use, the following general characteristics shall be considered: supplier assessment, economic issues, easy of introduction, reliability, efficiency and compatibility. In addition to these general level criteria researchers provided detailed characteristics of the refactoring tools. (11, 39)
- Some researchers explain and demonstrate how to use code complexity analysis to detect whether classes need a refactoring or not to make refactoring decision. (14)
- Recently search-based approaches of automating the task of software refactoring have been proposed such as: simulated annealing, genetic algorithm and multiple ascents hill-climbing. (19)
- An abstraction for methods viz. Data and Structure Dependency (DSD) graph and an algorithm viz. longest edge removal algorithm are techniques proposed to find extract-method refactoring candidates, which is one of the most frequently used refactorings to address code smells like long and cohesive methods, and duplicated code. (20)
- In most of the refactoring tools, the set of refactoring techniques are hard-coded into it, so that to achieve a slight different code transformation, the developers have to either apply a sequence of several built-in refactorings, or perform the transformation by hand. To address this limitation, researchers proposed refactoring based on synthesis from examples. (38)

Bad Smell and Refactoring: In an attempt to identify the most commonly used code smell, researcher find out reparative code components, frequent item sets, long classes and god classes are the main areas where restructuring are applied. (6)

- To make change of design philosophy accessible outside the security community it is documented as a collection of refactorings, which include problem templates that identify suspect design practice, and target patterns that provide solution. (7)
- Researches undertake a large numbers of relatively simple refactorings and found out an indirect relationship among most of the commonly used refactorings. (21)
- A study shows that the move method, move filed, rename method, rename filed refactorings techniques are “worker refactorings” that provide a service to the other (perhaps more complex) refactorings techniques. (33, 52)
- Using bad smells to direct refactoring and to address 'trouble' in code is not well clear. Developers investigate empirically the impact of bad smells on software in terms of their relationship to faults. (34)
- Researchers analyzed the relationships among different kinds of bad smells, and their impact on resolution orders of these bad smells. In the analysis, they recommend a resolution order of commonly used bad smells. (43)
- Some researches review the traditional object-oriented code smells in the light of aspect –orientation programming and propose some new code smells for detection of crosscutting concerns and new code smells that is specific to aspect-oriented programming. (51)

Software Artifice and Refactoring: A systematic approach to specification of both excitable UML model refactorings and associated bad smell in models, which are preliminary artifacts in Agile MDA (Model-Driven Architecture) development methodology, is provided. (33)

- Code smells and refactoring techniques are presented to improve the quality of OCL (Object Constraint Language) which is OMG standard and plays an important role in the elaboration of precise model. (1)
- Detecting defects at the model level during the design process can have great benefit to designers in particular within an MDE (Model-driven Engineering) process. Researchers suggested an automated approach to detect model refactoring opportunities related to various types of design defects using Genetic Programming. (30)
- A new approach to formalize model refactorings is presented based on the semantics of a predefined set of fine-grain transformations (FGTs) that can be used to construct any refactoring, and relies on logic-based representations of the UML model. (37)
- A rule-based inconsistency resolution, which enables reuse of different inconsistency resolutions across model refactorings that manages the flow of inconsistency resolution steps are offered for use. (40)
- To crate well-organized use case models, studies show the refactoring of use cases models based on the information captured in episode model. They introduce 10 refactoring rules for use case refactoring, including their validation of the behavioral-preserving property. (41)
- Refactoring of the design of an application is suggested using UML class diagrams and the behavior of each class diagram using state chart diagram. As preserving behavior is one of the defining attributes of a refactoring, the researches use a CSP-based formalism to check indeed the refactoring preserve the internal behavior. (42)
- To identify and analysis the identical design structures and obtain useful information about the design such as: commonly-used design patterns, most frequent design defects, domain-specific patterns, and reused design clones researchers propose a sub-graph mining-based approach. This would help developers to improve their knowledge about the software architecture. (45)
- Researchers identified that an automated refactoring to design patterns enables significant contribution to design pattern even for developer with little experience on the use of the required patterns. (20)

Agile Development Process and Refactoring: A case study on the impact of the software refactoring on the quality and productivity of the software product in an agile team identified that refactoring is a powerful mechanism to ensure the success of a system. (29, 56)

- In the study to investigate the three XP engineering activities: new design, refactoring and error fixing, the researchers figure out the more the new the design performed to the system the less refactoring and error fix needed. (49, 58)

Design Pattern and Refactoring: Design pattern, micro pattern, anti-pattern, code smell and software refactoring are highly intertwined concepts. Researchers in this area discovered different techniques and determine the association between these techniques and refactoring and show which technique to use at which time and how. (12)

- To follow a new approach to introduce parallelism using advanced refactoring techniques coupled with high-level parallel design pattern. The proposed refactoring approach can use these design patters to restructure programs defined as networks of software components that are more suited to parallel execution. (22)
- Refactoring of architectural models based on anti-patterns, that aims at providing performance improvement is proposed using a Role-based modeling language to represent pattern problems and anti-pattern solutions. (27)

- Researchers figure out many negative changes causing anti-patterns, at the package, class, and method levels. They also identified and grouped the causes into three basic categories: knowledge problems, artifacts problems and management problems. And, these problems can be easily prevented by promoting awareness of anti-patterns to the developers. (44)
- Software quality can be improved either by using collecting metrics scores for a given design or using design pattern. A study demonstrates that these two approaches are indeed congruent. (43)

Test Driven Development and Refactoring: Test driven development (TDD) is recently discovered new development approach with the motto test first. The approach has three phases red, refactor and green. Researchers already defined how and when to apply refactoring with unit testing to make the TDD more effective and efficient. (15)

- Refactoring and unit testing are hand in glove as the practice showed; behind each refactoring there is a rigorous testing to check the internal behavior. Studies indicate that unit testing is the commonly used efficient technique to validate the preservation of the internal behavior of the source code after refactoring. (26, 28)
- A TTCN-3 (Testing and Test Control Notation Version 3) development environment can notably provide suitable metrics and software refactorings to enable the assessment and automatic restructuring of test suites. (48)
- Aspectualization is an aspect-based refactoring which involves moving program code that implements cross-cutting concerns into aspects. One of the problems in aspectualization is the luck of an appropriate testing methodology. To overcome this problem, researchers proposed an iterative test driven approach for creating and introducing aspects. (29)
- Developers and researchers invented a framework for adaptation of clients and unit tests for all primitive refactorings that are defined by W. F. Opdyke. (31, 35)

Software Refactoring and Software Evolution: How the evolution of changes at source code line level can be inferred from CVS (Concurrent Versions System) repositories, by combining information retrieval techniques and the levenshtein edit distance. The application of this technique to the ArgoUML case study indicates a high precision and recalls one. (23)

- There is a proposed technique that combined visualization to identify software evolution patterns related to user requirement which can shows the evaluation metrics of a software system together with the implementation of its requirement. (32)
- Library developers need to evolve a library to accommodate changing requirements often face a dilemma, to address this dilemma they presented a lightweight approach for evolving application program interface (API) which does not depend on version control or configuration management system. (23)
- Researches distinguished three kinds of adaptation (evolution) according to their properties; namely refactoring, construction, and destruction. (54)
- A refactoring operation can only work correctly if all the code that needs to be changed is available to the integrated development environment (IDEs). But this cannot be always true for application programming interface (API) evolution or team development. To support refactoring in API and team development some studies suggested an extended IDE and version control to allow refactoring –aware merging and migration. (34)

Software Metrics and Refactoring: Some researchers proposed a set of metrics that suggest how code clones can be refactored. The tool gives metrics that indicates the need for certain refactoring methods rather than suggesting the refactoring methods themselves. (8)

- How the external qualities (robustness, reusability and performance) are affected with refactoring has studied theoretically and empirically. In this regard, the finding contradicted each other, some clam refactoring improve external qualities but some refuse the idea refactoring improve external qualities. (13)
- Most researchers recommend the need of a better validation technique than having the best refactoring tool to increase quality and productivity in software development. (16)
- A retrospective case study shows and shed light on how refactoring affects maintainability of software product. Furthermore, it indicates a positive impact on the coupling relationship with dependent software application. (17)
- Well known software metrics are proposed to evaluate the code and design quality of a system; to analyze the impact of refactoring on the quality of the system. (24)
- A taxonomy/classification of refactoring methods for aspect oriented programming based on their measurable effect on software quality attributes is proposed. (25)
- Researchers identified the dependency between the application area and the criteria for software quality measurement. As a result they figure out the criteria such as maintainability, comprehensibility and extensibility are the important once in refactoring. (37)
- Some researcher proposed a technique that uses software metrics to automatically detect refactoring opportunities. The technique measure the consistency of activity labels, the extent to which process overlap and the type of overlap that they have. (58)

Programming Language and Refactoring: A new preprocessor directive is proposed to embracing the C preprocessor directive during software refactoring. (9)

- Whenever the number of available refactoring increase researchers apply a genetic algorithm to generate the best refactoring schedule within a reasonable time to cope with the problem of refactoring techniques scheduling. (11)
- An appropriate refactoring approach identifies spot to be refactored and it suggests how they should be imported. Researchers proposed an approach that requires a light weight source code analysis for measuring several metrics, which can be applied to middle-or large-scale software system. This technique can make the refactoring process more effective and efficient. (14)
- Developers developed and used tidier to clean up and restructure various applications of Erlang source code and have tested it on many open source Erlang code based of significant size. (20)
- Binary refactoring browser for java aims to capture modifications that are often applied to source code, although they only improve the performance of the software application and not the code structure. (4)
- Refactoring in C programming language is complex because of the preprocessor directive that coexists with the code. New refactorings are proposed that defines preconditions and execution rules to maintain correctness of refactoring in the presence of macros and conditional directives. (18)
- A dynamic type of system such as small talk programming language offers little information about dependencies in the program that can be used by refactoring tool. So that, researchers come up with a solution of using static analysis for performing type inference to gather information about the dependencies in the program's source code and propose a prototype refactoring tool for small talk systems. (19, 46)

- It is important to change the data format from XML to JSON for efficiency purposes since JSON has light weight in nature and native support for JavaScript. There is a proposed refactoring system (X to J) to safely assist programmers migrate existing AJAX-based application utilizing XML into functionally equivalent AJAX-based applications utilizing JSON. (53)
- To migrate object-oriented systems to aspect-oriented ones, specific refactoring for aspect should be used which is complex and tedious task for the developer. Therefore, researcher contributed an expert software agent that can assist the developers in guiding how the refactorings should be applied and under what context. (4)

5.3. Gaps in Software Refactoring Researches

In this section, we presented the analysis result of the 58 papers that discussed on the gaps of the software refactoring researches.

Survey of Software Refactoring: There are between seventy and hounded refactoring techniques that can be apply in different situation. Identifying the most frequent types of refactoring can narrow down the refactoring options and it needs future study. (4, 19)

- Deviation from good design principles manifesting as code smell, which is a problem in the internal structure of the system. Researcher studied the identification of code smells to remove them with refactoring. But, this area still needs a compressive and historical review to find out the current state of the art and to identify the open challenges, current trends and research opportunities. (50)

Software Refactoring Tools: Extending the implementation to automatically synthesize aspects and pointcut expressions to produce aspect code is important. (14)

- No refactoring tool vender able to provide custom refactorings that can fit for all specific user needs because the possible number of refactoring is unlimited. Therefore, a customizable refactoring tools based on the demand of the developer is in demand. (5)
- Refactoring activities are semi-automated with tools, which should make it easier for programmers to refactor quickly and correctly. However many tools do poor job of communicating errors triggered by the refactoring. Besides that, they sometimes refactor slowly, conservatively and incorrectly, which need to be resolved. (38)

Bad Smell and Refactoring: Out of the 22 code smells Fowler identified, researchers' discovered that duplicate code smell received the most research coverage and Message chain received little concern in the research activity. Generally, there is little evidence to justify the use of code smell in refactoring. (6)

- Tools and methods to automate code smell have very few studies, so that a grate attention needs to pay to this direction in the future. (33)
- Though, researchers identified the types of code smells and the corresponding refactoring techniques. It is still a problem area to determine which code smell is effective to indicating the need of refactoring and what type of refactoring, with less programmer involvement is necessary. (43)
- In spite of the many discussion and claims about code smell, there are few empirical studies to support or contest the ideas behind code smell. In particular, the study of the human perception of what is a code smell and how to deal with it has been mostly neglected in the past. (51)

- Although, some developers propose code smell detectors that give programmers a quick, high-level overview of the code smells, there is no well accepted tool to detect and resolve code smell. (21)

Software Artifacts and Refactoring: To benefit from refactoring and implement it consistently, guidelines are needed which contain and support system refactoring. At the moment, establish these guidelines need the researchers' attention. (30)

- There are no guidelines to help aspect oriented software designer to decide which refactoring methods to apply and to optimize a software system with regard to certain design goal. (41)
- Refactoring can be applied on different software artifacts, how to do refactoring of testing artifacts; requirement and design model are an open area for study. (1, 37)

Agile Development Process and Refactoring: Refactoring does not fit very well in the linear waterfall or incremental spiral model of software engineering. Therefore, how refactoring can be included in the classical software refactoring development process need further study. (29)

- Researchers investigated Extreme Programming (XP) approach and they observed that the error fix effort is related to the number of days spent on each story, but relation between the refactoring effort and number of day's spent on each story was not conclusive, so that it needs a further study. (58)

Design Pattern and Refactoring: Though, the types, techniques and relationship among design patterns, micro-patterns and refactoring have been studies in detailed, how far they can be used in small, medium and big project and/or company need further study. (27)

Test Driven Development and Refactoring: Refactoring is the heart of the Test Driven Development (TDD), so that after and before the refactoring conducted a unit test should be applied. Writing the test may not fit with the code after refactoring if the test unit is logically dependent on the code. Consequently, how to write a test unit that cannot be logically dependent on the code need further study. (48, 55)

- Although unit testing support evaluates and changes in the application programs of a system and refactoring, there is currently a lack of testing strategies for database schema evolution. (35)

Software Refactoring and System Evolution: In software refactoring task, someone can modify little or large part of the source code. The little part is easier to fix and the large part is difficult and have an effect on the whole development process. Studying what fraction of code modification is refactoring or evolution needs further investigation. (32)

Software Metrics and Refactoring: There are many research works that use software metrics to validate software quality after software refactoring, but the output of this researches are inconsistent; so it is the researchers' responsibility to rectify the confusion by conduction more study on the external and internal quality of software and how they are affected. (13)

- Classifying refactoring method based on software quality attributes such as maintainability and reusability needs a theoretical and practical researches. (17, 25)

Programming Language and Refactoring: A sequence of software refactorings needs to be applied until the quality of the code is improved satisfactory. The final design after refactoring can vary with the application order of refactorings, thereby producing different quality improvements. It is necessary to determine a proper refactoring schedule to obtain as many benefits as possible. However, there is little on the problem of generating appropriate schedules to maximize quality improvement. (9)

- Currently there is a tremendous interest to apply refactoring on concurrency, parallelism, mobile platforms, web-based, cloud computing and GPU systems etc. (39)
- Despite its maturity and popularity, the C programming language still lacks tool support for reliably performing even simple refactoring techniques, browsing, or analysis operations. (53)

6. Discussions

6.1. Justifying the Evidence

In this section, we revisit the research question and discuss how the study results address them. To start the discussion let us remind the research questions once again here:

- How can we classify the software refactoring research activities?
- To identify the significant contributions of software refactoring literatures?
- To identify the gaps in software refactoring literatures?

The collected literatures covered a wide range of areas in software refactoring topics as shown in Section 5.1. This is evidence to confirmation that refactoring is one of the areas in software engineering, which attract researches attentions from different groups. The magnitude of the papers in each group can show the research intensity in each group, in this regard refactoring tool, agile, bad smell and design pattern have got the highest research coverage. On the contrary, Programming language, system evolution and survey have received less research consideration. We did the classification as a preprocessing step to lay the ground for the upcoming summarization task. The identification of the significant contribution and gaps was done respect to the established group during the classification for the sake of convenience to making the result reporting easier. In addition, the comparative study and assorting the similarity and difference among the various groups become simple.

6.2. Limitation of this Review

Here, we discuss the most important threats to the outcome quality of the systematic literature review. We use few search terms to formulate the query, but it is possible to use more key word to search the electronic database such as program analysis, program transformation, software evolution and reengineering that can increase the likelihood of more literatures into the study. The second area that may expose to threat is the inclusion and exclusion step of the study. Even though, the inclusion and exclusion criteria are well formulated and validated, the process was so subjective, especially when there was disagreement, the decision was done through discussion. In this case, it is better to have more group members to rich to a valid decision. Thirdly, the data extraction process was so difficult, even if the data extraction form was well designed. This is due to the high number of information pieces, that may lead to missing of information, since only have two persons for the data extraction.

7. Conclusion and Future Work

In software development, refactoring is highly desirable to assure the quality of the software process and product. This study is conducted to reveal the trends, opportunities and challenges of software refactor researches using systematic literature review. Although, this study is an extension of a previous work: classification and summarization of software refactoring researches, it has large coverage in terms of the number of literatures as well as the method used. Subsequently, we believe that the outcome of this study is better in quality and validity.

The study used nine prestigious electronic databases to collect the required literatures for the systematic literature review (SLRs). A total of 1358 papers are retrieved at early stage of the searching process, which later reduced to 186 and 58, once inclusion - exclusion criteria and quality assessment questions are applied respectively. To establish the ten slot classification scheme, we analyzed the title, abstract and conclusion section of the 186 papers. For the summarization part, we examined the whole section of the 58 papers and extract information using data extraction form. The information mined in this way, was used to articulate the significant contribution and gaps of the software refactoring researches as shown in Section 5.2 and 5.3.

For the last fifteen years researcher contribute a lot to the field of software refactoring, but there are still a great deal of unresolved issues, which need to be addressed in the future. Thereby, the identified gaps and even the significant contributions can guide the researchers, where to focus or not to focus. This can save the researchers time, effort and resources, at the same time it can reduce the redundancy and reinventing the wheel approach. Finally, this study can be extended in the future by incorporating literatures outside the electronic databases such as gray literature, books, dissertation, thesis and tutorials. Moreover, integrate this study with practices in the software industry can be a future work to increase the validity and credibility.

References

[1] D. M. Coleman, D. Ash, B. Lowther and P. Oman, “Using Metrics to Evaluate Software System Maintainability”, IEEE Computer, vol. 27, no. 8, **(1994)**, pp. 44-49.

[2] T. Guimaraes, “Managing Application Program Maintenance Expenditure”, Comm. ACM, vol. 26, no. 10, **(1983)**, pp. 739-746.

[3] H. Liu, Z. Ma, W. Shao and Z. Niu, “Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort”, IEEE Transactions on Software Engineering, **(2012)**.

[4] A. Garrido, G. Rossi and D. Distante, “Refactoring for Usability in Web Applications”, Comm. Software, IEEE, **(2001)**.

[5] I. Balaban, F. Tip and R. Fuhrer, “Refactoring Support for Class Library Migration”, ACM, **(2005)**.

[6] M. Lehman, “Laws of Program Evolution - Rules and Tools for Programming Management”, Proceedings Infotech State of the Art Conference, Why Software Projects Fail, vol. 11, **(1978)**, pp. 1-25.

[7] M. Lehman and J. Ramil, “Rules and Tools for Software Evolution Planning and Management”, Annals of Software Engineering, vol. 11, no. 1, **(2001)**, pp. 15-44.

[8] R. S. Arnold, “Tutorial on Software Restructuring, An Introduction to Software Restructuring”, IEEE Press, **(1986)**.

[9] W. G. Griswold, “Program Restructuring as an Aid to Software Maintenance”, Ph. D. Thesis, University of Washington, **(1991)**.

[10] M. Fowler, “Refactoring: Improving the Design of Existing Programs”, Addison-Wesley, **(1999)**.

[11] R. Shantnawi and W. Li, “An Empirical Assessment of Refactoring Impact on Software Quality Using a Hierarchical Quality Model, International Journal of Software Engineering and Its Applications, vol. 5, no. 4, **(2011)**.

[12] W. F. Opdyke, “Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks”, Ph. D. Thesis, Univ. of Illinois at Urbana-Champaign, **(1992)**.

[13] D. Arcelli, V. Cortellessa and Trubianic, "Antipattern-based Model Refactoring for Software Performance Improvement", The 8th International ACM SIGSOFT Conference, **(2012)**.
[14] S. Demeyer, S. Ducasse and O. Nierstrasz, "Object-Oriented Reengineering Patterns", Morgan Kaufmann and DPunkt, **(2002)**.
[15] http://sourcemaking.com/refactoring/defining-refactoring.
[16] R.d Mateosian, Software Development Patterns, IEEE, **(2008)**.
[17] T. Mens, "A Survey of Software Refactoring", IEEE, vol. 30, no. 2, **(2004)** February.
[18] K. Elish, M. Alshayeb and O. Karim, "A Classification of Refactoring Methods Based on Software Quality Attributes", Arab J Sci Eng., vol. 36, **(2010)** May, pp. 1253-1267.
[19] B. Du Bois, P. Van Gorp, A. Amsel, N. Van Eetvelde, H. Stenten and S. Demeye, "A Discussion of Refactoring in Research and Practice, **(2006)**.
[20] P. Meananeatra, "Identifying Refactoring Sequences for Improving Software Maintainability", ACM 978-1-4503, **(2012)** September.
[21] T. Mens and A. Van Deursen, "Refactoring: Emerging Trends and Open Problems", **(2003)** October 22.
[22] M. Abebe and C.-J. Yoo, "Classification and Summarization of Software Refactoring Research: A literature Review Approach", Advanced Science and Technology Letters, vol. 6 (Games and Graphics 2014), pp. 279-284.
[23] Systematic Reviews: CRD's Guidance for Undertaking Reviews in Health Care, University of York, **(2008)**.
[24] B. Kitchenham and O. Pearl Brereton, Systematic Literature Reviews in Software Engineering -A Systematic Literature Review, Information and Software Technology, vol. 51, **(2009)**, pp. 7-17.
[25] J. Biolchini and P. Gomes Mian, "Systematic Review in Software Engineering", System Engineering and Computer Science Department COPPE/UFRJ, **(2005)** May.

Appendix A: Studies included in the systematic review

No	*Digital Database*	*Author*	*Year*
1	Refactoring of a Database	Ayeesha et al	2009
2	A Field Study of Refactoring Challenges and Benefits	Miryung et al	2012
3	A Survey of Software Refactoring	Tom Men et al.	2004
4	A Survey of software testing in refactoring based software models	Pandimurug	2011
5	An expert system for determining candidate software classes for refactoring	Yasemin etal	2008
6	Bad Smelling Concept in Software Refactoring	Ganesh B.	2009
7	Design Patterns in Software Development	MU Huaxin	2011
8	Empirical investigation of refactoring effect on software quality	Mohammad	2009
9	Empirical Support for Two Refactoring Studies Using Commercial C# Software	M. Gatrell, ,	2011
10	Evaluating software refactoring tool support	Erica Mealy	2006
11	Formal specification of extended refactoring guidelines	Wafa Basit,	2012
12	From Software Architecture to Design Patterns	Jing Wang,	2005
13	Identifying Refactoring Sequences for Improving Software Maintainability	Panita	2012
14	Improving Usability of Software Refactoring Tools	Erica Mealy.	2007
15	Investigating the Effect of Refactoring on Software Testing Effort	Karim O.	2009
16	Quantifying Quality of Software Design to Measure the Impact of Refactoring	Tushar S	2012
17	Refactoring – Does it improve software quality	Konstantinos	2007
18	Refactoring : Emerging Trends and Open Problems	Tom Mens	2003
19	Refactoring Practice: How it is and How it should be Supported	Zhenchang	2006
20	Refactoring Tools: Fitness for Purpose	Emerson	2008
21	Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort	Hui Liu,	2012
22	Software refactoring at the package level using clustering techniques	A. Alkhalid,	2010
23	Strengthening Refactoring: Towards Software Evolution with Quantitative	Sérgio Bryto	2009
24	Using Software Metrics to Select Refactoring for Long Method	Panita	2011
25	A Classification of Refactoring Methods Based on Software Quality Attributes	Karim O.	2011
26	A test case refactoring approach for pattern-based software development	Peng-Hua	2012
27	Anti-pattern Based Model Refactoring for Software Performance	Davide	2012
28	Automated Acceptance Test Refactoring	RodrickBorg	2011
29	A Role for Refactoring in Software Engineering?	Tony Clear	2005
30	Drivers for Software Refactoring Decisions	Mika V.	2006
31	Testing During Refactoring: Adding Aspects to Legacy Systems	Panita	2006
32	Impact of Refactoring on Quality Code Evaluation	Francesca	2011
33	Perspectives on Automated Correction of Bad Smells	Javier Pérez	2009
34	Rank-based refactoring decision support: two studies	Liming Zhao	2011

35	Refactoring with Unit Testing: A Match Made in Heaven?	Frens	2012
36	The Impact of Refactoring to Patterns on Software Quality Attributes	Mohammad	2011
37	UML model refactoring: a systematic literature review	Mohammed	2013
38	Improving usability of refactoring tools.	Emerson Murp.	2006
39	Use, Disuse, and Misuse of Automated Refactorings	Mohsen Vakili	2012
40	Object-Oriented Reengineering Patterns	S. Demeyer	2002
41	Software Development Patterns	Richard Mateos	2008
42	An Approach to Refactoring of Executable UML Models	Dobrza	2006
43	Building Empirical Support for Automated Code Smell Detection	Schumacher	2010
44	Patterns for Refactoring to Aspects: An Incipient Pattern Language	Monteiro	2007
45	Refactoring the Documentation of Software Product Lines	Romanovsky	2011
46	Assessing the Impact of Refactoring Activities on the JHotDraw Project	Thapa	2010
47	Getting the Most from Search-based Refactoring	O'Keeffe	2007
48	Refactoring Test Suites Versus Test Behavior: A TTCN-3 Perspective	Counsell	2007
49	Refactoring-aware Version Control	Freese	2006
50	What Programmers Say About Refactoring Tools?: An Empirical Investigation of Stack Overflow	Pinto, Gustavo H	2013
51	On the refactoring of activity labels in business process models	Leopold	2012
52	Refactoring Towards a Layered Architecture	M Cornelio	2005
53	Refactoring legacy AJAX applications to improve the efficiency of the data exchange component	Ying, Ming and Miller	2013
54	Identifying refactoring opportunities in process model repositories	Dijkman, Remc	2011
55	Connection between Safe Refactorings and Acceptance Test Driven Development	Fontela, C. and Garrido, A.	2013
56	How We Refactor, and How We Know It	Murphy-Hill	2012
57	Tools for making impossible changes - experiences with a tool for transforming large Smalltalk programs	Roberts, D. and Brant, J.	2004
58	Agile programming: design to accommodate change	Thomas, D.	2005

Appendix B: Data Extraction Form

No	*Attribute*	*Description*
1	Study Identifier	Unique id for the study (e.g. DS1)
2	Date of data extraction	The data the extraction conducted (dd/mm/yy)
3	Name / Id of the data extractor	The person who does the data extraction activity
4	Reference citation	Author, year, title and source.
5	Publication type	Journal articles or conference papers
6	Aim of the study	What were the aims of the study?
7	Theoretical background	What were the objectives of the study?
8	Methodology of the study	What were the methodologies of the study?
9	Definition of Software refactoring	How it is defined in the context of the papers?
10	How refactoring is relate to the study	What other software r engineering concept is mentioned?
11	Short description	A short description about the study activity
12	The research activity strategy	What strategy they use to conduct the reassert?
13	Finding and Conclusion	What are the finding and the conclusion?
14	Contribution to Software refactoring	What is the significant contribution to ward software refactoring?
15	Gaps to Software refactoring	What is the gap identified toward Software refactoring?