www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 4 Issue 5 May 2015, Page No. 11733-11742

An Efficient Sdmpc Metric Based Approach For Refactoring Software Code

B Ramalkshmi[1], *D Gayathri Devi*[2]

[1] Bharathiyar University, Sri Ramakrishna College of Arts and Science for Women,
Coimbatore, Tamilnadu, India
ramyabalu.mdu@gmail.com

[2] Bharathiyar University, Sri Ramakrishna College of Arts and Science for Women,
Coimbatore, Tamilnadu, India
dgayadevi@gmail.com

Abstract

Software Engineering is an about development, design operation and maintenance of software. But there are some factors that make software maintenance difficult. A code clone is nothing a similar or duplicate code in a source code or created either by replication or some modification. Code clone is one of the factors that increase software maintenance and also cause code bloating. Thus the clone has to be removed. To remove clone, refactoring has to be determined and applied. Refactoring is done to improve the quality of a software systems' structure, which tends to degrade as the system evolves. While manually determining useful refactoring is a challenging, search-based techniques can automatically discover useful refactoring. Refactoring approach uses the concept of Pareto optimality which naturally applies to search-based refactoring. Before refactoring is done, the test case should be generated. A formal written test-case is characterized by a known input and by an expected output, which is worked out before the test is executed

This paper proposes a method for removing clone through refactoring. In order to do refactor the clone, first the concept of Pareto optimality and a Pareto front is defined. Jsync refactor tool is used to refactor the programs. The coupling between object classes (CBO) metric represents the number of classes coupled to a given class. The second metric LSCC is represents the classes. Meaningful class coupling and cohesion metric helps object-oriented software developers detect class design weaknesses and refactor classes accordingly. CBO, LSCC and SDMPC metrics are used to check the accuracy of the refactored programs. The advantage of this system is helps the developers to program faster and it takes less time for clone removal. It improve the design of the software and it makes softer easier to understand. Overall performance of the system is highly improved by the proposed system.

Keywords: Refactoring, Metrics, Parato optimality

1. Introduction

Different kinds of redundancy and replication in the code is called clone. Software systems often contain sections of code that are very similar, called code clones. Reusing code fragments by copying and pasting with or without minor adaptation is a common activity in software development. One of the major shortcomings of such duplicated fragments is that if a bug is detected in a code fragment; all the other fragments similar to it should be investigated to check the possible existence of the same bug in the similar fragments.

Code clone detection is one of the important fields of software engineering that helps in reducing or eliminating unnecessary duplication of code segment. It should be noted that almost every software industry is suffering from code cloning problem. The cloning problem normally arises in the areas where large and complex software projects are being developed. However, it is also widely agreed that clones should be detected. Therefore, Code clones are generally considered harmful in software development, and the predominant approach is to try to eliminate them through refactoring.

2 Refactoring

Code refactoring is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour", undertaken in order to improve some of the non-functional attributes of the software. Code Refactoring are used for code readability and to reduce complexity that improves the maintainability of the source code.

2.1 Overview of Refactoring

Refactoring is usually motivated by noticing a code smell. For example the method at hand is very long, or it is a near duplicate of another nearby method. Once recognized, such problems are addressed by refactoring the source code, or transforming it into a new form that behaves the same as before but that no longer "smells".

There are two general categories of refactoring.

1. **Maintainability** - It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp. This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods. It might be achieved by moving a method to a more appropriate class, or by removing misleading comments.

2. **Extensibility** - It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility.

2.2 Basic Procedure of Refactoring

One of the basic procedures of refactoring (besides eliminating duplication) is adding indirection. Indirection means defining structures (e.g. classes and methods) and giving them names. Using named structures makes code easy to read because it gives a way to explain intention (class and method names) and implementation (class structures and method bodies) separately. The same technique enables sharing of logic (e.g., methods invoked in different places or a method in super class shared by all subclasses). Sharing of logic, in turn, helps to manage change in systems. Finally, polymorphism (another form of indirection) provides a flexible, yet clear way to express conditional logic.

2.3 Benefits of Refactoring

Refactoring is used for several purposes. It helps the code to retain its shape. Without refactoring the design of the program will decay. As people change code (usually without fully understanding the design objectives behind the implementation) it gradually begins to lose its structure. Once the structure gets cluttered, the code becomes harder to understand and so the chances of cluttering the design further increase.

Refactoring makes the code more readable. This is essential for conveying the intention of the code to others. It also makes the code easier to read. That is equally important since it’s unrealistic to assume that it to be remembering the intentions for more than few weeks.

Refactoring is used to grasp the intention of unfamiliar code. When looking at a fragment of code try to understand. To find out how the code works, first refactor it to better reflect to understanding of its purpose. If everything goes well, that have understood and processed a part of the system correctly. If not, need to get a better understanding of the code fragment at hand.

2.4 Automated Refactoring

Despite the enormous success that manual and automated refactoring has enjoyed during the last decade, the software developers know little about the practice of refactoring. Understanding the refactoring practice is important for developers, refactoring tool builders, and researchers. Many previous approaches to study refactoring are based on comparing code snapshots, which is imprecise, incomplete, and does not allow answering research questions that involve time or compare manual and automated refactoring.

Refactoring is an important part of software development. Development processes like extreme Programming treat refactoring as a key practice. Refactoring has revolutionized how programmers design software: it has enabled programmers to continuously explore the design space of large codebases, while preserving the existing behaviour.

It is widely believed that refactoring improves software quality and developer productivity by making it easier to maintain and understand software systems. Many believe that a lack of refactoring incurs technical debt to be repaid in the form of increased maintenance cost.

3 JSync: Architecture Overview

Fig. 3.1 shows JSync's architectural overview. The main data structure of JSync is the Clone Management Database, storing the information about software projects, code fragments, clone relationship, clone changes, and consistency information.

Other modules access and store their working information in the Clone Database. Module IO has the responsibility of maintaining this database in the SVN Management repository of the project. It also accesses the Eclipse workbench, the SVN repository, and the file system to collect information about the project and source code and to store it in the Clone Management Database, thus providing working information for other modules.

Module GUI is associated with Eclipse and has various responsibilities, such as interacting with users to receive user requests, displaying the clone groups and corresponding inconsistency changes (e.g., groups having inconsistent changes are noted with red colour), and presenting the clone pairs, their matched and inconsistent code elements.

Two modules, Fragment Detector and Change Detector, working directly with source code, analyse and detect the code fragments and the code changes. The detected fragments and changes are stored in the Clone Management Database for further analysis.

One of the key functionality in JSync, incremental clone detection, is provided by module Incremental Clone Detector. This module reads the fragments, code changes information in the Clone Management Database and detects/ updates the clone groups of the project.

Clone consistency analysis and synchronization is provided by module Clone Consistency Manager. This module also accesses the clone information from the Clone Management Database, detects inconsistencies of clone pairs, and provides synchronization on user requests.

During the clone management process, the developer may not want to refactor/remove those clones, and may want to mark those to indicate such decisions so that they will not have to encounter those same sets of clones over and over. Moreover, the decision needs to be documented and shared among different programmers, and there should be facilities for the developers to review those clones at a later time, in case they want to re-evaluate their management decision.

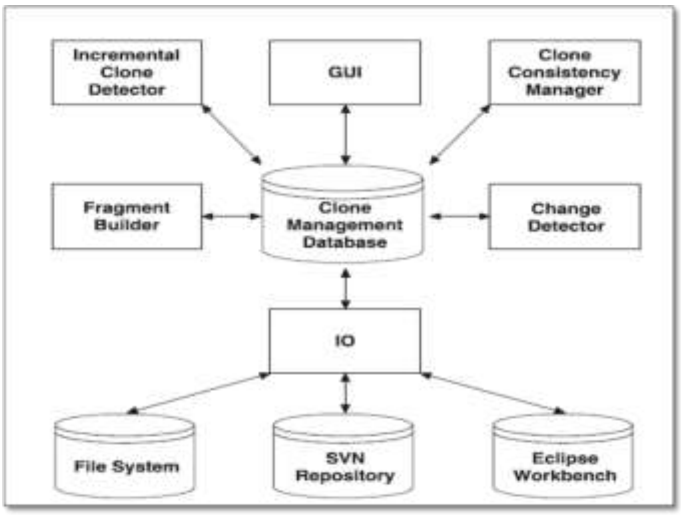

Figure 3.1: Architecture Overview of JSync.

4 Metrics

The proposed work combines coupling metrics with cohesion metric to produce a useful result. Therefore it is common to combine more than one metric when designing an appropriate fitness function, with the intuitive idea that the combination of metrics should prevent any one metric being unduly favoured. Number of metrics is calculating the clone refactoring technique.

WMC - Weighted methods per class

A class's weighted methods per class WMC metric is simply the sum of the complexities of its methods. As a measure of complexity we can use the cyclomatic complexity, or we can arbitrarily assign a complexity value of 1 to each method.

DIT - Depth of Inheritance Tree

The depth of inheritance tree (DIT) metric provides for each class a measure of the inheritance levels from the object hierarchy top. In Java where all classes inherit Object the minimum value of DIT is 1.

NOC - Number of Children

A class's number of children (NOC) metric simply measures the number of immediate descendants of the class.

CBO - Coupling between object classes

The coupling between object classes (CBO) metric represents the number of classes coupled to a given class (efferent couplings, Ce). This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.

RFC - Response for a Class

The metric called the response for a class (RFC) measures the number of different methods that can be executed when an object of that class receives a message (when a method is invoked for that object). Ideally, we would want to find for each method of the class, the methods that class will call, and repeat this for each called method, calculating what is called the transitive closure of the method's call graph. This process can however be both expensive and quite inaccurate.

LCOM - Lack of cohesion in methods

A class's lack of cohesion in methods (LCOM) metric counts the sets of methods in a class that are not related through the sharing of some of the class's fields. The original definition of this metric (which is the one used in ckjm) considers all pairs of a class's methods.

SDMPC- Standard Deviation of Methods Per Class

This metric is used the standard deviation of methods per class in the system which the user write as SDMPC(C) (note that the number of methods in the system stays constant no matter how many move method refactoring is use).

Ca - Afferent couplings

A class's afferent couplings are a measure of how many other classes use the specific class. Ca is calculated using the same definition as that used for calculating CBO (Ce).

LSCC- Low Level Class Cohesion Metric

Low Level Class Cohesion Metric is represents the classes. Meaningful class coupling and cohesion metric helps object-oriented software developers detect class design weaknesses and refactor classes accordingly.

NPM - Number of Public Methods

The NPM metric simply counts all the methods in a class that are declared as public. It can be used to measure the size of an API provided by a package.

In this proposed work, is combining LSCC with a simple ‘counter metric’ to CBO’s tendency to expand a small number of classes with large numbers of methods. The third metric is the standard deviation of methods per class in the system which it write as SDMPC(C) (note that the number of methods in the system stays constant no matter how many move method refactoring is use).

5. An Efficient SDMPC Metric Based Approach for Refactoring Software Code

5.1 JSync: Refactoring

In JSync, a software system is considered as a collection of source files. Each source file corresponds to a logical entity called compilation unit. A fragment corresponds to one or a collection of program entities for example, statement, method, class that is of user interest in clone management. Users are able to exclude the generated source files or annotate the portions of code that are generated or boilerplate code (e.g., getter/setter). JSync would totally ignore them in building fragments (i.e., similar handling for comments and Javadoc) or skip building the corresponding fragment(s) but still use the features extracted from them in building other fragments. Fragments are copied, pasted, and sometimes modified, thus producing code clones. Detected clones of object, class, method are refactored by JSync refactoring tool. In this type of clones, the cloned fragment is not necessarily copied from the original. JSync refactoring tool considers cloned code to have similar structures. It defines a pair of two fragments as a clone pair if their structural similarity, measured by a similarity measurement, exceeds a predefined threshold. Those fragments are called cloned fragments (or clones for short).

JSync provides several techniques to deal with the analysis and consistent updating of clones and their changes. Clone consistency analysis of JSync finds the matched and different entities between two cloned fragments and then validates them against the aforementioned clone consistency rules. Clone Synchronizing is the operation designed for two clone change scenarios, cloning and one-side change, that is, when there is only one clone that was changed. For the two side change, JSync uses Clone Merging.

Before refactoring is done, the test case should be generated. A formal written test-case is characterized by a known input and by an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post condition.

A test case has components that describe an input, action or event and an expected response, to determine if a feature of an application is working correctly. The basic objective of writing test cases is to validate the refactoring coverage of the application. A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test cases tend to have a high rate of code duplication, which is typically the result of a sequence of copy, paste and modify actions. The clone detection and refactoring capabilities of JSync would be used to remove a number of testing 'bad smells' and also to introduce common testing patterns.

5.2 Metrics for Refactoring

The first, metric (CBO) Coupling Between Object classes represents the number of classes coupled to a given class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.

This metric provides the average number of classes used per class in the package.

$$CBO = \frac{Numbers\ of\ Links}{Number\ of\ Classes}$$

The variable number of links represents the number of classes used (associations, use links) for all the package's classes. A class used several times by another class is only counted once.

The variable number of classes represents the number of classes of the package, by recursively processing sub-packages and classes. For the UML modelling project, this variable represents, therefore, the total number of classes of the UML modelling project.

The second metric (LSCC) Low Level Class Cohesion Metric is represents the classes. Meaningful class coupling and cohesion metric helps object-oriented software developers detect class design weaknesses and refactor classes accordingly. The results show that LSCC is better than CBO metric.

LSCC=

$$\begin{cases} 0 & \text{if } l\text{=0 and } k >1, \\ 1 & \text{if } (l\text{> 0 and } k\text{=0}) \text{ or } k\text{=1}, \\ \sum_{i=1}^{l} x_i(x_i - 1)/lk(k-1) & \textit{otherwise.} \end{cases}$$

The formula that precisely measures the degree of interaction between each pair of methods, and it used as a basic to introduce low level design class cohesion metric. Where l is the number of attributes, k is the number of methods, and x_i is the numbers of methods that reference attribute$i$.

The similarity between two methods is the collection of their direct and indirect shared attributes. It is an important objective in object oriented design. Class cohesion refers to the relatedness of the class members, and it indicates one important aspect of the class design quality.

Third, metric is used the standard deviation of methods per class in the system which the user write as SDMPC(C) (note that the number of methods in the system stays constant no matter how many move method refactoring is use).

The result shows that one of three metrics that explains more accurately the presence of faults in methods. SDMPC(C) is the only one among the three metrics to comply with important mathematical properties, and statistical analysis shows it captures a measurement dimension of its own. This suggests that SDMPC is a better alternative, when taking into account both theoretical and empirical results, as a measure to guide the refactoring of methods.

5.3 Search-Based Refactoring Approach Using Combining Metrics

More obvious ways to combine the CBO and SDMPC metrics into a fitness function are $\mathrm{CBO(C)} * \mathrm{SDMPC(C)} * \mathrm{LSCC}$ and $\mathrm{CBO(C)} + \mathrm{SDMPC(C)} + \mathrm{LSCC}$. Thus far in this project used the former of these two fitness functions to guide the search-based refactoring system. Using the latter fitness function on any of the systems under examination in this project, the programmer quickly find that it is non-inferior i.e. it produces distinct Pareto optimal values.

Using a single metric to guide search-based refactoring has obvious problems: optimizing only one aspect of the system can make other important measures of quality un-acceptably worse. Therefore it is common to combine more than one metric when designing an appropriate fitness function, with the intuitive idea that the combination of metrics should prevent any one metric being unduly favoured. In the case of the previous example, statistical theory provides a simple ‘counter metric’ to CBO’s tendency to bloat a small number of classes with large numbers of methods. The second metric use is the standard deviation of methods per class in the system which is written as SDMPC(C) (note that the number of methods in the system stays constant no matter how many move method refactoring use). It now comes up against an immediate problem: how should it combine these two metrics into one fitness function? Initial candidates include $\mathrm{CBO(C)} * \mathrm{SDMPC(C)} * \mathrm{LSCC}$ or $\mathrm{CBO(C)} + \mathrm{SDMPC(C)} + \mathrm{LSCC}$, possibly with weightings attached to the individual metrics. Previous search-based refactoring approaches combine metrics together in often complex fashions, and with the choice of weightings for various metrics often unclear. In similar fashion it initially arbitrarily defines in new fitness function to be $\mathrm{CBO(C)} * \mathrm{SDMPC(C)} * \mathrm{LSCC}$.

6 Result

6.1 Pareto Optimality

The optimized refactoring approach uses the concept of Pareto optimality naturally applies to search-based refactoring. In order to do that, first define the concept of Pareto optimality and a Pareto front. In economics a value is effectively a tuple of various metrics which would be made better or worse. A value is Pareto optimal if moving from it to any other value makes any of the constituent metrics better or worse; it is said to be a value which is not dominated by any other value. For any given set of values there will be one or more Pareto optimal values. The sub-set of values which are all Pareto optimal is termed the Pareto front. It also uses multiple fitness function to guide the search-based refactoring system. It will be more efficient than the refactoring system with simple or two fitness functions.

6.1.1 Test Case Performance of Pareto Optimality

In this proposed system, it builds search-based system in the Converge language which reads in arbitrary Java systems, performs search-based refactoring upon them, and returns a sequence of refactoring as its output. In this work first, develop a small Java application that has severe cohesion problems. Refractor of these programs are using JSync tool. It improves its design according to the combined CBO, LSCC and SDMPC metric.

The refractor is optimized with Pareto based approach and has multiple fitness function. Then generate test cases for the both versions of the program, before and after refactoring, and to compare the difficulty in generating the test cases. If generating test cases for the refactored version of the program proves significantly easier, then there is indicative evidence that automated refactoring indeed improve testability.

The advantage of the proposed system is to improve the code readability. Computational cost/complexity is reduced by using the proposed methods. It also improves the performance of the system.

Table 6.1

Average Performance for Code Clone Before and After Refactoring

Projects	**Before Refactoring**	**After Refactoring**
CFIS	10	5
PMCS	15	10
IE	10	5.5
OCMS	20	10
CLMS	5.5	5

Table 6.1 shows before and after refactoring of cloned code. Traditionally, automated refactoring inference relies on comparing five different versions of source code and describing the changes between versions of code using higher-level characteristic properties.

A refactoring is detected based on how well it matches a set of characteristic properties.

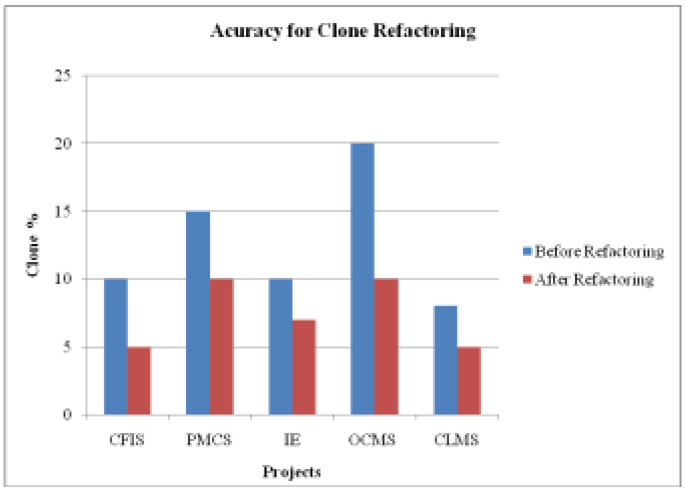

Figure 6.1: Accuracy for Code Clone Before and After Refactoring

The figure 6.1 shows accuracy for code clone of before and after refactoring. The clone level is decrease from after refactoring.

The Table 6.1 shows the average performance of test case of above said novel approaches. It illustrates the complete comparison among the enhanced clone detection through average.

Table 6.2

AVERAGE PERFORMANCE OF CBO And LSCC

Projects	**CBO**	**LSCC**
CFIS	7.9	5.45
PMCS	8.8	3.8
IE	5.4	7.3
OCMS	3.5	3.02
CLMS	3.5	1.7

Table 6.2 shows average performance of test case. The LSCC metric is better to the CBO metric.

A visual inspection of the performance of these metrics in fig 6.2 evidence that their results from one metric to another.

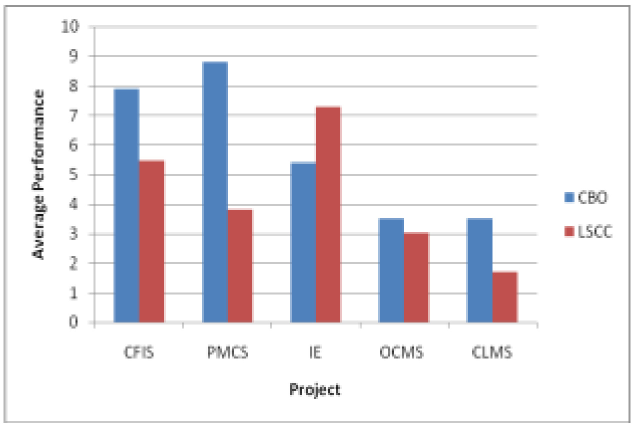

Figure 6.2: Average Performance of CBO and LSCC

The complexity of these metrics is occurring to during software development. LSCC metric is better than the CBO metric for calculate the quality of the product.

Table 6.3

AVERAGE PERFORMANCE OF CBO And SDMPC

Projects	**CBO**	**SDMPC**
CFIS	7.9	1.15
PMCS	8.8	0.15
IE	5.4	1.2
OCMS	3.5	0.16
CLMS	3.5	0.18

Table 6.3 shows average performance of test case. The CBO metric is better to the SDMPC metric.

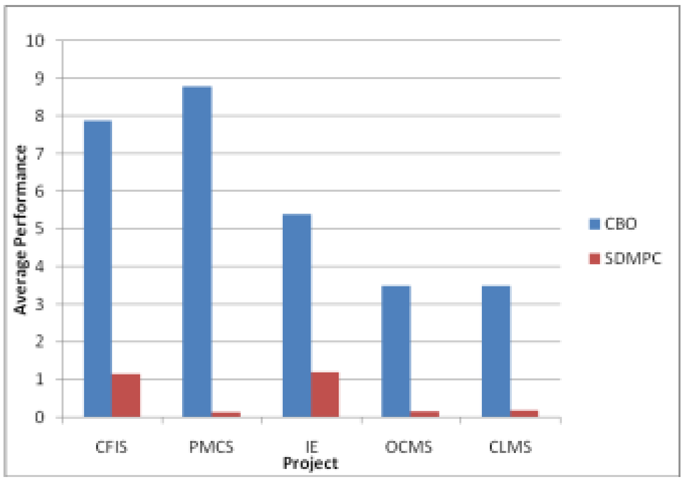

Figure 6.3: Average Performance of CBO and SDMPC

The test case performance of automated refactoring of quality is calculated by CBO and SDMPC metric. Fig 6.3 shows the SDMPC is better than the CBO metric for test case.

Table 6.4

AVERAGE PERFORMANCE OF LSCC AND SDMPC

Projects	LSCC	SDMPC
CFIS	5.45	1.15
PMCS	3.8	0.15
IE	7.3	1.2
OCMS	3.02	0.16
CLMS	1.7	0.18

Table 6.4 shows average performance of test case. The LSCC metric is better to the SDMPC metric.

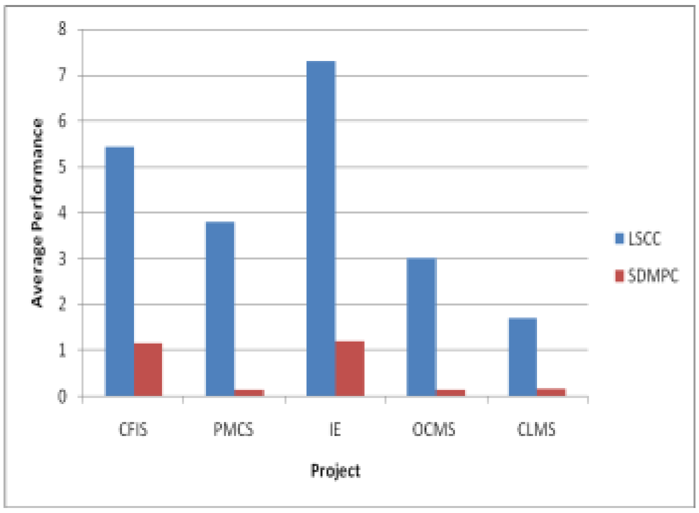

Figure 6.4: Average Performance of LSCC and SDMPC

Finally, LSCC and SDMPC metrics are calculated the performance of the test case. The fig 6.4 shows SDMPC metric is better for the deliberate the quality of the test case.

The figures 6.2 to 6.5 show the average performance of refactoring test case.

This experimental investigation includes five different kinds of java programs, namely Criminal Face Identification System (CFIS), Personal Mobile Crime System (PMCS), Image Encryption (IE), Online Courier Management System (OCMS), and College Library Management System (CLMS).

Table 6.5

AVERAGE PERFORMANCE OF CBO, LSCC AND ADMPC

Projects	CBO	LSCC	SDMPC
CFIS	7.9	5.45	1.15
PMCS	8.8	3.8	0.15
IE	5.4	7.3	1.2
OCMS	3.5	3.02	0.16
CLMS	3.5	1.7	0.18

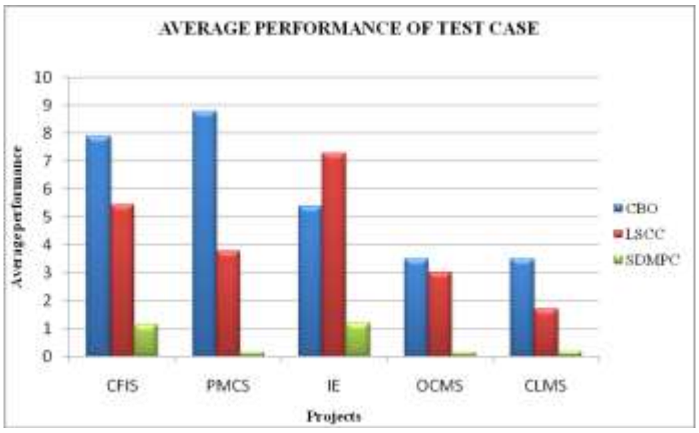

Figure 6.5: Test Case Performance for Various Files

7 Conclusion

The main objective of this paper is to achieve refactor the clone. In this proposed system is build search-based system in the Converge language which reads in arbitrary Java systems, performs search-based refactoring upon them, and returns a sequence of refactoring as its output. In this proposed work first, develop a small Java application that has severe cohesion problems. Refactor this program using JSync refactor tool in order to improve its design according to the combined CBO, LSCC and SDMPC metric.

The CBO metric represents the number of links and number of classes used for the package classes. The LSCC is detecting the refactored classes and metrics. And the SDMPC metric is represents the methods per class. These metrics are overcome the "good" refactoring solution as the combination of refactoring operations that should maximize as much as possible the number of corrected defects with minimal code modification/adaptation effort (*i.e.*, the cost of applying the refactoring sequence).

The refractor was optimized with Pareto based approach and has multiple fitness function. Then generate test cases for the both versions of the program, before and after refactoring, and to compare the difficulty in generating the test cases. If generating test cases for the refactored version of the program proves significantly easier, then there is indicative evidence that automated refactoring can indeed improve testability.

Rather than asking experiment participants to judge the difficulty in writing test cases for the original and refactored versions of a program, and it would be used an automated test cases generation tool to create the test cases. This would be applied to the original and refactored versions of the program and the resulting test suites compared on the basis of metrics such as code coverage, lines of test code and number of assert statements. The feasibility of this would need to be assessed in further research, but it has the advantage that it eschews the need for a costly experiment to assess the result.

By taking three simple metrics, are able to show how the concept of Pareto optimality can be usefully applied to search-based refactoring, and how it allows multiple fitness functions to present different Pareto optimal values to the user.

As an end result, SDMPC performances of the proposed approaches are evaluated through refactoring. The performance evaluation shows that the proposed SDMPC approach performs all the parametric standards than existing approach. The proposed SDMPC approach takes less time for clone removal. And it helps the developers to program to faster. It improves the design of the code. And it also makes software easier to understand.

8 Acknowledgements

Thanks to my college, department, lab which has been used to class room, and lab. Special thanks go to my guide Dr. D Gayathri Devi M.CA., M.Phil., P.hD., for encourage and help to develop my journal.

9 References

[1] Duala-Ekoko.E and Robillard.M.P, “Tracking Code Clones in Evolving Software,” Proc. IEEE 29th Int’l Conf. Software Eng., pp. 158-167, 2007.

[2] Ekwa Duala-Ekoko and Martin P. Robillard “Tracking Code Clones in Evolving Software”, 2007.

[3] Ekwa Duala-Ekoko and Martin P. Robillard “CloneTracker: Tool Support for Code Clone Management”, 2008.

[4] Estublier.J, Leblang.D, van der Hoek.A, Conradi.R, Clemm.G, Tichy.W, and Weber.D, “Impact of SE Research on the Practice of SCM,” ACM Trans. Software Eng. and Methodology, vol. 14, no. 4, pp. 383-430, 2005.

[5] Gatrell.M, Counsell.S and Hall.T “Empirical Support for Two Refactoring Studies Using Commercial C# Software”.

[6] Gode.N and Koschke.R, “Incremental Clone Detection,” Proc. 13th European Conf. Software Maintenance and Reeng., pp. 219-228, 2009.

[7] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen “Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection” 2009.

[8] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen, Member “Clone Management for Evolving Software”, IEEE 2012.

[9] Lingxiao Jiang, Zhendong Su and Edwin Chiu “Context-Based Detection of Clone-Related Bugs”, 2007.

[10] Mark Gabel, Lingxiao Jiang and Zhendong Su “Scalable Detection of Semantic Clones”, 2008.

[11] Miryung Kim, Thomas Zimmermann, Nachiappan Nagappan “A Field Study of Refactoring Challenges and Benefits” 2012.

[12] Nguyen.T.T, Nguyen.H.A, Pham.N.H, Al-Kofahi.J.M, and Nguyen.T.N, “Scalable and Incremental Clone Detection for Evolving Software,” Proc. IEEE Int’l Conf. Software Maintenance, 2009.

[13] Nils Gode “Incremental Clone Detection”, 2008.

[14] Patricia Jablonski and Daqing Hou “CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE”, 2007.

[15] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, Giancarlo Succi and agile team

"A case study on the impact of refactoring on quality and productivity".

[16] Ratzinger.J, Sigmund.T, and Gall.H.G "On the relation of refactorings and software defect prediction". In MSR '08: Proceedings of the 2008 international working conference on Mining software repositories, New York, NY, USA, 2008.

[17] Roy C.K, CordyJ.R, and Koschke.R, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," Science of Computer Programming, vol. 74, no. 7, pp. 470-495, 2009.

[18] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo "Comparison and Evaluation of Code Clone Detection Techniques and Tools", 2007.

[19] Yoshiki Higo, Yasushi Ueda, Shinji Kusumoto and Katsuro Inoue "Simultaneous Modification Support based on Code Clone Analysis" 2007.

[20] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code", 2006.

Author Profile

B Ramalakshmi received the degrees B.sc. Computer Science from Madurai Kamaraj University in 2008 and 2011. M.sc.from Annamalai University in 2011 and 2013 and M.phil. Computer Science from Bharathiyar University in 2014 and 2015.