# ANALYSING THE EFFECTS OF REFACTORING ON SOFTWARE QUALITY ATTRIBUTES

A DISSERTATION

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE AWARD OF DEGREE
OF

MASTER OF TECHNOLOGY
IN
**SOFTWARE ENGINEERING**

Submitted by:

**PRIYA SINGH**
**(2K16/SWE/10)**

Under the supervision of:

**DR. RUCHIKA MALHOTRA**
**(ASSOCIATE PROFESSOR, CSE)**
**Department of CSE, DTU**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi – 110042

JUNE 2018

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi – 110042

# CANDIDATE'S DECLARATION

I, PRIYA SINGH, 2K16/SWE/10 a student of M.TECH (Software Engineering) declare that the project Dissertation titled "Analysing the Effects of Refactoring on Software Quality Attributes" which is submitted by me to Department of Computer Science and Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma, Fellowship or other similar title or recognition.

Place: DTU, Delhi PRIYA SINGH

Date: (2K16/SWE/10)

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi – 110042

# CERTIFICATE

I, hereby certify that the Project titled "Analysing the Effects of Refactoring on Software Quality Attributes" submitted By PRIYA SINGH, Roll number: 2K16/SWE/10, Department of Computer Science and Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is a record of project work carried out by the student under my supervision. To the best of my knowledge, this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Place: DTU, Delhi
Date:

**Dr. RUCHIKA MALHOTRA**
(Associate Professor, CSE, DTU)
SUPERVISOR

# ACKNOWLEDGEMENT

I am very thankful to Dr. Ruchika Malhotra (Associate Professor, Computer Science Eng. Dept.) and all the faculty members of the Computer Science Engineering Dept. of DTU. They all provided immense support and guidance for the completion of the project undertaken by me. It is with their supervision that this work came into existence.

I would also like to express my gratitude to the university for providing the laboratories, infrastructure, test facilities and environment which allowed me to work without any obstructions.

I would also like to appreciate the support provided by our lab assistants, seniors and peer group who aided me with all the knowledge they had regarding various topics.

**PRIYA SINGH**
**M.TECH (SWE)**
**2K16/SWE/10**

# ABSTRACT

Bad-smell indicates code-design flaws and poor software quality that weaken software design and inversely affects software development. It also works as a catalyst for bugs and failures in the software-system. Refactoring methods are used by software practitioners as corrective actions for bad-smells. Refactoring is not only limited to removing bad-smells but it does have a strong correlation with quality attributes. Countless studies are present in the literature that studies the effect of refactoring-methods on software quality attributes. It is said to improve certain aspects of quality. Also, refactoring is a costly activity and the problem relies upon the fact that there are over seventy refactoring-methods available in literature and multiple refactoring methods can be used to nullify the effect of a particular bad-smell. So, it becomes very difficult to apply refactoring on complete source-code and almost impossible if software size is dramatically large. Thus, there arises a need for prioritizing classes in some way. This study aims to first provide a systematic literature review on the correlation of refactoring-methods and bad-smells and their improvement on internal as well as external quality attributes and second, it comes up with a way to apply refactoring to only severely affected classes to improve the overall software quality. The systematic literature review helps software developers in identifying the commonly prevalent bad-smell, their possible refactoring solution and effect of those refactoring methods on software quality attributes and guide the researchers in conducting future research. In the end, a framework is proposed that detects a small subset of classes from the entire

source-code instantly require refactoring. This prioritization of classes is based on two factors-severity of the presence of bad-smells and object-oriented characteristics. The approach is evaluated on eight open-source software-systems written in Java using ten most common bad-smells and six widely known Chidamber & Kemerer metrics. Both these factors help in calculating a new metric Quality Depreciation Index Rule (QDIR) that exposes those classes that are highly affected by bad-smells and demand an immediate refactoring solution. Results of the empirical study indicate that QDIR is an effective metric to remove bad-smells in an environment of stringent time constraints and limited cost, making the maintenance of software-system easier and effective with enhanced software quality.

# CONTENTS

| Title | Page No. |
|---|---|

# LIST OF TABLES

| Table No. | Title | Page No. |
|---|---|---|

# LIST OF FIGURE

| Figure No. | Title | Page No. |
|---|---|---|

# LIST OF ABBREVIATIONS

| | |
|---|---|
| **BoB** | **B**ase Of **Bad-smell** |
| **BoM** | **B**ase **o**f **M**etric |
| **C&K** | **C**hidamber **&K**emerer |
| **MOOD** | **M**etrics for **O**bject **O**riented **D**esign |
| **MV** | **M**etric **V**alue |
| **SLR** | **S**ystematic **L**iterature **R**eview |
| **SW** | **S**mell **W**eightage |
| **QDIR** | **Q**uality **D**epreciation **I**ndex **R**ule |
| **QMOOD** | **Q**uality **M**etrics for **O**bject **O**riented **D**evelopment |

# CHAPTER 1

# INTRODUCTION

Refactoring is a powerful technique to improve the internal source-code structure. It is considered to improve the code and make it less error-prone. It also helps in achieving better software quality by removing bad-smell, a design flaw poorly affecting the source-code. Research in the field of bad-smell and refactoring is active in recent years. Researchers are continuously identifying ways to detect bad-smells in the source-code and remove them by coming up with automatic and semi-automatic approaches to apply refactoring methods. Removing certain bad-smells by applying refactoring methods improves the internal design of code and its quality attributes (internal/ external/both). As software maintenance is a vital phase in software-development lifecycle because of the need of keeping the software-system operational over years, software practitioners are also relying on refactoring in order to keep their software maintainable and easily understandable.

There are seventy refactoring methods and over twenty-two bad-smells identified by Fowler [1] in his famous book in 1999. Since then, new refactoring methods have also been detected by the software researchers. Refactoring is a tedious activity involving a deep understanding of the code and good refactoring skills. Moreover, if software size is large, it is not possible to detect bad-smell and then apply refactoring to every possible portion of the source-code. So, there is a need to prioritize source-code portions that require refactoring solutions as early as possible so that satisfactory results can be achieved within time and budget.

In this study, we first provide a systematic literature review (SLR) to assess and compare the results of the primary studies that empirically evaluate the refactoring methods affecting internal and external quality attributes. Second, we provide a mechanism to remove bad-smells on a priority basis in order to apply refactoring to only a small portion of the source-code that has poor design characteristics and is severely affected by bad-smells.

## 1.1 BACKGROUND & MOTIVATION

Software quality is defined as the degree to which a given software-system, product or process adheres to desired requirements imposed on it and meets user's expectations. By quantifying the software quality, a better estimate of how a software-system adheres to specified requirement and user expectations can be made. The term "quality software" normally includes product-attributes of the software-system like maintainability, reliability, reusability, adaptability, completeness, and understand-ability [2]. An attribute means a property or characteristic of some entity (like number of lines in a class or time required in class-testing). One can measure an attribute either directly or indirectly: In the direct measurement of attributes of a source-code component, one can easily calculate it by using the elements that make up the syntax or behavior of the code-component [3]. On the other hand, in indirect measurement of an attribute of a source-code component, one cannot calculate it simply in terms of the elements that make up the syntax or behavior of the code-component. It requires calculating one or more direct attributes first [2]. There are plenty of object-oriented metrics proposed by researchers in the past to measure these quality aspects like MOOD's metrics [4], Chidamber & Kemerer metrics [5], Lorenz & Kidd metrics [6], Li & Henry [7] to name a few.

According to Fowler and Beck [1], a bad-smell is a poor internal structure prevailing in the source-code that does not stop it from executing but weakens its design and serves as a ground for bugs thereby degrading the overall software quality and increases the overhead of software maintenance. Fowler [1] has given a catalog of twenty-two code bad-smells that marks flaws in the design and if not handled, can cause serious damages.

Refactoring is regarded as a solution to bad-smells and can be defined as a technique that refines the internal structure of the source-code without compromising its functionality. As most of the cost in the entire software lifecycle is spent in maintaining it rather than making it operational, refactoring appears promising. It tends to reduce the source-code complexity and improve readability, maintainability and overall software quality. Fowler [1] has identified over seventy refactoring methods specific to the design flaws existing in the code. Refactoring is complex, error-prone and tiresome in nature that is why it is not possible to refactor all available classes in the software-system and there is a need to prioritize them in some way.

Many researchers have actively addressed the issue of bad-smells and their refactoring solutions in recent years. Most of them concentrate on removing one or more bad-smells, identifying opportunities of a particular refactoring, examining the impact of refactoring activities on software's internal as well as external quality, studying the overall effect of refactoring on overall code quality and developing tools to detect and apply refactoring. But there are only a few research articles that focus on removing bad-smells on a priority basis. And even if a few studies have tried to address it, they have not considered the severity of presence of bad-smells and object-oriented design characteristics at the same time. The current research addresses the issue by taking into consideration both the object-oriented characteristics and severity of presence of bad-smells in classes to prioritize them and hence achieve better software quality saving significant time and cost.

The motivation of this research work arrives from the need of a systematic literature review that can help the software practitioners in determining which refactoring methods are suitable for removing a particular design flaw in the source-code and improving certain internal (cohesion, coupling, inheritance, polymorphism etc.) and external (maintainability, adaptability, reusability etc.) that can ease out the maintenance phase activity effectively as well as guide the researchers in determining bad-smells and refactoring methods that are highly and least studied; techniques, tools and methodologies commonly adopted by researchers, overall benefit of refactoring methods on quality attributes and limitation of present literature. Also, we are motivated to prioritize classes based on the severity of presence of bad-smells and object-oriented characteristics of the software-system so that refactoring can be applied to only highly severe classes (generally a very small portion as compared to complete software-system) that help in achieving better software quality within budget in timeliness manner.

## 1.2 RESEARCH OBJECTIVE

The goal of this research work is to first provide a systematic literature review to answer questions related to effect of refactoring methods on software quality attributes (complete process and research questions addressed are discussed in detail in Chapter 3) and then provide a way to prioritize classes present in the software-systems so that refactoring can be applied to extremely severe classes in order to save overall maintenance effort and at the same time improve code-quality by removing bad-smells and improving object-oriented characteristics of the software. This work is based on the eighty-twenty principle that states that one can improve 80% software quality by concentrating on 20% of classes to apply refactoring on. For this purpose, a new metric, Quality Depreciation Index Rule (QDIR) with certain modifications to that proposed by Malhotra et al. [8] in 2015 is proposed. QDIR is computed using two other metrics- Base of Bad-smell (BoB) and Base of Metric (BoM). They have considered only four bad-smells in their preliminary study but in the current research work, we have selected ten bad-smells. The reason behind the selection of a wider set of bad-smells is taking into consideration the ill-effect of a larger number of bad-smells that are commonly present in the software-systems and are having proper tool support for their detection. This leads to reformation of BoB metric that helps us in ultimately improving the QDIR metric to remove the effect of a larger number of bad-smells from the studied software-systems. We have selected eight open-source software-systems having varying domains and sizes (medium-large) written in Java language to empirically evaluate the effort saved in prioritizing classes based on QDIR in contrast to only a single medium-sized software-system used by Malhotra et al. [8] in their preliminary study. Ten bad-smells and six C&K metrics [5] to capture object-oriented characteristics are selected to compute QDIR. Following research questions are addressed in the current research:

*RQ1: What percentage of classes is poorly affected by bad-smells?*

*RQ2: Which bad-smell predominantly harms classes in the selected software-systems?*

*RQ3: Is Quality of Depreciation Rule (QDIR) useful in providing treatment to critically affected class?*

*RQ4: Do use of Quality of Depreciation Rule (QDIR) leads to a reduction in Maintenance Effort?*

## 1.3 PROPOSED WORK

To conduct the systematic literature review, 104 primary studies from January 1991 to January 2018 were selected from premier journals and reputed conferences that apply refactoring methods on object-oriented systems to evaluate their effects on software quality attributes. The results from various primary studies are compared based on their impact on quality attributes and their statistical significance is also considered. To prioritize classes based on Quality Depreciation Index Rule (QDIR), eight open-source medium to large sized software-systems from various domains are selected for generality and wider-acceptability of the results. Object-oriented characteristics are captured by C&K metric-suite [5]. Ten most probable Bad-smells are identified for each class. Then, the combined effect of both the object-oriented characteristics and severity of presence of bad-smell is considered to calculate a new metric, QDIR to prioritize classes for the application of refactoring methods.

## 1.4 ORGANIZATION OF THESIS

The rest of the thesis is divided into following Chapters.

**Chapter 2** discusses the related work in the field applying refactoring to remove bad-smells and thereby improve software quality. **Chapter 3** provides Systematic Literature Review in the field of Bad-Smells & Refactoring. **Chapter 4** explains the research methodology adopted in the current research work. **Chapter 5** answers the research questions imposed at the start of the research work and discusses the results in detail. **Chapter 6** discusses threats to validity and in the end; **Chapter 7**concludes the research work with inferences drawn from analysis and gives directions for future work.

# CHAPTER 2

# RELATED WORK

The research in the field of refactoring and bad-smells is quite popular. There are a wide number of research articles that discuss bad-smells and refactoring or effect of refactoring on object-oriented characteristics. In the current work, we first provide a systematic literature review on the effect of refactoring methods on various internal and external quality attributes. Second, we provide a way to prioritize classes for refactoring based on the severity of presence of bad-smells and design characteristics of the source-code.

We identified many systematic literature reviews (SLRs) in the field of bad-smells and refactoring. Wangberg [9] in his master thesis reported results from the review on bad-smells & refactoring. The results showed that most of the selected studies are design-research contribution i.e. bad-smell detection's methods, related tools and refactoring support which account for sixty-one percent of the studies in contrast to only twenty-four percent of empirical work in the field of refactoring. Among these, only 13.8% accounted for any type of validation and again half of it has gone through thorough validation in the realistic setting. This SLR was conducted on studies prior to 2010 and its aim was not only to discuss empirical research in the field of refactoring. Moreover, there is an increase in empirical research in past few years. Dallal [10] conducted SLR to identify opportunities in refactoring object-oriented code with 47 selected primary studies. Most of the work in this area is done by academic researchers using non-industrial data which is generally open-source and repeatable too. These data-sets are generally small and generality of the results is in question. Results indicated that Extract Class and Move Method are very popular. They found that only 29.8% of studies are related to the evaluation of quality attributes. Singh and Kaur [11] conducted SLR on 238 primary studies w.r.t. refactoring in general and detection of code bad-smells and anti-patterns. Their SLR is an extension of Dallal's [10] work with large no. of primary studies. Their research indicated that god class and feature envy are mostly studied in the literature. Also, they identified that approximately 28% researchers used automatic detection of bad-smell while approximately 26% performed empirical studies in the field of refactoring. Abebe and Yo [12] conducted SLR on 58 studies since 1999 to reveal research pattern, important concerns and statistical information regarding published papers in the last fifteen years. They concluded that only 10.22% comprised of empirical work regardless of code and non-code refactoring applied which is second least after programming language and refactoring. Most of the work contributed to Refactoring tools and bad-smells. Cinnéide et al. [13] conducted a survey on benefits of refactoring to address various questions like whether it reduces bad-smells, whether applying refactoring methods improve non-functional requirements, whether there is an improvement in software quality metrics due to refactoring? They suggested making many direct measures of software quality like the number of defects, effort etc. in place of proxy -measures like maintainability-index, bad-smells, cohesion, coupling etc. and need to be more systematic in terms of the context of refactoring. Bassey et al. [14] performed analysis of empirical studies in refactoring of object-oriented systems based on the metric-based evaluation. They considered sixteen primary studies to identify the state-of-the-practice in finding refactoring opportunities by targeting, refactoring methods, coding language and their effect on software quality. They indicated move method and extract class are most commonly applied refactoring methods and software metrics help in detecting bad-smell and making decisions on applying refactorings. They suggested more empirical work should be done on languages other than java and a generic tool to detect refactoring opportunities and suggest where refactoring is required should be developed. Although it considered metric based measurements for empirically evaluating quality, it took very less number of papers and was very brief and non-systematic. Mens et al. [15] provided state-of-the-art in the field of refactoring based on criteria like available refactoring techniques, formalisms, and techniques for these refactoring, refactoring support for various software related artifacts etc. They followed a running example in the entire paper to explain important refactoring concepts. They also provided a list of open issues in the field of refactoring like appropriate tools, the impact of refactoring related activities on software processes etc. Dallal and Abdi [16] conducted an SLR to reflect the impact of refactoring-scenarios on quality attributes by taking seventy-six papers published before completion of 2015 from digital sources with the aim of identifying common refactoring methods, the impact of these refactoring methods on quality attributes, most commonly used data-sets. They concluded that move method and extract class are two most frequently used refactoring scenarios and the effect of refactoring scenarios on coupling and cohesion is widely analyzed. The results from their results are quite useful. This is the most significant research that overlaps with research under study. But they did not focus on identifying bad-smells that are common across the software-systems and identifying particular refactoring methods that positively affect certain quality attributes (internal and external). Also, they selected research papers published before 2015 and there are many relevant studies in recent times that empirically evaluate the effect of refactoring methods on internal and external quality attributes. Our study identified many such new relevant studies that address this issue and we decided to conduct an SLR that identifies common refactoring methods and studied bad-smells across relevant empirical studies, frequently used data-sets across empirical studies, statistical significance of results, commonly studied quality attributes, certain quality attributes improved by application of specific refactoring methods and overall effect of refactoring methods on certain quality attributes.

We also came across so many studies that studied techniques to identify and remove bad-smells and provide refactoring methods solution to source-code. Ouni et al. [17] proposed a technique on the basis of chemical reaction optimizations to expose appropriate refactoring methods that maximize the number of fixed riskiest bad-smells by also considering the preference of the software maintainer. They selected five medium to large sized open-source software-systems and studied seven different bad-smells to prioritize the bad-smell removal process. Results indicate that their meta-heuristic approach is better than other popular meta-heuristics in search-based software engineering as they use prioritization and preferences of maintainers to apply refactoring methods. Ouni et al. [18] also provided an approach to provide recommendations to fix prevailing bad-smells in source-code and improve software quality in terms of quality attributes by using same seven bad-smells and five software-systems used by Ouni et al. [17] with two additional software-systems, making a total of seven software-systems to conduct the experiment. Their approach was good for large size software-systems but they do not provide any prioritizing to apply refactoring methods. Fokaefs et al. [19] and Oliveto et al. [20] worked on removing feature-envy bad-smell. While Fokaefs et al. [19] came up with an Eclipse plug-in that helps in detecting Feature Envy Bad-smell and providing move method refactoring suggestion and its application, Oliveto et al. [20] provided an approach that studies two different types of relations-structural and conceptual between the methods of the source-code and identifies friend-methods (methods that share many responsibilities) using relational topic models to find target envied class which are probable to move the friend method to. Fokaefs et al. [19] evaluated the plug-in on two software-systems with a demonstration of the application of the plug-in but thesis limited to only java source-code and Oliveto et al. [20] evaluated it on a single project with preliminary results suggesting acceptability of their approach by suggesting appropriate refactoring suggestions in source-code. Higo et al. [21] generated a set of metrics to identify the way in which code-clones can be refactored. They also developed a tool-Aries based on their approach that gives metrics that are indicative of relevant refactoring methods instead of providing refactoring methods' themselves as suggestions. A case study of using the tool Aries is also provided on an open-source software which is quite simple to use. Fontana et al. [22] considered three smells- data class, duplicate code and god class and identified them in twelve open-source systems of different domains and sizes and focused on giving a recommendation on which design debt should be first paid, as per the bad-smells identified. Their study answers questions if it is possible to detect bad-smell that have more critical debts than others and if it is possible to detect such bad-smells that are domain-dependent and should not be regarded as bad-smell debt in particular domain. They suggested removing duplicate bad-smell prior to any other bad-smell stating it the most dangerous bad-smell out of the studied three smells. They also indicated improvement in complexity and cohesion metrics of the software thereby reducing maintenance effort. Also, they discussed that data class and god class bad-smells are dependent on the domain. Bavota et al. [23] proposed an approach to suggest portions of source-code that require extract class refactoring methods based on game theory. They evaluated it on two open-source java-based software-systems. The results indicated game theory approach superior to other two already existing approaches and applicability of preliminary results. Dallal [24] used univariate logistic regression technique that empirically investigates the capabilities of twenty-five metrics (belonging to size, cohesion, and coupling) to predict the classes requiring extract subclass refactoring method opportunity. The results reflected a strong correlation between some of the studied metrics and recommendation of whether extract subclass refactoring is required or not. Stroggylos and Spinellis [25] studied version control logs to identify refactoring requirement and examined how software code metrics vary correspondingly to assess whether refactoring method is effectively being used for enhancing software quality, particularly within an open-source software environment. Their results have enlightened that either the refactoring methods not always yield improvement in software quality or there is a lack of a proper way to be effectively apply refactoring methods on the developer's side. Malhotra et al. [8] proposed a novel approach to prioritize the classes demanding sudden treatment in terms of refactoring by giving equal weightage to both the design characteristics and bad-smells presence in the source-code. They developed a new metric to prioritize classes based on the severity of bad-smells and evaluated it on a single, middle-sized, open-source software-system written in java language considering C&K metric suite and four code bad-smells. The preliminary results show that by applying refactoring methods to only 10% of highly affected classes, 47% improvement can be achieved by saving quite a large amount of effort and cost. To prioritize classes to remove bad-smells and apply refactoring, this study adopts the idea of work done by Malhotra et al. [8]. We identified the need to address other bad-smells also that are commonly present in software-systems and have available tool support for their detection. For this, we have selected ten common bad-smells whose detection tools are easily available in the literature. This helps in removing the ill effect of a larger number of commonly prevailing bad-smells in the software-systems. Malhotra et al. [8] evaluated their approach on a single middle sized software-system written in java language whereas we evaluated the proposed approach on eight medium to large sized software-systems. Due to the inclusion of ten bad-smells in the current study, QDIR is modified and improved to take into consideration the severity of ten bad-smells instead of four selected by Malhotra et al. [8]. We have selected six other bad-smells along with four used by Malhotra et al. [8] in their preliminary study. The aim of the study is to prioritize the classes and provide refactoring methods as solutions to bad-smells to only those classes having both high severity of presence of bad-smells and poor object-oriented characteristics.

# CHAPTER 3

# SYSTEMATIC LITERATURE REVIEW

Software refactoring is a popular technique while maintaining a software-system to improve its complexity in terms of the internal structure without modifying how it behaves functionally. Quality attributes, both internal and external are considered to be indicators of overall software quality. In the past few years, research in the field of software refactoring to improve software quality by improving various internal and external quality attributes is quite active.

This systematic literature review (SLR) aims to assess and compare the results of the empirical studies in the field of refactoring that actively evaluate the refactoring methods affecting internal (coupling, cohesion, inheritance, abstraction, size etc.) and external quality attributes (understandability, maintainability, adaptability etc.) by applying refactoring methods on software-systems. Our study identified many relevant studies that address this issue and we were motivated to conduct an SLR that identifies common refactoring methods and studied bad-smells across relevant empirical studies, frequently used data-sets, statistical significance of the empirical results, commonly studied quality attributes, certain quality attributes improved by application of specific refactoring methods and overall effect of refactoring methods on certain quality attributes.

In this SLR, 104 primary studies from January 1991 to January 2018 were selected from premier journals and reputed conferences & workshops that apply refactoring methods on object-oriented systems to evaluate their effects on software quality attributes. The results from various primary studies are compared based on their impact on quality attributes and their statistical significance is also considered.

## 3.1 PHASES IN SYSTEMATIC LITERATURE REVIEW

The whole process of systematic review is planned, conducted and reported on the basis of the guidelines that are being provided by Kitchenham [26]. The complete process consists of three main phases viz. planning, conducting and reporting the review that is depicted in Figure 3.1. In the first phase, reasons for conducting review are identified and a review protocol is established. Review protocol encompasses identification of relevant research questions, designing search-strategy, adopting criteria for study selection, assessment of study quality, and the process of extracting data and finally the process of synthesis of data. It should be developed very wisely in order to avoid or at least reduce research bias in SLR. Once review protocol is developed, all the consequent steps to conduct SLR are carried out smoothly. In the second phase, research questions are stated that aptly answered the issues addressed by SLR. Then, the search strategy is identified that included identification of search term and key sources to be explored form where primary studies are captured. Inclusion-Exclusion criterion is set to include relevant primary studies in the context of empirical studies evaluating the effect of refactoring on quality attributes (internal external, or both) and exclude irrelevant studies w.r.t. context. Afterward, quality assessment is provided by developing a questionnaire of quality questions to assess the acceptability of primary studies for SLR. After the quality assessment is done, data extraction forms are prepared from the primary studies to gather all the important aspects of the studies in order to be able to answer the research questions. This extracted information is then synthesized i.e. tabulated in a very consistent way with the research questions in SLR. The third phase includes meaningfully and analytically reporting the findings from the SLR along with limitations and any future directions.

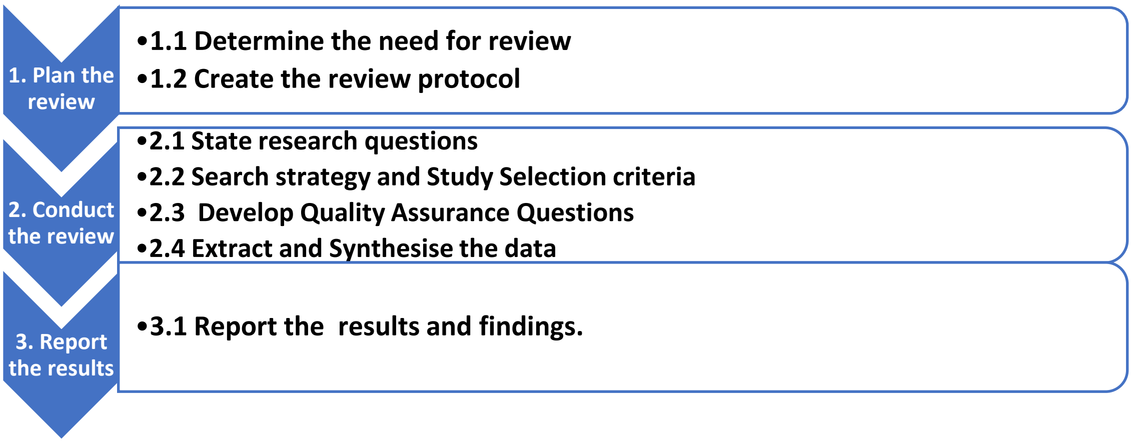

1. Plan the review
•1.1 Determine the need for review
•1.2 Create the review protocol
2. Conduct the review
•2.1 State research questions
•2.2 Search strategy and Study Selection criteria
•2.3 Develop Quality Assurance Questions
•2.4 Extract and Synthesise the data
3. Report the results
•3.1 Report the results and findings.

Figure 3.1 Phases in Systematic Literature Review

## 3.2 RESEARCH QUESTIONS

The aim of this SLR is to identify empirical evidence and evaluate consistency of results by primary studies that empirically evaluated performance of refactoring methods on various internal and external quality attributes, so that researchers, academicians, and developers all are aware of the implications of refactoring methods on quality attributes(internal, external, or both).

Table 3.1 presents six research questions addressed by the SLR. From primary studies, we identified refactoring methods applied by researchers (RQ1) and bad-smells detected in the primary studies (RQ2). Internal and external quality attributes that are evaluated empirically by the researchers are answered in RQ3, RQ3.1, and RQ3.2. Also, the software-systems/data-sets used by researchers to apply refactoring methods are identified (RQ4). RQ5 captures whether any statistical analysis or correlation analysis is being done to support the results. The impact of refactoring methods on various internal and external quality attributes is discussed in RQ6, RQ6.1, and RQ6.2.

Table 3.1 Research Questions addressed in SLR

| RQ# | RQ | Motivation |
|---|---|---|
| RQ1 | What refactoring methods have been applied across primary studies? | Identify the commonly adopted refactoring methods by researchers. |
| RQ2 | What bad-smells are analyzed in the primary studies? | Identify commonly detected bad-smells by researchers. |
| RQ3 | What quality attributes are | Identify popular quality measures in the |

| | | |
|---|---|---|
| | selected across primary studies? | field of refactoring. |
| **RQ3.1** | What internal quality attributes (object-oriented characteristics) are investigated in primary studies? | Identify common internal quality attribute and popular metric-suite for object-oriented systems. |
| **RQ3.2** | What external quality attributes are investigated in primary studies? | Identify commonly studied external quality measures by researchers. |
| **RQ4** | What software-systems/data-sets are selected in primary studies to perform refactoring methods? | Identify the data-sets found appropriate for applying refactoring by the researchers |
| **RQ5** | What statistical techniques are adopted by researchers? | Identify the conclusions are analyzed properly and are reliable enough. |
| **RQ6** | Does refactoring improve the quality attributes? | Identify the performance of various refactoring methods on the quality metrics based on current research. |
| **RQ6.1** | Which quality attributes (internal/ external) are overall benefitted by refactoring? | Find internal and external quality attributes that have a positive impact of refactoring in general. |
| **RQ6.2** | Which refactoring method and quality attribute combination yield a good result? | Find out which internal and external quality attributes will have a positive impact after applying a particular refactoring method. |

## 3.3 SEARCH STRATEGY & STUDY SELECTION

The purpose of Study Selection is to pick up those primary studies that are in correspondence with providing direct evidence of research questions. To reduce bias-likelihood, the selection criterion is established at the time of research-protocol definition. Figure 3.2 depicts steps followed for study selection in the SLR. In the first place, five popular electronic databases are searched to arrive at potentially relevant articles against an exhaustive set of search terms. For this SLR, following set of sophisticated search terms are created that are merged using Boolean OR for alternatives and synonyms and important terms are grouped using Boolean AND. The search terms are as below:

1) Refactor* OR Restructur*
2) Refactoring method OR Refactoring Technique
3) Object Oriented OR Object-Oriented OR Source-code OR Class OR Method OR Attribute OR Code OR Software-system
4) Quality Attributes OR Quality Metrics OR Quality Measures OR Software Quality Attributes OR Software Quality Measures OR Software Quality Metrics OR Software Metrics
5) Evaluat* OR Estimat* OR Predict* OR Detect* OR Apply OR Applied OR Study OR Studied
6) Software Quality OR Coupling OR Cohesion OR Maintainability OR Adaptability OR Reusability OR Modifiability OR Understand-ability OR Size OR Inheritance OR Flexibility OR Testability OR Extendibility OR Fault Proneness OR Efficiency OR Integrity OR Readability OR Accuracy

The terms on refactoring and quality attributes were captured from textbooks in required areas [1 and 7]. Once search terms are finalized, potential popular electronic databases were selected. Table 3.2 represents details about selected five electronic databases that were used to retrieve potentially relevant primary studies.

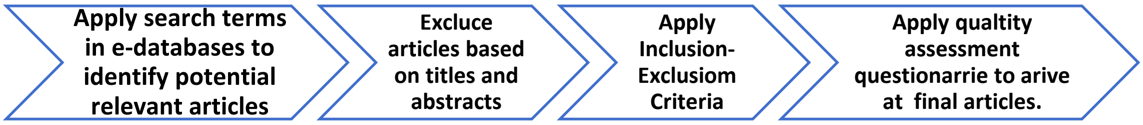
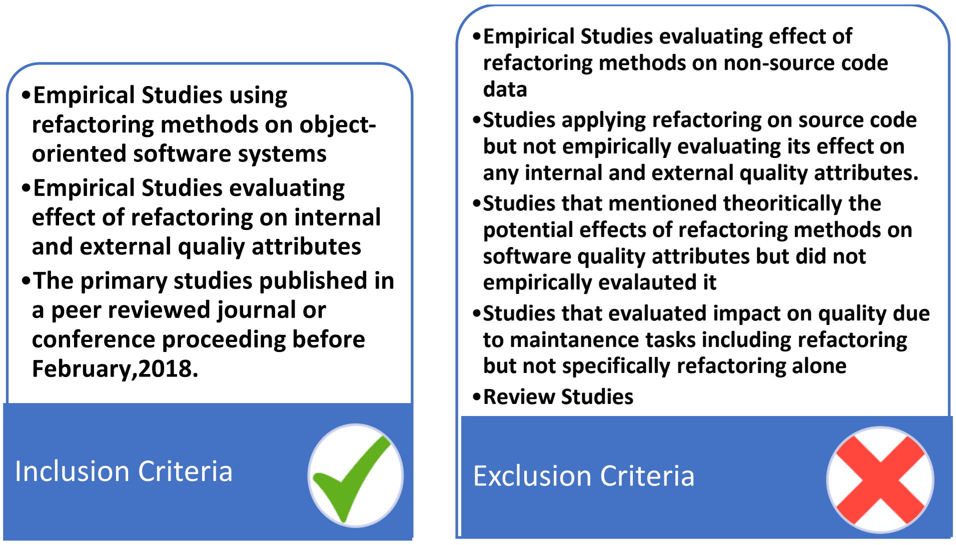

Apply search terms in e-databases to identify potential relevant articles
Excluce articles based on titles and abstracts
Apply Inclusion-Exclusiom Criteria
Apply qualtity assessment questionarrie to arive at final articles.

Figure 3.2 Steps for Study Selection

The search was made on articles from 1991 till January 2018. An initial search using the search terms was made to capture potentially relevant articles as primary studies. All these studies were collected in full-text and afterward inclusion and exclusion criterion was applied that is discussed in next section. Studies that are found in the reference section and seemed useful were also selected. We included those empirical studies in SLR that empirically evaluate the effect of refactoring methods on software quality attributes (internal and external, or both). We found 334 relevant studies after exploring electronic databases using the defined search strategy and then further added 30 relevant studies after scanning the reference lists of relevant studied identified from the electronic databases. As a result, overall 364 relevant studies were captured for subsequent processing. The following inclusion-exclusion criteria are applied to the identified studies.



- Empirical Studies using refactoring methods on object-oriented software systems
- Empirical Studies evaluating effect of refactoring on internal and external qualiy attributes
- The primary studies published in a peer reviewed journal or conference proceeding before February,2018.

Inclusion Criteria

- Empirical Studies evaluating effect of refactoring methods on non-source code data
- Studies applying refactoring on source code but not empirically evaluating its effect on any internal and external quality attributes.
- Studies that mentioned theoritically the potential effects of refactoring methods on software quality attributes but did not empirically evalauted it
- Studies that evaluated impact on quality due to maintanence tasks including refactoring but not specifically refactoring alone
- Review Studies

Exclusion Criteria

Figure 3.3 Inclusion-Exclusion Criteria

The papers related to computer science and engineering are only included. The SLR incorporated articles from the initial dates of the electronic libraries to January 2018. However same articles from different electronic data sources are removed so as to remove duplicity. The inclusion-exclusion criteria as given in Figure 3.3 were tested by two independent researchers to arrive at a same decision after meetings and discussions. In case of doubt, thorough analysis of entire text was made to decide the inclusion or exclusion for that article. The quality of the selected studies was also being identified so as to capture their relevance regarding research questions. Adopting the aforementioned inclusion-exclusion criteria, 123 studies were selected. In the end, the quality questionnaire is prepared for each of the selected studies to arrive at a final list of primary studies.

Table 3.2List of Electronic Databases explored

| S. No. | E-Resource | Link to Access |
|---|---|---|
| 1 | IEEE Xplore | ieeexplore.ieee.org/ |
| 2 | Elsevier ScienceDirect | www.sciencedirect.com |
| 3 | ACM Digital Library | www.acm.org/ |
| 4 | SpringerLink | http://www.springer.com/in, |
| 5 | Wiley Online Library | https://onlinelibrary.wiley.com/ |

## 3.4 QUALITY ASSESSMENT CRITERIA

To achieve confidence in the relevant studies, a quality questionnaire is formed that includes fifteen quality questions to assess their significance and provide weights to the studies with regards to the research questions being addressed in the SLR. Table 3.3 represents all the selected fifteen questions and the percentage of primary studies answering these quality assessment questions as per the suggestions of Wen et al. [27]. The articles are ranked from 0 to 1. 1 signifies YES, 0 signifies NO and 0.5 signifies PARTLY. For each article, the rank for each question is summed to calculate the final score. At max, any study could score a maximum of 15 and a minimum of 0.

Two independent researchers were occupied with assessing the quality questions and preparing final scores for each of the selected studies. In case of any doubt or disagreement, reviews and brainstorming sessions were performed to arrive at a consensus.

Table 3.3 Quality Assessment Questions

| QA# | Quality Assurance Question | Yes | Partly | No |
|---|---|---|---|---|
| **Design** | | | | |
| **QA1** | Are the refactoring methods stated and defined appropriately? | 63.6 | 49.1 | 14.5 |
| **QA2** | Is aim of the primary study clearly stated? | 100 | 0 | 0 |
| **QA3** | Are evaluated internal quality attributes and external quality attributes clearly stated and defined? | 58.7 | 32.1 | 9.2 |
| **Conduct** | | | | |
| **QA4** | Are methods used for collecting data properly stated/described? | 68.4 | 25.7 | 5.9 |
| **Analysis** | | | | |
| **QA5** | Is the purpose of the analysis clear? | 98.1 | 1.2 | 0.7 |
| **QA6** | Are the results of applying the refactoring techniques evaluated? | 89.2 | 7.2 | 3.6 |
| **QA7** | Are the datasets properly described? (programming language, size, venue) | 63.8 | 19.7 | 16.5 |
| **QA8** | Are the statistical methods described? | 29 | 1.2 | 69.8 |
| **QA9** | Are the statistical methods justified? | 29 | 0 | 75 |

| | | | | |
|---|---|---|---|---|
| **QA10** | Do the statistical significance of the results reported? | 12.9 | 2.2 | 89.4 |
| **Conclusion** | | | | |
| **QA11** | Are all questions from the primary study answered? | 75.2 | 9.6 | `15.2 |
| **QA12** | Are negative findings reported? | 35.3 | 1.2 | 63.5 |
| **QA13** | Do the results and findings contribute to the literature? | 38.4 | 40.4 | 21.2 |
| **QA14** | Do the results support the conclusions? | 96.9 | 2.8 | 0.3 |
| **QA15** | Are validity threats discussed? | 49.2 | 5.4 | 45.4 |

## 3.5 DATA EXTRACTION AND DATA SYNTHESIS

Data extraction was performed by following these steps:

(a) One author analyzed 104 primary studies selected quality assessment to extract relevant data with respect to various parameters like refactoring methods (number of refactoring methods performed, name of refactoring methods, automated or manual), bad-smells (number of bad-smells analyzed, common bad-smells), quality attributes (internal or external or both, number of each type of attributes, effect of refactoring on quality attributes, process to evaluate quality attributes), data-set (venue-open-source, academic, industrial, programming language, size, source) etc.

(b) Another author, a prominent professor in the area of Software Engineering validated the extracted data by evaluating the primary studies on selected parameters.

(c) In case of any discrepancy among the results, a meeting was conducted to arrive at an appropriate result.

By adopting above mentioned process, data extraction forms were filled for each of the primary studies. The main aim to fill in these forms is to gain all the desired information that is required to answer the research questions imposed by SLR. The final data is saved in excel spreadsheets to use this data as input for data synthesis.

The data synthesis task consists of gathering, accumulating and summarizing the facts and figures from the selected primary studies to answer research questions imposed by SLR. It is a meta-analysis task to classify the results and findings by different researchers so as to provide evidence to arrive at a certain conclusion that answers the research questions. Both the quantitative aspects that contains impact of refactoring methods on quality attributes (internal and external), number of refactoring methods applied, number of quality attributes under study, as well as qualitative aspects like the data-sets used, details and classification of refactoring methods, statistical significance, strengths and weaknesses of refactoring methods with respect to their effect on quality attributes, etc. are considered. To answer the research questions, visualization mechanisms like pie-charts, bar-charts, line-graphs along with tables to precisely represent the results are being used.

## 3.6 RESULTS & FINDINGS OF SYSTEMATIC LITERATURE REVIEW

In this section, results captured from the selected primary studies are discussed. Firstly, an overview of selected primary studies is presented here. Later, the results are interpreted and presented meaningfully in form of suitable charts and tables.

### 3.6.1 Description of Primary Studies

Here, we briefly describe all the 104 selected primary studies. Out of 123 relevant studies identified by us before quality assessment, we have selected 104 primary studies based on the scores obtained in the quality assessment.

**(a) Publication Source**

Table 3.4 lists out the number and the corresponding percentage of primary studies from top journals and conferences. Majority of publications were in Journal of Systems and Software, IEEE Transactions on Software Engineering, Information and Software Technology and Empirical Software Engineering, Asia Pacific Software Engineering Conference (APSEC), IEEE Conference on Advances in Computing, Communications and Informatics (ICACCI), IEEE International Working Conference on Source-code Analysis and Manipulation (SCAM) and so on. The top 19 publication sources encompass 57.31% of primary studies.

Also, Figure 3.4 shows that majority of primary studies are present in IEEE Xplore followed by ACM and Springer.

Table 3.4 Summary of Key Publication Sources

| Publication Name | Publication Type | Primary Studies | Count | Percent |
|---|---|---|---|---|
| **Asia Pacific Software Engineering Conference (APSEC)** | Conference | [56], [60], [95], [110], [119] | 5 | 4.80 |
| **IEEE International Working Conference on Source-code Analysis and Manipulation (SCAM)** | Conference | [28], [107], [114], [116] | 4 | 3.84 |
| **IEEE Conference on Advances in Computing, Communications and Informatics (ICACCI)** | Conference | [46],[59], [62], [63] | 4 | 3.84 |
| **IEEE International Conference on Software Engineering (ICSE)** | Conference | [40],[51], [12] | 3 | 2.88 |
| **IEEE International Conference on Software Maintenance and Evolution (ICSME)** | Conference | [58], [73],[77] | 3 | 2.88 |
| **IEEE Conference on Software Maintenance and Reengineering (CSMR)** | Conference | [35], [105], [98] , [106] | 4 | 3.84 |
| **IEEE International Software Metrics Symposium** | Conference | [53], [57] | 2 | 1.92 |
| **International Conference on the Quality of Information and Communications Technology (QUATIC)** | Conference | [64], [75] | 2 | 1.92 |
| **IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)** | Conference | [66],[70] | 2 | 1.92 |
| **Journal of Systems and Software** | Journal | [31],[42], [82], [83], | 6 | 5.77 |

| | | | | |
|---|---|---|---|---|
| | | [102], [117] | | |
| **IEEE Transactions on Software Engineering** | Journal | [72], [41], [52], [61], [65], [104] | 6 | 5.77 |
| **Information and Software Technology** | Journal | [39], [55], [16], [84], [85],[103] | 6 | 5.77 |
| **Empirical Software Engineering** | Journal | [30], [47], [90] | 3 | 2.88 |
| **ACM Transactions on Software Engineering and Methodology** | Journal | [29], [91] | 2 | 1.92 |
| **Arabian Journal of Science and Engineering** | Journal | [36], [89] | 2 | 1.92 |
| **Journal of Software Evolution and Processes** | Journal | [88], [10] | 2 | 1.92 |
| **Expert Systems with Applications** | Journal | [54], [94] | 2 | 1.92 |
| **Journal of Software Maintenance and Evolution: Research & Practice** | Journal | [81], [87] | 2 | 1.92 |
| **Software Practice and Experience** | Journal | [86], [90] | 2 | 1.92 |

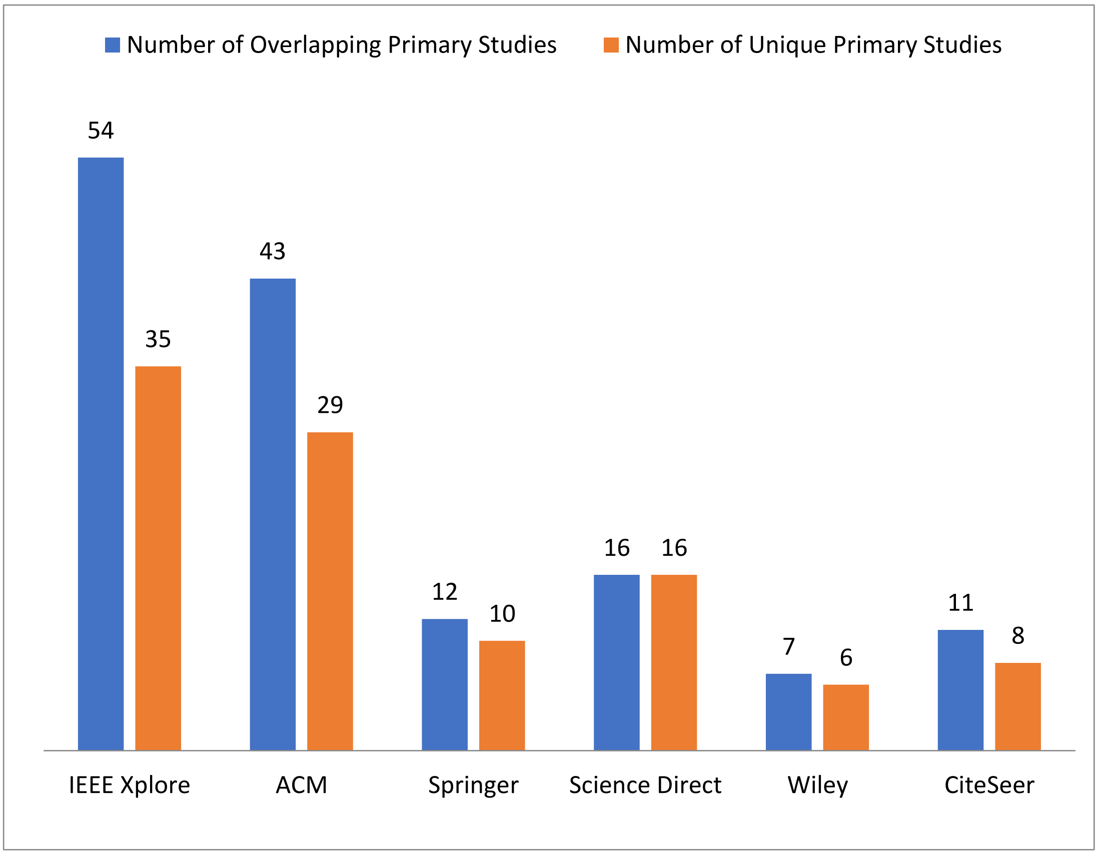


Figure 3.4 Distribution of Primary Studies across Electronic Data Sources

### (b) Quality Assessment Questions

The quality assessment questions were assigned scores that were divided into three categories i.e. high (13 ≤ scores ≤ 15), medium (10 ≤ scores ≤ 12) and low (0≤ score ≤ 9). Six primary studies- [36], [39], [46], [50], [97] and [99] obtained highest scores. Therefore, the willing readers can go through these studies for further reading. Studies with the low score were discarded and as a result, nineteen studies- [125]-[143], having a score of <=8 were removed from the relevant studies to arrive at a total of 104 primary studies that are found fit for SLR.

### (c) Publication Year

In Figure 3.5, distribution of primary studies from the year 2000 to 2017 is shown. It can be observed that before 2006, only 8 primary studies evaluated the effect of one or more refactoring methods on software quality attributes. Among these studies, mainly all examined the effect of refactoring on quality attributes from 2006 onwards. Number of studies in 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016 and 2017 are 6, 2, 5, 8, 4, 13, 13, 7, 11, 8, 9 and 10 respectively. The maximum number of relevant primary studies is in the years 2011, 2012, 2014, 2017 and 2016. In the recent, quite a good number of studies are focusing on evaluating the impact of refactoring methods on quality attributes. This confirms the inclusion of related and current studies in the SLR.

Also, 10 primary studies ([60], [61], [62], [63], [65], [67], [72], [73], [76] and [10]) in 2017, 9 primary studies ([46], [64], [66], [68], [69], [70], [71], [75] and [77]) in 2016, 8 primary studies in 2015 ([48],[55],[58], [74], [83], [90], [103] and [109]), 11 primary studies in 2014 ([29], [32], [44], [52], [54], [56], [96], [99], [107], [114] and[118]), 7 primary studies in 2013 ([30], [49], [85], [104], [110], [115] and [121] ), 13 primary studies ([28], [33], [34], [38], [47], [59], [16], [84], [91], [92], [93], [14] and [102]) in 2012, 13 primary studies ([31], [36], [40], [42], [51], [79], [80], [89], [12], [97], [111] and[37]) in 2011, 4 primary studies ([82], [15], [108] and [122]) n 2010, 8 primary studies ([37], [41], [86], [94], [95], [106] and[119]) in 2009, 5 primary studies ([45], [81], [13], [112], [123]) in 2008, 2 primary studies ([98] and [17]) in 2007, 6 primary studies ([35], [43], [87], [88], [105] and [113]) in 2006, 1 primary study ([53]) in 2005, 2 primary studies ([100] and [120]) in 2004, 3 primary studies ([57], [101] and [117]) in 2003,1 primary study ([50]) in 2002 and 1 primary study ([78]) in 2000 are conducted. Only in the year 2001, there is no relevant primary study.

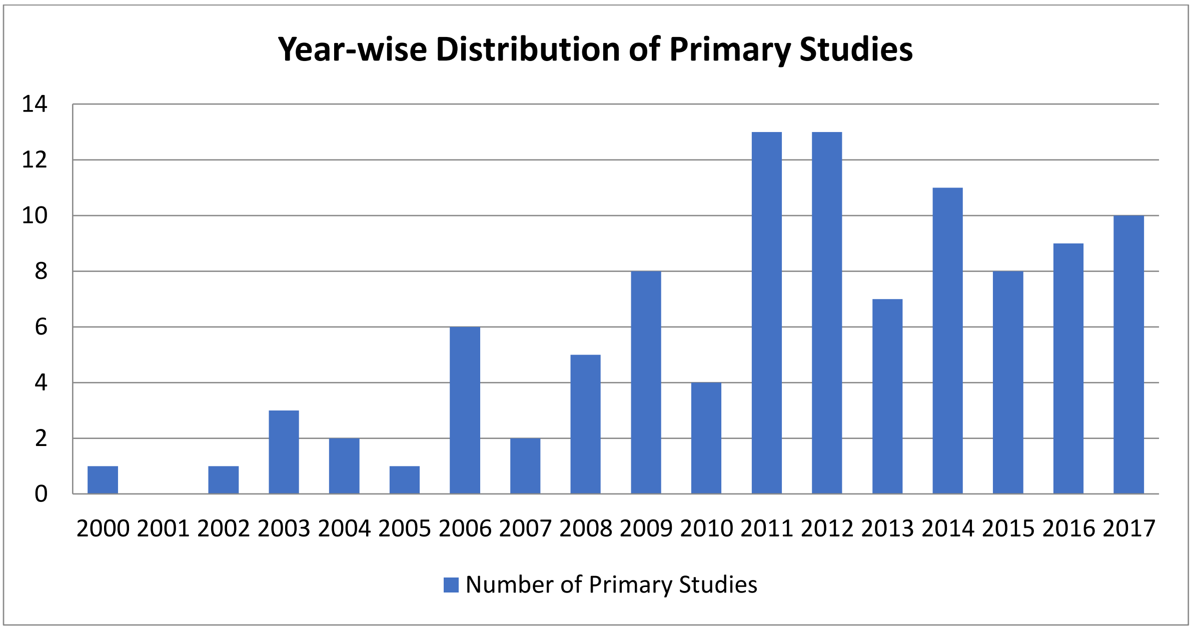


Figure 3.5Year-wise distribution of Primary Studies

### 3.6.2 RQ1: What refactoring methods have been applied across primary studies?

Following results can be drawn from the selected primary studies:

1) Table 3.5 gives the number of refactoring methods used by listed primary studies. It is evident from the table that there are two types of studies; first that identifies the effect of a particular refactoring method on quality attributes and second that identifies the effect of two or more refactoring methods on quality attributes. 25.96% primary studies belong to the first category and studied the impact of applying a particular refactoring method on certain quality attributes whereas 50.96% primary studies studied the impact of two or more refactoring methods on quality attributes. It should also be noted that 23.07% studies do not mention the number of refactoring methods applied in their research.

Table 3.5 Number of Refactoring Methods performed in Primary Studies

| **Number of Refactoring Methods** | **Primary Studies** | **Number of Primary Studies** |
|---|---|---|
| 1 | [29],[30], [31], [32], [40], [41], [42], [53], [55], [61], [65], [71], [77], [16], [15], [12], [95], [96], [102], [103], [105], [109], [110],[111] and [113] | 27 |
| 2 | [44], [57], [79], [84], [85], [91 and [104] | 7 |
| 3 | [54], [68], [72], [75], [78], [80], [87], [101], [107] and [121] | 10 |
| 4 | [64], [106], [108] and[116] | 4 |
| 5 | [46], [98], [100], [119] and [120] | 5 |
| 6 | [35], [14] and [117] | 3 |
| 7 | [59] | 1 |
| 8 | — | 0 |
| 9 | [51], [63], [89] and [13] | 4 |
| 10 | [49], [73], [74] and [90] | 4 |
| >10 | [28], [33], [34], [36], [38], [45], [47], [48], [62], [76], [76], [81], [83], [10], [118] and [124] | 15 |
| Not | [37], [39], [43], [50], [52], [56], [58], [60], [66], [67], | 24 |

| | | |
|---|---|---|
| Mentioned | [69], [70], [82], [86], [88], [92], [93], [94], [97], [99], [112], [115], [122] and [123] | |

2) A total of 156 refactoring methods have been identified in the selected studies out of which 70 have been used by two or more primary studies and rest 86 have been used by a single primary study.
3) Out of 156 identified refactoring methods, there are several new refactoring methods not identified by Fowler like test-based bad-smells (like resource optimism, indirect testing, test run war) given in [73], accessibility/security related bad-smells (increase security field, decrease security field) given in [76], new variants of move method (like move a method to its parameter types, move a method to a type of a randomly chosen attribute of the class the method belongs to, move a static method) etc.
4) Figure 3.6 shows the common refactoring methods among primary studies. Move method, extract method and encapsulate field are among the most frequently used refactoring methods as they are used in 30.76%, 29.80% and 28.84% of studies respectively.

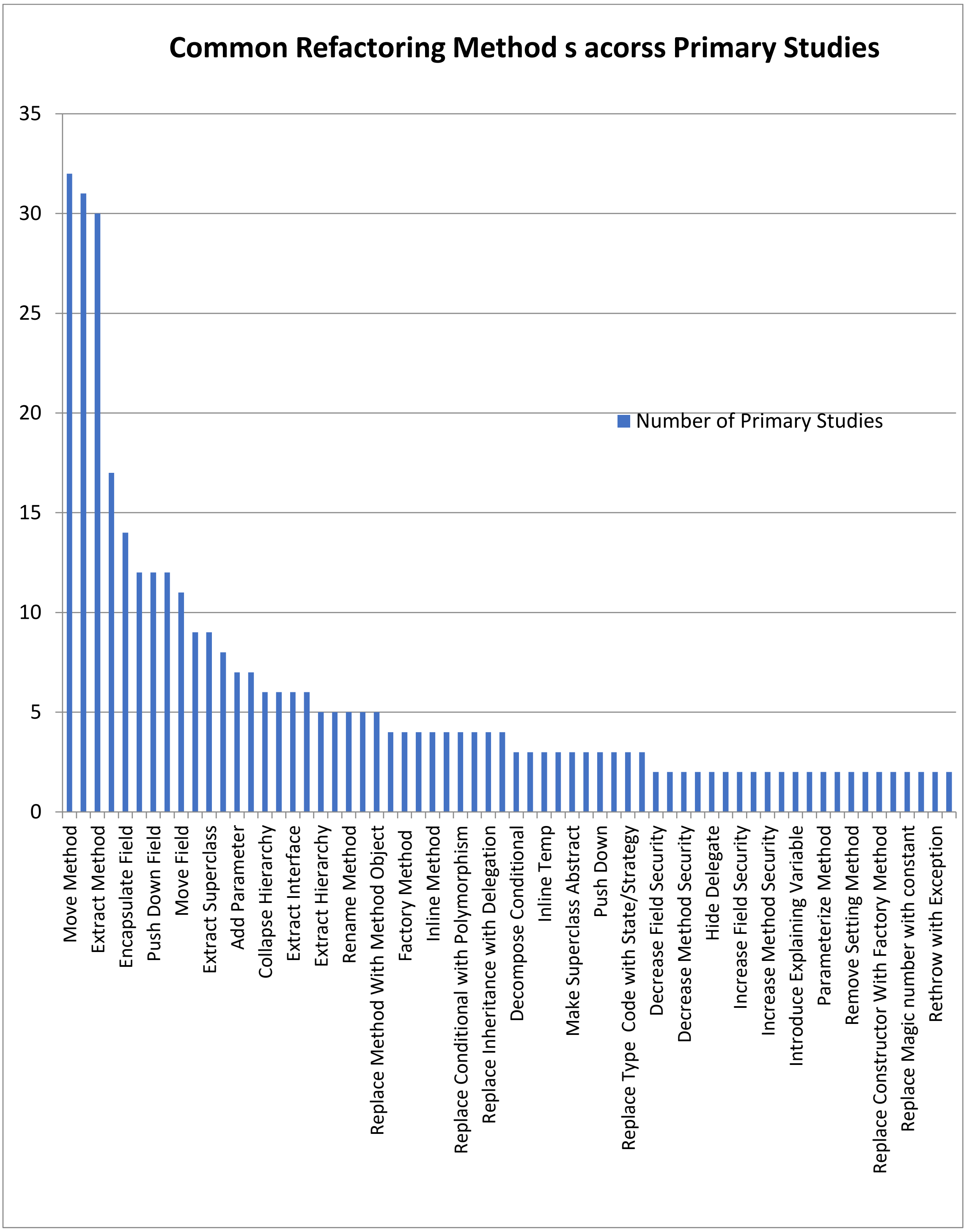


Figure 3.6 Commonly used Refactoring Techniques

### 3.6.3 RQ2: What bad-smells are analyzed in the primary studies?

Following results can be drawn about bad-smells from the selected primary studies:

1) Out of 104 primary studies, 24 primary studies tried to apply refactoring method specific to bad-smells selected by them whereas remaining 80 primary studies do not apply refactoring method to remove particular bad-smells. Figure 3.7 depicts that only 23% of primary studies applied refactoring method to remove specific bad-smells.
2) Out of 23% primary studies that tried to apply refactoring methods to remove selected bad-smells, 37.5% primary studies ( [32], [40], [41], [12], [95], [108], [109], [116] and [121]) tried applying refactoring method for removing only one bad-smell whereas remaining 62.5% primary studies ([54], [57], [59], [62], [63], [70], [75], [80], [83], [85], [10], [14], [107] and [114]) applied refactoring method to remove more than one bad-smell.

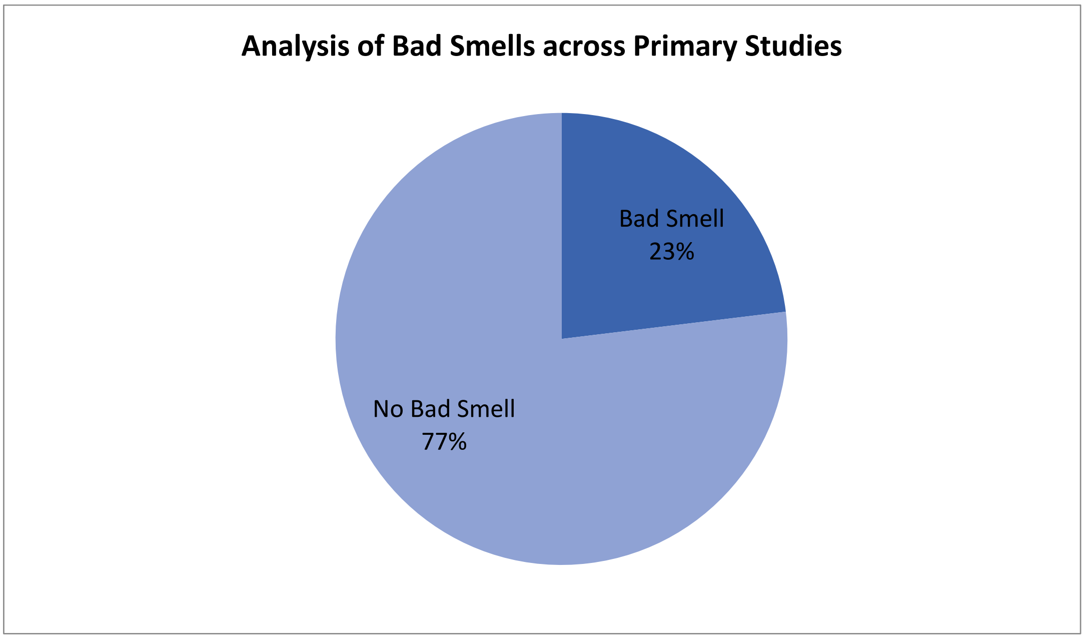


Figure 3.7 Percentage of Primary Studies analyzing Bad-smells

3) Figure 3.8 shows bad-smells that are commonly detected and removed after applying appropriate refactoring methods. Feature Envy ([32], [41], [54], [57], [59], [80], [83], [10], [12] and [109]), Long Method ([57], [59], [63], [75], [80], [83], [95]

and [114]), God Class ([40], [59], [63], [75], [10] and [14]), duplicate code/ code clone ([29], [14], [108], [114], [116] and [121]), long parameter list ([59], [63], [70], [75], [83 and [10]) are some of the most frequently detected bad-smells across empirical studies. On the other hand, refused parent bequest ([83]), middle man ([104]) and inappropriate intimacy ([107]) are detected by only one primary study.

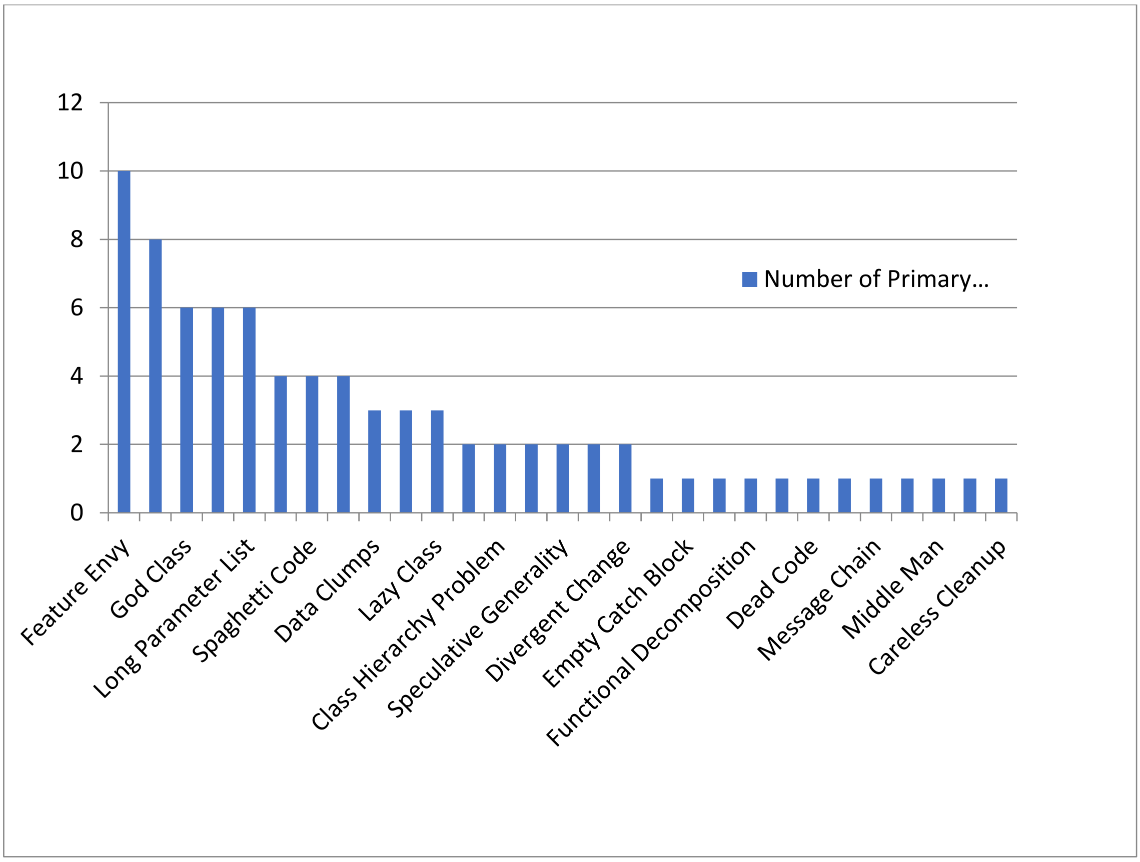


Figure 3.8 Commonly detected Bad-smells in Primary Studies

### 3.6.4 RQ3: What quality attributes are selected across primary studies?

Quality attributes are indicators of overall software quality. Two kinds of quality attributes are present- internal and external. Internal quality attributes are measured by software code-artifacts. On the contrary, external quality attributes cannot be directly computed and require calculation of one or more internal quality attributes. Due to this, internal and external quality attributes are respectively called direct and indirect attributes. To evaluate external quality attributes, software environment and interactions between software artifacts and environment are considered. Many times, all the desired details to measure the external quality attributes is not available. As a result, some researchers propose formulae and models to estimate the external quality attributes. Internal quality attributes are used as independent variables to identify external quality attributes that are treated as dependent variables.

**(a) RQ3.1: What internal quality attributes (object-oriented characteristics) are investigated in primary studies?**

Following results can be drawn regarding internal quality attributes across the primary studies:

1) Among the selected primary studies, we identified ten internal quality attributes that are coupling, cohesion, size, complexity, inheritance, abstraction, polymorphism, data encapsulation, composition and information hiding. Table 3.6 lists out the details of the internal quality attributes studied in the primary studies.
2) Coupling, cohesion, size, and complexity are the highly studied internal quality attributes that are evaluated by 70.19%, 58.65%, 47.11% and 39.42% primary studies respectively. On the other hand, polymorphism, data encapsulation, composition and information hiding are evaluated by least number of primary studies as they are used in 6.73%, 6.73%, 6.73% and 2.88% of primary studies respectively.

Table 3.6 Internal Quality Attributes evaluate in Primary Studies

| Internal Quality Attribute | Number of Primary Studies | Primary Studies |
|---|---|---|
| **Coupling** | 73 | [29],[30], [31], [32], [33], [34], [35], [36], [37], [39], [40], [41], [43], [45], [46], [50], [52], [56], [58], [59], [62], [63], [65], [67], [71], [75], [76], [77], [80], [81], [82], [83], [16], [85], [86], [87], [88], [10], [89], [90], [92], [15], [93], [12], [94], [95], [14], [96], [97], [99], [100], [101], [102], [13], [104], [105], [106], [107], |

| | | |
|---|---|---|
| | | [108], [109], [110], [111], [113], [17], [114], [115], [116], [119], [120], [121], [122], [123] and [124] |
| **Cohesion** | 61 | [30], [31], [33], [34], [35], [36], [39], [40], [41], [42], [43], [45], [46], [52], [56], [59], [61], [62], [63], [65], [67], [80], [81], [82], [83], [16], [85], [86], [10], [89], [90], [91], [92], [15], [12], [95], [14], [96], [97], [100], [101], [102], [3], [104], [106], [107], [108], [109], [110], [111], [113], [17], [114], [115], [116], [120], [121], [122], [123] and [124] |
| **Size** | 49 | [28], [29], [30], [33], [35], [36], [38], [39], [41], [43], [44], [45], [57], [58], [62], [66], [68], [77], [78], [81], [82], [83], [16], [84], [86], [87], [88], [10], [89], [90], [93], [94], [96], [97], [98], [99], [104], [106], [108], [113], [115], [119], [117], [119], [121], [123] and [124] |
| **Complexity** | 44 | [36], [43], [44], [45], [46], [52], [58], [58], [59], [62], [63], [66], [67], [68], [69], [71], [75], [77], [80], [83], [84], [87], [12], [94], [95], [14], [97], [99], [101], [103], [13], [104], [105], [106], [107], [109], [110], [113], [114], [115], [117], [118], [119], [120], [121] and [122] |
| **Inheritance** | 28 | [33], [35], [39], [45], [58], [63], [76], [81], [82], [83], [88], [89], [95], [97], [98], [99], [101], [13], [106], [107], [108], [109], [114], [115], [116], [120], [122] and [124] |
| **Abstraction** | 9 | [45], [46], [76], [80], [10], [14], [108], [116] and [124] |
| **Polymorphism** | 7 | [45], [81], [10], [90], [108], [116] and [124] |
| **Data encapsulation** | 7 | [33], [35], [10], [90], [108], [116] and [124] |

| | | |
|---|---|---|
| **Composition** | 7 | [35], [45], [10], [90], [108], [116] and [124] |
| **Information Hiding** | 3 | [59], [63] and [67] |

3) Table 3.7 presents the common metrics used in primary studies to evaluate internal quality attributes. Chidamber & Kemerer metrics suite [5], QMOOD metrics suite [144], MOOD metrics suite [4], Lorenz & Kidd Metrics suite [6] and Li & Henry metrics suite [7] are most commonly used metrics suite among the primary studies. Chidamber & Kemerer and QMOOD are used in the majority of the primary studies.

Table 3.7 Metrics identified across primary study for respective quality attribute

| **Internal Quality Attribute** | **Metrics** |
|---|---|
| Coupling | Class Method Export Coupling (OCMEC), Afferent Coupling, Aggregated import coupling, Coupling Between Objects (CBO), Conceptual Coupling Between Classes(CCBC), Message Passing Coupling (MPC), CCC, CDBC, Coupling Factor (CF), Class Coupling (CC), Data Abstraction Coupling (DAC), Direct Class Coupling (DCC), Efferent Coupling, Export Coupling, Fan In, Fan Out, General Coupling, Information-flow-based-Coupling (ICP), Low Data Coupling (LD), NOCM (Number of Outward Coupling Methods), Number of Remote Methods (NR), Number of Parameters, Response For a Class (RFC), Semantic Coupling , Structural Coupling, Access To Foreign Data (ATFD), Return Value Coupling |
| Cohesion | Conceptual Cohesion of Classes (C3), Classified Accessor Attribute Interactions (CAAI), Classified Attribute Interaction Weight (CAIW), Cohesion Among Methods of Class (CAM), Cohesion |

| | |
|---|---|
| | Based on Member Connectivity (CBMC), Classified Mutator Attribute Interactions (CMAI), Classified Methods Weight (CMW), Coh, Connectivity, coverage, Degree of Cohesion Direct (DCD), Degree of Cohesion Indirect (DCI), Improved Cohesion Based on Member Connectivity (ICBMC), Information flow based cohesion (ICH), Loose Class Cohesion (LCC), LCOM1, LCOM2, LCOM3, LCOM4, LCOM5, Low level design Similarity-based Class Cohesion (LSCC), Methods Similarity Cohesion (MSC), Non-normalized Cohesion, Normalized Cohesion, Path Connectivity Class Cohesion (PCCC), Semantic Cohesion, Structural Cohesion, Tight Class Coupling (TCC), tightness, Locality of Attribute Access (LAA), Class Cohesion (CC) |
| Size | Number of Blocks, Number of Classes, Number of Functions, Number of Local Variables, Number of Parameters, AMS, ANA, Attributes/ Class, CIS, CS, DSC, Duplicate Code Blocks, Comment Blocks, Lines of Code (LOC), Source Lines of Code (SLOC), Number of Methods (NOM), Number of Static Methods, Class design proportion (CDP), NLM, NOCL, NIV, NCV |
| Complexity | Cyclomatic Complexity (CC), Class Definition Entropy (CDE), Classes in a Cycle, Function Parameters, Immediate Base Class, Lines of Code Per Class, Lines of Code Per Method, Max_Loc, Max_MCC, McCabe Per Method (MVG), Member reads, Member writes, Method Size, Methods lines of code per method (MLOC), Number Of Attributes (NOA), Number Of Methods (NOM), Number of Public Methods of a class (NPM), Type Declarations in Local Method, Weighted Method per Class (WMC), Average Method Weight (AMW), Average Line of Code per Method (ALCM), OCavg |
| Abstraction | Measure of Functional Abstraction (MFA) |
| Polymorphism | Number of Polymorphic Method (NOP) |
| Data | Classified Class Data Accessibility (CCDA), Classified Instance |

| | |
|---|---|
| encapsulation | Data Accessibility (CIDA), Classified Operation Accessibility (COA), Data Access Metric (DAM) |
| Composition | Measure Of Aggregation (MOA), Composite Part Critical Classes (CPCC) |
| Inheritance | Depth of Inheritance (DIT), Number of Children (NOC), Average Number of Ancestors (ANA) |
| Information Hiding | Attribute Hiding Factor (AHF), Method Hiding Factor (MHF) |

**(b) RQ3.2: What external quality attributes are investigated in primary studies?**

1) Among the selected primary studies, we identified fifteen external quality attributes that are maintainability, reusability, understandability, flexibility, adaptability, testability, extensibility, effectiveness, completeness, functionality, modularity, reliability, security, modifiability, and traceability, Table 3.8 lists out the details of the external quality attributes studied in the primary studies.

2) Maintainability, understandability, reusability, flexibility, extensibility are the highly studied external quality attributes that are evaluated by 25%, 15.38%, 14.42%, 10.57% and 10.57% primary studies respectively. On the other hand, adaptability, completeness, and traceability are evaluated by least studied external quality attributes as they are evaluated by 1.92%, 1.92% and 0.96% primary studies respectively.

3) Out of 104 primary studies, 24 primary studies evaluated the effect refactoring method(s) on single external quality attribute whereas remaining 76.93% evaluated the effect of refactoring method(s) on two or more external quality attributes.

Table 3.8 External Quality Attributes evaluated in Primary Studies

| **External Quality Attribute** | **Number of Primary** | **Primary Studies** |
|---|---|---|

| | Studies | |
|---|---|---|
| **Maintainability** | 26 | [36], [38], [44], [45], [47], [49], [50], [55], [56], [57], [58], [59], [66], [69], [72], [77], [85], [89], [98], [99], [107], [117], [118], [122] and [123] |
| **Reusability** | 15 | [35], [36], [38], [43], [46], [48], [54], [70], [10], [89], [90], [108], [116], [118] and [124] |
| **Understandability** | 16 | [35], [36], [45], [46], [70], [72], [74], [81], [10], [89], [90], [98], [102], [108], [116] and [118] |
| **Flexibility** | 11 | [35], [38], [45], [54], [70], [10], [89], [90], [108], [116] and [124] |
| **Adaptability** | 2 | [36] and [89] |
| **Testability** | 6 | [36], [71], [73], [89], [118] and [119] |
| **Extensibility** | 11 | [38], [44], [45], [46], [54], [70], [10], [90], [108], [116] and [124] |
| **Effectiveness** | 9 | [38], [45], [54], [70], [10], [90], [108], [116] and [124] |
| **Completeness** | 3 | [36], [108] and [118] |
| **Functionality** | 6 | [45], [10], [89], [90], [108] and [116] |
| **Modularity** | 3 | [38], [52] and [69] |
| **Reliability** | 10 | [28], [51], [52], [53], [62], [76], [77], [93], [112] and [118] |
| **Security** | 2 | [33] and [79] |
| **Modifiability** | 5 | [46], [48], [49], [53] and [74] |
| **Traceability** | 1 | [64] |

### 3.6.5 RQ4: What software-systems/data-sets are selected in primary studies to perform refactoring methods?

With respect to the software-systems/data-sets used in the primary studies, following results are drawn:

1) Figure 3.9 shows the number of data-sets used in primary studies. Majority of primary studies evaluated the effect of refactoring methods using a single data-set for study. It can be observed that on an average, researchers used less number of data-sets in their studies. It means that generality and consistency are not guaranteed.
2) A total of 178 unique data-sets are identified, out of which 144 are open-source, 10 are academic and 16 are industrial whereas 8 are unidentified. Figure 3.10 reflects the venue of the primary studies reflecting that most of the data-sets are open-source and hence repeatable.
3) The maximum number of data-sets is written using Java as the programming language. Figure 3.11 represents that a large number of data-sets contributing to 86% are written in Java and only 3% and 7% of the total number of data-sets are written in C++ and C# despite their huge popularity otherwise. It reflects a need to involve primary studies from other languages as well.
4) Out of the total number of data-sets, 32.7% are large, 16.3% are medium and 48.8% are small. 2.2% data-set sizes are unidentified. The size of almost half of the total data-sets are small, therefore there is a need for including large and medium data-sets for analysis among primary studies.
5) Table 3.9 reports the popular open-source data-sets among the primary studies. JHotDraw, Apache, AgroUML are most frequently used open-source data-sets/software-systems that are written in Java language and are of medium-large size. The advantage of open-source data-sets is easy accessibility and repeatability in similar types of studies.

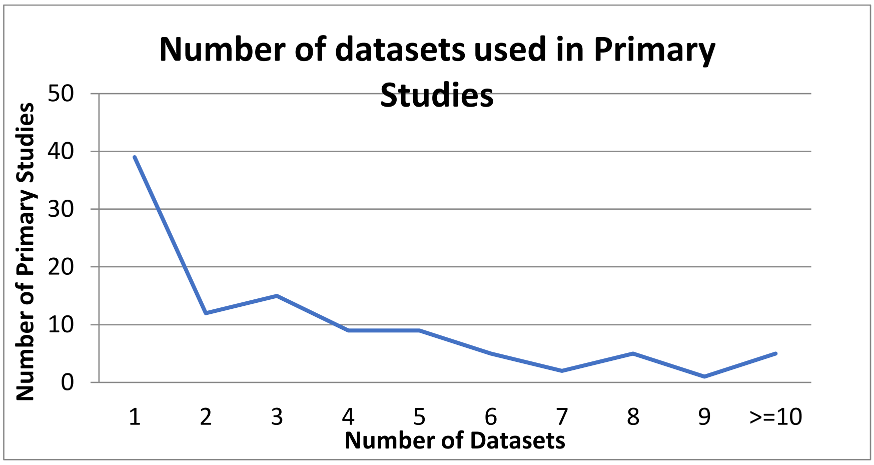


Figure 3.9 Number of Data-sets used across Primary Studies

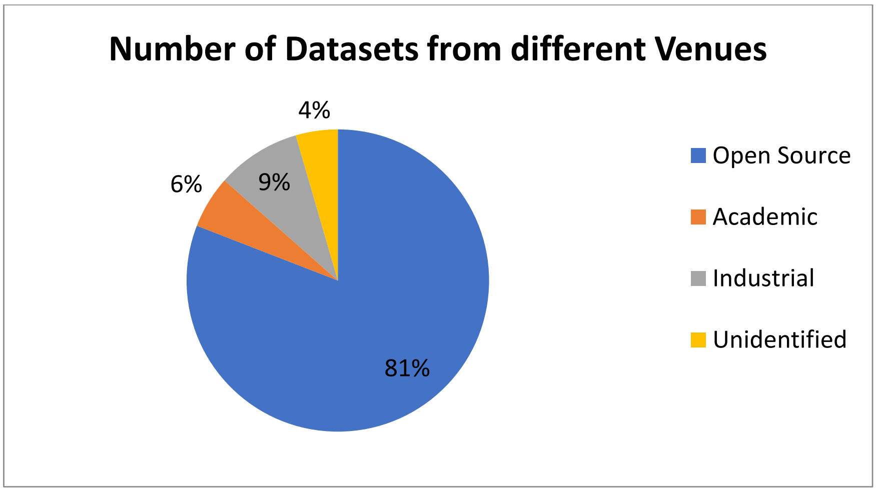


Figure 3.10 Venue of the Data-sets used in Primary Studies

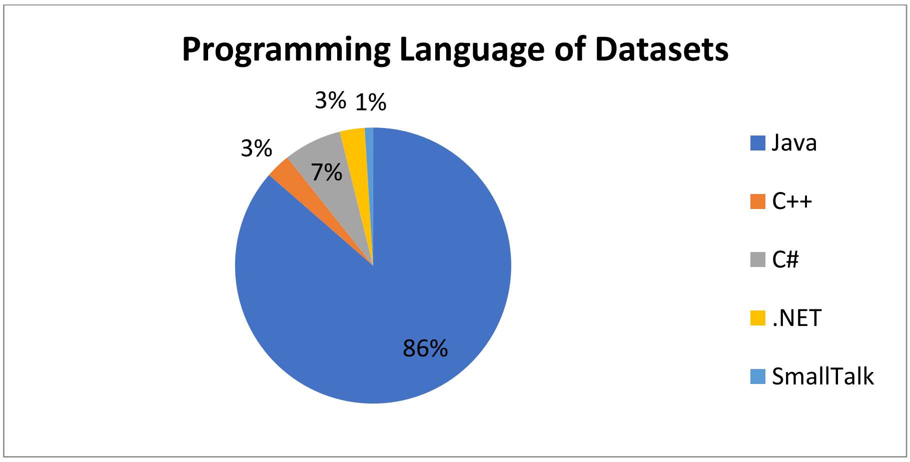


Figure 3.11 Number of data-sets from each language

Table 3.5 Frequently used data-sets among Primary Studies

| **Data-set** | **Primary Studies** | **Number of Primary Studies** |
|---|---|---|
| Apache | [28], [65], [71], [72], [83], [84], [88], [10], [97], [103] and [121] | 11 |
| Agro UML | [28], [31], [47], [70], [72], [83], [90], [92], [93], [12] and [112] | 11 |
| Xerces | [1], [54], [55], [72], [83], [10], [90], [93] and [103] | 9 |
| Gantt Project | [29], [34], [70], [80], [16], [80], [90], [93], [14], [115], [120] and [121] | 12 |
| jEdit | [29], [32], [41], [51], [54], [56], [85] and [104] | 8 |
| jHotDraw | [29], [31], [34],[62], [65], [70], [73], [78], [16], [10], [91], [92], [14] and [102] | 14 |
| jFreeChart | [32], [39], [42], [62], [73], [10], [103], [108] and [115] | 9 |
| ArtOfIllusion | [34], [62], [16], [10] and [115] | 5 |
| JabRef | [34], [16] and [115] | 3 |
| JGraphX | [34], [16] and [115] | 3 |
| JLOC | [36] and [119] | 2 |
| J2Sharp | [36], [119] | 2 |
| JNFS | [36], [119] | 2 |
| log4J | [54], [73], [82], [88] and [104] | 5 |
| Columba | [55], [56], [85] and [14] | 4 |
| Jgit | [55], [65] and [85] | 3 |
| Antlr | [60], [66], [14] and [108] | 4 |

| | | |
|---|---|---|
| Junit | [60], [66] and [67] | 3 |
| MapDB | [60] and [66] | 2 |
| mcMMO | [60] and [66] | 2 |
| mct | [60] and [66] | 2 |
| oryx | [60[ and [66] | 2 |
| Titan | [60] and [66] | 2 |
| Apache Nutch | [73], [84] and [103] | 3 |
| Apache Struts | [73], [14] and [124] | 3 |
| Hibernate | [73] and [17] | 2 |
| Mango | [76] and [80] | 2 |
| Beaver | [76] and [80] | 2 |
| Violet | [84] and [103] | 2 |
| Jade | [84] and [103] | 2 |

### 3.6.6 RQ5: What statistical techniques are adopted by researchers?

Following results can be drawn corresponding to the use of statistical techniques among primary studies:

1) As shown in Figure 3.12, out of 104 primary studies, only 30 primary studies (29%) used statistical techniques to determine the effect of refactoring method on internal and external quality attributes. Remaining 74 primary studies (71%) did not exploit any statistical techniques in their research work.
2) The primary studies that do not use statistical techniques noted the values of the studied quality attributes before and after applying refactoring method and arrive at conclusions by looking at the difference/ percent of difference between quality attributes (internal and external) values before and after applying refactoring method without identifying if the changes in the observed quality attributes (internal and external) are statistically significant or not. This reflects the lack of use of statistical

techniques across empirical studies identifying the effect of one or more refactoring methods on the selected quality (internal/external/both) attributes.

3) Among the studies using statistical techniques, Wilcoxon Rank Sum Test, t-test and Mann Whitney U Test are among the popular statistical tests as shown in Table 3.10. On the other hand, Z-Test, Logistic Regression, and Wilcoxon Signed Rank Test are least used across the primary studies.

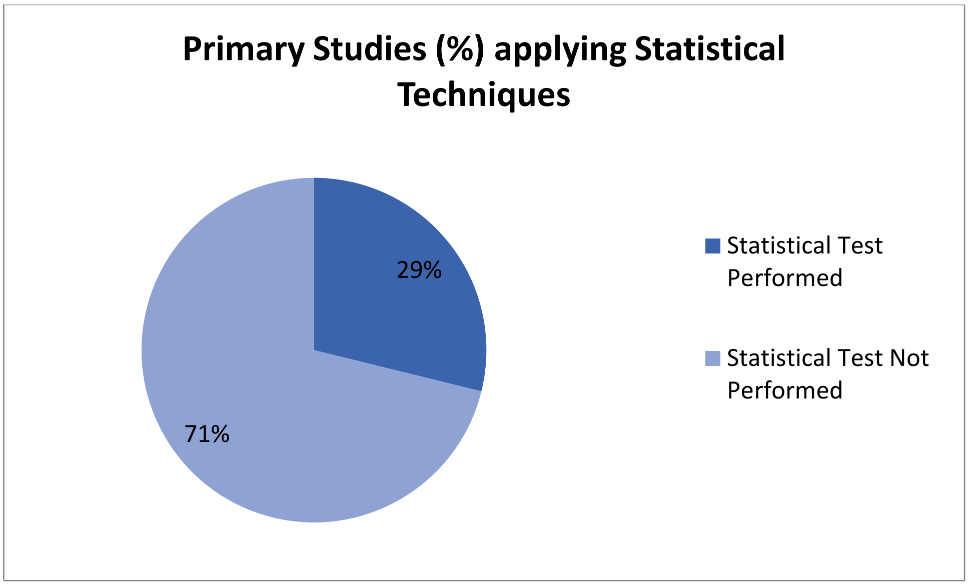


Figure 3.12 Percentage of Statistical Test performed among Primary Studies

Table 3.6Commonly used Statistical Tests among Primary Studies

| **Statistical Test** | **Primary Studies** | **Number of Primary Studies** |
|---|---|---|
| **Wilcoxon Rank Sum Test** | [31],[44], [53], [70] [10] and [90] | 6 |
| **t-Test** | [34], [49], [51], [53], [54] and [60] | 6 |
| **Spearman and Pearson Correlation** | [61], [115], [82], [49] and [48] | 5 |
| **Mann Whitney U Test** | [48], [66] and [82] | 3 |
| **Mann Whitney Test** | [29] and [30] | 2 |
| **Wilcoxon Test** | [31] and [52] | 2 |
| **Fisher's Exact Test** | [28] | 1 |
| **PCA Proportion of** | [29] | 1 |

| | | |
|---|---|---|
| **variance** | | |
| **ANOVA Test** | [31] | 1 |
| **2 tail Test** | [46] | 1 |
| **Wilcoxon Paired Test** | [47] | 1 |
| **Mean Re-test** | [57] | 1 |
| **Paired t-test** | [71] | 1 |
| **Kendall's Rank Correlation** | [73] | 1 |
| **Kruskal-Wallis Test** | [81] | 1 |
| **Wilcoxon Signed Rank Test** | [82] | 1 |
| **Logistic Regression** | [91] | 1 |
| **z- Test** | [49] | 1 |

### 3.6.7 RQ6: Do refactoring methods improve the quality attributes?

Before discussing the effect of refactoring methods on software quality attributes, following observations can be noted:

1) While analyzing the selected primary studies, the effectiveness of refactoring methods on two categories of internal and external quality attributes can't be discussed. First, there are some internal as well as external quality attributes that are not considered by more than one researcher. Second, there are some internal and external quality attributes on which the effect is rather contradictory among studies and arriving at either positive or negative impact is not possible. Therefore, the effect of refactoring on such two categories of internal and external quality attributes cannot be included as findings because it is not possible to comment on the effectiveness of applying refactoring methods in improving such quality attributes.
2) During the research, it is identified that it is rather wrong in saying that refactoring always improves quality aspects, both internal and external and that is good for overall software quality. The effect of refactoring methods on various internal as well as external quality attributes is contradictory among different studies. There can be multiple reasons for this scenario. Firstly, many of the researchers do not pay much attention to identifying portions of source-code that are in dire need of refactoring. Some may overly apply to refactor while others may refactor the source-code to only a little extent. One possible reason for this is 81% data-sets used

by the primary studies are open-source projects that are not written by the researchers themselves and it becomes really difficult understanding someone else's code and then also identifying potential problems in it to apply refactoring methods. This has a direct effect on the values of studied internal and external quality attributes and they are poorly affected. Secondly, as identified in RQ4, almost half of the data-sets are small-sized; effect of refactoring is not truly visible to much extent. Say, for example, there is a software-system that is very small and do not have any inheritance feature, then the effect of refactoring on inheritance related metrics can never be seen in such software-system. So, there is a need to include many media and large sized software to actually see the effect of refactoring on quality attributes.

3) Refactoring improves overall quality only when applied carefully while properly identifying places in the source-code that really require refactoring solution. It is therefore rightly said that refactoring is a time demanding, error-prone and tiresome activity that can or cannot improve software quality in general. It depends on the ability of the researcher as well in determining source-code portions that require refactoring.

4) To arrive at the positive effectiveness of refactoring methods, we included only those results that are at least in accordance with two or more researchers. Findings of the effect of refactoring methods on internal and external quality attributes by only single primary study cannot be resulted here. In case of a contradiction of findings (positive/ negative) in different primary studies, we included that effect that is supported by a maximum number of primary studies. This is because techniques like meta-analysis are not suitable for identifying the effect of refactoring methods on internal and external quality attributes. Based on this assumption, the results are discussed.

### a) RQ 6.1: Which quality attributes (internal/ external) are overall benefitted by refactoring?

Following results can be derived from the overall benefit of refactoring methods on internal and external quality attributes on the basis of findings achieved from the primary studies:

i. The overall effect of refactoring methods is available for five out of ten internal quality attributes that are cohesion, coupling, size, complexity, and inheritance. For the rest, five internal quality attributes, i.e. abstraction, polymorphism, data encapsulation, composition, information hiding, and the overall effect of refactoring cannot be concluded.

ii. Cohesion, coupling, size, complexity, and inheritance are improved in approximately 42.62%, 42.46%, 40.82%, 34.09% and 14.28% primary studies respectively.

iii. It can be observed from Table 3.11 that cohesion and coupling are improved in the maximum number of primary studies whereas size and inheritance are improved in the least number of primary studies.

iv. The overall effect of refactoring methods is available for ten out of fifteen external quality attributes that are maintainability, reusability, testability, understandability, flexibility, effectiveness, extensibility, functionality, reliability and modifiability whereas not available for adaptability, completeness, modularity, traceability, and security.

v. As seen from Table 3.12, understandability, maintainability, reliability and reusability are improved in the maximum number of primary studies as they show the highest percentage of improvement in 68.75%, 57.69%, 90% and 40% of primary studies respectively.

Table 3.7 Overall Positive Impact of Refactoring Methods on Internal Quality Attributes

| **Internal Quality Attribute** | **Number of Primary Studies** | **Positive Impact (+)** |
|---|---|---|
| **Coupling** | 31 | [32], [33],[41], [43], [51], [55], [56], [65], [66], [67], [68], [70], [71], [76], [82], [16], [85], [88], [10], [91], |

| | | |
|---|---|---|
| | | [12], [96], [99], [102], [104], [106], [111], [113], [17] and [123] |
| **Cohesion** | 26 | [32], [33],[35], [37], [41], [42], [55], [56], [65], [70], [16], [84], [85], [10], [92], [12], [102], [106], [107], [110], [111], [113], [17], [115] and [123] |
| **Complexity** | 15 | [44], [66], [67], [69], [87], [94], [95], [99], [103], [104], [107], [109], [113], [117] and [123] |
| **Size** | 20 | [33], [44], [57], [66], [67], [70], [71], [77], [78], [82], [16], [84], [87], [88], [10], [98], [99], [107], [117] and [121] |
| **Inheritance** | 4 | [35], [88] and [109] |

Table 3.8Overall Positive Impact of Refactoring Methods on External Quality Attributes

| **External Quality Attribute** | **Number of Primary Studies** | **Positive Impact (+)** |
|---|---|---|
| **Maintainability** | 15 | [53], [55], [56], [58], [59], [60], [63], [65], [69], [77], [85], [98], [99], [107] and [117] |
| **Understandabilit y** | 11 | [35], [40], [49], [60], [70], [74], [81], [10], [102], [108] and [116] |
| **Reusability** | 6 | [43], [45], [70], [10], [108] and [116] |
| **Flexibility** | 5 | [35], [45], [10], [108], [116] |
| **Testability** | 2 | [71] and[73] |
| **Effectiveness** | 5 | [45], [70], [10], [108] and [116] |
| **Extensibility** | 5 | [45], [70], [10], [108] ad n[116] |
| **Functionality** | 4 | [45], [10], [108] and[116] |
| **Reliability** | 9 | [47], [51], [52], [62], [76], [77], [93], [112] and[119] |
| **Modifiability** | 4 | PS33, [47], [49] and [74] |

### b) RQ6.2: Which refactoring method and quality attribute combination yield good result?

As answered in RQ1, 150+ refactoring methods are identified in the primary studies but the effect of only a few refactoring methods is consistent on internal as well as external quality attributes.

Following comments can be made regarding refactoring method and quality attribute combination that yields the good result:

i. Table 3.13 represents the refactoring methods along with corresponding internal and external quality attributes that are benefitted by the application of that refactoring method. Ten refactoring methods: move field, extract method, pull up method, pull-down method, consolidate conditional expression, remove assignment to parameter, move method, extract class, encapsulate field and inline class are listed. They are considered to be improving corresponding internal and external quality attributes listed along with them in Table 3.13 in the majority of the primary studies. On the other hands, refactoring methods not listed have either contradictory effects on internal and external quality attributes or are studied by only one primary study. So, it is not possible to list them here.

Table 3.9 Refactoring Methods and their impact on Quality Attributes

| Refactoring Method | Positive Impact | | Negative Impact | |
|---|---|---|---|---|
| | Internal Quality Attribute | External Quality Attribute | Internal Quality Attribute | External Quality Attribute |
| **Move Field** | Cohesion | Reusability,<br>Understandability | | |
| **Extract Method** | Cohesion, Size | Modifiability,<br>Maintainability,<br>Understandability<br>Reusability,<br>Testability | Complexity,<br>Coupling | |

| | | | | |
|---|---|---|---|---|
| **Pull Up Method** | | | Coupling, Inheritance | |
| **Pull Down Method** | | Reusability, Reliability | | |
| **Consolidate Conditional Expression** | | Maintainability, Understandability | Complexity, Size | |
| **Remove Assignment to Parameter** | | Maintainability | | |
| **Move Method** | Coupling, Cohesion, Size, Complexity, Inheritance | Reusability, Understandability, Flexibility, Extensibility | | |
| **Extract Class** | Cohesion, Coupling, Complexity | Understandability, Maintainability | | |
| **Encapsulate Field** | Coupling, Cohesion, Complexity | Maintainability, Reusability, Testability | | |
| **Inline Class** | Cohesion, Coupling | Reusability, Understandability | Size | |

ii. Effect of refactoring methods on internal quality attributes based on the results of the majority of primary studies:

**Cohesion:** It is benefitted by move field, extract method, move method, extract class, inline class and encapsulate field tends to improve cohesion.

**Coupling:** It is benefitted by move method, extract class, and encapsulate field but deteriorated by extract method and pull up method.

**Complexity:** It is improved by move method, extract class and encapsulate field but deteriorated by extract method and consolidate conditional expression.

**Size:** It is improved by extract method, move method but ill-affected by inline class and consolidate conditional expression.

**Inheritance:** It tends to improve using move method but deteriorate using pull up method.

iii. Effect of refactoring methods on external quality attributes based on the results of the majority of primary studies:

**Reusability:** Move field, extract method, pull down field, move method, encapsulate field and inline class tend to improve reusability.

**Maintainability:** Extract method, remove assignment to parameter, consolidate conditional expression, extract class and encapsulate field improves maintainability.

**Understandability:** Move field, extract method, move method, consolidate conditional expression, extract class and inline class improve understandability.

**Testability:** Encapsulate field and extract method improve testability.

**Flexibility:** Move method improves flexibility whereas the effect on flexibility by other refactoring methods is not general.

**Extensibility:** Move method improves extensibility.

**Reliability:** Pull down method improves reliability.

iv. It can be observed that extract class, encapsulate field, inline class and extract method are identified to be improving the maximum number of quality attributes (internal and external) based on the findings of the majority of primary studies.

# CHAPTER 4

# RESEARCH METHODOLOGY

In this section, the methodology that is followed in the current empirical research has been discussed elaborately. Figure 4.1 depicts all the events followed while conducting the empirical research. Section 4.1 details about the selected C&K metrics to capture object-oriented characteristics of the software-system. Section 4.2 explains the process of empirical data collection. Section 4.3 describes the selected bad-smells along with the way of detecting way. Section 4.4 discusses the refactoring methods that are selected as a remedy to bad-smells present in source-code and Section 4.5 explains the way of evaluating Quality Depreciation Index Rule (QDIR) for each class by which we can assign priorities to them. Finally, in Section 4.6, all the classes are prioritized based on their QDIR values into one of the four types (Critical, Bad, Mild and Low) where the Critical value for a class means that it requires instant refactoring whereas the Low value for a class means it requires no or very little refactoring.

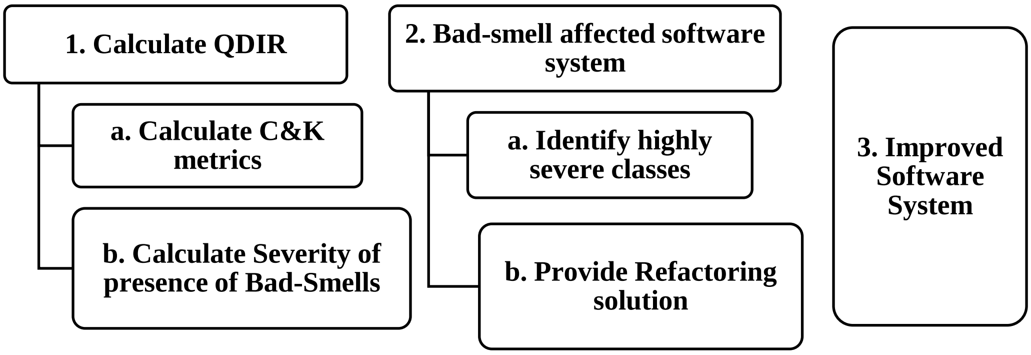

1. Calculate QDIR
a. Calculate C&K metrics
b. Calculate Severity of presence of Bad-Smells
2. Bad-smell affected software system
a. Identify highly severe classes
b. Provide Refactoring solution
3. Improved Software System

Figure 4.1 Flow of Research Events

## 4.1 OBJECT-ORIENTED CHARACTERISTICS CALCULATION

Software metric is defined as the characteristic feature of the software product, process, or resource [145]. Object-oriented characteristics for given software are captured by software metrics. These object-oriented metrics are indicators of inheritance, coupling, abstraction level, cohesion etc. for the system under study. There are a large number of software metrics available in the literature to capture the design characteristics of the software-systems like Li & Henry metrics [7], MOOD's metrics [4], Chidamber & Kemerer metrics [5], Lorenz & Kidd metrics [6]. We have selected Chidamber & Kemerer (C&K) metric suite that evaluates the structural quality of object-oriented software-systems. It consists of a set of six metrics namely Response for a Class (RFC), Depth of Inheritance Tree (DIT), Lack of Cohesion on Methods (LCOM), Number of Children (NOC), Coupling between Objects (CBO) and Weighted Method per Class (WMC). It is worthwhile mentioning that DIT and NOC capture inheritance, CBO and RFC capture the coupling, LCOM captures cohesion and WMC captures complexity (size). These metrics have been computed using Understand metric-tool [146] and the computed values are then compared with the threshold values of the given metrics. These threshold values for C&K metrics are suggested by Shatnawi et al. [147] and are listed in Table 4.1 below. The classes that have computed values of metrics higher than their threshold values are treated as that of compromised quality. And finally, these metric values together with the severity of presence of bad-smells are used in order to prioritize classes.

Table 4.1 C&K metrics along with threshold values

| C&K metric | Definition | Threshold Value |
|---|---|---|
| **RFC (Response For a Class)** | In response to a message received by an instance of a given class, it gives the total count of class' methods that can potentially get executed. As a result, it captures coupling. | 44 |
| **DIT (Depth of Inheritance Tree)** | It is the maximum path-length from a given node to its root node in the entire inheritance tree. It, therefore, captures inheritance for a class. | 7 |
| **LCOM (Lack of Cohesion on** | It is the count of pair-wise methods in a class which does not have shared instance variables minus the | 7 |

| | | |
|---|---|---|
| **Methods)** | count of pairs of methods in that class that has shared instance variables. It is used to capture cohesion in a class. | |
| **NOC (Number of Children)** | It represents the total number of classes that inherit a given class i.e. the count of all its children. | 1 |
| **CBO (Coupling between Objects)** | It gives the number of distinct classes that are coupled to a given class (i.e. uses methods/attributes/both of coupled class) except those classes that are inheritance based related to the given class. | 13 |
| **WMC (Weighted Methods per Class)** | It computes the sum of complexities of all the methods present in a given class. It, therefore, captures the complexity of the entire class. | 24 |

## 4.2 EMPIRICAL DATA COLLECTION

This empirical study uses eight open-source software-systems for evaluation that are written in Java language with different domains and sizes. The reason behind selecting a larger set of software-systems for analysis is due to a wider acceptability of results across software-systems with different domains and sizes. Table 4.2 shows the details of eight selected software-systems. The analysis and review of the software-systems are done using Eclipse [148] tool. All software-systems are downloaded from SourceForge [149] except Frogger that is downloaded from GitHub [150].

A brief description of the selected software-systems is provided as under:

1) **Frogger:** Frogger is a popular open-source game that requires a player (treated as a hopping frog) to cross the river safely with certain hurdles in the way like moving cars and flying objects among others in order to win. It also offers interesting global warming effects like wind-gusts and over-heated pavements to make it even more interesting. It is the only small project selected for the research work and is written in Java consisting of 20 classes and 1284 LOC (lines of code) in studied version.
2) **JEdit-5.5.0:** JEdit is an open-source text-editor for programmers with built-in macro language and extensible plug-in architecture that allows downloading

plug-ins easily from its Plug-in Manager. It allows a feature to automatically indent and highlight syntax for over two hundred languages and so many character encodings including ACIII, Unicode, UTF etc. It is written in Java and has 1336 classes and 319404 LOC in studied version.

3) **JGraphX-3.9.3:** JGraph is an open-source Java Swing based library under BSD license. It allows visualizing and interacting with graphs having nodes and edges. It also supports XML stencils, automatic layouts, and import/export convenience. One can write a variety of applications like organizational-charts, UML tools, workflow editors to name a few. Its studied version has 572 classes and 70,216 LOC.

4) **Jsettlers-1.1.20:** Jsettlers is an open-source board game- Settlers of Catan's web-based version. It is based on client-server architecture and is written in Java language. It supports multiple games to be played parallelly between players and computer-based opponent. Its studied version consists of 255 classes and 54,226 LOC.

5) **jVLT-1.3.3:** jVLT is an open-source vocabulary learning tool that allows improving one's vocabulary. One can generate one's own dictionary of words and also participate in vocabulary quizzes. It is written in Java and studied version contains 409 classes and 23,858 LOC.

6) **jHotDraw-7.0.6:** jHotDraw is a popular open-source Java-based GUI framework for technical as well as structural graphics. It was initially developed as a design exercise but gained instant popularity. The studied version has 1068 classes and 97553 LOC.

7) **Xerces-2.11.0:** Xerces is an open-source XML parser that simplifies the reading and writing of XML data. It allows a shared library to parse, create, analyze and validate XML documents using DOM, SAX and SAX2 APIs with high performance and scalability. It is written in Java and studied version contains 976 classes and 141,609 LOC.

8) **ArtOfIllusion-3.0.3:** ArtOfIllusion is an open-source 3D modeling and rendering studio. Its features include subdivision surface modeling tools, skeleton-based animations and graphical language support for procedural textures and material objects.

Table 4.2 Details of the software-systems under study

| Sr. No. | Data-set | Source | Number of Classes | Lines of Code |
|---|---|---|---|---|
| 1 | Frogger | https://github.com/denodell/frogger | 20 | 1284 |
| 2 | JEdit-5.5.0 | https://sourceforge.net/projects/jedit/ | 1336 | 319,404 |
| 3 | JGraphX-3.9.3 | https://github.com/jgraph/jgraphx | 572 | 70,216 |
| 4 | Jsettlers-1.1.20 | https://sourceforge.net/projects/jsettlers2/ | 255 | 54,226 |
| 5 | jVLT-1.3.3 | https://sourceforge.net/projects/jvlt/ | 409 | 23,858 |
| 6 | jHotDraw-7.0.6 | https://sourceforge.net/projects/terppaint/ | 1068 | 97,553 |
| 7 | Xerces-2.11.0 | https://sourceforge.net/projects/xercesframework/ | 976 | 141,609 |
| 8 | ArtOfIllusion-3.0.3 | https://sourceforge.net/projects/aoi/ | 893 | 119,203 |

The descriptive statistics of the object-oriented metrics for each of the six selected software-systems is computed. Table 4.3 represents the maximum, mean, minimum and standard deviation values for C&K metrics for all the six selected software-systems.

The interesting observations and revelations from the descriptive statistics of C&K metrics of software-systems in Table 4.3:

1) Inheritance in the software-systems is limited as for mean and mean values for DIT are very less. The maximum level of inheritance present is 2.

2) Value of LCOM for classes is quite large. Therefore, cohesion in classes is less. The maximum value of cohesion for almost all systems is 100.
3) Coupling between classes is high as mean and median of CBO is large.
4) Complexity is high among classes as WMC and RFC are quite large.

Table 4.3 Descriptive Statistics of C&K metrics of selected Software-systems

| **Software-system** | **Metric Value** | **Min** | **Max** | **Mean** | **Standard Deviation** |
|---|---|---|---|---|---|
| **Frogger** | **CBO** | 1 | 13 | 5.4 | 3.69 |
| | **DIT** | 0 | 2 | 1.15 | 0.93 |
| | **LCOM** | 0 | 100 | 27.91 | 39.16 |
| | **NOC** | 0 | 10 | 0.5 | 2.24 |
| | **RFC** | 16 | 33 | 18.8 | 3.79 |
| | **WMC** | 1 | 17 | 4.8 | 3.68 |
| **jedit5.5.0** | **CBO** | 0 | 189 | 9.9 | 12.23 |
| | **DIT** | 0 | 3 | 0.23 | 0.49 |
| | **LCOM** | 0 | 100 | 27.46 | 34.18 |
| | **NOC** | 0 | 38 | 0.22 | 1.68 |
| | **RFC** | 0 | 987 | 134.26 | 16.08 |
| | **WMC** | 0 | 351 | 6.1 | 16.08 |
| **JGraphX-3.9.3** | **CBO** | 0 | 80 | 7.94 | 9.54 |
| | **DIT** | 0 | 3 | 0.19 | 0.47 |
| | **LCOM** | 0 | 100 | 27.91 | 39.16 |
| | **NOC** | 0 | 13 | 0.21 | 1.08 |
| | **RFC** | 12 | 924 | 70.41 | 182.72 |
| | **WMC** | 0 | 924 | 70.41 | 182.72 |
| **Jsettlers 1.1.20** | **CBO** | 0 | 173 | 9.98 | 19.95 |
| | **DIT** | 0 | 3 | 0.57 | 0.67 |
| | **LCOM** | 0 | 100 | 44.25 | 31.41 |
| | **NOC** | 0 | 84 | 0.48 | 5.3 |
| | **RFC** | 13 | 752 | 117.56 | 204.21 |
| | **WMC** | 0 | 159 | 11.31 | 19.49 |

| | | | | | |
|---|---|---|---|---|---|
| **jVLT-1.3.3** | **CBO** | 0 | 126 | 9.99 | 12.63 |
| | **DIT** | 0 | 4 | 0.45 | 0.72 |
| | **LCOM** | 0 | 100 | 39.88 | 34.95 |
| | **NOC** | 0 | 16 | 0.34 | 1.41 |
| | **RFC** | 0 | 905 | 156.27 | 282.03 |
| | **WMC** | 0 | 52 | 4.88 | 5.38 |
| **jHotDraw 7.0.6** | **CBO** | 0 | 98 | 11.15 | 11.32 |
| | **DIT** | 0 | 6 | 0.56 | 1.07 |
| | **LCOM** | 0 | 100 | 32.46 | 37.94 |
| | **NOC** | 0 | 28 | 0.33 | 1.56 |
| | **RFC** | 0 | 871 | 85.56 | 183.93 |
| | **WMC** | 0 | 80 | 6.65 | 9.25 |
| **Xerces 2.11.10** | **CBO** | 0 | 115 | 9.18 | 11.66 |
| | **DIT** | 0 | 5 | 1.02 | 1.6 |
| | **LCOM** | 0 | 100 | 50.41 | 38.48 |
| | **NOC** | 0 | 52 | 0.41 | 2.36 |
| | **RFC** | 12 | 849 | 61.5 | 88.91 |
| | **WMC** | 0 | 129 | 10.22 | 14.12 |
| **ArtOfIllusion 3.0.3** | **CBO** | 0 | 134 | 10.78 | 12.66 |
| | **DIT** | 0 | 5 | 0.45 | 0.69 |
| | **LCOM** | 0 | 100 | 33.72 | 35.07 |
| | **NOC** | 023 | 55 | 0.26 | 2. |
| | **RFC** | 0 | 880 | 33.73 | 67.32 |
| | **WMC** | 0 | 108 | 8.15 | 11.03 |

## 4.3 BAD-SMELL DETECTION

Bad-smells are poor design structures violating design principles that make the system complex and difficult to manage. Often people confuse them with bugs but they are not bugs and unlike bugs, they do not stop a program from executing. Instead, they, if not handled properly, can weaken a program and attract bugs or even much serious problems. Fowler [1] appropriately describes a bad-smell as a surface indication that usually corresponds to a deeper problem in the system". It reflects poor design in the software-system that has poor readability & understand-ability, high level of complexity, less maintainability making it poor its quality. Fowler had provided a catalog of twenty-two bad-smells. In this study, ten different bad-smells are selected and they are identified for each class of all the eight studied software-systems. Table 4.4 lists all the selected bad-smells along with the possible refactoring methods to remove it. Also, three Eclipse plug-ins namely: JDeodorant [151], JSpirit [152] and Robusta [153] are used to detect bad-smells from all the classes of the selected software. Out of the ten selected bad-smells. In the current research, four bad-smells- God Class, Feature Envy, Long Method and Type Checking are those that were also selected by Malhotra et al. [8] in their preliminary study whereas six new bad-smells- Nested Try Statement, Empty Catch Block, Shotgun Surgery, Refused Parent Bequest, Brain Method, and Intensive Coupling have been selected in the current study to take into consideration the severity of a wider set of bad-smells in the selected software-systems.

Table 4.4 Selected Bad-smells and their respective Refactoring Solution

| **Bad-smell** | **Definition** | **Bad-smell Detection tool** | **Refactoring Method** |
|---|---|---|---|
| **God Class** | It refers to a large class in a system that majorly controls all of its intelligence/ workings. | JDeodorant | Extract Class |
| **Feature Envy** | It occurs when a method relies on another object's data more than that of its own. | | Move Method |
| **Long Method** | It is a method that is very large in a class and has got so much to do instead of performing a single responsibility well. | | Extract Method |
| **Type Checking** | It occurs when multiple methods are assigned responsibility to perform a single functionality instead of the need for only one method. | | Replace Type Code with State/Strategy |
| **Nested Try** | It is a chain of a large number of try statements just like an if-else ladder. | Robusta | Extract Method |

| | | | |
|---|---|---|---|
| **Statement** | | | |
| **Empty Catch Block** | It is a catch block without body showing negligence of the developer in handling the respective error condition. | | Re-throw with Exception |
| **Shotgun Surgery** | It occurs when a method of one class is being called extensively by methods of another class. | JSpirit | Move Method |
| **Refused Parent Bequest** | It happens when a child class either can't inherit base class' data or it uses a small subset of methods defined in its base class and fields of its base class. | | Replace Inheritance with Delegation |
| **Brain Method** | It is a method where the major functionality of a class resides. | | Move Method |
| **Intensive Coupling** | It occurs when a client-method tends to communicate too much with one or more classes it depends upon. | | Move Method |

## 4.4 REFACTORING

As defined by Fowler [1], Refactoring is a technique that does not change how a software-system behaves functionally yet improves its internal structure making it much easier to read, understand, and manage. Following six refactoring methods are selected to remove the studied bad-smells. A brief description of all of them is given below in Table 4.5:

Table 4.5 Details of Refactoring Methods

| Refactoring Method | Definition |
|---|---|
| **Move Method** | Create a new method in a class that most frequently use a given method in a particular class and place the content of this method in the newly created class. Finally, either completely remove the method from that particular class or provide it with reference to the new method, if required. |
| **Extract Class** | For a given class that has too much responsibility to handle, move all the desired fields and methods in a newly created |

| | |
|---|---|
| | class. |
| **Extract Method** | In a long method where some part of the code can be grouped together, move that part of the code in a newly created method and make a reference for this new method in an old existing method. |
| **Replace Type Code with State/Strategy** | Replace type-code with state-object in case there is a coded-type that affects functionality base class can't be used to avoid the related issue. Also, plug-in another state-object if it is required to replace a field-value with its type-code. |
| **Replace Inheritance with Delegation** | Put base-class' object in a new field and delegate methods to base-class' objects instead of having generalization relationship between class and get-rid-of inheritance. |
| **Re-throw with Exception** | Re-throw an exception in catch block in spite of keeping it empty or printing an error statement. |

## 4.5 QUALITY DEPRECIATION INDEX RULE (QDIR)

A new metric, Quality Depreciation Index Rule (QDIR) with certain modifications, as suggested by Malhotra et al. [8] is calculated based on the severity of presence of bad-smells and object-oriented characteristics of software-systems under study. The current study takes into consideration ten most common bad-smells with available tool support in contrary to four bad-smells considered by Malhotra et al. [8] in their preliminary study. The motivation behind selecting ten bad-smells is to remove the effect of a wider set of bad-smells from the software-systems that are commonly present in them and also have proper detection tool support. QDIR further helps in prioritizing classes so that refactoring methods can be applied to severely affected classes only saving considerable maintenance cost under strict time constraints and most importantly improving overall software quality.

### 4.5.1 Calculation of Base of Bad-smell(BoB′)

Malhotra et al. [8] in their preliminary study calculated BoB by giving equal Smell Weightage (SW) of 0.25 to all the bad-smells and then taking an average of SW of four identified bad-smells as shown in equation (1).

$$\text{BaseofBadSmell (BoB)} = \frac{1}{4}\sum_{i=1}^{4} \text{SWi} \qquad (1)$$

In the current study, we have detected ten different bad-smells in the classes and given equal Smell Weightage (SW) to all the bad-smell which is 0.10. Afterward, BoB′ for a class is calculated by taking an average of SW of all the detected bad-smells in that class to that of the ten selected bad-smells as shown in equation (2).

$$\text{BaseofBadSmell}' \text{ (BoB')} = \frac{1}{10}\sum_{i=1}^{10} \text{SWi} \qquad (2)$$

### 4.5.2 Calculation of Base of Metric (BoM)

All the selected six metric values are first identified for each class of all the software-systems under study as discussed in Section 4.1 and then compared to their corresponding threshold values. This computed value is then divided by its corresponding threshold value to arrive at Metric Value (MV) as shown in equation (3).

$$\text{Metric Value(MV)} = \frac{\text{CalculatedMetric}}{\text{ThresholdMetric}} \qquad (3)$$

After this, BoM can be calculated by capturing average values of MV for all six selected metrics as shown in equation (4).

$$\text{BaseofMetric(BoM)} = \frac{1}{6}\sum_{j=1}^{6} \text{MVj} \qquad (4)$$

### 4.5.3 Calculation of Quality of Depreciation Rule (QDIR)

Finally, QDIR is calculated using (2) and (4) as given in equation (4)

$$\text{Quality Depreciation Index Rule (QDIR )} = \frac{1}{2}\text{BoB'} + \frac{1}{2}\text{BoM} \qquad (5)$$

QDIR is calculated in the same as calculated by Malhotra et al. [8]. The only difference here is the modified BoB′ metric instead of BoB metric.

**ILLUSTRATION BY EXAMPLE**

To illustrate an example, a java class named ObjectViewer from ArtOfIllusion-3.0.3 software-system is considered.

Firstly, BoB′ for the selected class is calculated. For this, we identified the presence of ten bad-smells under study in the class and found that it is affected by five bad-smells, namely Long Method, Shotgun Surgery, Refused Parent Bequest, Intensive Coupling, and Brain Method.

$$\mathrm{BoB'} = \frac{(0.1 + 0.1 + 0.1 + 0.1 + 0.1)}{10}$$

$$\therefore \qquad \mathrm{BoB'} = 0.05$$

Now, BoM is calculated for ObjectViewer class by first calculating MV for the class. $\mathrm{MV}_{\mathrm{metric}}$ is calculated by computing its values and dividing it by respective threshold of that metric. LCOM, WMC, DIT, NOC and RFC for ObjectViewer are 81, 22, 2, 1 and 118 respectively.

$\therefore \mathrm{MV}_{\mathrm{LCOM}} = 81/7 = 11.571$,

$\mathrm{MV}_{\mathrm{WMC}} = 22/24 = 0.917$,

$\mathrm{MV}_{\mathrm{CBO}} = 30/13 = 2.307$,

$\mathrm{MV}_{\mathrm{DIT}} = 2/7 = 0.286$,

$\mathrm{MV}_{\mathrm{NOC}} = 1/1 = 1$ and

$\mathrm{MV}_{\mathrm{RFC}} = 118/44 = 2.681$.

$$\mathrm{BoM} = \frac{11.571+0.917+2.307+0.286+1+2.681}{6}$$

$\therefore \mathrm{BoM} = 18.762/6 = 3.127$.

Finally, QDIR is calculated by taking the average of BoB′ and BoM.

QDIR= (0.05+3.127)/2

QDIR= 3.177/2 = 1.588

Likewise, QDIR is calculated for all the available classes in the software-system and based on the value of QDIR, ranges for the four severity levels i.e. Critical, High, Mild and Low, ranges are identified.

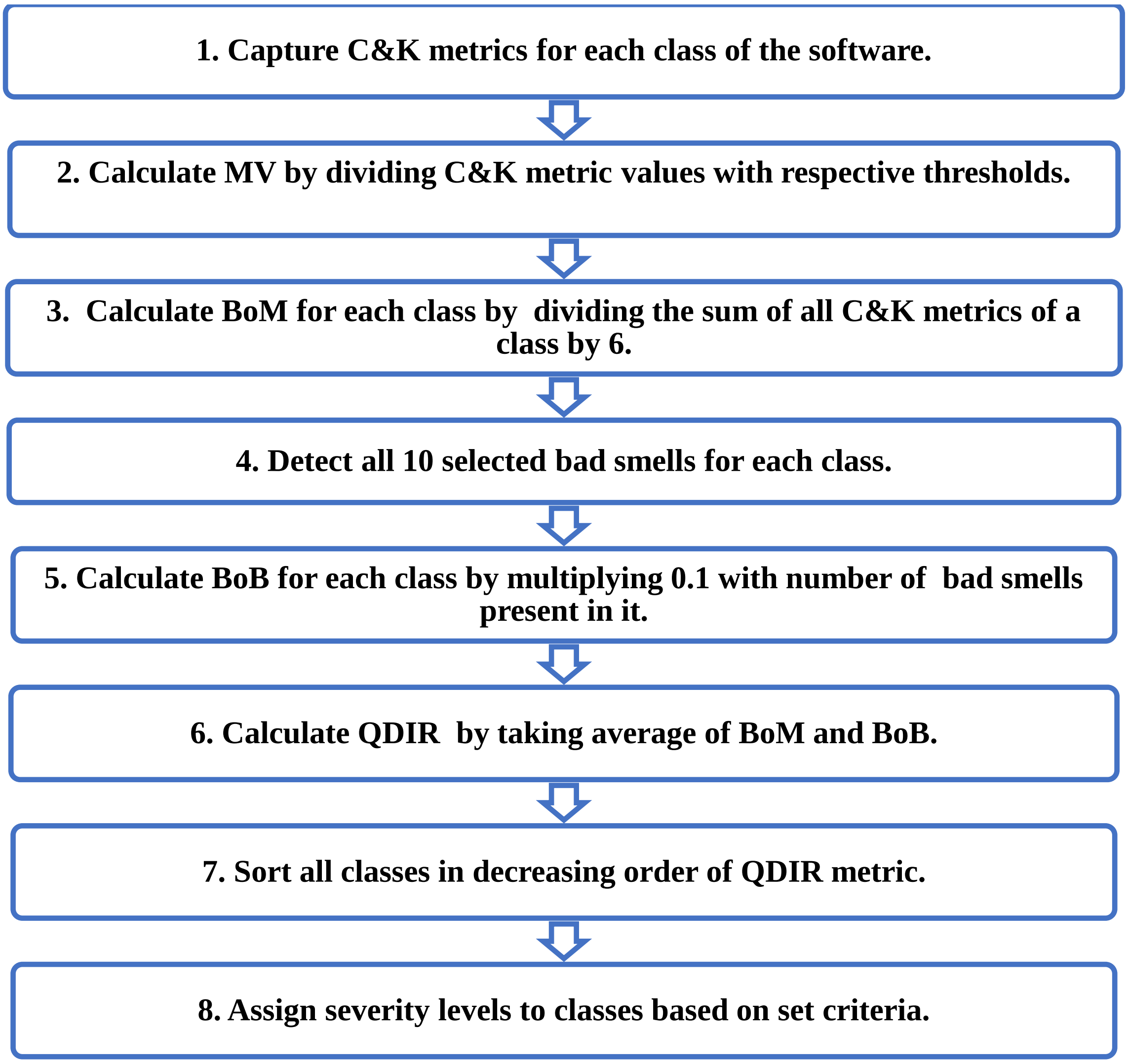
## 4.6 Prioritization of Classes

This study aims to save cost and time by refactoring only highly severe classes having poor object-oriented characteristics as well as high level of severity of presence of bad-smells instead of refactoring the entire software-system as per Algorithm is shown in Figure 4.2. As per the algorithm, we first compute Chidamber & Kemerer (C&K) metrics for the classes and then calculate their Metric Value (MV) by dividing C&K metric-values with their respective thresholds. MV is then used to find (Base of Metric (BoM) by finding the average of MVs computed for a class. Afterward, ten selected bad-smells are detected in all the classes. Each bad-smell is given equal Smell

Weightage (SW) of 0.1. The Base of Bad-smell (BoB') is calculated by finding the average of SWs present in a class. Quality Depreciation Index Rule (QDIR) metric is then computed by adding BoM and BoB' and dividing it by two. QDIR then helps in assigning priority to classes so that refactoring to only a small subset of software-system's classes can be provided. For this purpose, we arrange the classes in decreasing order of their QDIR value. A higher value of QDIR for a class indicates critical flaws in the design and higher priority is assigned to it whereas a class having a lower value of QDIR signifies better design and assigns less priority to it. Four severity levels are suggested for the classes of the studied software-system as given in Table 4.6.

Table 4.6 Four Severity Levels for Classes

| Severity Level | Description | Range |
|---|---|---|
| **Critical** | Classes that are highly severe having and require refactoring at the earliest. | 10% |
| **High** | Classes that are comparatively less severe and require refactoring but not at the earliest. | 25% |
| **Mild** | Classes that require refactoring but not instantly. | 25% |
| **Low** | Classes that are least severe having no harmful bad-smells. | 40% |



1. Capture C&K metrics for each class of the software.

2. Calculate MV by dividing C&K metric values with respective thresholds.

3. Calculate BoM for each class by dividing the sum of all C&K metrics of a class by 6.

4. Detect all 10 selected bad smells for each class.

5. Calculate BoB for each class by multiplying 0.1 with number of bad smells present in it.

6. Calculate QDIR by taking average of BoM and BoB.

7. Sort all classes in decreasing order of QDIR metric.

8. Assign severity levels to classes based on set criteria.

Figure 4.2 Algorithm to prioritize classes based on QDIR

# CHAPTER 5

# RESULTS AND DISCUSSIONS

This chapter is dedicated to report the findings and discuss the results arrived by empirically evaluating our approach on eight open-source software-systems written in java language. Research questions imposed at the start of conducting results are discussed in detail.

The current empirical study is conducted on eight open-source java-based software-systems of varying sizes and domains to reduce the maintenance-effort during refactoring process by utilizing Quality Depreciation Index Rule (QDIR) metric. This section is dedicated to discussing the results and answering the research questions that were raised before conduction the empirical research.

**RQ1: What percentage of classes is poorly affected by bad-smells?**

Figure 5.1 represents the distribution of bad-smells in selected software-systems. It is found that on an average, 30% of the classes are poorly affected by bad-smells. As a considerable amount of classes are poorly-affected by the bad-smells, there is a need for removing bad-smells. This revelation supports our attempt to remove bad-smells from source-code by prioritizing them based on the severity of presence of bad-smells.

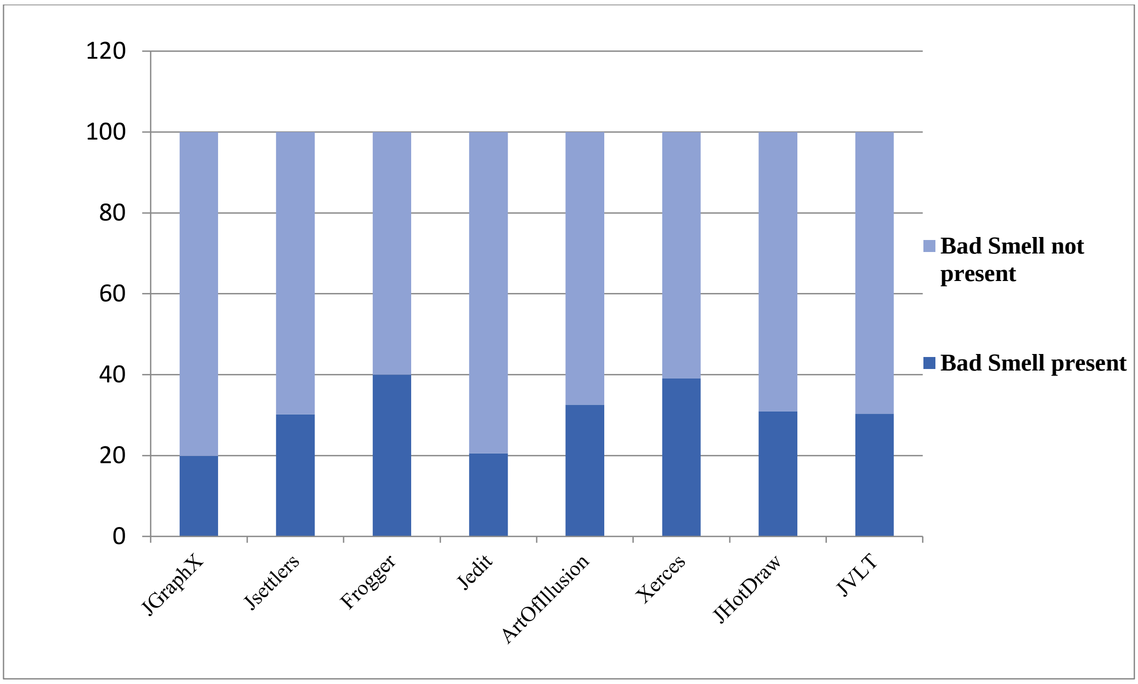


Figure 5.1 Percentages of Bad-smells in Software-systems

### RQ2: Which bad-smell predominantly harms classes in the selected software-systems?

Figure 5.2 – 5.9 depicts the presence of ten selected bad-smells across eight studied software-systems. As bad-smells are present in all the softwares in significant amount, remedial actions are required in an efficient manner.

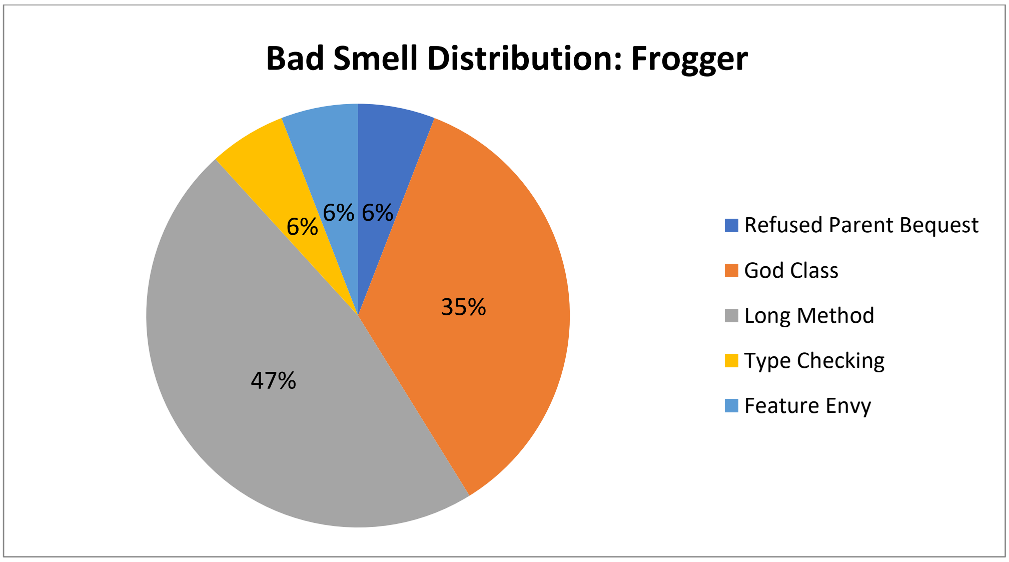


Figure 5.2 Distribution of selected Bad-Smells in Frogger

Frogger is the only small-sized software and it has the presence of five bad-smells out of ten studied bad-smells as seen in Figure 5.2. The long method followed by god class bad-smell is highly dominant across it.

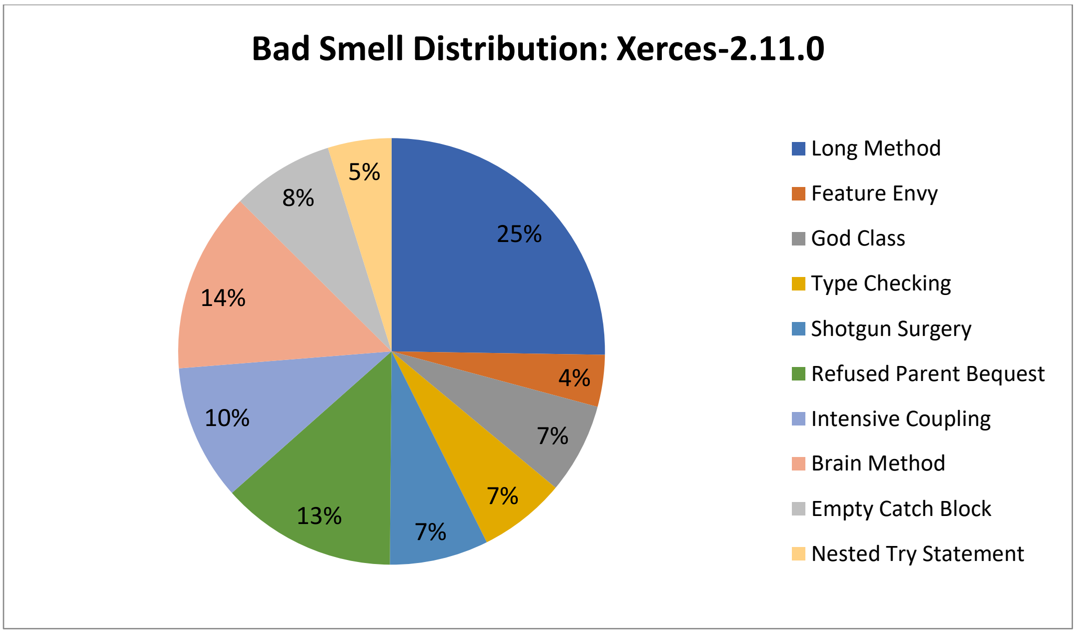


Figure 5.3 Distribution of selected Bad-Smells in Xerces-2.11.0

In Xerces-2.11.0, it can be observed that brain method and refused parent bequest are predominant whereas feature envy is present in less number of classes (see Figure 5.3).

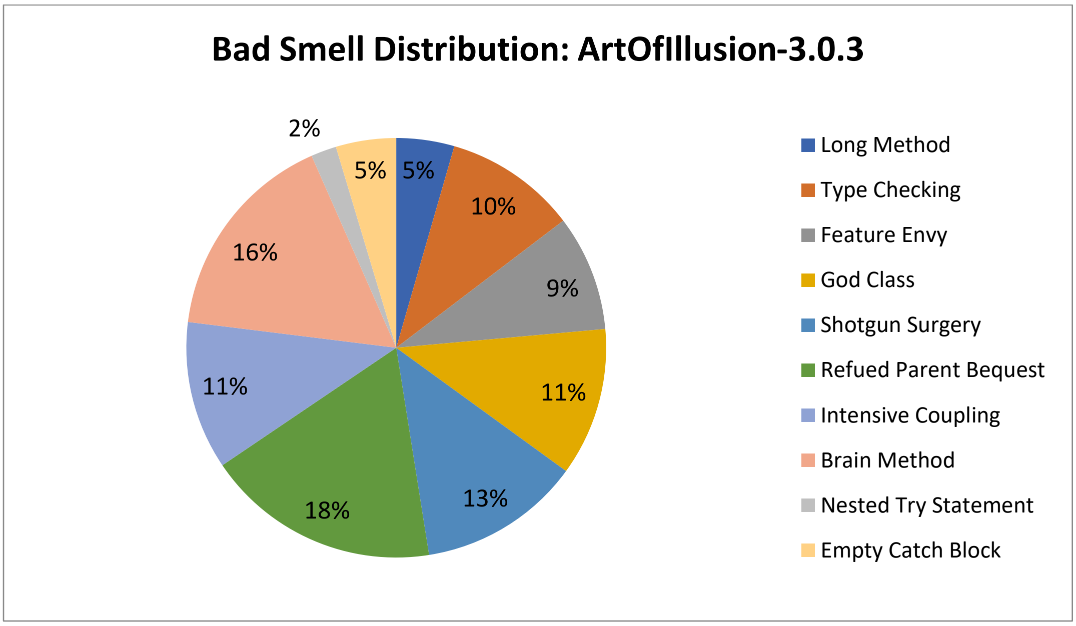


Figure 5.4 Distribution of selected Bad-Smells in ArtOfIllusion-3.0.3

In Figure 5.4, it can be seen that refused parent bequest and long method are present in the maximum number of classes whereas feature envy is present in the least number of classes in ArtOfIllusion-3.0.3.

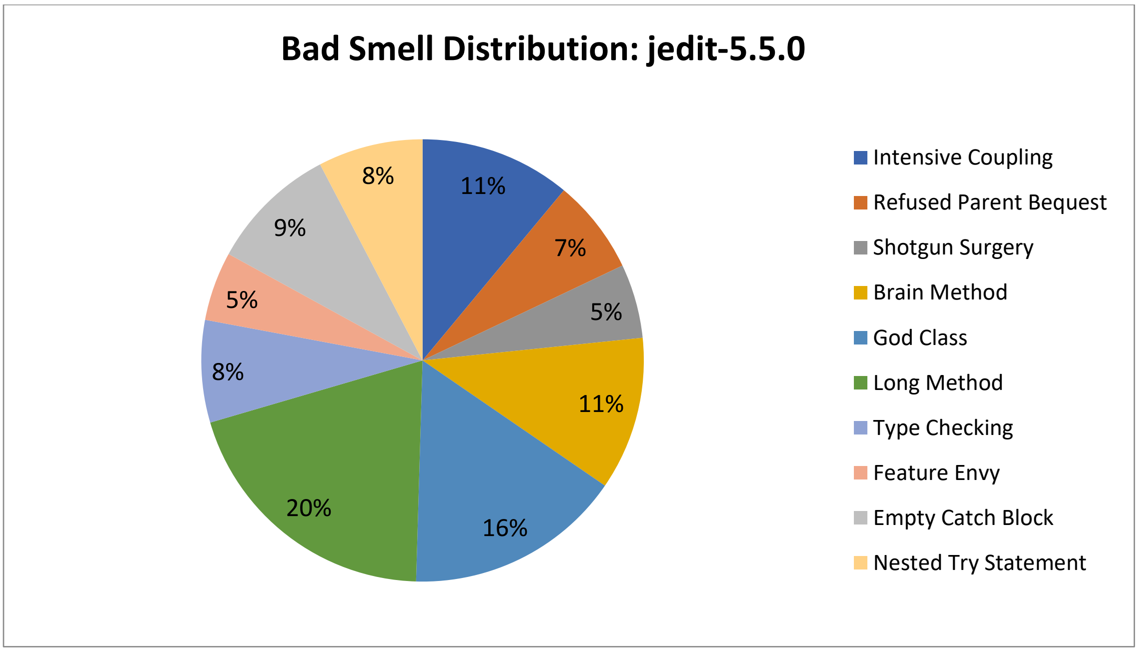


Figure 5.5 Distribution of selected Bad-Smells in JEdit-5.5.0

Jedit-5.5.0 has the maximum number of classes affected by long method and type checking bad-smell and least number of classes affected by empty catch block and feature envy bad-smell (see Figure 5.5).

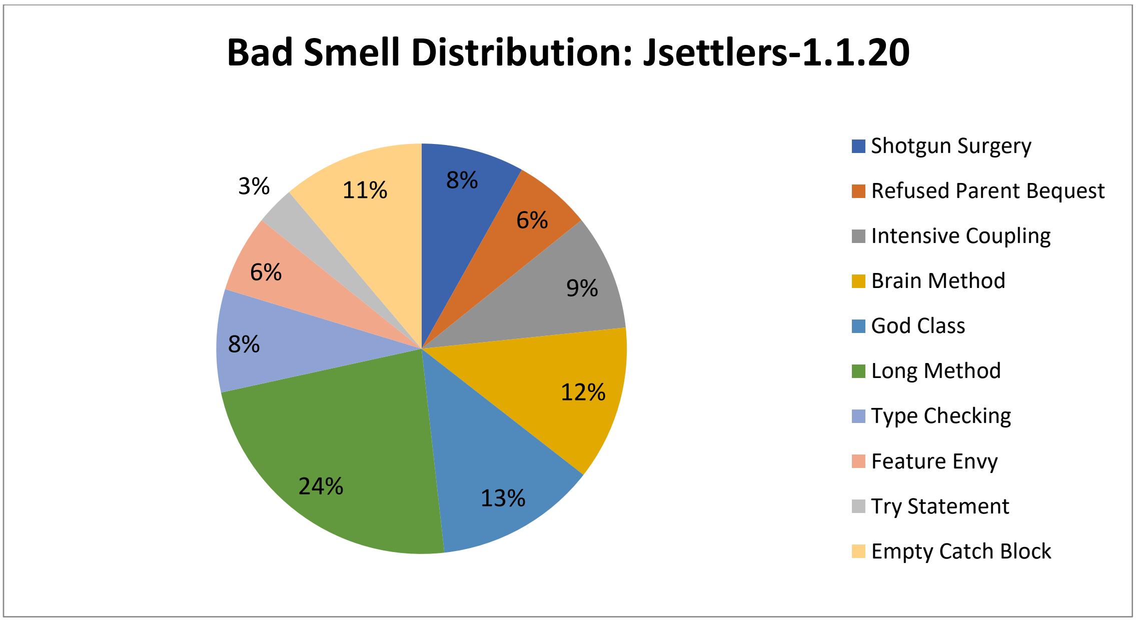


Figure 5.6 Distribution of selected Bad-Smells in JSettlers-1.1.20

Jsettlers-1.1.20 is poorly affected by long method, god class and brain method in the maximum number of classes whereas least affected by intensive coupling bad-smell as visible in Figure 5.6.

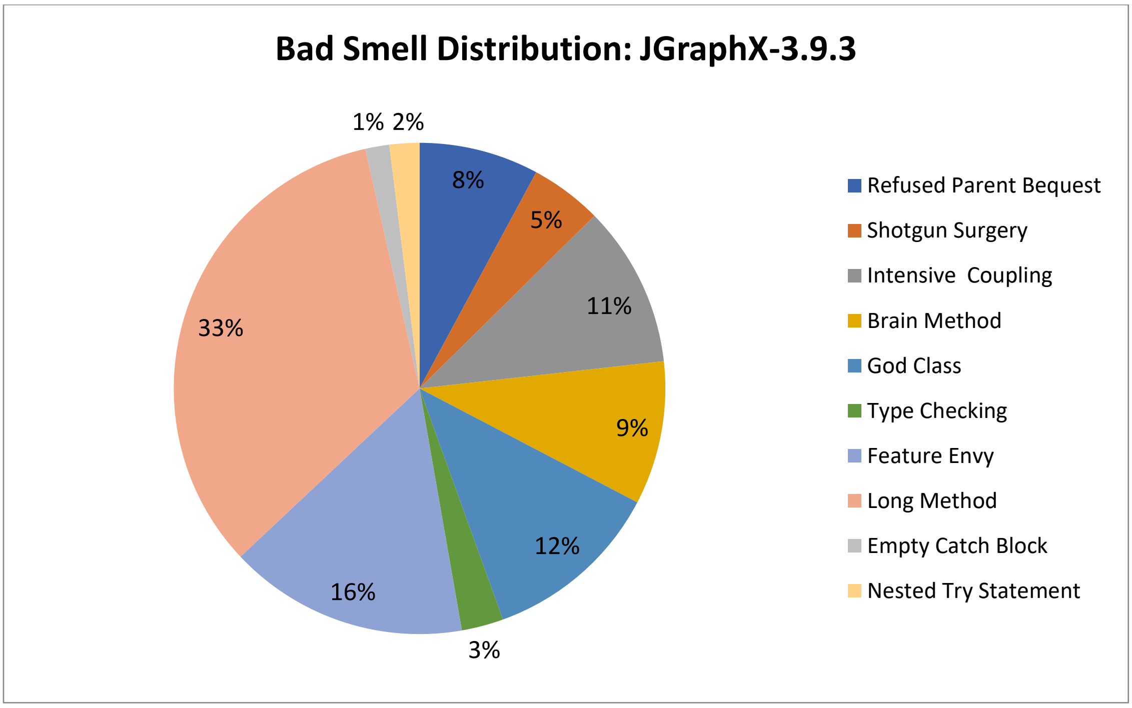


Figure 5.7 Distribution of selected Bad-Smells in JGraphX-3.9.3

In Figure 5.7, it can be seen that long method is present in a very large number of classes whereas empty catch block and nested try statement is present in a negligible number of classes of JGraphX-3.9.3.

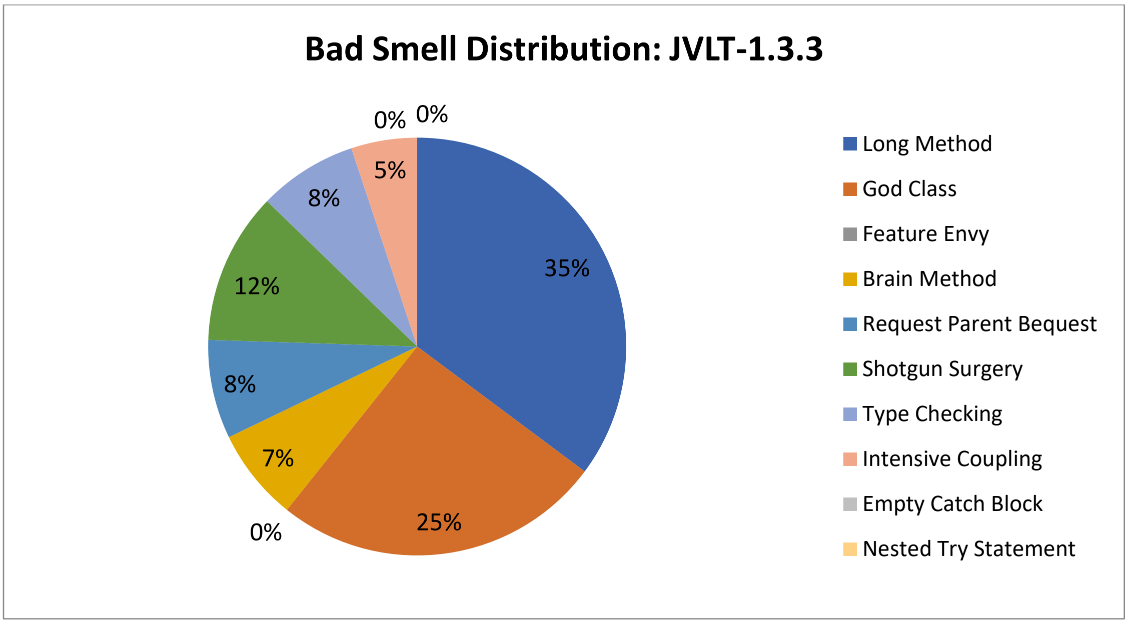


Figure 5.8 Distribution of selected Bad-Smells in JVLT-1.3.3

In JVLT-1.3.3, a large portion of classes is badly affected by long method and god class whereas not at all by empty catch block and nested try statement as shown in Figure 5.8.

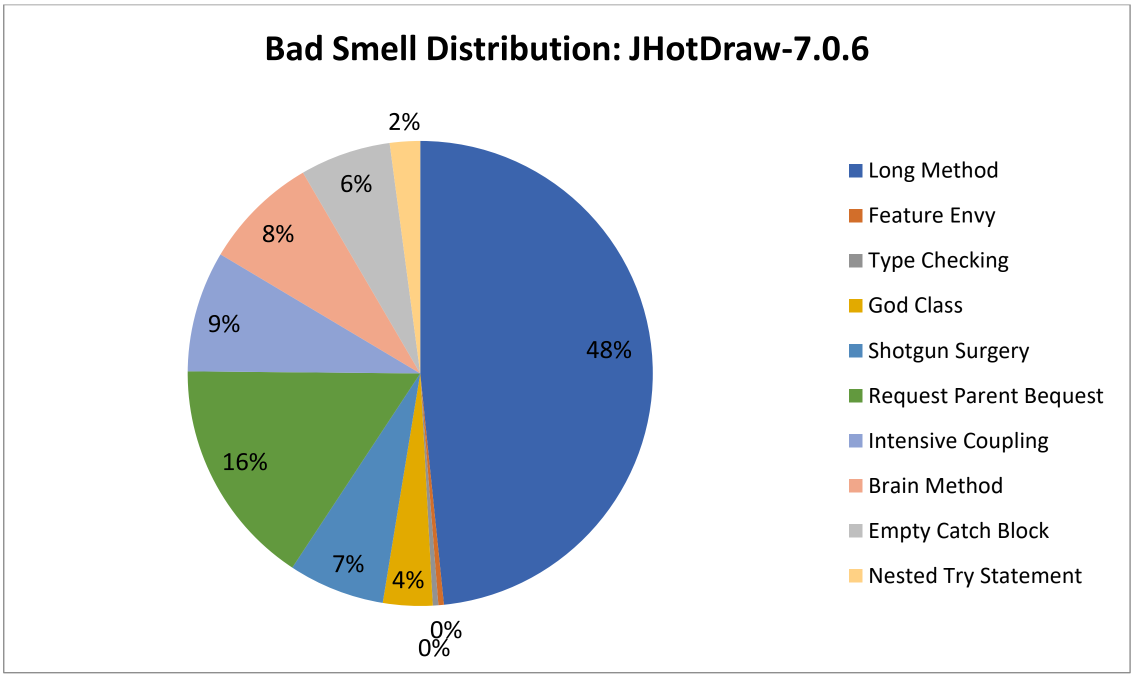


Figure 5.9 Distribution of selected Bad-Smells in JHotDraw-7.0.6

As shown in Figure 5.9, in JHotDraw-7.0.6, an alarming portion of code is poorly affected by long method whereas least affected by nested try block and type checking bad-smell.

Following general comments can be made from the results of eight selected software-systems:

1) Bad-smells in decreasing order of their presence across selected software-systems: long method, god class, brain method, refused parent bequest, intensive coupling, shotgun surgery, empty catch block, type checking, feature envy and nested try block.

2) It can be noted that long method is present in the maximum number of classes. Apart from it, god class and brain method are also highly present. These bad-smells require immediate attention and need to be removed as early as possible.

3) Nested try block and feature envy are present in the least number of classes so they can be given least importance.

4) It can also be observed that degree of presence of bad-smell increases as complexity and size of a software-system increase, Xerces-2.11.10 and ArtofIllusion-3.0.3 account for the largest amount of presence of bad-smells which corresponds to 39.04% and 32.59% respectively.

**RQ3: Is of Quality of Depreciation Rule (QDIR) useful in providing treatment to critically affected class?**

Once bad-smells are detected and C&K metrics are captured, QDIR is calculated. Both bad-smells and C&K metrics are given equal weight-age in calculating QDIR. A higher value of QDIR metric for a class is an indicator of high severity and reflects the need for instant refactoring method application to that class. On the other hand, a lower value of QDIR metric means less severity and no sooner or completely no need of refactoring. Correspondingly, four severity levels have been identified (see Table 4.6) after arranging QDIR values for a system in decreasing order. It is not possible to arrive at a common range for eight systems to fit percentage requirement of 10, 25, 25 & 40 as per Table 4.6 because all of them are having a completely different domain and a difference in sizes. Table 5.1 provides ranges of four severity levels and Table 5.2 gives the distribution of classes in each of those levels. Following observations can be made:

1) QDIR helps in reducing the number of refactoring operations and restricting it to only a small portion of classes by putting highly severe classes into the critical level which is only 10% of the total available classes (see Table 5.1 and 5.2).

2) On an average, classes with QDIR metric value >=1.75 fall under critical severity level.

3) On an average, classes with QDIR value >= 1.75 surely falls under critical severity level (see Table 5.3).

Table 5.1 Range for various Severity Levels based on QDIR

| Software-system | Severity Level | Range |
|---|---|---|
| **Frogger** | **Critical** | >=1.28 |
| | **High** | 1.27-1.06 |
| | **Mild** | 1.05-1.0 |
| | **Low** | <=1.0 |
| **jedit5.5.0** | **Critical** | >=1.84 |
| | **High** | 0.83-1.83 |
| | **Mild** | 0.11-0.82 |
| | **Low** | <=0.10 |
| **JGraphX-3.9.3** | **Critical** | >=1.50 |

| | | |
|---|---|---|
| | **High** | 0.59-0.63 |
| | **Mild** | 0.62-0.16 |
| | **Low** | <=0.15 |
| **JSettlers-1.1.20** | **Critical** | >=2.11 |
| | **High** | 0.95-2.10 |
| | **Mild** | 0.65-0.94 |
| | **Low** | <=0.64 |
| **jVLT-1.3.3** | **Critical** | >=2.45 |
| | **High** | 1.05-2.44 |
| | **Mild** | 0.50-1.04 |
| | **Low** | <=0.49 |
| **JHotDraw-7.0.6** | **Critical** | >=1.70 |
| | **High** | 0.95-1.69 |
| | **Mild** | 0.14-0.94 |
| | **Low** | <=0.13 |
| **Xerces-2.11.10** | **Critical** | >=1.70 |
| | **High** | 1.15-1.69 |
| | **Mild** | 0.66-1.14 |
| | **Low** | <=0.65 |
| **ArtOfIllusion-3.0.3** | **Critical** | >=1.40 |
| | **High** | 0.85-1.39 |
| | **Mild** | 0.11-0.84 |
| | **Low** | <=0.10 |

Table 5.2 Division of classes of selected software-systems in various Severity Levels

| **Software-system** | **Critical** | **Number of Classes** | **%age of Classes** |
|---|---|---|---|
| **Frogger** | **Critical** | 1 | 10 |
| | **High** | 6 | 25 |
| | **Mild** | 9 | 25 |
| | **Low** | 4 | 40 |
| **Jedit-5.5.0** | **Critical** | 135 | 10 |

| | | | |
|---|---|---|---|
| | **High** | 334 | 25 |
| | **Mild** | 332 | 25 |
| | **Low** | 535 | 40 |
| **JGraphX-3.9.3** | **Critical** | 56 | 10 |
| | **High** | 143 | 25 |
| | **Mild** | 140 | 25 |
| | **Low** | 233 | 40 |
| **Jsettlers-1.1.20** | **Critical** | 25 | 10 |
| | **High** | 63 | 25 |
| | **Mild** | 60 | 25 |
| | **Low** | 107 | 40 |
| **jVLT-1.3.3** | **Critical** | 38 | 10 |
| | **High** | 99 | 25 |
| | **Mild** | 108 | 25 |
| | **Low** | 154 | 40 |
| **JHotDraw-7.0.6** | **Critical** | 108 | 10 |
| | **High** | 266 | 25 |
| | **Mild** | 264 | 25 |
| | **Low** | 430 | 40 |
| **Xerces-2.11.10** | **Critical** | 95 | 10 |
| | **High** | 240 | 25 |
| | **Mild** | 247 | 25 |
| | **Low** | 394 | 40 |
| **ArtOfIllusion-3.0.3** | **Critical** | 88 | 10 |
| | **High** | 224 | 25 |
| | **Mild** | 229 | 25 |
| | **Low** | 352 | 40 |

Table 5.3 Average QDIR metric for Critical Severity Level

| **Severity Level** | **Average QDIR value** |
|---|---|
| Critical | >=1.75 |

**RQ4: Do use of Quality of Depreciation Rule (QDIR) leads to the reduction in Maintenance Effort?**

QDIR classifies classes in one of the four severity-levels so that refactoring methods can be applied to only critically affected classes to save Maintenance Effort and time of the developer. Table 5.4 represents the Effort Estimation (EE) and Effort Saved (ES) for all the selected eight software-systems and Table 5.5 provides average Effort Estimation (EE) and Effort Saved (ES) for Critical and High severity levels. CE and HE in Table 5.4 and 5.5 refer to Critically Effected and Highly Effected classes respectively.

1) It can be observed that on an average, EE (CE) and EE (HE) classes are 9.20 %and 25.45% respectively (see Table 3.8).

2) Also, on an average, ES (CE) and ES (HE) is 90.79% and 74.56% respectively (see Table 3.8).

Table 5.4 Effort Estimation of Critically and Highly Affected Classes

| **Software-system** | **EE(CE)** | **ES (CE)** | **EE (HE)** | **ES(HE)** |
|---|---|---|---|---|
| **Frogger** | 5 | 95 | 30 | 70 |
| **JEdit-5.5.0** | 10.10 | 89.90 | 25 | 75 |
| **JSettlers-1.1.20** | 9.80 | 90.20 | 24.70 | 75.30 |
| **JGraphX-3.9.3** | 9.79 | 90.21 | 25 | 75 |
| **JVLT-1.3.3** | 9.29 | 90.71 | 24.20 | 75.80 |
| **JHotDraw-7.06** | 10.11 | 89.89 | 25 | 75 |
| **Xerces-2.11.10** | 9.73 | 90.27 | 24.59 | 75.41 |
| **ArtOfIllusion-3.0.3** | 9.85 | 90.15 | 25.08 | 74.92 |

Table 5.5 Average Effort Estimation of Critically and Highly Affected Classes

| **EE (CE)** | **ES(CE)** | **EE(HE)** | **ES(HE)** |
|---|---|---|---|
| 9.20 | 90.79 | 25.45 | 74.56 |

# CHAPTER 6

# THREATS TO VALIDITY

By capturing object-oriented characteristics and identifying bad-smells in the classes upon eight selected software-systems, we tried to arrive at the best possible generalized results that are unbiased and true to our knowledge. But, the results of this empirical study would not be complete, lacking the discussion of construct, internal and external validity that are quite evident across empirical studies.

Construct validity is how much accurately the object-oriented characteristics are calculated from the selected software-systems. To reduce this threat to the maximum, we have used a professional industrial metric-tool Understand [147] to calculate these object-oriented characteristics. Internal validity can be described as the extent to which conclusions can be arrived at about the consequences of certain debts prevailing in the design of the software-system. We have minimized this threat quite successfully by investigating ten bad-smells and their effect on software quality. The results drawn from the study are consistent in nature. External validity can be regarded as the extent to which the results of an empirical study can be obtained in generalized form. To reduce this threat, we have selected a wider set of medium to large sized open-source software-systems written in Java language across different domains. For even more generality across different object-oriented languages, we are further planning to repeat this empirical work on languages like C#, C++, and Python so that developers working on languages other than Java can also be benefitted from the results of this empirical study.

# CHAPTER 7

# CONCLUSION AND FUTURE WORK

In this empirical study, we tried to prioritize only small portion in the software-system for application of refactoring on the basis of the bad-smell severity and their object-oriented design characteristics. It is widely known that refactoring is a tedious and time-consuming activity and that software developers constantly apply refactoring to source-code for various reasons like reducing the likelihood of errors and improving maintainability and understandability of the code. Therefore our research in the direction of reducing considerable portions in the software-system to apply refactoring appears appealing. Quality Depreciation Index Rule (QDIR) metric is proposed that takes into effect both the object-oriented characteristics and bad-smell severity to help prioritize classes into decided four priority levels. Four severity levels- Critical, High, Mild and Low severity levels are assigned 10, 25, 25 and 40% of classes from the entire range of classes. A range for all these four levels is then decided and classes falling in the critical level of severity are given instant refactoring solution.

First, we detected classes that are poorly affected by bad-smells in all the eight selected software-systems. Second, we identified bad-smells that are dominant across the software-systems so that we can target to remove them first. Third, we assessed the usefulness of Quality Depreciation Index Rule (QDIR) to prioritize classes for refactoring operations. Fourth, we determined if QDIR is useful in reducing maintenance effort.

The main findings obtained from this empirical research are:

1) On an average, 30% classes in all the selected software-systems are poorly affected by bad-smells.
2) Certain bad-smells like long method, god class, and brain method are severely present across studied software-systems whereas Feature Envy and Nested try block are present in the least number of software-systems.
3) Quality Depreciation Index Rule (QDIR) is effective in reducing the number of classes in an object-oriented system for application of refactoring and thereby reduces the number of refactoring operations.
4) It reduces Maintenance Effort to a great extent and on an average, 90% of effort is saved while refactoring classes based on Quality Depreciation Index Rule (QDIR).

Below given are the guidelines for software developers and researchers for carrying out future research in the field of refactoring the source-code to on priority basis:

1) The current empirical study used a wide set of eight medium to large size software-systems written in java language for the generality of results corresponding to java language for the generality of results in java based systems. For overall generality and wider acceptability of results across different languages as well, a future direction would be analyzing the results on other object-oriented languages like C++, python, and C#.
2) There are over seventy bad-smells present in literature and this research captured most common ten bad-smells whereas there are other bad-smells that can also be explored by researchers too.
3) In this study, widely known Chidamber & Kemerer metric-suite is used to capture the design characteristics of the software-systems. But authors can take into consideration other popular metric-suites like Li & Henry, QMOOD, Li & Henry.

We finally hope that significant quality can be improved by providing refactoring solution to only highly severe classes that have instant refactoring requirements saving considerable maintenance effort and time.

# REFERENCES

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman Publishing Co., Inc., 1999.

[2] K.P. Srinivasan, "Unique Fundamentals of Software Measurements and Software Metrics in Software Engineering", *International Journal of Computer Science & Information Technology,* vol 7, no. 4, 2015.

[3] I. Sommerville, *Software Engineering*, 9th ed., Addison Wesley Publishing Company, , 2010.

[4] Fernando BeA, "The MOOD2 metrics set (in Portuguese)", *Technical Report. Grupo de Engenharia de Software, INESC*, 1998.

[5] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Transactions on Software Engineering,* vol. 20, no. 6, pp. 476–493, 1994.

[6] M. Lorenz and J. Kidd, "Object-Oriented Software Metrics", *Prentice Hall Object-Oriented Series*, Prentice Hall, 1994.

[7] W. Li and S. M. Henry. "Object-oriented metrics that predict maintainability". *Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, 1993.

[8] R. Malhotra, A. Chug and P. Khosla, “Prioritization of Classes for Refactoring: A step towards improvement in Software Quality”, *In Proceedings of the Third International Symposium on Women in Computing and Informatics,* pp. 228-234. 2015.

[9] R. D. Wangberg, "A Literature Review on Code Smells and Refactoring," *University of Oslo*, 2010.

[10] J. Al. Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, vol. 58, pp. 231-249, 2015.

[11] S. Singh, S. Kaur, "A systematic literature review: Refactoring for disclosing code smells in object-oriented software," *Ain Shams Engineering Journal , 2017.*

[12] M.Abebe and C. J. 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.

[13] M. Ó. Cinnéide, A. Yamashita, S. Counsell "Measuring Refactoring Benefits: A Survey of the Evidence," *In Proceedings of the 1st International Workshop on Software Refactoring,* pp. 9-12, 2016.

[14] I. Bassey, N. Dladlu, B. Ele "Object-Oriented Code Metric-Based Refactoring Opportunities Identification Approaches: *analysis," 4th Itnternational Conference on Applied Computing and Information Technology,* 2016.

[15] T. Mens, T. Tourwé, and F. Muñoz, "Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring," *In Proceedings of the 6th International Workshop on Principles of Software Evolution*, 2003.

[16] J. Al. Dallal and A. Abdin "Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review," *IEEE Transactions on Software Engineering*, pp. 44-69, 2016.

[17] A. Ouni, M. Kessentini, S. Bechikh and H. Sahraoni, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Software Quality Journal*, vol. 23, no. 2, pp 323-361, Jun. 2015.

[18] A. Ouni, M. Kessentini, M. Houari, Sahraoui, K. Deb and K. Inoue1, "MORE: A multi objective refactoring recommendation approach to introducing design patterns and fixing code smells", *Journal of Software Evolution and Practices*, vol. 29, no. 5, May. 2016.

[19] M. Fokaefs, N, Tsantails and A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Feature Envy Bad Smells," *IEEE International Conference on Software Maintenance*, 2007.

[20] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk and A. Lucia, "Identifying Method Friendships to Remove the Feature Envy Bad Smell (NIER Track)", *13th International Conference on Software Engineering*, 2011.

[21] Y. Higo, S. Kusumoto and K. Inoue "A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system", *Journal of Software Maintenance and Evolution: Research and Practice*s, vol. 20, no. 6, pp 435-461, 2008.

[22] F. Fontana, V. Ferme and S. Spinelli, "Investigating the Impact of Code Smells Debt on Quality Code Evaluation", *3rd International Workshop on Managing Technical Debt, 2012.*

[23] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, G. Gueheneuc "Playing with refactoring: Identifying extract class opportunities through game theory*", IEEE International Conference on Software Maintenance*, Timisoara, 2010.

[24] Al. Dallal, "Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics", *Journal of Information and Software Technology*, vol. 54, no. 10, pp 1125-1141, 2012.

[25] K. Stroggylos and D. Spinellis, "Refactoring–Does it improve software quality?",*5th International Workshop on Software Quality*, 2007.

[26] B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering," *Technical Report EBSE, Keele University*, 2007.

[27] J. Wen, S. Li, Z. Lin, Y. Hu, C. Huang, "Systematic literature review of machine learning based software development effort estimation models", *Information and Software Technology*, vol. 54, no. 1, pp. 41–59, 2012.

[28] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo, "When Does a Refactoring Induce Bugs? An Empirical Study," *In Proceedings of the IEEE International Work-ing Conference on Source Code Analysis and Manipulation, ser. (SCAM '12)*, pp. 104–113, 2012.

[29] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A.D. Lucia, "Improving Software Modularization Via Automated Analysis of Latent Topics and Dependencies," *Transactions on Software Engineering and Methodologies*, vol. 23, no. 1, 2014.

[30] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating Extract Class Refactoring: An Improved Method and Its Evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617-1664, 2014.

[31] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying Extract Class Refactoring Opportunities Using Structural and Semantic Cohesion Measures," *Journal of Systems and Software*, vol. 84, pp. 397–414, 2011.

[32] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending Move Method Refactor-ings via Relational Topic Models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671-694, 2014.

[33] S. Ghaith and M. O. Cinnéide, "Improving Software Security Using Search-based Refactoring," *In Proceedings of the 4th international conference on Search Based Software Engineering*, pp. 121–135, 2012.

[34] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, "Experimental Assessment of Software Metrics Us-ing Automated Refactoring," In *Proceedings of Empirical Software Engineering and Management*, pp. 49-58, 2012.

[35] M. O'Keeffe and M. O'Cinneide, "Search-Based Software Main-tenance," In *Proceedings of Conference on Software Maintenance and Reengineering*, pp. 249-260, 2006.

[36] K. Elish and M. Alshayeb, "Classification of Refactoring Methods Based on Software Quality Attributes," *Arabian Journal for Science and Engineering*, vol. 36, no. 7, pp. 1253-1267, 2011.

[37] M. Alshayeb, "Refactoring Effect on Cohesion Metrics," *International Conference on Computing, Engineering and Information*, pp. 3-7, 2009

[38] M. Alshayeb and F. Banaeamah, "Approaches for Refactoring to Frameworks," *International Journal of Information Technology*, vol. 18, no. 1, 2012.

[39] M. Alshayeb, "Empirical Investigation of Refactoring Effect on Software Quality," *Information and Software Technology*, vol. 51, no. 9, pp. 1319-1326, 2009.

[40] M. Fokaefs, N. E. Stroulia, A. Chatzigeorgiou, "JDeodorant: Identification and Application of Extract Class Refactorings," *International Conference on Software Engineering*, 2011.

[41] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347-367, 2009.

[42] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[43] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does Refactoring Improve Reusability?," *In Proceedings of the International Conference on Software Reuse*, pp. 287-297, 2006.

[44] W. Liu, Z.G. Hu, H.T. Liu, and L. Yang, "Automated Pattern-directed Refactoring for Complex Conditional Statements," *Journal of Central South University*, vol. 21,pp.1935-1945, 2014.

[45] H. Liu, G. Li ,Z.Y. Ma and W. Z. Shao, "Conflict aware schedule of software refactorings," *The Institution of Engineering and Technology*, vol. 2, no. 5, pp. 446-460, 2008.

[46] R. Malhotra, ,A. Chug, "An Empirical Study to Assess the Effects of Refactoring on Software Maintainability", *International Conference on Advances in Computing, Communications and Informatics*, 2016.

[47] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "How Changes Affect Software Entropy: An Empirical Study," *Empirical Software Engineering*, vol. 19, no. 1, pp. 1–38, 2014.

[48] M.Gatrell and S. Counsell, "The Effect of Refactoring on Change and Fault- proneness in Commercial C# Software," *Science of Computer Programming*, vol. 102, pp. 44-56, 2015.

[49] S.H. Kannangara and W.M.J.I. Wijayanayake, "Impact of Refactoring on External Code Quality Improvement: An Empirical Evaluation," *International Conference on Advances in ICT for Emerging Regions*, pp. 60-67,2013.

[50] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring," *In Proceedings of International Conference on Software Maintenance*, pp. 576-585, 2002.

[51] M. Kim, D. Cai, and S. Kim, "An Empirical Investigation into the Role of Refactorings During Software Evolution," *In Proceedings of 33rd International Conference on Software Engineering*, pp. 151-160, 2011.

[52] M. Kim, T. Zimmermann, and N. Nagappan, "An Empirical Study of Refactoring Challenges and Benefits at Microsoft," *IEEE Transactions on Software Eng*ineering, vol. 40, no. 7, pp. 633-649, 2014.

[53] B. Geppert, A. Mockus and F. Rößler, "Refactoring for Changeability: A way to go?," *11thIEEE International Software Metrics Symposium, 2005.*

[54] B. dos Santos Neto, M. Ribeiro, V. Silva, C. Braga b, C. de Lucena, E. de Barros Costa, "AutoRefactoring: A Platform to Build Refactoring Agents," *Expert Systems with Applications.* vol. 42. no. 3, pp. 1652-1664, 2015.

[55] A.R. Han, D.E. Bae, and S. Cha, "An Efficient Approach to Identify Multiple and Independent Move Method Refactoring Candidates," *Information and Software Technology*, vol. 59, pp. 53-66, 2015.

[56] A.R. Han, and D.H. Bae, "An Efficient Method for Assessing the Impact of Refactoring Candidates on Maintainability Based on Matrix Computation," *21st Asia Pacific Software Engineering Conference*, pp. 430-437, 2014.

[57] R. Leitch and E. Stroulia "Assessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis," *International Software Metrics Symposium*, 2003.

[58] G. Szoke, C. Nagy, P. Hegedus, R. Ferenc, and T. Gyimóthy, "Do Automatic Refactorings Improve Maintainability? An In-dustrial Case Study" *IEEE International Conference on Software Maintenance and Evolution*, 2015.

[59] S. Tarwani and A. Chug, "Sequencing of Refactoring Techniques by Greedy Algorithm for maximizing Maintainability," *Intl. Conference on Advances in Computing, Communications and Informatics*, 2012.

[60] L. Kumar and A. Sureka, "Application of LSSVM and SMOTE on Seven Open Source Projects for Predicting Refactoring at Class Level," *Asia Pacific Software Engineering Conference, 2017.*

[61] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis and P. Avgeriou "Identifying Extract Method Refactoring Opportunities based on Functional Relevance," *IEEE Transactions on Software Engineering*, vol. 43, no. 10, pp. 954 - 974, 2017.

[62] A. Chug and M. Gupta, "A Quality Enhancement through Defect Reduction using Refactoring Operation,"*International Conference on Advances in Computing, Communications and Informatics*, 2017.

[63] A. Chug and S. Tarwani, "Determination of Optimum Refactoring Sequence using A* Algorithm after Prioritization of Classes,"*International Conference on Advances in Computing, Communications and Informatics*, 2017.

[64] F. Faiz, R. Easmin and A. Ul Gias, "Achieving better requirements to code traceability: Which refactoring should be done first?" *International Conference on the Quality of Information and Communications Technology*, 2016.

[65] A. R. Han and S. Cha, "Two phase Assessment Approach to Improve the Efficiency of Refactoring Identification," IEEE Transactions on Software Engineering, pp. 1-1, 2017.

[66] I. Kadar, P. Heged˝us, R. Ferenc and T. Gyimothy, "A Code Refactoring Dataset and Its Assessment Regarding Software Maintainability," *IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2016.

[67] G. Kaur and B. Singh, "Improving the Quality of Software by Refactoring," *International Conference on Intelligent Computing and Control Systems*, 2017.

[68] Y. Khrishe and M. Alshayeb, "An Empirical Study on the Effect of the Order ofApplying Software Refactoring," *International Conference on Computer Science and Information Technology*, 2016.

[69] A. Mart, E. Sikander and N. Medlani, "Estimating and Quantifying the Benefits of Refactoring to Improve a Component Modularity: a Case Study," *Euromicro Conference on Software Engineering and Advanced Applications*, 2016.

[70] R. Morales, A. Sabané, P. Musavi, F. Khomh, F. Chicano and G. Antoniol, "Finding the Best Compromise Between Design Quality and Testing Effort During Refactoring," *IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2016.

[71] M. Badri, L. Badri, O. Hachemane and A. Ouellet, "Exploring the Impact of Clone Refactoring on Test Code Size in Object Oriented Software," *IEEE International Conference on Machine Learning and Applications*, 2016.

[72] F. Palomba, A. Zaidman, R. Oliveto and A. D. Lucia. "An Exploratory Study on the Relationship between Changes and Refactoring, "IEEE*/ACM International Conference on Program Comprehension*, 2017.

[73] F. Palomba and Andy Zaidman, "Does Refactoring of Test Smells Induce Fixing Flaky Tests?,"*IEEE International Conference on Software Maintenance and Evolution*, 2017.

[74] A. Shahjahan,W. H. Butt and A. Z. Ahmad, *"Impact of Refactoring on Code Quality by using Graph Theory: An Empirical Evaluation,"* 2015.

[75] A.Vasileva and D. Schmedding, "How to Improve Code Quality by Measurement and Refactoring,"

*International Conference on the Quality of Information and Communications Technology*, 2016.

[76] M. Vimaladevi and G. Zayaraz, "Stability Aware Software Refactoring Using Hybrid Search Based Techniques," *International Conference on Technical Advancements in Computers and Communications*, 2017.

[77] M.Wahler and W. Snipes, "Improving Code Maintainability: A Case Study on the Impact of Refactoring," *IEEE International Conference on Software Maintenance and Evolution*, 2016.

[78] S. Demeyer, S. Ducasse and O. Nierstrasz, "Finding Refactorings via Change Metrics," *ACM SIGPLAN Conference on object-oriented programming, systems, languages, and applications*, 2007.

[79] K. Maruyama, and T. Omor, "A Security Aware Refactoring Tool for Java Programs," *In Proceedings of the 4th Workshop on Refactoring Tools*, ACM, pp. 22-28, 2011.

[80] F. Arcelli Fontana and S. Spinelli, "Impact of Refactoring on Quality Code Evaluation," In *Proceedings of 4th Workshop on Refactoring Tools*, pp. 37–40, 2011.

[81] M. O'. Keeffe and M. Ó. Cinnéide, "Search-based refactoring: an empirical study" *Journal of Software Maintainence and Evolution*, vol. 20, no. 5, pp. 345-364, 2008.

[82] E. Nasseri, S. Counsell and M. Shepperd, "Class movement and relocation: An empirical study of Java inheritance evolution," *Journal of Systems and Software*, vol. 3, no. 2, pp. 303-315, 2010.

[83] G. Bavota, A.D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1-14,2015.

[84] A. Christopoulou, E. A. Giakoumakis, V. E. Zafeiris, and V. Soukara, "Automated Refactoring to the Strategy Design Pattern," *Information and Software Technology*, vol. 54, no. 11, pp. 1202–1214, 2012.

[85] A. R. Han and D. H. Bae, "Dynamic profiling based approach to identifying cost effective refactorings," *Information and Software Technology*, vol. 55, no. 6, pp. 966-985, 2013.

[86] F. Castor, N´.Cacho and E. Figueiredo," *Journal of Software: Practice and Experience*, vol. 39, no. 17, pp. 1377-1417, 2009.

[87] R. Kolb, D. Muthig, T. Patzke and K.Yamauchi, "Refactoring a legacy component for reuse in a software product line: a case study," *Journal of Software Maintenance and Evolution: Research and Practice IEEE International Conference on Software Maintenance*, vol. 18, no. 2, pp. 109-132, 2006.

[88] Y. Bian, M. A. Parande, G.Koru and S. Zhao, "Testing the theory of relative dependency from an evolutionary perspective: higher dependencies concentration in smaller modules over the lifetime of software products," *Journal of Software: Evolution and Process*, vol. 28, no. 5, pp. 340-371, 2006.

[89] M. Alshayeb, "The Impact of Refactoring to Patterns on Software Quality Attributes," *The Arabian Journal for Science and En-gineering*, vol. 36, 2011.

[90] M. Wiem Mkaouer, M. Kessentini, S. Bechikh, M. Ó. Cinnéide and K. Deb, "On the use of many quality attributes for software refactoring: a many objective search based software engineering approach," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2503-2545, 2015.

[91] A. Dallal, L. C. Briand, "A Precise Method Method Interaction Based Cohesion Metric for Object Oriented Classes," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 2, 34 pages, 2012.

[92] G. Bavota, A.D. Lucia, A. Marcus and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901-932, 2012.

[93] R. Mahouachi1, M. Kessentini and Khaled Ghedira, "A new design defects classification: marrying detection and correction," *International conference on Fundamental Approaches to Software Engineering*, 2012.

[94] Y. Kosker , B. Turhan and A. Bener, "An expert system for determining candidate software classes for refactoring," *Expert Systems with Applications*, vol. 36, no. 6, pp. 10000-10003, 2009.

[95] L. Yang, H. Liu, and Z. Niu, "Identifying Fragments to be Ex-tracted from Long Methods," *In Proceedings of Software Engineering Conference,* 2009, pp. 43 –49, 2009.

[96] G. Bavota, R. Oliveto A. De Lucia, and A. Marcus, "In Medio Stat Virtus: Extract Class Refactoring Through Nash Equili-bria," *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering, Software Evolution Week*, 2014.

[97] D. Boshnakoska and A. Mišev, "Correlation between Object-Oriented Metrics and Refactoring," *ICT Innovations, Communication in Computer and Information Science*, vol. 83, pp. 226-235, 2011.

[98] F. Bourqun and R.K. Keller, "High Impact Refactoring Based on Architecture Violations," In *Proceedings of 11th European Conf. Software Maintenance and Reengineering*., pp. 149-158, 2007.

[99] C. Dibble II and P. Gestwicki, "Refactoring Code to Increase Readability and Maintainability: A Case Study," *Journal of Computing Sciences in Colleges,* vol. 30, no. 1, pp. 41-51, 2014.

[100] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring—Improving Coupling and Cohesion of Existing Code," *In Proceedings of 11th Working Conference on Reverse Engineering,* pp. 144-151, 2004.

[101] B. DuBois and T. Mens, "Describing the Impact of Refactoring on Internal Program Quality," *In Proceedings of International Workshop on Evolution of Large-scale Industrial Software Applications*, pp. 37-48, 2003.

[102] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and Application of Extract Class Refactorings in Object-Oriented Systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241-2260, 2012.

[103] M. Gaitani, V. Zafeiris, N. Diamantidis, E. Giakoumakis, "Automated Refactoring to the NULL OBJECT Design Pattern," *Information and Software Technology*, vol. 59, pp. 33–52, 2015.

[104] H.C. Jiau, L.W. Mar, and J.C. Chen, "OBEY: Optimal Batched Refactoring Plan Execution for Class Responsibility Redistribution," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp.

1245-1263, 2013.

[105] P.Joshi and R.K.Joshi, "Microscopic Coupling Metrics for Re-factoring," *In Proceedings of 10th European Conf. on Soft-ware Maintenance and Reengineering*, pp. 145-152, 2006.

[106] P. Joshi and R.K. Joshi, "Concept Analysis for Class Cohesion," *In Proceedinds of 13th European Conference on Software Maintenance and Reengineering*, pp. 237- 240, 2009.

[107] M. Kallen, S. Holmgren, and E. poraHvannberg, "Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application," *14th International Working Conference on Source Code Analysis and Manipulation*, pp. 125-134, 2014.

[108] S. Lee, G. Bae, H. S. Chae, D. H. Bae and Y. R. Kwon, "Automated Scheduling for Clone based Refactoring Using a Competent GA," *Journal of Software: Practice and Experience*, vol. 41, no. 5, pp. 521-550, 2010.

[109] K. Nongpong, "Feature Envy Factor: A Metric for Automatic Feature Envy Detection," *7th International Conference on Knowledge and Smart Technology*, IEEE, 2015.

[110] G. Pappalardo and E. Tramontana, "Suggesting Extract Class Refactoring Opportunities by Measuring Strength of Method Interactions" *In Proceedings of Asia Pacific Software Engineering Conference (APSEC)*, 2013.

[111] A.A. Rao and K.N. Reddy, "Identifying Clusters of Concepts in a Low Cohesive Class for Extract Class Refactoring Using Me-trics Supplemented Agglomerative Clustering Technique," *International Journal of Computer Science*, vol. 8, no. 5, pp. 185-194, 2011.

[112] J. Ratzinger, T. Sigmund, and H.C. Gall, "On the Relation of Refactorings and Software Defect Prediction", *In Proceedings of International Working Conference on Mining Software Repositories*, pp. 35-38, 2008.

[113] O. Seng, J. Stammel, and D. Burkhart, "Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems," *InProceedings of Genetic and Evolutionary Computation Conf*erence, pp. 1909-1916, 2006.

[114] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Bulk Fixing Coding Issues and Its Effects on Software Quality: Is it Worth Refactoring?" In *Proceedings of 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 95–104, 2014.

[115] V. Veerappa and R. Harrison, "An Empirical Validation of Coupling Metrics Using Automated Refactoring," *International Symposium on Empirical Software Engineering and Measurement*, 2013.

[116] M. F. Zibran and C.K. Roy, "A Constraint Programming Approach to Conflict aware Optimal Scheduling of Prioritized Code Clone Refactoring," *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2011.

[117] L. Tahvildari, K. Kontogiannis and J. Mylopoulos, "Quality-Driven Software Re Engineering," *Journal of Systems and Software - Special issue on: Software architecture - Engineering quality attributes*, vol. 66, no. 3, pp. 225-239, 2003.

[118] N. Kumari and A. Saha, " Effect of Refactoring on Software Quality," *InProceedings of Confenrence on Software Maintenance, 2014.*

[119] K. O. Elish and Mohammad Alshayeb, "Investigating the effect of refactoring on software testing effort," *In Proceedings of Asia Pacific Software Engineering Conference*, 2009.

[120] L. Tahvildari and K. Kontogiannis, "Improving Design Quality Using Meta-pattern Transformations: A Metric-based Ap-proach," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no.4–5, pp. 331–361, 2004.

[121] F. Arcelli Fontana, M. Zanoni, A. Ranchetti, D. Ranchetti, "Software Clone Detection and Refactoring." *ISRN Software Engineering*, vol. 2013, pp. 1-8, 2013.

[122] Č. Gerlec and M. Heričko, "Evaluating Refactoring with a Qual-ity Index," *World Academy of Science, Engineering and Technology* 63, pp. 76-80, 2010.

[123] R. Moser, P. Abrahamsson, W. Pedrycz, A.Sillitti and G. Succi, "A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team," *LNCS*, vol. 5082, pp. 252-266, 2008.

[124] R. Shatnawi and W. Li., "An Empirical Assessment of Refactoring Impact on Software Quality Using a Hierarchical Quality Model," *International Journal of Software Engineering and Its Ap-plications*, vol. 5, no. 4, 2011.

[125] M. Alshayeb, H. Al. Jamimi and M. O. Elish, "Empirical taxonomy of refactoring methods for aspect oriented programming," *Journal of Software: Process and Evolution*, vol. 25, no. 1, pp. 1-25, 2013.

[126] P. Meananeatra, "Identifying Refactoring Sequences for Improving Software Maintainability," *In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2013.

[127] R. S. Bashir, S. P. Lee, C. C. Yung, K. A. Alam, R. W. Ahmad, "A methodology for impact evaluation of refactoring on external quality attributes of a software design," *International Conference on Frontiers of Information Technology*, 2017.

[128] H. Lu, S. Wang, T. Yue, S. Ali, J. F. Nygård, "Automated Refactoring of OCL Constraints with Search," *IEEE Transactions on Software Engineering*, 2017.

[129] A. Imazato, Y. Higo, K. Hotta and S. Kusumoto"Finding Extract Method Refactoring Opportunities by Analyzing Development History," *IEEE 41st Annual Computer Software and Applications Conference, 2017.*

[130] Semih Okur, "Understanding, Refactoring, and Fixing Concurrency in C#," *30th IEEE/ACM International Conference on Automated Software Engineering*, 2015.

[131] C. Sahin, L. Pollock, J. Clause, "How Do Code Refactorings Affect Energy Usage?," *International Symposium on Empirical Software Engineering and Measurement*, no. 36, 36 pages, 2014.

[132] B. Fonseca , M. Ribeiro , V.Torres,C. Braga, C. José, E. Costa, "AutoRefactoring: A platform to build refactoring agents," *Expert Systems with Applications*, vol. 42, no. 3, pp. 1652-1664, 2015.

[133] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Feature Envy Bad Smells," *IEEE International Conference on Software Maintenance*, 2007.

[134] Y. Higo, T. Kamiya, S. Kusumoto1 K. Inoue, "ARIES: Refactoring support environment based on code clone analysis," In *Proceedings of the IASTED Conference on Software Engineering,* 2004.

[135]Y. Higo, S. Kusumoto and K. Inoue, "A metric-based approach to identifying Refactoring pportunities for merging code clones in a Java software system," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 20, no. 6, pp. 435-461, 2008 .

[136] V. Sales, R. Terrayz, L. F. Miranda and M. T. Valente, Recommending Move Method Refactorings using Dependency Sets," *20th Working Conference on Reverse Engineering*, 2013.

[137] L. Zhao, J. H. Hayes, "Predicting Classes in Need of Refactoring: An Application of Static Metrics," *International Conference on Frontiers of Information Technology, 2009.*

[138] M. Abbes, F. Khomh, Y. G. Gu´eh´eneuc, G. Antoniol, "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension," *15th European Conference on Software Maintenance and Reengineering*, 2011.

[139] F. S. Barbosa, A. Aguiar, "Removing Code Duplication with Roles," *12th IEEE International Conference on Intelligent Software Methodologies, Tools and Techniques, pp.* 22-24, 2013.

[140] I. Griffith, S. Wahl, C. Izurieta, "Evolution of Legacy System Comprehensibility through Automated Refactoring," *In Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering,* pp. 35-42, 2011.

[141] W. F. Opdyke, "Object-oriented refactoring, legacy constraints and reuse," Bell Laboratories - Innovations for Lucent Technologies, *8th Workshop on Institutionalizing Software Reuse* ,1996.

[142] M. V. Mäntylä, "An Experiment on Subjective Evolvability Evaluation of Object-Oriented Software: Explaining Factors and Interrater Agreement," *International Symposium on Empirical Software Engineering,* 2005.

[143] J. Al. Dallal, "Qualitative Analysis for the Impact of Accounting for Special Methods in Object-Oriented Class Cohesion Measurement ," Journal of Software, vol. 8, no. 2, pp. 327–336, 2013.

[144] J. Bansiya J, C. G. Davis, " A hierarchical model for object-oriented design quality assessment", *IEEE Transactions on Software Engineering* 2002; vol. 28, no. 1, pp. 4–17, 2002.

[145] R. Pressman, *Software Engineering: A Practitioner's Approach*, 6th edn., McGraw Hill, New York, 2005.

[146] Understand Metrics Tool, Available at: https://scitools.com/feature/metrics/ (Accessed: 20 May 2018).

[147] R. Shatnawi , W.L. James, Swain, T. Newman, Finding software metrics threshold values using ROC curves, *Journal of Software Evolution and Processes*, 2010.

[148] Eclipse website, [Online], Available: https://www.eclipse.org (Accessed: 15 April, 2018).

[149] Github, Available: https://github.com/ (Accessed: 14 February, 2018).

[150] SourceForge, Available: https://sourceforge.net (Accessed: 14 February, 2018).

[151] JDeodorant plug-in, Available: http://jdeodorant.com/ (Accessed: 21 February, 2018)

[152] JSpirit plugin. Available: http://sites.google.com/site/santiagoavidal/projects/jsirit/ (Accessed: 21 February, 2018).

[153] Robusta plugin, Available: https://marketplace.eclipse.org/content/robusta-eclipse-plugin/ (Accessed: 21 February, 2018).