Code refactoring using slice-based cohesion metrics and aspect-oriented programming

Jagannath Singh*, Pabitra Mohan Khilar and Durga Prasad Mohapatra

Department of Computer Science and Engineering, National Institute of Technology, Rourkela, Orissa, India
Email: jagannath.singh@gmail.com
Email: pmkhilar@nitrkl.ac.in
Email: durga@nitrkl.ac.in
*Corresponding author

Abstract: Software restructuring is essential for maintaining software quality. It is a usual practice that we first design the software and then go for coding. After coding, if there is any change in the requirement or if the output is incorrect, then we have to modify the code again. For each small code modification, it is not feasible to alter the design. These minor changes made to the code causes decay in the software design. Software refactoring is used to restructure the code to improve the design and quality of the software. In this paper, we propose an approach for performing code refactoring. We use slice-based cohesion metrics to identify the target methods that require refactoring. After identifying the target methods, we use program slicing to divide the target method into two parts. Finally, we use the concept of aspects to alter the code structure in a manner that does not change the external behaviour of the original module.

Keywords: software refactoring; program slicing; AOP; cohesion metrics; code restructuring; AspectJ.


Biographical notes: Jagannath Singh is currently working as an Assistant Professor in the School of Computer Engineering at Kalinga Institute of Industrial Technology, Bhubaneswar, India. He is a PhD Research Scholar at the National Institute of Technology Rourkela. He has completed MTech from the National Institute of Technology Rourkela in the year 2012. He had received BE from Biju Pattnaik University of Technology, Orissa. His research interest includes software engineering, artificial neural networks and microprocessor.

Pabitra Mohan Khilar is currently working as an Assistant Professor in the Department of Computer Science and Engineering at the National Institute of Technology, Rourkela, India. He has been involved with teaching undergraduate/graduates and research since his joining as a faculty in the Department of Computer Science and Engineering, NIT, Rourkela, India in the year 1999. He is a member of many international and national professional bodies from where he has received many awards for technical contributions.

Copyright © 2018 Inderscience Enterprises Ltd.
1 Introduction

Any software in the real world emerges by acquiring new requirements, new techniques, and newly changed scenarios. In the early days, the need for change in software raised at long intervals of time. But in the present days, the requirements change at small intervals of time. In order to incorporate all these changes, the code of the software undergoes many modifications and additions. As a result, the code of the software becomes more and more complex and drifts away from its original design (Tom and Tom, 2004). This drifting of code from design is undesirable for software management (Kevin et al., 2013) and risk assessment (Robin and Uma, 2013). It also affects all the descendant activities, like testing, maintenance, etc., to be followed correctly.

In order to solve the above-said problem, we need a technique that will incorporate all the evolving requirements and changes, and simultaneously ensures the quality of the software (Bavota et al., 2011). One such technique is software restructuring. Software refactoring as defined in Tom and Tom (2004) is “the process of changing an object-oriented software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure”. In this process, we redistribute the classes, variables and methods, to make the overall software less complex and of better quality.

But, refactoring of the full software is a tedious task in terms of time and cost involved in it (Tom and Tom, 2004). There exist many research papers dealing with the refactoring techniques (Bavota et al., 2011; Ettinger and Verbaere, 2004; Mohsin et al., 2010). However, these papers do not reveal the techniques used to select the target methods for refactoring. We observe that instead of refactoring all the modules of the software, we should refactor only those modules that need refactoring. Also, we know that the software quality can be evaluated using software metrics (Sudhaman, 2011). Hence, we use slice-based cohesion metrics (Weiser, 1981) to identify those modules that need refactoring. We compute the cohesion of each module in the software and then check their cohesion metric values. If some modules have less cohesion metric value than the acceptable threshold, then that module is restructured. In this paper, we address the modules fit for restructuring as target modules.

Now the problem is to develop a technique to refactor the target modules such that it reduces their complexity. Here, program slicing and aspect-oriented programming (AOP) come into the picture (Walker et al., 1999). Each target module is sliced into a number of
slices by taking the output variables (Green et al., 2009) as slicing criteria. Then, among
the resultant slices, the most similar slices are combined to form a new module. The new
modules that are obtained are defined as *advices* of an *aspect*. As a result of this
technique, the target module will produce the same output at a reduced complexity.

The primary objective of this paper is to develop a technique for code refactoring. We
need to identify these methods, which need refactoring before the start of refactoring
process. We show that the slice-based cohesion metrics are useful in this task. The
division of one method into two partitions is a difficult task. In this paper, we show that
program slicing is useful for partitioning a method and how AspectJ programs are used to
restructure the code into different methods.

*Organisation:* in Section 2, we discuss the basic concepts to understand our work. In
Section 3, we present some related works. Section 4 introduces our proposed refactoring
approach for object-oriented software. Also in this section, we propose the refactoring
algorithms and present the working of our proposed refactoring algorithm along with an
example. In Section 5, we discuss the experimental evaluation of our proposed
refactoring approach by taking some JAVA projects. Section 6 concludes the paper with
some insights into our future work.

## 2 Basic concepts

In this section, we present some of the basic concepts those are required to understand
our proposed approach. First, we present the concepts of program slicing and the
slice-based metrics. Then, we discuss the concepts of AOP and its features. Subsequently,
we discuss some intermediate representations such as program dependence graph and
system dependence graph (SDG). Finally, we present the concept of software refactoring
and its advantages.

### 2.1 Program slicing

In order to reduce the complexity of a program (so that it will be easy for testing and
maintenance), one approach is to break the whole program into smaller parts. Program
slicing is an efficient technique in this regard. Program slicing was first introduced by
Weiser (1981). It is an analysis and transformation technique. It uses the dependency
relationship between the program statements to identify the parts of a program that affect
or get affected by a point of interest. This point of interest is called the *slicing criterion*.
All the program statements that influence or are influenced by the variables mentioned in
the slicing criterion are added to the slice (Binkley and Gallagher, 1996).

The construction of a program slice starts with the definition of a slicing criterion. A
slicing criterion is a set \(<s, v>\), where ‘s’ denotes the statement number and ‘v’ denotes
the subset of variables of the program. Program slicing has been applied to procedural
programs (Goswami et al., 2000), object-oriented programs (Zhao, 2001; Mohapatra
et al., 2006) and aspect-oriented programs (Mohapatra et al., 2008). Program slicing is
very much useful in software testing (Gupta et al., 1996), debugging (Nagarajan et al.,
2012), software maintenance (Beszedes et al., 2001), software reuse (Lanubile and
Visaggio, 1997) and cohesion metrics computation (Green et al., 2009).
2.2 Slice-based cohesion metrics

Software metrics are used to quantify the complexity of software (Meyers and Binkley, 2004). Slice-based cohesion metrics were proposed by Weiser (1981). He informally presented five slice-based metrics: tightness, coverage, overlap, parallelism and clustering. Out of these five, parallelism and clustering are highly correlated with tightness, coverage and overlap. Hence, we can drop these two metrics (Meyers and Binkley, 2004). Two more metrics: MinCoverage and MaxCoverage are proposed by Ott and Thuss (1993). The formalisation of the five metrics is shown below (Meyers and Binkley, 2004):

\[ \text{Tightness}(M) = \frac{|SL_{int}|}{\text{length}(M)} \]  
\[ \text{MinCoverage}(M) = \frac{1}{\text{length}(M)} \min_{v_i} |SL_i| \]  
\[ \text{Coverage}(M) = \frac{1}{|V_o|} \sum_{i=1}^{|V_o|} |SL_i| \]  
\[ \text{MaxCoverage}(M) = \frac{1}{\text{length}(M)} \max_{v_i} |SL_i| \]  
\[ \text{Overlap}(M) = \frac{1}{|V_o|} \sum_{i=1}^{|V_o|} \frac{|SL_{int}|}{|SL_i|} \]

where \( M \) is the module under consideration, \( V_o \) is the set of variables in \( M \), \( V_o \) is the set of output variables, \( SL_i \) is the slice obtained for \( v_i \in V_o \), and \( SL_{int} \) is the intersection of \( SL_i \) over all \( v_i \in V_o \).

After analysing 63 programs, Meyers and Binkley (2004) obtained that there is a strong correlation between tightness and minCoverage, between minCoverage and overlap and between tightness and overlap. Hence, it is not necessary to compute all the metrics. Depending upon this analysis, they gave benchmark values for overlap that lies between 0.6908 and 1.0, and the benchmark value for tightness lies between 0.2973 and 0.3039. We use these values in Section 4 for explaining our refactoring approach.

2.3 Aspect-oriented programming

OOP creates a coupling between the core and cross-cutting concerns; that is undesirable. Cross-cutting concerns are those parts of the program that are scattered across multiple modules of the program and are also tangled with other modules (Singh et al., 2016; Aouag et al., 2014). The most simple and common example of a cross-cutting concern is logging. Logging is a cross-cutting concern because it affects many modules or classes across the software, and it intrudes on the business logic. AOP is also useful in handling concurrent programs (Vajapeyam, 2014). Adding new cross-cutting features and even certain modifications to existing cross-cutting functionalities require modifying the relevant core modules. But AOP provides separation of cross-cutting concerns from the core modules by introducing a new unit of modularisation, called aspect (Kiczales et al.,
Code refactoring using slice-based cohesion metrics

1997). In AOP, we implement cross-cutting concerns in aspects instead of fusing them into the core modules.

2.3.1 Some basic terminologies in AOP

Below, we provide some basic definitions of AOP that are used in our proposed approach (Laddad, 2003).

- **Aspect**: aspects are like classes in OOP that contain functionalities. But aspects are different from classes because aspects are meant to compute cross-cutting concerns to be injected into other codes.

- **Joinpoints**: aspects cross-cut objects at only well-defined points, such as at objects construction, method call, or member variable access points. Such well-defined points are known as joinpoints. Joinpoints include method calls, constructor calls, field accesses, object and class initialisation and others.

- **Point-cut**: the specification for naming a joinpoint is called a point-cut. Point-cut is the collection of joinpoints.

- **Advice**: once the joinpoints are spotted in a program, the intended behaviour must be defined. This behaviour is called advice. An advice can contain anything that an arbitrary Java method can have.

- **Code introduction**: with code introduction, programmers can add variables and methods into a program entity by using aspects.

2.4 Procedure dependence graph

Procedure dependence graph (PDG) is a graphical representation of individual procedures or methods in a program (Sinha et al., 1999). In PDG the statements of the procedure are represented as nodes of the directed graph. There can be two types of edges between any two nodes, data dependence edge and control dependence edge. There exists a data dependence edge between two nodes x and y of the PDG, when the value of a variable is passed from node x to node y. A control dependence edge exists between two nodes of a PDG if the execution of one node depends on the other node. In a program, there can be one or more procedures. Hence, to represent the whole program in a dependency graph, a PDG is not suitable. We required a SDG.

2.5 System dependence graph

SDG is a collection of PDGs (Horwitz et al., 1990) with some additional nodes and edges. SDG contains some extra nodes than PDG such as formal-in, formal-out, actual-in and actual-out nodes. To represent parameter passing, the SDG uses formal parameter vertices: a formal-in vertex for each formal parameter of the procedure and a formal-out vertex for each formal parameter that may be modified by the procedure. In the counterpart side, i.e., the called procedure side, parameter passing is shown through actual parameter vertices: an actual-in vertex for each actual parameter at the call site and an actual-out vertex for each actual parameter that may be modified by the called
procedure. SDG combines the PDGs by call edges. Two PDGs are linked with call edges, when one procedure calls another procedure.

2.6 Software refactoring

According to Fowler (1999), “refactoring is the process of modifying the original structure of the software system in order to reduce the complexity, but without altering its external behaviour”.

When we perform refactoring, we improve the design of the code after it has been written (Fowler, 1999). In the current practice of software development, we first design and then start coding. But over time the requirement keeps on changing and hence the code requires modification. As a result of repeated modification, the integrity of the system and its structure according to the original design gradually fades. But on the other hand, refactoring can take a bad design and rework it into a well designed code.

2.6.1 Advantages of software refactoring

Software refactoring has several advantages (Fowler, 1999). Below we mention some of them.

1. Refactoring improves the design of software

   Generally we change the code to realise short-term goals and as a result of accumulation of these small changes, the design of the program decays. Refactoring helps to redesign the code in order to develop the code in accordance to a good design.

2. Refactoring makes software easier to understand

   The code for a program must be written in a more descriptive manner and easily understandable format. In general practice, we go on changing and adding code until it gives the desired result. As a result, our program is no easier to understand. Software refactoring rearranges the code to improve its understandability.

3. Refactoring helps find the bugs

   Refactoring helps in improving the understandability, and hence also helps in finding the bugs. This is because finding bugs in a whole structured code is easier than finding them in an unstructured code.

4. Refactoring helps developing code faster

   All the earlier points mentioned above, conclude that refactoring helps us writing programs faster. A good design is essential for rapid software development and refactoring enhances the design of a program. Hence, it helps in developing software faster.

3 Related works

Not much work has been done in the field of refactoring of software using slice-based metrics. Here, we compare our proposed technique with some of the existing work. Wang
et al. (2014) have developed a tool called SEGMENT that inserts blank lines into the given method to increase the readability of the program. They tried to identify the important points in a program where there is a need of vertical space between the lines of code to improve readability. But, their technique does not make any change into the internal structure of the program so as to improve the program complexity. Our approach identifies those methods of a given program which are more complex and reduces their complexity by splitting them into a number of methods.

Bavota et al. (2011) proposed an extract class refactoring method based on graph theory. They used structural and semantic analysis to identify the relationships between the methods of a class. One semi-automated tool was developed to improve the cohesion of a class by identifying refactoring opportunities. Here, the target class for refactoring was to be identified by the software engineers. The developed tool did not provide any support for the detection of more complex classes. Also, the proposed approach did not consider the class inheritance while performing the class refactoring; hence, this may cause compilation error or an unexpected change in the behaviour of the program. In our proposed refactoring technique, we provide an approach to find complex methods in given software by calculating their slice-based cohesion metrics. Also, we do not change the overall behaviour of the program, so as to avoid the compilation error or an unexpected error.

Mohsin et al. (2010) discussed code restructuring by using program slicing. In this work, they have shown that program slicing can be used for decomposition of modules. They found that by decomposition, the coupling was lowered to 40% and cohesion grown to 70% more, than before. But their work did not address the most crucial point, i.e., how to decompose a module. In this paper, we discussed in detail the decomposition process of a module.

Monteiro and Fernandes (2008) presented an approach for refactoring of object-oriented programs and conversion into aspect-oriented programs. They collected 17 refactoring techniques to identify the cross-cutting concerns from the programs. Then, in the first phase all the cross-cutting concerns are moved into aspects, leaving behind only the core object-oriented programs. In the next phase, the refactoring techniques are again applied onto the newly created aspects to remove duplicate codes. This work concentrates on application of refactoring, not on any new refactoring technique. In our work, we present a new refactoring technique. Our proposed technique can be applied recursively on a program till the cohesion metrics of all its methods improve up to the threshold values.

Sward et al. (2004) have proposed that cohesion, coupling and cyclomatic complexity (CC) can be used to determine the target module that needs refactoring. With detail examples, they have shown that by refactoring the existing module into two modules, coupling and CC reduces. They have proposed that the modules could be sliced with respect to a set of slicing criteria. It will reduce the average complexity of the module by the same amount as that it had been sliced with respect to individual slicing criterion. In our example program, we have adopted this technique. But, in their approach they have not mentioned clearly how to compute the cohesion, coupling, and CC. Also, they have not presented any idea how to identify the target modules for refactoring and what should be the benchmark values for cohesion, coupling, and CC. We have addressed all these issues in our work.
Applying the refactoring process on given software will enhance the design, if a collection of different refactoring techniques is applied to the software in a proper sequence. Generating a proper sequence of refactoring is a NP-hard problem. Lee et al. (2011) have developed a genetic algorithm-based technique to generate the optimised sequence of refactoring to be applied on the target software. Our proposed refactoring technique can be applied on software after all the remaining refactoring sequences are followed, because our proposed technique works on the individual methods to further enhance the complexity of the software.

4 Our proposed approach

In this section, we present our proposed refactoring approach. First, we describe the proposed refactoring approach with the help of a block diagram. We explain the function of each component of the block diagram. Then, we present the proposed algorithm for refactoring of a given program. Our proposed algorithm is a collection of three algorithms. The first algorithm is the main algorithm that in turn calls the other two algorithms. The second algorithm calculates the cohesion metrics for a given method and the third algorithm splits the target method into two parts – an advice and a method.

4.1 Block diagram

The block diagram of our proposed approach is given in Figure 1. This is a collection of seven basic blocks of our approach, and it shows the stepwise flow of activities that must be carried out to perform code refactoring.

As shown in Figure 1, the class file of the JAVA program, i.e., the byte code of the program, is given as input. The SDG constructor (block 1) produces the SDG for the whole JAVA class. The SDG consists of representations for all the methods in the class. But, we need SDG for each method individually. So, we give the SDG of the whole class to the Modulariser.

Modulariser component (block 2) takes the SDG of JAVA class file as input and produces the PDG for individual methods in the class. Then, we need to compute the slice-based cohesion metrics for each method. Using equations (1)-(5) in Section 2.2 (Meyers and Binkley, 2004), we compute the values of the different slice-based cohesion metrics (block 3). According to the research by Meyers and Binkley (2004), among the five slice-based cohesion metrics, tightness and overlap are most important and if these two are computed then rest three can be covered. Hence, in our approach we consider tightness and overlap to simplify the technique and decrease the computation time.

Now, the values of the computed cohesion metrics are compared with the threshold benchmark values (block 4), to find the methods that need refactoring. According to the research conducted by Meyers and Binkley (2004), the tightness of any method must be in between 0.2973 and 0.3039. Similarly, the value of overlap should not be less than 0.6908. Hence in our approach, we have taken those values (i.e., 0.3039 for tightness and 0.6908 for overlap) as threshold values to identify the methods those require refactoring.
Figure 1  Block diagram of our approach

1. SDG Constructor
   - SDG for Class

2. MODULARIZER
   - PDG of individual method

3. SLICE-BASED COHESION
   - Cohesion values

4. Threshold > Cohesion Value
   - Yes: NOOP *
   - No: REFACTORING

5. No. of Output Variables > 1
   - No: NOOP *
   - Yes: SDG Output & Variables

6. Slicer
   - Generated Slices

7. AspectJ Modularizer

END

Note: *NOP: no operation.
Refactoring:

In refactoring, we consider each method identified during the above process. We first fetch the number of output variables present in the method. For simplification, we have considered only the printed variables as output variables. If the number of output variables is greater than 1 (block 5), then only refactoring is possible. Then, the output variables and SDGs of the methods are supplied to the Slicer component (block 6). The Slicer component computes the slices of the input method w.r.t. the output variables.

Now, the computed slices are given as input to the AspectJ modulariser component (block 7). The main task of AspectJ modulariser is to increase the cohesion by creating an Aspect and fitting the computed slices into the Aspect. Here, we create an aspect using AspectJ program. Aspects can contain advices, which are similar to the methods in Java. So, we accommodate the computed slices into the aspects as their advices. After creation of the aspects, we remove the codes in the slice from the original method.

After the completion of the above said steps, we are at the end of one iteration of the refactoring process. Again, we have to compute the cohesion metrics for all the methods using equation (1)–(5) and check whether these values agree with the threshold values or not. We repeat the above steps again till the cohesion metrics of all the methods satisfy the threshold benchmark values. If all the methods satisfy the prescribed threshold benchmark values, then the process of refactoring is stopped.

Algorithm 1: Refactoring (SDG, threshold)

Require: SDC < Node, Edge >

Require: Threshold for < tightness, coverage >

• generate procedure dependence graph (PDG) from SDG

repeat
  • identify output variable nodes \( n \)
  • call Compute_Cohesion(PDG, \( n \))
  if Threshold values > (tightness, coverage) then
    • call Binary_Refactoring(PDG, \( n \))
  end if
until all methods are processed

4.2 Methodology

In this section, we discuss our methodology for refactoring of object-oriented software. We propose a refactoring algorithm (Algorithm 1), that takes the SDG of the target program (that we want to refactor) and the threshold values for slice-based cohesion metrics, as input. It then forms the PDG by deleting the call and parameter passing edges from the given SDG. Then the algorithm calls Algorithm 2, i.e., Compute_Cohesion algorithm, which takes the PDG, output, and \( n \) as the input and calculates the slice-based cohesion metrics for each PDG. These computed cohesion metrics values are returned to refactoring algorithm (Algorithm 1).

After checking the cohesion metrics values (tightness and coverage) for each method, against the corresponding threshold values, the refactoring algorithm decides which methods must be refactored. It then calls another algorithm called Binary_Refactoring
(Algorithm 3). The Binary_Refactoring algorithm then splits the identified methods into two parts, one of which is a new advice in an aspect program, and the other is the given module itself having less complex code. This process is recursively carried out till the cohesion metrics values (tightness and coverage) of the resultant modules are greater than the corresponding threshold values.

Correctness proof of the refactoring algorithm

The main controlling algorithm in our approach is Refactoring() (Algorithm 1). In the proof of the correctness of any algorithm, we must follow the steps of completeness, finiteness, and correctness. Hence, the proof of our algorithm consists of three parts. First, we prove that our algorithm is complete, i.e., it covers all the possible cases. Secondly, we show that our algorithm terminates after a finite number of iterations. Finally, we prove that the algorithm is correct.

For the proof of completeness of our algorithm, let’s assume that the SDG for a given program is generated and readily available to us. The Refactoring() algorithm takes this SDG as input. All the call edges of SDG are removed, in order to get the PDGs. Suppose, we get $p$ number of PDGs out of the given SDG after removing all call edges. Each PDG represents one method in the input program. It means that there are $p$ numbers of methods in the given input program. The Refactoring algorithm repeats the intended process for $p$ times. It shows that all the methods present in a given program are handled by our algorithm. Hence, our proposed refactoring algorithm is complete.

The refactoring algorithm itself terminates after $p$ iterations, as shown above. The Refactoring algorithm calls two more algorithms, i.e., Compute_Cohesion() and Binary_Refactoring(). Compute_Cohesion() algorithm does only the calculation of different slice-based cohesion metrics values, and it terminates after the computation. The other algorithm is Binary_Refactoring(), which computes the slices of given PDG and groups them into two modules. As there are finite number of slices computed by this algorithm, we can assure that after finite time the Binary_Refactoring() algorithm terminates. As a result the whole refactoring process terminates after finite execution time.

We prove the correctness of our algorithm by proof by cases. In the Refactoring() algorithm, first we compute the cohesion metrics of a method and then check whether the method needs refactoring or not. Depending upon the computed cohesion metrics values of a given method and the input threshold values provided by the user, there can be three possible cases. Our proposed algorithm handles these three cases as given below:

Case 1 Threshold > computed cohesion metrics values

According to the main aim of refactoring, the cohesion metrics values for a method should be less than the standard threshold metrics values. When the given threshold values are greater than the method’s cohesion metrics values then the method needs refactoring. In our Refactoring() algorithm, it calls the Binary_Refactoring() algorithm that refactors the given method. So, this case can be handled correctly by our algorithm.

Case 2 Threshold < computed cohesion metrics values

When the given threshold cohesion metrics values is less than the method’s computed cohesion metrics values, the method remains unchanged. In the
proposed Refactoring() algorithm, the same thing happens. In the algorithm, when it finds that the given threshold values are less than the method’s cohesion metrics values, it does nothing and proceeds for next iteration.

Case 3  Threshold == computed cohesion metrics values

Any method’s cohesion metrics values should be greater than or equal to the threshold values provided by the user. So, when the given threshold values are equal to the method’s cohesion metrics values, then the method needs no refactoring. In the proposed Refactoring() algorithm, when this situation arises, the algorithm does nothing and the given method remains as it is.

Form the above cases, it can be deduced that our proposed refactoring algorithm works correctly in all the possible conditions and hence, the algorithm is correct. □

4.2.1 Working of the algorithm and implementation

In this section, we explain the working of our proposed algorithm and its implementation. For implementation of our technique, we have developed one SDG generator for Java programs. The basic part of the tool is based on a Java SDG generation API, that is an open source API which can be download from JAVA API for SDG Generation. This API generates SDG according to the representation proposed by Walkinshaw et al. (2003). Our tool analyses the byte-code of the class file and produces the SDG for the given program.

Figure 2  An example program

```java
class Metrics{
    public static void main(String args[]){
        int[] sizes = {12, 6, 8, 10, 6};
        int module = 20;
        int v0 = 6;
        calcMetrics(sizes, module, slint, v0);
    }

    static void calcMetrics(int[] sizes, int module, int slint, int v0) {
        double min = 99999.9;
        double max = 0.0;
        int sumslice = 0;
        double tightness, coverage, overlap;
        int ti;
        for (ti = 0; ti < v0; ti++) { // ...
            tightness = (double) slint / module;
            coverage = (double) sumslice / (v0 * module);
            overlap = calcoverlap(sizes, slint, v0);
            min = min / module;
            max = max / module;
            System.out.println("tightness = " + tightness);
            System.out.println("coverage = " + coverage);
            System.out.println("overlapping coverage = " + max);
            System.out.println("overlapping = " + overlap);
        }

    }

    static double calcoverlap (int sizes[], int slint, int v0) {
        double total = 0.0;
        int t;
        for (t = 0; t < v0; t++) { // ...
            total = total + (float) slint/sizes[t];
        }
        return total/v0;
    }
}
Figure 3  SDG of the example program given in Figure 2
We have considered an example program shown in Figure 2 that calculates the values of five cohesion metrics, i.e., tightness, minCoverage, coverage, maxCoverage, overlap and also displays their values (Green et al., 2009). The byte-code of the class `metrics` is given as input to the SDG generator tool (block 1). Our tool analyses the byte-code of the class file and produces the SDG for the given program, as shown in Figure 3.

Now the user has to enter the threshold values for the cohesion metrics. The benchmark values for only tightness and overlap are maintained in a study by Meyers and Binkley (2004). In our approach, we consider the threshold values for tightness as 0.3039 and for overlap as 0.6908.

### 4.2.1.1 Working of refactoring algorithm

We supply the generated SDG and the threshold values for tightness and coverage, as input to the refactoring algorithm.

The algorithm first searches for the class dependence edges, call edges and parameter edges in the given SDG. It deletes all class and parameter edges and separately generates PDG for each method present in the program as shown in Figure 4.

**Figure 4** PDG of the example program given in Figure 2

Then the algorithm processes all PDGs one-by-one. It first identifies all the output nodes from the PDG of the current method. For simplification of explanation, we consider the output nodes as those nodes which have out-degree = 0. The algorithm counts such nodes and stores in a variable \( n \).

Then, the refactoring algorithm calls Algorithm 2 (Compute_Cohesion algorithm) to compute the tightness and coverage of each method. Once it gets those values, then it compares them with the threshold values provided by the user. If the metrics values for a method, are less than the corresponding threshold values, then it refactors that method by calling `Binary_Refactoring()`.
Algorithm 2: Compute_Cohesion (PDG, n, output)

Require: PDG < Node, Edge >
Require: Integer n
Require: List of output variables

for i = 1 to n do
• compute the slice taking the output [i] as slicing criterion
• store each node of the slice in List [i]
end for

• compute
\[ SL_i = \text{number of nodes in the List }[i] \]
\[ SL_{int} = \text{number of common nodes in all the List}\]
\[ length(M) = \text{number of nodes in the PDG} \]

• compute cohesion metrics using following formulae:
\[ \text{Tightness}(M) = \frac{SL_{int}}{length(M)} \]
\[ \text{Coverage}(M) = \frac{1}{n} \sum_{i=1}^{n} \frac{|SL_i|}{length(M)} \]

return (tightness, coverage)

4.2.1.2 Working of Compute_Cohesion algorithm

The cohesion metrics for a method are computed by calling Compute_Cohesion() algorithm. The Compute_Cohesion algorithm takes the following as input: the PDG of the current method, n and the output nodes. It then computes the slices from the PDG by taking each output node as the slicing criterion. Hence, we get n number of slices which are stored in an array called List[]. The detailed analyses for all the three methods in the example program are shown in Table 1. Then, the algorithm computes the tightness and overlap values according to the equations given in Section 2.2. Table 2 shows the computed metrics values for the example program. These values are returned to the called algorithm, i.e., Refactoring().

<table>
<thead>
<tr>
<th>Method name</th>
<th>length (no. of nodes in PDG)</th>
<th>Output node</th>
<th>Nodes in the slice list { }</th>
<th>SLi</th>
<th>Intersection nodes</th>
<th>SLint</th>
</tr>
</thead>
<tbody>
<tr>
<td>calcMetrics</td>
<td>31</td>
<td>40</td>
<td>{14, 30, 16, 17}</td>
<td>4</td>
<td>{14}</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>41</td>
<td>{14, 31, 16, 18, 21, 28, 15, 22, 23}</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>42</td>
<td>{14, 38, 16, 19, 25, 15, 22, 24, 23, 18}</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>43</td>
<td>{14, 39, 16, 20, 27, 15, 22, 26, 23, 18}</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>44</td>
<td>{14, 37, 36, 32, 34, 17, 33, 15, 35, 18}</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>calcOverlap</td>
<td>11</td>
<td>49</td>
<td>{45, 55, 48, 50, 53, 46, 47, 51, 52}</td>
<td>10</td>
<td>All</td>
<td>10</td>
</tr>
<tr>
<td>main</td>
<td>11</td>
<td>None</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 2  Original slice-based metrics calculated for the example program given in Figure 2

<table>
<thead>
<tr>
<th>Metrics</th>
<th>main()</th>
<th>calcMetrics()</th>
<th>calcOverlap()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tightness</td>
<td>1</td>
<td>0.032</td>
<td>1</td>
</tr>
<tr>
<td>Coverage</td>
<td>1</td>
<td>0.213</td>
<td>1</td>
</tr>
<tr>
<td>MinCoverage</td>
<td>1</td>
<td>0.129</td>
<td>1</td>
</tr>
<tr>
<td>MaxCoverage</td>
<td>1</td>
<td>0.322</td>
<td>1</td>
</tr>
<tr>
<td>Overlap</td>
<td>1</td>
<td>0.132</td>
<td>1</td>
</tr>
</tbody>
</table>

4.2.1.3 Working of Binary_Refactoring algorithm

Now after checking the values of cohesion metrics of all methods, we observed that only calcMetrics() fails to satisfy the threshold values. Hence, it must be refactored. Now the Refactoring algorithm will call Binary_Refactoring() for the method calcMetrics. Binary_Refactoring algorithm takes the PDG of the method and the number of output variable nodes (n) as the input and computes the slices of the PDG for the given method.

Algorithm 3: Binary_Refactoring(PDG, n)

Require: PDG < Node, Edge >
Require: Integer n
for i = 1 to n do
  • compute the slice taking the output [i] as slicing criterion
  • store each node of the slice in List [i]
end for
• compute $m = \lfloor n/2 \rfloor$
• find $List_{aspect} = \bigcup_{i=1}^{m} List[i]$ most common nodes
• declare an after() advice into the Aspect
• move codes represented by $List_{aspect}$ from current method to after() advice
from current method to after() advice

As the name suggests, this algorithm divides the given method into two parts. According to this algorithm, we compute the value of $m$, as the lower bound of $n/2$. Now, we have to choose $m$ number of slices, those are having most common nodes. Then, we find the union of these $m$ slices and store the result in an array $List_{aspect}$. Now we have to create an Aspect using any AOP language. Here, for our implementation, we have used AspectJ for creating aspects. Then, within the aspect, we declare an advice after(), that handles the call of the target method. Initially, advice after() does not contain anything.

Next, we have to identify the codes, in the target method, those are represented by nodes of the $List_{aspect}$ in the PDG. Move all the codes from the body of the method, into the advice after(). After the completion of Binary_Refactoring() algorithm, the newly created aspect will look like the code as shown in Figure 5. The code for the modified calcMetrics is shown in Figure 6. We have calculated the slice-based cohesion metrics for the newly created advice after() and the modified calcMetrics(), as shown in Table 3 and found that now all cohesion metrics values are within the threshold. So, the process of refactoring stops at this point.
Figure 5  The newly created aspect for the example program in Figure 2 after refactoring

```java
public aspect aspect_calcMetrics {
    after(int sizes[], int module, int slint, int vo):
        call(void calc.calcMetrics(int [],int,int))
        && args(sizes[],module,slint,vo)
    {
        double min=9999.9;
        double max=0.0;
        int i;
        for(i=0; i<vo;i++)
        {
            if(sizes[i]>min) min=sizes[i];
            if(sizes[i]>max) max=sizes[i];
        }
        min=min/module;
        max=max/module;
        System.out.println("Min-coverage="+min);
        System.out.println("Max-coverage="+max);
    }
}
```

Figure 6  Final modified example program after refactoring

```java
public class calc {
    public static void main(String args[])
    {
        int sizes[]={12,3,4,5,6,7,8,9,0};
        int module=20;
        int slint=4;
        int vo=8;
        calcMetrics(sizes,module,slint,vo);
    }
    public static void calcMetrics( int sizes[], int module, int slint, int vo)
    {
        int sumslice=0;
        double tightness, coverage, overlap;
        int i;
        for(i=0; i<vo;i++)
        {
            sumslice=sumslice+sizes[i];
        }
        tightness=(double)slint/module;
        coverage=(double)sumslice/(vo/module);
        overlap=calcOverlap(sizes,slint,vo);
        System.out.println("Tightness="+tightness);
        System.out.println("Coverage="+coverage);
        System.out.println("Overlap="+overlap);
    }
    static double calcOverlap( int sizes[],int slint, int vo)
    {
        double total=0.0;
        for( int j=0; j<vo;j++)
        {
            total+=slint/sizes[j];
        }
        return total/vo;
    }
}
```

Table 3  Updated Slice-based Metrics for calcMetrics() and after()

<table>
<thead>
<tr>
<th>Metrics</th>
<th>calcMetrics()</th>
<th>after()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tightness</td>
<td>0.04</td>
<td>0.353</td>
</tr>
<tr>
<td>Coverage</td>
<td>0.306</td>
<td>0.588</td>
</tr>
<tr>
<td>Min-Coverage</td>
<td>0.16</td>
<td>0.588</td>
</tr>
<tr>
<td>Max-Coverage</td>
<td>0.4</td>
<td>0.588</td>
</tr>
<tr>
<td>Overlap</td>
<td>0.153</td>
<td>0.588</td>
</tr>
</tbody>
</table>
5  Experiment and results

We have developed a tool named JSlicer for the construction of SDG of Java programs and to compute the slices. We have applied our proposed refactoring technique on some benchmark open-source Java projects to evaluate the effectiveness of our approach. For our study, we have taken 11 Java projects, whose size ranges from 100 to 1,000 lines of code. In the study, the cohesion metrics values for 651 methods are calculated. A total of 5,000 lines of code are analysed. The details of the case study projects are given in Table 4. Table 4 shows the project name, lines of code (LOC), number of classes and methods present in a particular project, number of slicing criteria considered for refactoring, average size of the slices computed, and average slice computation time.

Table 4  Details of the case study projects

<table>
<thead>
<tr>
<th>Sl. no.</th>
<th>Project name</th>
<th>LOC</th>
<th>No. of Classes</th>
<th>No. of methods</th>
<th>No. of slicing criteria considered</th>
<th>Avg. slice size</th>
<th>Avg. slice computation time (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alarm clock</td>
<td>125</td>
<td>6</td>
<td>20</td>
<td>12</td>
<td>7.44</td>
<td>3.08</td>
</tr>
<tr>
<td>2</td>
<td>Binary search tree</td>
<td>130</td>
<td>4</td>
<td>23</td>
<td>6</td>
<td>12.37</td>
<td>5.25</td>
</tr>
<tr>
<td>3</td>
<td>Cruise control</td>
<td>261</td>
<td>4</td>
<td>32</td>
<td>17</td>
<td>9.10</td>
<td>2.61</td>
</tr>
<tr>
<td>4</td>
<td>Groovy</td>
<td>361</td>
<td>2</td>
<td>34</td>
<td>14</td>
<td>5.91</td>
<td>2.79</td>
</tr>
<tr>
<td>5</td>
<td>Daisy</td>
<td>883</td>
<td>22</td>
<td>106</td>
<td>28</td>
<td>17.33</td>
<td>4.05</td>
</tr>
<tr>
<td>6</td>
<td>Deos</td>
<td>838</td>
<td>24</td>
<td>133</td>
<td>15</td>
<td>10.07</td>
<td>3.18</td>
</tr>
<tr>
<td>7</td>
<td>Double linked list</td>
<td>277</td>
<td>1</td>
<td>32</td>
<td>6</td>
<td>15.14</td>
<td>2.85</td>
</tr>
<tr>
<td>8</td>
<td>Elevator</td>
<td>934</td>
<td>12</td>
<td>97</td>
<td>11</td>
<td>7.29</td>
<td>2.10</td>
</tr>
<tr>
<td>9</td>
<td>Lang</td>
<td>990</td>
<td>4</td>
<td>101</td>
<td>21</td>
<td>24.51</td>
<td>6.39</td>
</tr>
<tr>
<td>10</td>
<td>Vector</td>
<td>254</td>
<td>1</td>
<td>49</td>
<td>7</td>
<td>15.45</td>
<td>4.98</td>
</tr>
<tr>
<td>11</td>
<td>Red black tree</td>
<td>334</td>
<td>1</td>
<td>24</td>
<td>8</td>
<td>6.02</td>
<td>3.27</td>
</tr>
</tbody>
</table>

Table 5  Details of change in cohesion metrics due to refactoring

<table>
<thead>
<tr>
<th>Sl. no.</th>
<th>Project name</th>
<th>Tightness</th>
<th>Coverage</th>
<th>Max-coverage</th>
<th>Min-coverage</th>
<th>Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>before</td>
<td>after</td>
<td>before</td>
<td>after</td>
<td>before</td>
</tr>
<tr>
<td>1</td>
<td>Alarm clock</td>
<td>0.44</td>
<td>0.51</td>
<td>0.66</td>
<td>0.69</td>
<td>0.55</td>
</tr>
<tr>
<td>2</td>
<td>BST</td>
<td>0.66</td>
<td>0.73</td>
<td>0.75</td>
<td>0.76</td>
<td>0.83</td>
</tr>
<tr>
<td>3</td>
<td>Cruise control</td>
<td>0.25</td>
<td>0.32</td>
<td>0.53</td>
<td>0.63</td>
<td>0.625</td>
</tr>
<tr>
<td>4</td>
<td>Groovy</td>
<td>0.42</td>
<td>0.53</td>
<td>0.66</td>
<td>0.69</td>
<td>0.83</td>
</tr>
<tr>
<td>5</td>
<td>Daisy</td>
<td>0.57</td>
<td>0.66</td>
<td>0.71</td>
<td>0.73</td>
<td>0.71</td>
</tr>
<tr>
<td>6</td>
<td>Deos</td>
<td>0.5</td>
<td>0.66</td>
<td>0.75</td>
<td>0.77</td>
<td>0.83</td>
</tr>
<tr>
<td>7</td>
<td>Double linked list</td>
<td>0.333</td>
<td>0.52</td>
<td>0.66</td>
<td>0.68</td>
<td>0.666</td>
</tr>
<tr>
<td>8</td>
<td>Elevator</td>
<td>0.07</td>
<td>0.12</td>
<td>0.2</td>
<td>0.38</td>
<td>0.28</td>
</tr>
<tr>
<td>9</td>
<td>Lang</td>
<td>0.11</td>
<td>0.32</td>
<td>0.37</td>
<td>0.46</td>
<td>0.27</td>
</tr>
<tr>
<td>10</td>
<td>Vector</td>
<td>0.19</td>
<td>0.35</td>
<td>0.34</td>
<td>0.44</td>
<td>0.35</td>
</tr>
<tr>
<td>11</td>
<td>Red black tree</td>
<td>0.26</td>
<td>0.47</td>
<td>0.47</td>
<td>0.58</td>
<td>0.52</td>
</tr>
</tbody>
</table>
5.1 Results

We have calculated the cohesion metrics of all 651 methods. In our study, we considered only those methods for refactoring whose tightness and overlap values are less than the threshold values. On those selected methods, we have applied our proposed refactoring algorithm. After refactoring, we got the modified methods and we have again calculated their cohesion metrics. We found that the values of the cohesion metrics of all the methods have increased, as shown in Table 5. The effect of refactoring on tightness, coverage, max-coverage, min-coverage and overlap are also shown in Figures 7(a)–7(e). From Figure 7(a), it can be observed that there is 56.39% increase in tightness. Similarly, Figures 7(b)–7(e) show that there is 13.47% increase in coverage, 19.48% increase in max-coverage, 0.6% increase in min-coverage, and 35.3% increase in overlap. From these plots, we clearly observe that, we achieve significant improvement in the values of cohesion metrics by applying our proposed refactoring approach.

Figure 7 Effect of refactoring on slice-based metrics, (a) comparison of tightness (b) comparison of coverage (c) comparison of max-coverage (d) comparison of min-coverage (e) comparison of slice overlap
Figure 7 Effect of refactoring on slice-based metrics, (a) comparison of tightness (b) comparison of coverage (c) comparison of max-coverage (d) comparison of min-coverage (e) comparison of slice overlap (continued)

The results from this study form the paper’s three main contributions:

- first, the result of our study indicates that refactoring of methods improves overall quality of the project
- second, slice-based cohesion metrics are very effective in quantifying the cohesiveness of the methods, for code refactoring
- finally, this study shows that cohesion of the method increases by refactoring.
5.2 Refactoring impact analysis

We have conducted experiments with EclEmma plug-in for Eclipse to check the effect of our proposed refactoring technique on the behaviour of the case study projects. EclEmma is an open source code coverage analysis tool that comes as an Eclipse plug-in. For any Java application executed in Eclipse, EclEmma collects coverage data and automatically calculates code coverage percentage as soon as the application terminates. EclEmma analyses each class and method of a project during its execution. We have used EclEmma plug-in for eclipse to show the effect of our proposed refactoring technique on execution of overall project. We have first designed some JUnit test suit for each case study project. Before applying the proposed refactoring technique, we have noted the code coverage of each case study project. Then, we applied our proposed refactoring technique on each of the project and again calculated the code coverage percentage of each project. The finding of our testing is provided in Table 6. Table 6 contains the name of projects, EclEmma code coverage percentage before and after refactoring, and finally the percentage of change in code coverage. We found that, there is negligible change in the code coverage percentage in each of the case study projects ranging from 0.02% to 3.43%. It shows that our refactoring technique only changes the structure of programs, and does not affect their original functionality. The increase in code coverage percentage is due to the fact that, during refactoring we are dividing one method into two methods and some statements are added as the header of newly created method. When the test cases are applied on the module after refactoring, then more number of program statements will be executed. Hence, there is an increase in code coverage percentage after refactoring of software.

<table>
<thead>
<tr>
<th>Sl. no.</th>
<th>Project name</th>
<th>Percentage of code coverage before refactoring</th>
<th>Percentage of code coverage after refactoring</th>
<th>Percentage of change in code coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alarm clock</td>
<td>69.2</td>
<td>71.5</td>
<td>3.32</td>
</tr>
<tr>
<td>2</td>
<td>Binary search tree</td>
<td>87.5</td>
<td>88.7</td>
<td>1.37</td>
</tr>
<tr>
<td>3</td>
<td>Cruise control</td>
<td>66.6</td>
<td>66.9</td>
<td>0.45</td>
</tr>
<tr>
<td>4</td>
<td>Groovy</td>
<td>46.5</td>
<td>47.2</td>
<td>1.50</td>
</tr>
<tr>
<td>5</td>
<td>Daisy</td>
<td>40.7</td>
<td>42.1</td>
<td>3.43</td>
</tr>
<tr>
<td>6</td>
<td>Deos</td>
<td>24.3</td>
<td>24.8</td>
<td>2.05</td>
</tr>
<tr>
<td>7</td>
<td>Double linked list</td>
<td>74.6</td>
<td>74.9</td>
<td>0.40</td>
</tr>
<tr>
<td>8</td>
<td>Elevator</td>
<td>21.4</td>
<td>22.0</td>
<td>2.80</td>
</tr>
<tr>
<td>9</td>
<td>Lang</td>
<td>30.1</td>
<td>30.8</td>
<td>2.32</td>
</tr>
<tr>
<td>10</td>
<td>Vector</td>
<td>67.8</td>
<td>69.0</td>
<td>1.76</td>
</tr>
<tr>
<td>11</td>
<td>Red black tree</td>
<td>76.7</td>
<td>76.9</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Table 6 Impact of refactoring on code coverage
5.3 Threats to validity

In this section, we present some of the threats to the validity of our proposed approach.

1. We have considered method return and print statements as output variables. We have not considered all types of output variables during the slicing criterion selection of our experiment. We believe that our refactoring approach will produce similar results even after altering the type of output variable.

2. Our proposed technique is developed by keeping in view Java and AspectJ programming languages. We have not tested our technique with other languages. However, we believe that our algorithm may work fine for C++ and AspectC++ after making suitable changes in the SDG construction, considering the relevant features of C++ and AspectC++.

6 Conclusions and future work

In this paper, we presented a novel technique for code refactoring using slice-based cohesion metrics and AOP. We clearly mentioned the cohesion metrics benchmark values that should be satisfied for refactoring. We presented a detailed process for refactoring a given program. To explain our technique, we considered an example Java program. In order to verify the working of our refactoring approach, we considered 11 open-source Java projects. We applied our proposed technique on all the 651 methods present in these projects. We found significant increase in all cohesion metrics after refactoring. We have observed that the increase in tightness is 56.39% and increase in coverage is 13.47%. The limitation of our approach is that it cannot handle multithreaded programs.
Some research papers (Murgia et al., 2011; Reijers and Vanderfeesten, 2004) have stated that an increase in the cohesion metrics value of a module also decreases the coupling metrics value. In future, we plan to evaluate the effect of cohesion change due to the refactoring on coupling. Also, we plan to consider some more slice-based metrics such as CC, coupling, etc. for software refactoring.

References


