A Combined Approach for Identifying Crosscutting Concerns in an Object Oriented System

Shantanu Kumar Biswal

Roll. 213CS3181

under the supervision of

Prof. Ramesh Kumar Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela – 769008, India

A Combined Approach for Identifying Crosscutting Concerns in an Object Oriented System

Dissertation submitted in

MAY 2015

to the department of

Computer Science and Engineering

of

National Institute of Technology Rourkela

in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Shantanu Kumar Biswal

(Roll. 213CS3181)

under the supervision of

Prof. Ramesh Kumar Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela – 769 008, India

Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, India. www.nitrkl.ac.in

MAY, 2015

Certificate

This is to certify that the thesis work entitled *A Combined Approach for Identifying Crosscutting Concerns in an Object oriented system* by *Shantanu Kumar Biswal*, of computer science and engineering department bearing roll number 213CS3181, is a research work carried out by him under the supervision and guidance of me for the partial fulfillment for the award of the degree of *Master of Technology* in *Computer Science and Engineering Department* at National Institute of Technology Rourkela. To the best of my knowledge, neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Ramesh Kumar Mohapatra
Assistant Professor
Department of CSE
NIT, Rourkela

Acknowledgment

I would like to express my great sense of gratitude and respect towards my supervisor Prof. Ramesh Kumar Mohapatra, who has been the guiding me continuously for my research work. He introduced me the research work in the field of Aspect Mining and Aspect Oriented Programming and giving me an opportunity to work under him. This would not be possible without his guidance,support and encouragement. I am also grateful to the persons whose work i have referred in the bibliography.

I would also like to thank all the professors, PhD scholars,and my friends for encouraging me during my work.

At last but not the least I am thankful to my family members my wife Sagarika and to support and help me during my hard times.

Shantanu Kumar Biswal

Abstract

A concern is an essential guideline and an important principle of software engineering development. It refers to some particular concept, the ability to identify functionalities,or some goal to encapsulate the related parts of a software system. Crosscutting concerns of a program are the concerns that affect or crosscut other concern. Usually these concerns are very hard to identify and cannot be clearly separated from the rest of the system, as they are mixed with many core concerns from the system leading to code scattering and code tangling. Identifying crosscutting concerns will automatically improve the maintainability, reliability, understandability and the evolution of the software system. Aspect mining is a reverse engineering process that tries to find out crosscutting concerns in an object oriented software system which is already developed. Aspect mining can be done without using Aspect Oriented Software Development(AOSD) paradigm. Our goal is to locate and identify the crosscutting concerns and then to re-factor them into aspects, to obtain a system that can be easily understood, maintained and modified. In our work we have implemented a combined approach for identifying such crosscutting concerns as one approach is not efficient to identify some of the crosscutting concerns. The first technique is the fan-in analysis which is a static approach for identifying scattered codes whereas the second technique is the dynamic analysis approach where execution traces are examined and recurring execution patterns are obtained for identifying the tangled code.

Keywords :Aspect mining, Concern, Crosscutting concern, Aspects, Aspect oriented programming.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The problems of object oriented programming are the improper decomposition of modules into discrete concerns. Separation of concern is an important role of software design. But some concerns which cannot be cleanly disintegrated from the rest of the systems and mixed with many other core concerns leading to code scattering and code tangling. Code scattering refers to the code which is spread across the software system and code tangling refers to the code which is mixed with other code. Crosscutting concerns are the concerns which affect or crosscut other concerns. The symptoms of crosscutting concerns are code scattering and code tangling. During maintenance phase a developer should localize the code that implements the concern. This may perhaps oblige him to review a wide range of modules as the concern is scattered over the software system. Identifying these crosscutting concerns will automatically enhance the maintainability, reliability,understandability and evolution of the software system. Examples of crosscutting concerns are persistence, synchronization, exception handling, error management and logging. The Aspect Oriented Programming (AOP) paradigm encapsulates all crosscutting concerns and implement them in a localized manner. It defines crosscutting concerns in a new language technique that uses point cuts and advices. All the crosscutting concerns are put in a separate module called aspect. But aspect mining is a reverse engineering process that tries to locate and

recognize crosscutting concerns without using aspect oriented programming in an already developed software system. Different refactoring techniques are used to encapsulate these crosscutting concerns into aspects. This is also called aspect refactoring. This migration of legacy system (Object Oriented Programming) to a new a programming paradigm(Aspect Oriented Programming) can improve the system which can be easily understood and maintained.

1.1 Software Evolution

Evolution of software system is required for developing new systems, reducing complexity, easy understanding and maintenance of the system. Different programming paradigms are developed to help in the evolution of software development. The programming paradigms are as follows:-

1.1.1 Procedural programming

Procedural programing is a programming paradigm which is based on the concept of procedure call. It is derived from the structural programming and inherits some of its features. Procedures focus on the functionality of the program rather than the structure of the program. Procedures are also called methods or functions or subroutines that perform certain functionality. It performs the required functionality by executing a set of commands in a given sequence or procedure. Examples of such programming paradigms are as follows:

- BASIC
- COBOL
- FORTRAN
- C

1.1.2 Functional Programming

It is a programming paradigm that builds the structure and elements of computer programs. Any function of the program can be evaluated in terms of other functions. Examples of such programming paradigms are as follows:-

- Lisp
- ML
- OCaml

1.1.3 Logic Programming

Logic programming is a programming paradigm that is based on the concept of formal logic. In this programming language to perform certain functionality a set of instructions to be written in logical form. The instructions should express some rules and facts about the functionality. Major logic programming language families are as follows:-

- Answer Set Programming(ASP)
- Datalog

1.1.4 Object Oriented Programming (OOP)

It is a programming paradigm where the functionalities are encapsulated in objects. It encapsulates the features of data and functions into a single unit. The objects are the basic run-time entities of a class where as a class encapsulates attributes and functions. Now a days many programming languages support object oriented programming for writing programs for all the domains. Examples of object oriented programming paradigms are as follows:- Smalltalk, Java, C++ (to some extent).

- Smalltalk

- Java
- C++ (to some extent)

1.1.5 Aspect Oriented Programming (AOP)

It is a programming paradigm which complements object oriented programming. It increases the modularity by encapsulating crosscutting concerns into a separate module called aspect. So the key unit of modularity in AOP is aspect which is same as a class in object oriented programming. It can't replace object oriented programming but complements it. It implements techniques like advices and point cuts to refactor the crosscutting concerns into aspects. Example of AOP paradigm is AspectJ

1.2 Problem statement

In our approach the different problem statements that we identified are as follows:-

1. Some programming tasks can't be cleanly encapsulated into objects, but the code is spread across the system called code scattering and some code is mixed with other code called code tangling.
2. The result is crosscutting code, the code that cuts across many different classes and methods.
3. Identifying these crosscutting concerns will automatically improve maintainability, evolution and reliability of the system.

1.3 Objectives

Our approach focuses on the research direction of aspect mining where crosscutting concerns are identified by using a combined approach. The various objectives of our approach are as follows:-

- **Objective-1:**

  Our first objective is to provide a solution for identifying maximum number of crosscutting concerns from the source code. Aspect mining technique in both static and dynamic analysis approaches are implemented for a better result.

- **Objective-2:**

  Our solution is obtained by taking case studies of different benchmark programs like SquareRootDisk and JHotDraw.

- **Objective-3:**

  Our solution is compared with other approaches and the detailed report is documented as a result.

- **Objective-4:**

  The crosscutting concerns should be managed in a proper way. They can be refactored into aspects like pointcuts and advices which is beyond our scope.

1.4 Motivation

Since the size and overall complexity of the software systems are increased day by day, encapsulation of modules into separate concerns is very important. Crosscutting concerns are the concerns which are scattered and tangled with other codes, and hence very difficult to maintain and debug the software system. Aspect mining is a reverse engineering process that tries to identify crosscutting concerns in an already developed software system. Identifying these crosscutting concerns will automatically improve the maintainability, understandability, evolution and reliability of the system. This will be helpful for the users who are new to aspect oriented programming. Different aspect mining techniques like fan-in analysis, formal concept analysis, clone detection technique, clustering techniques and graph based approaches are there. Some techniques are static in nature and others are

dynamic in nature. Combining any techniques appropriately can give better result as these techniques alone cannot identify all the crosscutting concerns in the system.

1.5 Challenges

Identifying crosscutting concerns in smaller programs is relatively easier. But when a larger program having 20K LOC or more than that it is very challenging to identify and manage the crosscutting concerns. The various solutions and techniques available are difficult to integrate with each other and they don't provide any common criteria for combining techniques. Concern mining are addressed at different levels of granularity and difficult to compare and combine solutions. Manually checking each module from the comment lines is very challenging factor. Availability of open source softwares and tools as well as case studies are rather scarce.

1.6 Organization of the Thesis

Our thesis includes the following chapters.

1. Chapter 1: We have discussed the the introduction to aspect oriented programming,various programming paradigms ,objectives and challenges of aspect mining.

2. Chapter 2: We have gone through different related papers and research work based on aspect mining.

3. Chapter 3: This chapter is based on our proposed work on fan-in analysis and dynamic analysis.

4. Chapter 4: We evaluated our approach by taking a case study of a benchmark program.

5. Chapter 5: In this chapter we focus on the conclusion and scope for future work.

Chapter 2

Aspect Mining:Background and Related Work

2.1 Aspect Mining

Aspect mining is a reverse engineering process that aims at identifying crosscutting concerns in an existing, non-aspect oriented code. Identifying these crosscutting concerns will automatically improve maintainability, reliability, understandability and evolution of software system. It increases our understanding of crosscutting code. Generally aspect mining techniques are not fully automatic. Most of the aspect mining techniques are semi-automatic. It requires human involvement for analyzing the seeds. Aspect mining tools generate all the candidate seeds from the source code but the human experts can choose only the confirmed seeds which are part of a crosscutting concern and reject all other seeds. Several aspect mining approaches are developed and they are categorized under Query-based approach or Generative approach.

2.1.1 Query-Based approach

In this approach the user provides a search pattern for which the source code locations are matched. Different tools based on this approach are developed. The first tool is the Aspect Browser tool which uses lexical pattern for matching the query code to the source code. An extension of this tool is called Aspect Mining Tool (AMT) that provides support for type based pattern. AMTEX is an extension of AMT that provides support for characterizing particular aspects. PRISM is another aspect mining tool which uses lexical and type based patterns. FEAT is another tool developed by Robillard and Murphy [1] which is an Eclipse plug-in that aims at finding and analyzing concerns in source code.

2.1.2 Generative Approach

These approaches try to capture the crosscutting concerns automatically. They use program analysis techniques to identify scattered and tangled code. A clone detection technique [2] is a generative approach which is based on matching of tokens at different locations in the source code. A fan-in analysis technique is a semi-automatic process that tries to identify scattered code by using a fan-in value for each method. Dynamic analysis approaches can automatically identify crosscutting concerns by examining the execution traces and obtain the recurring execution patterns and by applying formal concept analysis. Some clustering approaches are applied for identifying crosscutting concerns where related concerns are grouped together into the same cluster by using fan-in analysis [3] .

2.2 Related Work

Aspect mining is a relatively new research direction for identifying crosscutting concerns. Different aspect mining techniques have been proposed. There are different types of aspect mining techniques like Metrics analysis, Formal

concept analysis, Execution relations, Clone detection techniques, Natural language processing and Clustering approach.

Marin et al [4] have proposed an aspect mining technique based on the concept of metric calculation. They calculate the fan-in metric for each method of the software system. Their idea is to determining methods whose fan-in value is more than that of a given threshold value. Analyzing these high fan-in methods for determining crosscutting concerns in a software system.

Serban, G. and Moldovan, G. S. [5] have proposed a graph based aspect mining technique to determine similar methods by using graph theory. In their approach a graph is constructed between methods where nodes represent methods and edges represent relationship between them. They determine a connex component between methods and put them in a cluster. Then the cluster is analyzed to identify crosscutting concerns.

Czibula et al [6] have proposed a new hierarchical agglomerative clustering algorithm in aspect mining where a clustering approach is used to make clusters of methods by considering a distance matrix. The methods representing a crosscutting concern are grouped together in the same cluster.

Tonella and Ceccato [7] have also proposed an aspect mining technique based on dynamic analysis. In their approach execution traces are generated by executing the main functionalities for each use cases of a system. They use the concept of formal concept analysis where each computational unit is subject to a concept. The relationship between these computational units and the execution traces are considered for identifying crosscutting concern.

Shepherd et al [8] have proposed an aspect mining approach based on clone detection technique. They used program dependency graph to identify the codes

that are present at different locations in the source code. These duplicate codes at different places are further analyzed for discovering crosscutting code.

He and Bai [9] have proposed an aspect mining technique based on dynamic analysis. In their approach execution traces are generated by executing each use case of the system. From the execution trace clusters are formed and applying some association rules to identify crosscutting concerns.

Brue and Krinke [10] have proposed an aspect mining technique based on dynamic analysis. In their approach program traces are generated by running the program under some data pool. These traces are then investigated for recurring execution patterns based on different constraints.

2.3 Basic Terminology

The following different terms we can use in our approach.

2.3.1 Concern

A concern is an essential guideline and an important principle of software engineering development. It refers to some particular concept, the ability to identify,or some goal to combine and manipulate some parts of a system that are related. A concern is a set of information that affects the code of a computer program. Concerns are the design issues that the reflects in the stakeholders requirement. Examples of concerns are performance, security, specific functionality etc. As each concern is created by using modularity structure hence separation of concern would create a good modular design. When concerns are well separated, individual sections can be reused effectively.

A concerns intent is defined as the role of the concern where as a concerns extent is the concrete representation of that concern. We can clearly trace the program

from the requirements to implementation by separating of concerns in a program.

Core concerns are the primary concerns that represents the functionality of the system where as secondary concerns are the concerns that reflect nonfunctional parameters like security,reliability and QoS requirements.

2.3.2 Crosscutting Concern

Crosscutting concerns are the concerns which are spread across a number of program components and crosscuts other concern. These concerns cant be clearly separated from the rest of the system as they are mixed with other code. This results in a problem to implement the changes is not localized as it is spread. The symptoms of crosscutting concern lead to code scattering and code tangling. Code scattering refers to the code which is spread across the system where as code tangling refers to the code which is mixed with other code. Examples of crosscutting concerns are logging, persistence, exception handling, synchronization etc.

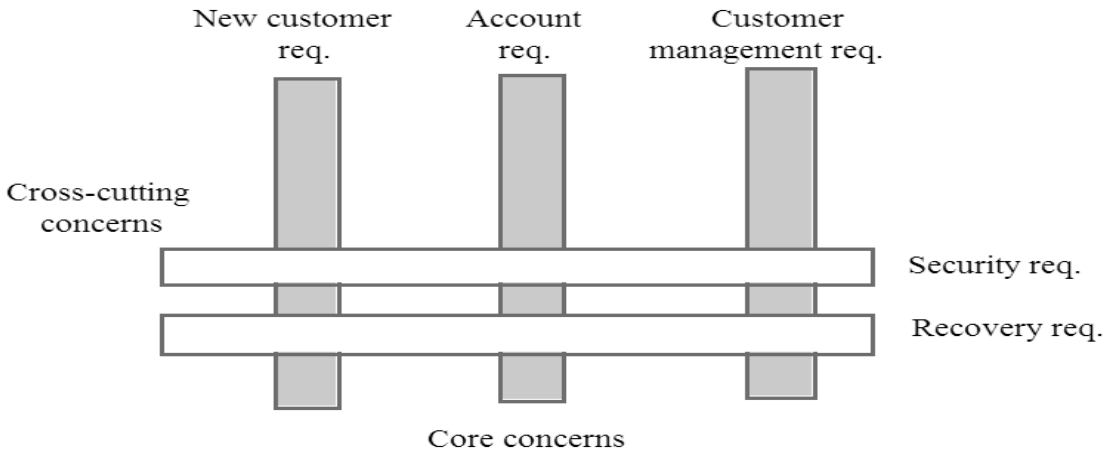

Figure 2.1: Crosscutting concern

2.3.3 Aspects

Aspects are tangled or scattered codes making it harder to understand and maintain. In AOP, crosscutting concerns are implemented as aspects instead of fusing them into core concerns. These aspects are additional units added to the program which can be reused as the dependency among these modules is less. Aspects can reduce code scattering and tangling and hence easy to understand the functionality of the module. In AOP crosscutting concerns can be defined by using advices and point cuts. Hence the combination of advices and point cuts is termed as aspect. For example we can add a logging aspect to our application by defining a point cut and giving a correct advice. An aspect weaver takes the aspects and core modules and composes the final system.

2.3.4 Aspect Oriented Programming (AOP)

Aspect oriented programming is a programming paradigm that complements object oriented programming by separating concerns into core modules. Aspect oriented programming captures crosscutting concerns into a new module called aspect by using advices and point cuts. AOP includes programing methods and tools that support the modularization of concern at the level of source code, while Aspect Oriented Software Development (AOSD) refers to the whole engineering process. In AOP, aspects can be implemented by using new language technique like point cuts and advices. AOP breaks the whole system into aspects as the basic module. AOP implements all crosscutting concerns by encapsulating them into a separate module.

Chapter 3

A Combined Approach for Identifying Crosscutting Concerns

3.1 Introduction

Crosscutting concerns are the concerns which crosscut other concerns. The symptoms of crosscutting concerns are code scattering and code tangling. Identifying these crosscutting concerns can automatically enhance maintainability,reliability, understandability and evolution of the software system. Different aspect mining techniques are proposed to identify these crosscutting concerns. Aspect mining techniques may be static or dynamic. In static aspect mining technique the source code is taken into consideration for identifying crosscutting concerns. From the source code the abstract syntax tree is generated and hence the static call graph is obtained from the syntax tree for calculating the number of callers for each method. But in dynamic analysis, instead of taking the source code, the program is run under certain inputs and execution traces are observed. From the execution traces different execution relations are obtained. The recurring execution patterns of these execution relations are generated and hence analyzed for crosscutting concerns.

3.2 A combined approach for identifying crosscutting concerns

Different aspect mining techniques are implemented to identify crosscutting concerns in already developed software system. In our approach we have combined two techniques and tried to identify crosscutting concerns and we identified more numbers of crosscutting concerns as compared to the other techniques. Our first approach is fan-in analysis which tries to identify all scattered code whereas our second approach is dynamic analysis approach which tries to identify all tangled code.

3.2.1 Fan-in Analysis

It is a semi-automatic aspect mining approach which tries to identify all the scattered code by computing the fan-in values of each and every method in the software system. This approach consists of the following three steps:-

1. **Fan-in Calculation**

   Calculation of fan-in metric for all the methods in the system.

2. **Methd Filtration**

   Filtering methods to obtain a smaller range of methods which are likely to implement crosscutting behavior. This step is called as Method filtration phase.

3. **Seed Analysis**

   Our last step is to analyze the remaining methods manually to identify those methods which are actual crosscutting concerns. This step is called seed analysis phase which can be done by considering the source code.

Calculation of fan-in metric

The callers of each method is calculated from the static call graph and the cardinality will give the required fan-in value of the method. Hence fan-in can be defined as a measure of number of distinct method bodies that call some other method. It is therefore the number of incoming calls of a method. For example P,Q,R and S are methods where P → Q(P calls Q),S→Q and R→S then in that case the fan-in value of Q is two because P is called from two different methods P and S whereas the fan-in value of S is one as it is called from a single method R. But the polymorphic methods affect the fan-in value of other methods. If a single abstract method is implemented in two different sub classes or super classes then these implementations are separate callers. Let us consider a class hierarchy as shown below.

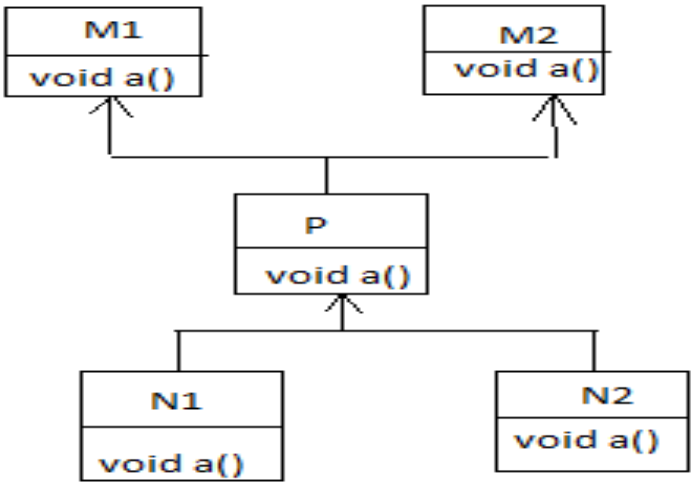

Figure 3.1: Class Hierarchy

In this class hierarchy, polymorphic method a() is called from various classes. Fan-in value of method a() in class M1 is not affected by the calls to method a() in

class M2 and vice-versa. Again fan-in value of method a() in class N1 is not affected by the calls to method a() in class N2 and vice-versa. But calls to method a() in class N1 can affect to classes B, M1, M2. Similarly calls to method a() in class M1 can affect to classes B, N1, N2. However for polymorphic methods the fan-in values are changed and are obtained as shown in table below.

Table 3.1: Fan-in values for the class hierarchy

Call site	M1.a()	M2.a()	N1.a()	N2.a()	P.a()
Call to method a() in M1	1	0	1	1	1
Call to method a() in M2	0	1	1	1	1
Call to method a() in N1	1	1	1	0	1
Call to method a() in N2	1	1	0	1	1
Call to method a() in P	1	1	1	1	1
Total fan-in value	4	4	4	4	5

Method Filtration

After calculating the fan-in values of all the methods, the next step is to obtain a small range of methods with a higher chance of implementing crosscutting concerns. In our approach we focus on high fan-in methods which are called many times and hence spread across the system. The high fan-in value can be obtained by putting some threshold value by observing the program. This value may be any absolute value or any relative percentage. The threshold value is not fixed and may vary from program to program. In our case study of SquareRootDisk and JHotdraw the value may be 3 and 10 respectively. Different values may be taken and experiments are done to fix the threshold value.

As all the high fan-in values above the threshold are not part of crosscutting concerns. We apply another filtration technique to remove all the getters and setters from the list of obtained methods. This can be based on the naming convention (method matching the *get** and *set** pattern) of the methods.

At last we apply a manual step to filter the utility methods like toString(),classes such as XMLDocumentUtils containg *util* in their name, collection manipulation methods etc. As this is a manual step, we use some heuristics to identify utility methods.

Seed Analysis

Our last step is to manually check the filtered methods to identify crosscutting concerns. As this is a manual step we follow some heuristics and guidelines, so that it will be easier to us for identifying crosscutting concerns.

- The callers should call the method at the beginning or at the end of its execution.
- All callers should be all refinements of a single abstract method.
- The calls at the call site should occur with similar names.For example when we use key or mouse events the call site should be like mouse handler or key handler.
- All calls occur in methods implementing a certain objective or function or role.
- The high fan-in method should represent a concern that is known to be a crosscutting concern.
- The methods concern or functionality should be conceptually different form the key functionality of the calling class.

The call sites must be checked regularly to capture the caller methods in a point cut mechanism and the high fan-in method into advices. For this seed analysis the user must have some domain knowledge about the program.

3.2.2 Pitfalls of Fan-in Analysis

- It addresses crosscutting concerns that are scattered across the system. It can't identify the tangled code that is mixed with other code.
- It is very difficult task to decide the threshold value.
- Taking a high fan-in value can discard some of the crosscutting concern. Taking a low fan-in value can increase the task of seed analysis and hence increase the percentage of false positives.
- It is a semi-automatic process .Some parts are done automatically with the help of fan-in tool. But the seed analysis part is done manually by the human engineers.
- It requires some domain expertise to describe the concerns identified.

3.2.3 Dynamic Analysis

Static aspect mining techniques can observe the potential behavior of the program where as dynamic analysis can reflect the run time behavior of a program. Due to the static nature of fan-in analysis technique we missed some of the tangled code. Code tangling refers to the code which is exist in several times in a software system and can't be encapsulated into a separate module. So here we implemented a dynamic programming approach which tries to identify the tangled code. In this approach we run the program and the execution traces are obtained. From these execution traces we identify recurring execution patterns and from these patterns we determine the patterns that describe certain role of the software system by applying certain execution constraints.

Program Trace and Execution Relation

A program trace is a sequence of method executions that are obtained under certain program run. In our approach we are tracing object oriented programs where methods describe the logically related functionality. We focus on sequence of method executions rather than their functionality. So we are interested only the entry and exit as well as the signature of the methods.

We can define two terms to identify crosscutting concerns.

1. *Outside aspects:* Outside aspects are the patterns where execution of one method is always followed by another method.

2. *Inside aspects:* Inside aspects are patterns where execution of one method is always called inside another method.

Again these two classes of aspects can be further categorized as before or after and first or last respectively. Hence we can define four terms such as outside before, outside after, inside first and inside last. All these four terms are defined in the algorithm below.

Algorithm 1 : (Dynamic Analysis)

1. Consider a program P,whose execution trace is T_p

2. Let the method signature (N_p) where only entry and exit of a method is considered.{represents entry and }represents exit of a method.Hence method B can be represented as B () {}

3. (a) Define *outside before* execution relation (S $\rightarrow$ T_p)

   a$\rightarrow$ b,a,b ϵ N_p where [(a,ext),(b,ent)] is a sublist of T_p

   (b) Define *outside after* execution relation (S $\leftarrow$ T_p)

   a$\leftarrow$ b,a,b ϵ N_p where [(a,ext),(b,ent)] is a sublist of T_p

(c) Define *inside first* execution relation (S $\vdash$ (T_p))

a $\vdash$ b, a,b ϵ N_p where [(b,ent),(a,ent)] is a sublist of T_p

(d) Define *inside last* execution relation (S $\dashv$ (T_p))

a $\dashv$ b, a,b ϵ N_p where [(a,ext),(b,ext)] is a sublist of T_p

4. Define $Uniformity$ constraint ($U^{\sim}$)

   An execution relation s=a $\sim$ b ϵ $S^{\sim}$, where $\sim$ ϵ ($\rightarrow$,$\leftarrow$, $\vdash$,$\dashv$)is uniform if $\forall$ c $\sim$ b ϵ $S^{\sim}$:a=c and a,b,c ϵ $N_p \cup \{\psi\}$where ψ is the empty method signature.

5. Define $Crosscutting$ constraint ($R^{\sim}$)

   An execution relation s=a $\sim$ b ϵ $U^{\sim}$, where $\sim$ ϵ ($\rightarrow$,$\leftarrow$, $\vdash$,$\dashv$)is crosscutting if $\exists$ s' =a $\sim$ c ϵ $U^{\sim}$:$b \neq c$ and a,b,c ϵ N_p

Execution Relation

Consider a program P,whose execution Trace is T_p and N_p defines the method signature. A program trace T_p is shown in figure 3.2.

In a program trace we only focus on sequence of method execution,its entry and exit points. The entry and exit points of a method can be represented by {(opening curly brace) and }(closing curly brace) respectively. For example signature of method B can be represented B () {}.

The crosscutting concerns can be reflected by either outside aspects or inside aspects. The outside aspects may be outside before or outside after and the inside aspects may be inside first or inside last. These four aspect candidates are defined in the algorithm and can be explained below.

1. **Outside before execution relation**

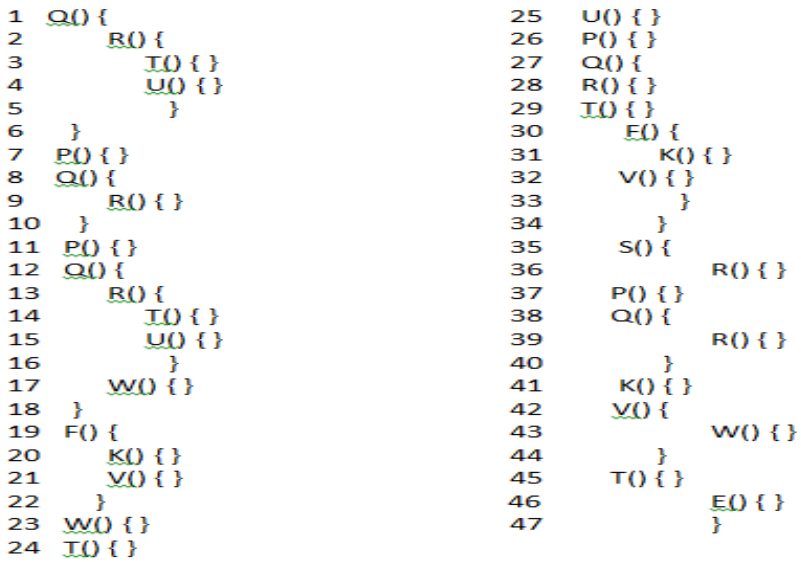

Figure 3.2: Example of program trace

The method which is executed before the execution of another method and the methods are executed sequentially is called as outside before execution relation. It is denoted as ($\rightarrow$). The relation a $\rightarrow$ b, a,b ϵ N_p is called outside before execution relation if [(a,ext),(b,ent)] is a sublist of T_p. The set of such execution relations can be denoted as S $\rightarrow(T_p)$.After executing method a,method b will be executed and the execution will be done sequentially.

2. **Outside after execution relation**

   The method which is executed after the execution of another method and the methods are executed sequentially is called as outside after execution relation. It is the reverse of outside before execution relation. It is denoted as ($\leftarrow$). The relation a $\leftarrow$ b, a,b ϵ N_p is called outside after execution relation if [(a,ext),(b,ent)] is a sublist of T_p. The set of such execution relations can be denoted as S $\leftarrow(T_p)$. Hence method b is executed only after the execution of method a and this will be executed sequentially.

3. **Inside first execution relation**

The method which is executed as the first method inside another method is called as inside first execution relation. It is denoted as ($\vdash$). The relation a $\vdash$ b, a,b ϵ N_p is called inside first execution relation if [(b,ent),(a,ent)] is a sublist of T_p. The set of such execution relations can be denoted as S $\vdash(T_p)$. Hence method a is only method executed as the first method inside method b .

4. **Inside last execution relation**

   The method which is executed as the last method inside another method is called as inside last execution relation. It is denoted as ($\dashv$). The relation a $\dashv$ b, a,b ϵ N_p is called inside last execution relation if [(a,ext),(b,ext)] is a sublist of T_p. The set of such execution relations can be denoted as S $\dashv(T_p)$. Hence method a is the only method executed as the last method inside method b .

The different execution relations that are obtained from the example trace as shown in figure 3.2 as follows.

- *The set of all outside before relation ($S \rightarrow (T_p)$)*

  Q () $\rightarrow$ P(),T () $\rightarrow$ U(),P () $\rightarrow$ Q(),R () $\rightarrow$ W(),Q () $\rightarrow$ F(),K () $\rightarrow$ V(),F () $\rightarrow$ W(),W () $\rightarrow$ T(),H () $\rightarrow$ P(),Q () $\rightarrow$ S(),R () $\rightarrow$ T(),T () $\rightarrow$ F(),R () $\rightarrow$ P(),Q () $\rightarrow$ K(),V () $\rightarrow$ T(),T () $\rightarrow$ E() .

- *The set of all outside after relation ($S \leftarrow (T_p)$)*

  P () $\leftarrow$ Q(),U () $\leftarrow$ T(),Q () $\leftarrow$ P(),W () $\leftarrow$ R(),F () $\leftarrow$ Q(),V () $\leftarrow$ K(),W () $\leftarrow$ F(),T () $\leftarrow$ W(),P () $\leftarrow$ U(),S () $\leftarrow$ Q(),T () $\leftarrow$ R(),F () $\leftarrow$ T(),A () $\leftarrow$ R(),K () $\leftarrow$ Q(),T () $\leftarrow$ V(),E () $\leftarrow$ T() .

- *The set of all inside first relation ($S \vdash(T_p)$)*

  R ()$\vdash$ Q (), T ()$\vdash$ R (), K ()$\vdash$ F (), R ()$\vdash$ S (), W ()$\vdash$ V ().

- *The set of all inside last relation ($S \dashv(T_p)$)*

  U ()$\dashv$ R (),R ()$\dashv$ Q (), W ()$\dashv$ Q (),V ()$\dashv$ F (), F ()$\dashv$ Q (),W ()$\dashv$ V (), E ()$\dashv$ S ().

Execution Relation Constraints

The execution relations obtained so far are further refined to obtain crosscutting concerns by applying our constraints to the execution relations. Recurring patterns in the execution relations are thus potential crosscutting concerns. Thus we have to find out recurring execution relations from our obtained execution relations by applying some constraints to it. The first constraint is the uniformity constraint where as the second one is a recurring pattern.

Before applying our constraints we have taken an assumption that there is no other method execution between another method i.e, methods shouldn't be nested. There is no other method execution between method entry and exit. This absence of nested method is called as empty method signature and can be denoted by ψ. Now this empty method signature is added to the program trace T_p. Hence our definitions can be changed from simple program trace to program trace including ϵ relation.

1. **Uniformity Constraint ($U^{\sim}$)**

   By considering the empty method signature we can define our *uniformity* constraint as follows:-

   An execution relation s=a $\sim$ b ϵ $S^{\sim}$,wher $\sim$ ϵ $\{\leftarrow, \rightarrow, \vdash, \dashv\}$is called uniform if $\forall$ c $\sim$ b ϵ $S^{\sim}$;a=c and a,b,c ϵ $N_p \cup \{\psi\}$holds.

   Uniformity defines the same pattern that is followed in the execution relation.For example a $\rightarrow$ b is uniform if for every execution of b is preceded by a that means a is always executed before b. If any other method rather than a is executed before b ,for example c $\rightarrow$ b means c is executed before b then this is not uniform as a and c are two methods executed before b.

   Similarly we can apply this uniformity constraint to our other execution relations like outside after, inside first and inside last. In outside after

execution relation the method which is executed after another method and the same pattern should be followed i.e, for every execution of the after method should be executed after the before method. For example b $\leftarrow$ a where b is executed after a. If b $\leftarrow$ c that means b is executed after c is followed then this is not uniform as b is executed after a and c. Always b should be executed after a then it would be uniform.

Consider an inside first execution relation a $\vdash$ b where method a is executed as the first method inside method b. This execution relation is called as uniform if for every execution of a should occur as the first method inside method b. If another relation c $\vdash$ b exist i.e, c is executed as the first method inside method b and this relation is not uniform as methods a and c are executed as the first method inside method b in different method executions.

Similarly consider an inside last execution relation a $\dashv$ b where method a is executed as the last method inside method b.This execution relation is called as uniform if for every execution of a should occur as the last method inside method b. If another relation c $\dashv$ b exist i.e, c is executed as the last method inside method b and this relation is not uniform as methods a and c are executed as the last method inside method b in different method executions.

After applying these uniformity constraints to our execution relations we can get the set of uniformity relation as follows:-

$U^{\rightarrow}$={Q () $\rightarrow$ S(),T () $\rightarrow$ E(),T () $\rightarrow$ U(),K() $\rightarrow$ V() }

$U^{\leftarrow}$={Q () $\leftarrow$ P(),V () $\leftarrow$ K() }

$U^{\vdash}$={R ()$\vdash$ Q (), R ()$\vdash$ S (), K ()$\vdash$ F () }

$U^{\dashv}$={E ()$\dashv$ S (),V ()$\dashv$ F () }

2. **Crosscutting constraint**

Uniformity constraint can reduce the potential crosscutting concern seeds and hence for better analysis of the remaining relations. After getting the uniformity relation we can apply the crosscutting constraint to it. For this analysis we have to remove the ψ -relation.

An execution relation s= a $\sim$ b ϵ $U^{\sim}$ is called crosscutting if $\exists$ s'=a $\sim$ c ϵ $U^{\sim}$ and b $\neq$ c where a,b,c ϵ N_p holds i.e, it occurs repeatedly in the program trace T_p. This constraint can be applied to all the execution relations and hence the aspect candidates can be determined. After applying the constraint to our execution relations we got the following patterns that represent the crosscutting concern seeds.

$R^{\rightarrow}$= {T() $\rightarrow$ U (),T () $\rightarrow$ E () }

$R^{\leftarrow}$= {ϕ }

$R^{\vdash}$= {R () $\vdash$ Q (),R () $\vdash$ S () }

$R^{\dashv}$= {ϕ }

From the above relation we found one aspect from the outside before execution relation. The method T() is called before the execution of methods U() and E (). Hence method T() is our first crosscutting concern obtained. There is no such pattern exist while considering outside after execution relation. In the inside first execution relation method R() is executed as the first method inside methods Q() and S(). Hence method R() represents another crosscutting concern. Again there is no such pattern exist in the inside last execution relation.

.

3.3 Summary

In this chapter, we have discussed about the two important techniques of aspect mining. The first technique is the fan-in analysis which is static in nature. It tries to identify all the scattered codes in a software system. The three steps of fan-in analysis such as fan-in metric calculation, method filtration and seed analysis are explained properly with suitable examples. This technique is a semi-automatic approach which needs some domain experts for identifying crosscutting concerns. The second technique is the dynamic analysis approach where different execution relations are obtained from a program trace. We described the four different execution relations such as outside before, outside after, inside first and inside last with an example. Then different constraints are applied on these execution relations to obtain crosscutting concerns. This analysis tries to identify all the tangled code of a software system. Hence combining these two techniques can identify both scattered as well as tangled code. Hence combining these two techniques can give better result.

Chapter 4

CASE STUDY

4.1 Introduction

In order to identify all the crosscutting codes i.e, code scattering and code tangling in an object oriented system we combined two approaches such as fan-in analysis and dynamic analysis. Our approach can be implemented to any object oriented program for our evaluation purpose we applied our approach to a bench mark programs. There are different bench mark programs are developed for aspect mining.They are SquareRootDisk, JHotDraw,JDraw, PetStore, JSokoApplet.JHotDraw and JDraw are two dimensional drawing tools to draw different figures. JSokoApplet is a gaming tool. For our case study analysis we have taken SquareRootDisk which is a small java program used for scanning files and folders. It is a good implementation of object oriented design pattern. Our next goal is to apply our approach to some other bench mark programs.

4.2 SquareRootDisk

SquareRootDisk is a java application program which is used to scan files and folders. It is a good implementation of object oriented design pattern. It has different versions but we used version 1.4.2 for our analysis. It is small in size. It contains

624 numbers of LOC, one package, 7 classes and 47 methods. Our analysis focuses on these 47 methods.

4.2.1 Fan-in analysis Result

We applied our first technique to SquareRootDisk,a benchmark program to obtain methods having fan-in value more than that of the threshold value. In order to calculate fan-in metric and for method filtration we used a tool called FINT. It is an eclipse plugin and requires 3.1 to 3.3 versions of eclipse. It is an open source software which can be easily downloaded from Internet. The installation procedure is very easy and to simply put jar version of the tool to the eclipse plugin file and restart the IDE. It consists of three views such as fan-in analysis view, Seeds view and Redirection-layer view.

In order to calculate the fan-in metric the tool first imports the source code into the package explorer view of eclipse. Name the project as SquareRootDisk. The tool first bulids the abstract syntax tree of the project and then creates its static call graph. From the static call graph the callers for each callee is obtained and the cardinality gives the required fan-in value. After calculating the fan-in values of all the methods the result can be shown in the fan-in analysis view. The obtained result is shown in figure 4.1 below.

Our second step is method filtration phase where we have to apply some restrictions to the methods. We filter out all the getters and setters as well as library and utility methods. After applying this step we found only 24 methods out of 47 methods. We filtered around 50 percentage of the methods. Then the important criteria is the threshold value. We put a threshold value of 4 and obtained only one method. Hence we put the threshold value 2 and found six methods. So around another 60 percentage of the methods are filtered out. Hence for our case study the

```
SaveJDialog.SaveJDialog(Qjava.awt.Frame;Z) : 2
 Callers:
  > GUISquareRootDisk.saveMenuItemActionPerformed(Qjava.awt.event.ActionEvent;)
  > SaveJDialog$1.run()
HelpJDialog.initComponents() : 1
 Callers:
  > HelpJDialog.HelpJDialog(Qjava.awt.Frame;Z)
SelectJDialog.SelectJDialog(Qjava.awt.Frame;Z) : 3
 Callers:
  > SelectJDialog$1.run()
  > GUISquareRootDisk.analyzeMenuItemActionPerformed(Qjava.awt.event.ActionEvent;)
  > GUISquareRootDisk.analyzeMenuItem2ActionPerformed(Qjava.awt.event.ActionEvent;)
GUISquareRootDisk.saveMenuItemActionPerformed(Qjava.awt.event.ActionEvent;) : 1
 Callers:
  > GUISquareRootDisk$2.actionPerformed(Qjava.awt.event.ActionEvent;)
SquareRootDisk.SquareRootDisk() : 1
 Callers:
  > SaveJDialog.jFileChooser1ActionPerformed(Qjava.awt.event.ActionEvent;)
SharedData.SharedData() : 4
 Callers:
  > SquareRootDisk.SquareRootDisk(QFile;Z)
  > GUISquareRootDisk.analyzeMenuItemActionPerformed(Qjava.awt.event.ActionEvent;)
  > GUISquareRootDisk.analyzeMenuItem2ActionPerformed(Qjava.awt.event.ActionEvent;)
  > GUISquareRootDisk.initComponents()
SquareRootDisk.processFile(QFile;I) : 2
 Callers:
  > SquareRootDisk.run()
  > SquareRootDisk.getDirectorySize(QFile;I)
GUISquareRootDisk.aboutMenuItemActionPerformed(Qjava.awt.event.ActionEvent;) : 1
 Callers:
  > GUISquareRootDisk$7.actionPerformed(Qjava.awt.event.ActionEvent;)
GUISquareRootDisk.initComponents() : 1
 Callers:
  > GUISquareRootDisk.GUISquareRootDisk()
```

Figure 4.1: Fan-in values

threshold value is two. The six high fan-in methods and their callers can be shown in figure 4.2 as below.

These six high fan-in methods are our are our crosscutting concern seeds. We can analyze these seeds by using the point cuts rules as defined in the chapter-3.

The two high fan-in methods $SelectJDialog()$ ans $sharedData()$ are not crosscutting concerns as their definition is a not a concern. Their call sites are visited and call position is tracked. But we found no point cut rule is implemented for these two methods.

Next we focused on the high fan-in values and obtained four methods as

```
No. methods: 6

SaveJDialog.SaveJDialog(Qjava.awt.Frame;Z) : 2
 Callers:
  > GUISquareRootDisk.saveMenuItemActionPerformed(Qjava.awt.event.ActionEvent;)
  > SaveJDialog$1.run()
SelectJDialog.SelectJDialog(Qjava.awt.Frame;Z) : 3
 Callers:
  > SelectJDialog$1.run()
  > GUISquareRootDisk.analyzeMenuItemActionPerformed(Qjava.awt.event.ActionEvent;)
  > GUISquareRootDisk.analyzeMenuItem2ActionPerformed(Qjava.awt.event.ActionEvent;)
SharedData.SharedData() : 4
 Callers:
  > SquareRootDisk.SquareRootDisk(QFile;Z)
  > GUISquareRootDisk.analyzeMenuItemActionPerformed(Qjava.awt.event.ActionEvent;)
  > GUISquareRootDisk.analyzeMenuItem2ActionPerformed(Qjava.awt.event.ActionEvent;)
  > GUISquareRootDisk.initComponents()
SquareRootDisk.processFile(QFile;I) : 2
 Callers:
  > SquareRootDisk.run()
  > SquareRootDisk.getDirectorySize(QFile;I)
HelpJDialog.HelpJDialog(Qjava.awt.Frame;Z) : 2
 Callers:
  > GUISquareRootDisk.helpMenuItemActionPerformed(Qjava.awt.event.ActionEvent;)
  > HelpJDialog$1.run()
SquareRootDisk.SquareRootDisk(QFile;Z) : 2
 Callers:
  > srd.main([QString;)
  > SelectJDialog.jFileChooser1ActionPerformed(Qjava.awt.event.ActionEvent;)
```

Figure 4.2: Methods after filtration

$HelpJDialog()2$,$SaveJDialog()2$, $processfile()2$, $SquareRootDisk()2$

Table 4.1: High fan-in values

Callee	**Caller**
HelpJDialog()	helpMenuItemActionPerfomed(), run()
SaveJDialog()	saveMenuItemActionPerformed(), run()
SelectJDialog()	analyzeMenuItem2ActionPerformed(), analyzeMenuItemActionPerformed(), run()
SharedData()	analyzeMenuItem2ActionPerformed(), analyzeMenuItemActionPerformed(),initComponents(),SquareRootDisk()
processfile()	getDirectorySize() ,run()
SquareRootDisk()	jFileChooser1ActionPerformed(), main()

Out of these four methods, three methods are constructors and they invoke automatically. They are not part of a concern and their call position is not at the beginning or at the end and they are not the refinements of a single abstract method. Hence they are not a part of crosscutting concern.

Our last method $processfile()$ is a concern which is used to scan the files and folders. It is called from two different methods $run()$ and $getDirectorySize()$ which are again representing two concerns. So it satisfies the point cut guidelines. Again we checked its functionality which is different from the callers functionality. Hence one crosscutting concern we found by fan-in analysis is processfile().

Table 4.3: Fan-in Analysis Result

Crosscutting concern	**Fan-in value**	**Methods**
Scan File	2	processfile()

4.2.2 Dynamic Analysis Result

We applied our second approach,Dynamic analysis to the same benchmark program SquareRootDisk. We run the project and tried to cover all the functionalities of the project. We used a dynamic call graph tool called Call Graph Viewer which is an eclipse plugin. It can be easily downloaded from the eclipse market place and installed. We run the program and it simply tracks its method executions. We only considered method execution sequence rather than its functionality. We run our project under different functionalities and obtained the following program trace as shown in figure 4.3 .

From the above program trace in figure 4.3 we obtained the following execution relations.

```
GUISquareRootDisk(){
        initComponents(){
                sharedData(){ }
                        }actionPerformed(){
                analyzeMenuItem2ActionPerformed(){ }
                analyzeMenuItemActionPerformed(){ }
                saveMenuActionPerformed(){ }
                helpMenuActionPerformed(){ }
                exitMenuActionPerformed(){ }
                        } }
HelpJDialog(){
        initComponents(){ }
        actionPerformed(){
                analyzeMenuItem2ActionPerformed(){ }
                analyzeMenuItemActionPerformed(){ }
                saveMenuActionPerformed(){ }
                helpMenuActionPerformed(){ }
                exitMenuActionPerformed(){ }
}
keyPressed(){ }
        JButtonActionPerformed(){ }
        }
SelectJDialog(){
        initComponets(){ }
        JFileChooser1ActionPerformed()
                {
                        SquareRootDisk(){ }
                } }
SaveJDialog(){
initComponents(){ }
JFileChooser1ActionPerformed(){
```

Figure 4.3: Program Trace

- **Outside before execution relation**

  GUISquareRootDisk() → HelpJDialog(),initComponents() → actionPerformed(),HelpJDialog () → SelectJDialog(),SelectJDialog() → SaveJDialog(),SaveJDialog() → SquareRootDisk(),sharedData() → run(),run() → getdirectory(),getDirectorySize() → printlist(),printlist() → initialDepth(),actionPerformed() → keyPressed(),keyPressed() → actionPerformed(),initComponents() → JFileChooser1ActionPerformed(),run() → processfile(),getDirectorySize() → processfile().

- **Outside after execution relation**

  Outside after relations can be determined by reversing the outside before

execution relation.

HelpJDialog() ← GUISquareRootDisk(),actionPerformed() ← initcomponets(),SelectJDialog () ← HelpJDialog(),SaveJDialog()← SelectJDialog(),SquareRootDisk() ← SaveJDialog(),run() ← sharedData(),getdirectory() ← run(),printlist() ← getDirectorySize(),initialDepth() ← printlist(),keyPressed() ← actionPerformed(),JFileChooser1ActionPerformed() ← initComponents(),processfile() ← run(),processfile() ← getDirectorySize().

- **Inside first execution relation**

initComponents() ⊢ GUISquareRootDisk(),initComponents() ⊢ SelectJDialog(),initComponents() ⊢ SaveJDialog(), initComponents() ⊢ HelpJDialog(),sharedData() ⊢ SquareRootDisk().

- **Inside last execution relation**

actionPerfomed() ⊣ GUISquareRootDisk(),JButtonactionPerfomed() ⊣ HelpJDialog(),JFileChooser1ActionPerfomed() ⊣ SelectJDialog(),JFileChooser1ActionPerfomed() ⊣ SaveJDialog(),initialDepth() ⊣ SquareRootDisk().

After getting all these execution relations we apply uniformity and crosscutting constraints to it and we got the crosscutting concerns as below in table 4.5.

Dynamic analysis identifies three methods that are part a crosscutting concern. These methods are not identified by fan-in analysis as their fan-in values are below the threshold and they are tangled code. It identifies a concern which was earlier identified by fan-in analysis as we have considered a small program. For large programs it can't identify the scattered code. The initComponents is the first method executed in four different methods such

Table 4.5: Dynamic Analysis Result

Crosscutting concern	**Execution Relation**	**Methods**
initComponents()	Inside first	GUISquareRootDisk(), SaveJDialog(), ,SelectJDialog(),HelpJDialog()
processfile()	Outside after	run(),getDirectorySize()
JFileChooser1ActionPerformed()	Inside last	SelectJDialog(),SaveJDialog()

as GUISquareRootDisk,SaveJDialog,SelectjDialog,HelpJDialog.Similarly processfile is a method which is executed after methods run and getDirectorySize. Atlast we found another method JFileChooser1ActionPerformed which is executed as the last method inside methods SelectJDialog and SaveJDailog.

After combing the two techniques we got the result as shown in the table.

Table 4.7: Combined Result

Crosscutting concern	**fan-in value**	**Type of code**	**Technique**
initComponents()	1	Code Tangling	Dynamic
processfile()	2	Code Scattering	Fan-in
JFileChooser1ActionPerformed()	1	Code tangling	Dynamic

4.3 Summary

In this chapter, we have implemented our combined approach to a bench mark program *SquareRootDisk* and obtained the crosscutting concerns. Our first technique,fan-in approach identifies only one crosscutting concern which is used to scan the file. The method processfile() is spread across two classes having different functionalities. Our second approach, dynamic analysis technique identifies three crosscutting concerns out of which two crosscutting concerns are not identified by fan-in analysis. Hence combining techniques can give better result.

Chapter 5

Conclusion

Crosscutting concerns are the concerns which are spread across the system as well as mixed with other code making it difficult to maintain,debug and understand the code. Hence identifying these crosscutting concerns will automatically improve the code modularity by reducing the tangled code as well as scattered code. We implemented a combined approach for identifying these crosscutting concerns. The first technique is the fan-in analysis approach which tries to identify all the scattered code. The complementary of this approach is a dynamic analysis which tries to identify all the tangled code. Hence combining the techniques can give better result. We implemented our approach to a benchmark program "SquareRootDisk" to identify the crosscutting concerns. It contains 7 classes and 47 methods. We analyzed these methods and classes for our result. Next we will implement our approach to other benchmark programs having more number of methods and classes.

Scope for Further Research

- We can apply our technique to other benchmark programs like PETStore and JHotDraw for better result.
- The obtained crosscutting concerns can be re-factored into aspects by using

AspectJ Language.

- To use program slicing techniques for identifying crosscutting concerns.
- Finding the complexities and solution to reduce these complexities occur in our approach.
- We will make it automatic by designing a tool.

Bibliography

[1] Martin P Robillard and Gail C Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th international conference on Software engineering*, pages 406–416. ACM, 2002.

[2] B Magiel, Av Deursen, Rv Engelen, and T Tourwe. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proc. Intl. Conf. Software Maintenance (ICSM). IEEE Computer Society*, 2004.

[3] Danfeng Zhang, Yao Guo, and Xiangqun Chen. Automated aspect recommendation through clustering-based fan-in analysis. In *23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. ASE 2008*, pages 278–287. IEEE, 2008.

[4] Marius Marin, Arie Van Deursen, and Leon Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(1):3, 2007.

[5] Gabriela Serban and GRIGORETA SOFIA Moldovan. A graph algorithm for identification of crosscutting concerns. *Studia Universitatis Babes-Bolyai, Informatica, LI (2)*, pages 53–60, 2006.

[6] Istvan Gergely Czibula, Gabriela Czibula, and Grigoreta Sofia Cojocar. Hierarchical clustering for identifying crosscutting concerns in object oriented software systems. *INFOCOMP Journal of Computer Science*, 8(3):21–28, 2009.

[7] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the 11th Working Conference on Reverse Engineering, 2004*, pages 112–121. IEEE, 2004.

[8] David Shepherd, Emily Gibson, and Lori L Pollock. Design and evaluation of an automated aspect mining tool. In *Software Engineering Research and Practice*, pages 601–607. Citeseer, 2004.

[9] Lili He and Hongtao Bai. Aspect mining using clustering and association rule method. *International Journal of Computer Science and Network Security*, 6(2A):247–251, 2006.

[10] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Proceedings of the 19th International Conference on Automated Software Engineering, 2004*, pages 310–315. IEEE, 2004.

Dissemination

Conference

1. Shantanu Kumar Biswal,Ramesh Kumar Mohapatra,A Combined Approach for Identifying Crosscutting concerns in an Object Oriented System. *International Conference on Recent Innovations Science, Engineering and Management* (ICRISEM -2015).