June 2014

# Identifying Extract Class and Extract Method Refactoring Opportunities Through Analysis of Variable Declarations and Uses

Mehmet Kaya
*Syracuse University*

# Abstract

For small software systems, with perhaps a few thousand lines of code, software structure is largely an esthetic issue. When software systems grow large, including perhaps a million or more lines of source code, their structures become much more important.

Developing a large system requires teams of developers working in concert to provide a finished product in a reasonable amount of time. That means that many people will read each component to use, test or modify towards accomplishing new features.

In the software development life cycle, the maintenance phase is a dominant stage that impacts production cost of the system dramatically. This is mainly because, for a successful system, the maintenance phase lasts until the system's retirement and includes crucial operations such as enhancing performance, fixing newly discovered bugs and adopting/expending the software to meet new user requirements. Moreover, a software component may be modified or fixed by someone who is not the original author of that component. In this case, all the operations conducted during maintenance or initial development may lead to insertion of code into a unit that may be unrelated to the original design concept of that unit.

As software systems become large and complex they grow too long to read and understand completely by a single person. After their initial implementations, maintenance operations tend to make the system even less maintainable, increasing the time and effort needed for future maintenance.

In this research, we are interested in finding ways to successfully detect code defects and propose solutions to increase the overall maintainability of software systems that are larger than any one person can completely comprehend from its code alone. This process of refactoring software impacts the total production cost of the system positively by improving the quality of software code such as its comprehensibility and readability.

To reduce the total development cost for a system, we suggest three main re-factorings. These novel forms of refactoring techniques aim to eliminate code defects such as large classes and long methods. The main goal of these re-factorings is to create smaller and cohesive software units with clear intentions to improve the maintainability of software.

We provide analysis and visualization tools to help a user identify candidate code fragments to be extracted as separate unites. With these automation tools, developers do not have to manually inspect a foreign code base to detect possible refactoring opportunities. Through the visual representations we provide, one can observe all suggested re-factorings effectively on large scale software systems and decide whether a particular refactoring needs to be applied. To show the effectiveness of our techniques, we also provide some experiments conducted using these tools and techniques both on our own project's source code and other open-source projects.

# Identifying Extract Class and Extract Method Refactoring

# Opportunities Through Analysis of Variable Declarations and Uses

by

Mehmet Kaya

B.S., Suleyman Demirel University, 2007

M.S., Syracuse University, 2010

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering

in the Graduate School of Syracuse University

Syracuse University
June 2014

# Contents

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1

## Introduction

In this research, we investigate some common software code defects and propose practical solutions to improve design and quality of source code. The main focus of this research is on the use of variable references to detect code defects and propose solutions for them. Using variable references in code, we seek cohesive code fragments that can be separated into new units in a large software system such as classes or methods. The motivation behind this is to improve quality and design of the software to make it more readable, maintainable and less complex.

Techniques and solutions that we provide are supported by analysis and visualization tools. Since reading and trying to understand source code for refactoring would not be efficient without such tools, they play an important role in software refactoring process for especially large professional software. While analysis tools make the identification of code defects in the software refactoring process easier; by using visualization tools, one can observe the suggested refactoring more effectively.

## 1.1 Problem Presentation

The problem addressed in this research is caused by software units, such as methods and classes, with complex and/or undesirably long implementations. Software units with such poor quality reduce maintainability of a system and, in effect, impact production cost. Hence, solving such quality or design issues reduces the time, effort and cost of the maintenance phase which is often a dominant part of the software development life cycle.

### *1.1.1 Software Maintenance*

Software development is a process that proceeds in stages. There are several widely adopted software life cycle models (SLCM) all of which include phases for pre-development, development, and post-development stages. Although the IEEE standard does not dictate or define a particular SLCM to be used, it requires one to adopt a model that defines a specific approach to producing software with a possibly time-ordered set of activities [49].

After delivery or release of a software product, software development enters what is known as the maintenance phase where the focus is "to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment" [1]. Therefore maintenance is the primary process in SLCMs that comes after all other activities and continues until the project's retirement. Software maintenance operations can be grouped into four different maintenance categories: adaptive maintenance, corrective maintenance, emergency maintenance, and perfective maintenance [1].

During adaptive maintenance, one aims to adapt the software program to a changing or changed environment. For instance, changes to make the software support new file formats or to make the program operational on new platforms can be considered as an adaptive maintenance activity.

Sometimes new user requirements may surface causing addition of new features in order to meet those requirements. Alterations to support these additional features can also be considered as adaptive maintenance.

Corrective maintenance is performed in case a fault is discovered in the behavior of the program after its delivery. The scope of this maintenance operation usually consists of fixing defects that cause erroneous behavior in the system. Sometimes, when supporting multiple threads, there can be serious race conditions that are not discovered during implementation and halt operations of the system. Such changes to overcome these faulty behaviors are considered as corrective maintenance activities.

Emergency maintenance involves unscheduled operations to modify a software system to keep it operational by handling unexpected errors.

The last category of software maintenance operations, perfective maintenance, aims to improve the performance and maintainability of software systems. Changing software code to improve its quality, such as refactoring activities to reduce the size or complexity of a method, is a maintenance activity that falls under this category.

The maintenance phase is a crucial part of the software development process as the operations performed in this phase are important to meet the user requirements by producing expected correct results within a reasonable period of time. According to IEEE standards [50], the maintenance phase also "supports the software product from its inception through migration to new environments, to its retirement and ends when the software product is finally retired".

Maintenance operations require a lot of time and effort, and therefore, increase the cost of production dramatically, especially for large systems. Some software developers, for example,

spend two to three times more effort maintaining and enhancing software than they spend creating new software [2]. Throughout the maintenance phase the changes made to enhance, correct and/or to adapt the software code may degrade its quality and make it increasingly difficult to maintain. Therefore, the maintenance phase is a costly process comprising 60-75% of the overall development effort [3, 72, 51]; and the maintenance phase usually lasts for 10 to 20 years for successful systems [3] or until their retirement [50].

The main reason behind a costly maintenance process lies in the following difficulties: understanding the software and explaining its intention to others for software reuse, correcting defects, improving performance of software code and extending its functionalities for maintenance. These difficulties are attributed to software complexity in [4] and they are also closely related to other software quality aspects such as readability and reusability. Therefore, when software systems are improved in terms of design and quality, maintenance cost will be reduced along with the overall cost of development.

For small software systems, with perhaps a few thousand lines of code, software structure is largely an esthetic issue. Maintenance phase for larger systems with a million or more of lines of code, will become a lot more costly due to potentially poor structure and quality, as each component in the system will be read and used by many other developers - those whose code is dependent on some other components will have to read and understand components written by others. Therefore, if these components have been designed and implemented in a way that they can be understood and reused in other systems easily, maintenance cost, or time and effort spent, will be reduced significantly. Thus, developers may need to redesign their code regularly during its maintenance phase and sometimes during development, to increase the quality of code by creating less complex, more readable and cohesive components.

Due to the significance of the maintenance phase within the entire software development life cycle and its impact on the total production cost, researchers have spent a great deal of effort to figure out possible ways of improving the maintainability of existing code. This is accomplished by improving internal design and quality of the code, while keeping the external behavior intact. This process is usually known as refactoring and is still an active and important field of research in software engineering discipline.

### 1.1.2 Software Refactoring

Refactoring techniques have been suggested to improve several different quality aspects of software programs such as cohesion, complexity and readability. In this research, the main focus is on refactoring through the analysis of variable declarations and uses. Variable references or techniques based on this can be used to effectively find possible refactoring opportunities that improve design and quality of the code by eliminating code defects.

Tool support plays a fundamental and significant role in effective and practical execution of refactoring approaches. Often, implementation of such tools not only requires a detailed code analysis, it necessitates a visualization feature as well. Therefore, rather than only a theoretical approach, it will be an important contribution to introduce a software refactoring technique supported by such analysis and visualization tools. Suggesting a refactoring technique based mainly on variable references is a practical approach to this critical problem in terms of the convenience of its use for both analysis and visualization.

Implementation of the refactoring techniques introduced here is straightforward and can be applied to real software effectively without any major limitations on programming languages. According to the experimental results presented in the next chapters, refactoring approaches

through variable references can effectively resolve some software code defects related to the design and quality of software programs.

Techniques and tools presented in this research address code defects including Large Class, Long Method, and Long Parameter List code defects. We have chosen these amongst other code defects presented in [5] due to the fact that they are closely related to each other and are considered to be important.

### 1.1.3 Common Code Defects

There are certain groups of code defects that reduce the quality of software code in a certain way usually related to one or multiple quality aspects at the same time. In this research, code defects that we focus on are named Large Class, Long Method, and Long Parameter List in [5].

In an object oriented program, the Large Class code defect refers to classes with more than one abstraction. If a class is doing more than one main activity, the class should be split into separate classes extracting each distinct task from the large class as a separate unit. Long Method code defects on the other hand, can be found in many programs regardless of their implementation language. We think that methods should be short enough to be easily read and simple enough to be easily understood and tested. Long Methods, just like Large Classes, should be decomposed into smaller and less complex methods to improve understandability and readability of code. Long parameter list for a method is another code defect that we address in this research. While extracting methods from existing long methods, we should also be aggressive in creating methods with short parameter lists, as the long parameter list is another code defect that usually increases the complexity of methods [5].

Proposing some new tools and techniques, the goal of this research is to detect such code defects, to identify code refactoring opportunities for eliminating them, and to visualize possible code refactoring opportunities.

## 1.2 Organization of This Dissertation

After this short introduction, Chapter 2 discusses technical background materials such as code refactoring and program slicing followed by a discussion of techniques and tools that have been used in this research as well as a discussion of what has already been done in this area. Chapter 3 discusses the first contribution of this research – an effective method for class extraction. This contribution addresses the first code defect we listed: Large Class code defect. The main motivation of the work presented in Chapter 3 is to show that variable references can be used as a good indicator of a range of refactoring opportunities. In Chapter 4, a novel technique for method refactoring is presented. Chapter 4 also includes some discussion related to *Placement Tree* structure and *Tree-map Visualization* that we have adopted. Following this, in chapter 5, another method refactoring technique is presented. The focus of this chapter is to extract methods to eliminate Long Method code defects while keeping the argument list of the extracted methods within a desired length. Finally in Chapter 6, we discuss future work and summarize the research and its contributions.

# Chapter 2

## Background and Literature Survey

This section describes background information regarding the concept of refactoring, code defects to be resolved, and other techniques/tools that are used in this research. After the discussion of these concepts and techniques, other studies related to the problems that we tackle are explored.

### 2.1 Software Refactoring

Refactoring is defined by Fowler et al. as "the exact reverse of the normal notion of software decay" [5]. Software code ages and evolves over time and quality of the code degrades due to the changes that code is exposed to. Restructuring techniques help the source code regain its quality after maintenance operations or sometimes even before the first release of the product during the initial implementation.

Software programs may be modified by people who are not the original authors. This means that, changes are sometimes made by people who do not thoroughly understand the original design concept and this almost always causes the quality or design of the programs to degrade. Over time, after many such changes, software evolves in a way that no one understands the code any more, not even the original authors of the program [6]. Software that has evolved in this way, at some point, becomes very costly to modify or improve and fixing bugs in such code, likewise turns into a very challenging task due to the complexity of the code.

The key for an effective maintenance stage in large scale software development is the quality of the software design and implementation. As explained earlier, if code is not readable, well-structured, well-documented, and relatively simple, it should be restructured in a way that its quality and design are improved to increase maintainability while its external behavior is preserved by using only behavior preserving refactoring of the code.

Refactoring can be a process as simple as renaming a software attribute, or it can be a more complicated process such as extracting new classes or functions from existing code. Comprehensibility of code, for example, can be improved by providing less complex and more readable code initially and by keeping code simple and readable throughout its life time via refactoring activities. Software refactoring techniques, when applied to software regularly during implementation and maintenance, also help to reduce the size of individual software units such as methods and classes to improve readability.

## 2.2 Common Code Defects

Code defects are listed and explained in detail in [5] by Fowler et al. Yet, there is no a priori approach to solve these code defects automatically or semi-automatically as an aid to developers. Fowler states that refactoring is based on human intuition and it is believed that the developer is the last authority to decide where to apply the refactoring [46]. Especially in large systems, effective refactoring necessitates tools and techniques to find code defects and solve them rather than relying on human intuition. From the code defects described by Fowler et al., in this research we propose solutions and tool support for three of them as we think they are the most common ones among others and they are related to each other.

### *2.2.1 Large Class Code Defect*

Classes are the key units of object oriented programming. Therefore, developers aim to design classes with high quality so that they can be reused, maintained, and tested easily. To reduce maintenance cost, these key units are expected to be simple, understandable, and readable as well. Cohesion as a quality metric measures "the degree to which the elements of a module belong together" [8] and high cohesion impacts understanding, reuse, and maintenance of code positively [7]. To meet this property for a class, developers try to create classes with a clear task. That is, every class must have a well defined task or capture one and only one key abstraction within the larger operation of the whole system [78].

As software is exposed to changes during the maintenance phase, classes may be altered in a way that they include some irrelevant components to their intended operations and therefore, they no longer preserve their original simplicity and reusability properties by including more than one abstraction.  A class may initially be designed having more than one abstraction as well, because

of complex requirements, time constraints, or other reasons. Even though the effect cannot be observed directly, most of the time code defects or design anomalies cause failure of the system indirectly [76]. Therefore, such classes should be re-factored to improve their internal design and thus maintainability. Such refactoring is known as Extract Class Refactoring where each distinct abstraction or task in a less cohesive class is extracted to generate a new cohesive and possibly reusable class.

### 2.2.2 Long Method Code Defect

Long methods are usually the source of many other code defects in large systems [1]. In contrast to longer methods, smaller methods are easier to read, comprehend, and maintain. As stated earlier, software development is a team work and code written by one developer in a large scale system will very often be used, read and/or modified by other people. Rather than trying to accomplish many subtasks together, methods should be shorter with one clear intention for better comprehensibility, readability and simplicity. When this is achieved, maintenance of such methods will be a lot less costly.

As a result of maintenance operations such as bug fixing and performance improvements, quality of methods in large scale systems tends to degrade over time with increased length and complexity. This is mainly due to irrelevant code insertions into the original method or implementing more than one distinct functionality in the same method. Maintenance of such methods will gradually become more and more difficult if proper precautions are not taken in time.

Long Methods with higher complexity and length should be restructured to produce smaller, more cohesive and comprehensible methods. This decomposition process is known as Extract

Method Refactoring where pre-identified code fragments are extracted from an original method as new separate methods and they are replaced with appropriate method calls in the original method. Extract Method Refactoring has two major parts: identification of code fragments and their extraction for refactoring. While extraction of code fragments is trivial and supported by IDEs such as Eclipse [81] and MS Visual Studio [82], identification of code fragments for method refactoring is less straightforward and sometimes a very challenging task. For this reason, we devote our attention to the identification part of the refactoring in this research.

### 2.2.3 Long Parameter List Code Defect

In most programming languages, there are usually three fundamental ways of providing data needed by a method. Data can be passed as arguments to the methods, data can be made member data of an enclosing class of the method, or it can be global and therefore accessible by all the methods at an appropriate access level. It is widely accepted that using global data is not a good programming principle and should be avoided [5]. Therefore, there are only two practical ways of providing data to the methods: passing in as an argument or creating as member data. In this section, we are concerned with data passed as an argument.

Passing all data needed as arguments may result in methods with long parameter lists. Long Parameter List code defect can impact the quality of software programs dramatically, as such methods are typically difficult to understand and test [5]. As a result, the maintenance phase requires more time and effort increasing overall production cost.

Therefore, it is very important to generate methods with fewer arguments during refactoring long methods. In other words, while applying extract method refactoring, one should try not to introduce a new code defect, Long Parameter List. Therefore, in this part of the research, a novel

technique is introduced as extract method refactoring technique for an effective improvement of quality and design by providing a control over the number of parameters that new extracted methods would take.

## 2.3 Tools and Techniques

In this research, we suggest or identify refactoring opportunities using tools and techniques based on static source code analysis. This section describes the tools and techniques we use to solve aforementioned design and quality problems.

### 2.3.1 Rule Based Parser

Dr. Fawcett developed a rule based ad-hoc parser which I and others in our research group have extended for our source code analysis. This parser analyzes source code, extracting only the information we need using rules that work on specific collections of tokens called "SemiExpressions". We chose this approach over using a traditional parser generator because the results we seek depend on only a small part of the (C++) language grammar. This parser has a simple design and very flexible to extend for detecting any new language grammar. It can also easily be applied to other similar programming languages.
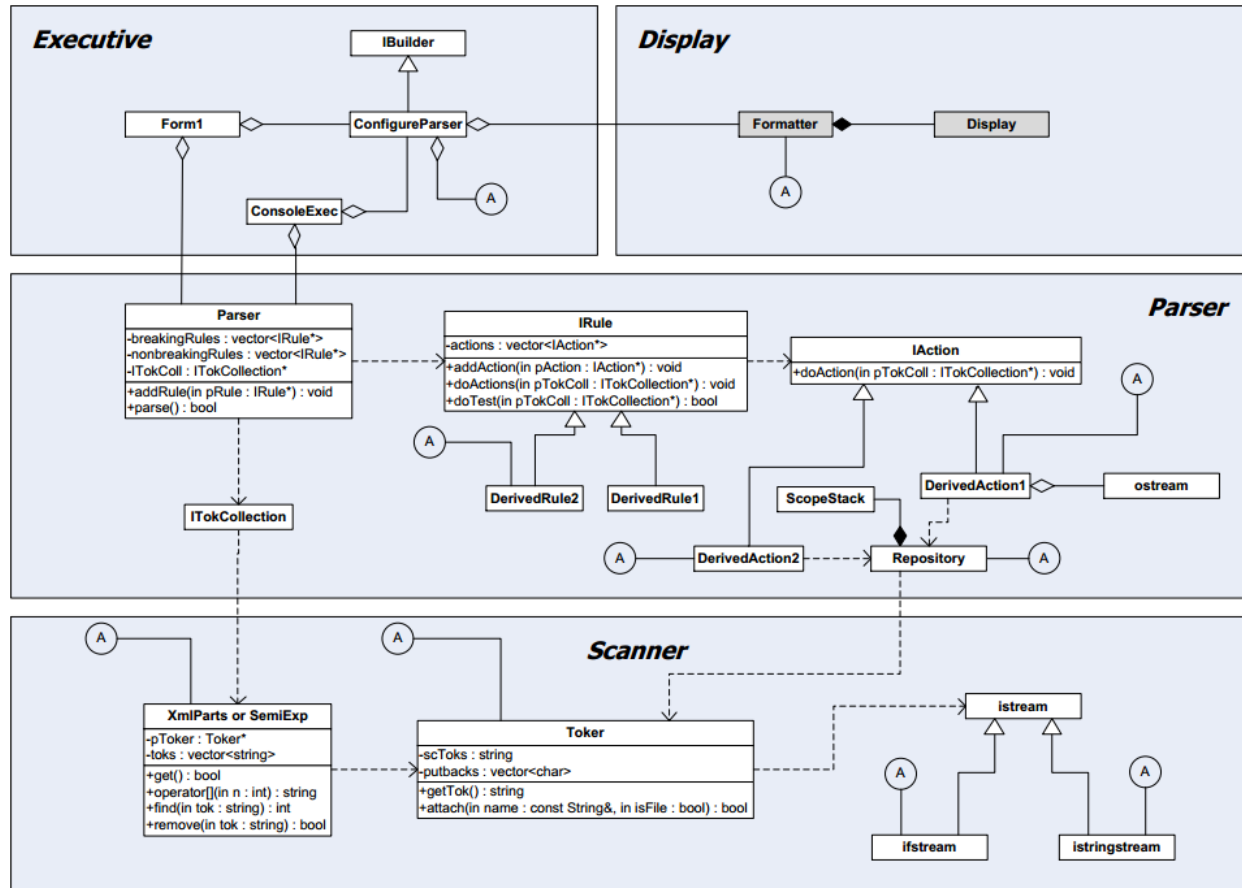
Figure 2.1 Parser Facility

Figure 2.1 shows the class diagram of the Parser facility that we have built. This parser facility provides modules to analyze code and XML texts. It enables us to extract information from source code according to a small part of the language grammar that we define in rules. The parser facility is designed in a way that it can be extended easily to extract additional information by defining new rules corresponding to additional parts of the language grammar.

The parser facility is designed on an Actions and Rules based approach. It provides an IRule interface for the user that can be used to define detectors for specific grammatical constructs like classes and functions. Each rule may include one or more actions that define what should be done when the grammar defined in the corresponding rule is detected. When we want to include a specific part of the language grammar, we simply define a concrete Rule with the IRule

interface and one or more actions with the IAction interface that define what to do when that particular grammatical construction is detected. We then attach action or actions to the rule that we defined.

By using this parser facility, theoretically, it is possible to parse source code according to the entire language grammar (C++) by defining rules and actions that cover the entire language grammar. But, the actual power of the parser facility is that it allows us to define a small portion of the language grammar and extract only the information that interests us.

## *2.3.2 Program Slicing*

The concept of program slicing was first proposed by Mark Weiser [9, 10, and 11]. Weiser describes program slicing as the method of automatically decomposing programs by analyzing their relationships between statements based on data and control flow. Given the criterion C=(s, V), where s is a program statement and V is a subset of variables in a program P, program slicing is the process of finding all the program statements that affect value of a variable v in statement s. While a backward slice is defined as all the statements that the computation at the slicing criteria may depend on, forward slice consists of all the statements computation of which may be affected by the slicing criteria [37].

| 1  | int i;                 |
|----|------------------------|
| 2  | int sum = 0;           |
| 3  | int product = 1;       |
| 4  | for(i = 0; i < N; ++i) |
| 5  | {                      |
| 6  | sum = sum + i;         |
| 7  | product = product *i;  |
| 8  | }                      |
| 9  | cout<< sum;            |
| 10 | cout<< product;        |

Figure 2.2 Example Program Fragment P

Figure 2.3 shows the program slice on the example program fragment P given in Figure 2.2, with respect to the criterion C= (9, sum). Here the number 9 represents the statement in line 9 of program P, i.e. *cout << sum;*

| 1 | int i; |
|---|--------|
| 2 | int sum = 0; |
| 3 | for(i = 0; i < N; ++i) |
| 4 | { |
| 5 | sum = sum + i; |
| 6 | } |
| 7 | cout<< sum; |

Figure 2.3 Slice of P with resp. to C=(9,sum)

## 2.3.3 Graph Theory

Graph Theory in general is an effective way of representing relationships between objects in almost any collection. For this reason, graph theory is widely used in many disciplines of computer science, physics, biology and the social sciences [12].

A simple graph consists of nodes and edges connecting these nodes to represent the existence of a certain relationship between them. Graphs are practical to apply to any problem where there are elements and a definition of a relationship between these elements. There are some theorems in graph theory, defined to gather information from a given graph that we find useful for our research. There are also different types of graphs that hold a particular property that is very useful for modeling code structure important for our research. In this subsection, we give the definition of a graph property, connected component, and a special type of graph, hammock graph, that we use in this research to introduce new refactoring techniques and tools.

### A.  Connected Components

A connected component in a graph, G, is a sub-graph in which there exists a path between every pair of nodes. Connected components can be used effectively to find out groups of objects that

directly or indirectly have a relationship and therefore usually construct a cluster of objects with at least one common attribute.
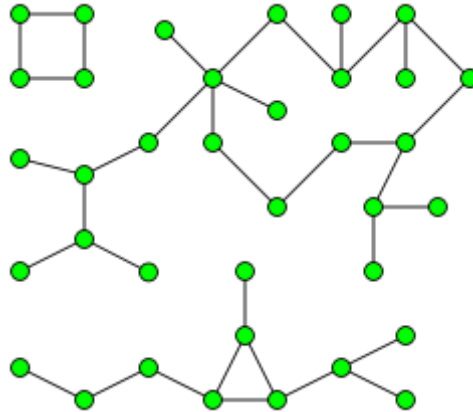


Figure 2.4 A graph with three connected components.

In Figure 2.4, for instance, we see a graph that includes three connected components. This property can be used to address some design flaws and quality issues in software programs. For example cohesiveness of a software unit can be measured using connected components and restructuring can be suggested using component analysis, once the attributes of that unit are represented with a graph.

## B.    *Hammock Graphs*

A Hammock Graph is a special kind of graph defined usually on a control flaw graph of a program as a sub-graph that has a single entry and a single exit points. This is an important property that we use in this research to confine code regions as extract method refactoring candidates. Definition and an example of hammock graphs are given in [13] as follows:

*Definition: Let G be a control flow graph for program P. A hammock H is an induced sub-graph of G with a distinguished node V in H called the entry node and a distinguished node W not in H called the exit node such that*

*(1) All edges from (G - H) to H go to V.*

*(2) All edges from H to (G - H) go to W.*

In Figure 2.5, we see an example program and its control flow graph. A hammock graph is also shown on this control flow graph in dashed lines.
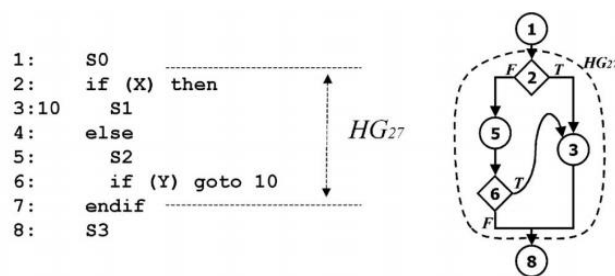


Figure 2.5 Example  program, its control flow graph and a hammock graph [13].

## 2.3.4 Placement Trees and Tree-Maps

Most programming languages, such as C++, C#, C and Java, obey a set of inclusion rules that govern how key constructs are placed within source code such as functions and controls. We will see that the placement structure of controls within a method is a tree which we will refer to as *a placement tree* where each element has only one parent.  The placement tree provides a representation of scopes within methods.

Every method, therefore, can be represented with a placement tree where each node represents a different scope. A scope is defined by all program code enclosed within braces, e.g., "{" and "}". A scope may include multiple other scopes and this hierarchical placement of scopes constitutes the placement tree. Placement tree structure is an important property that we use in creating
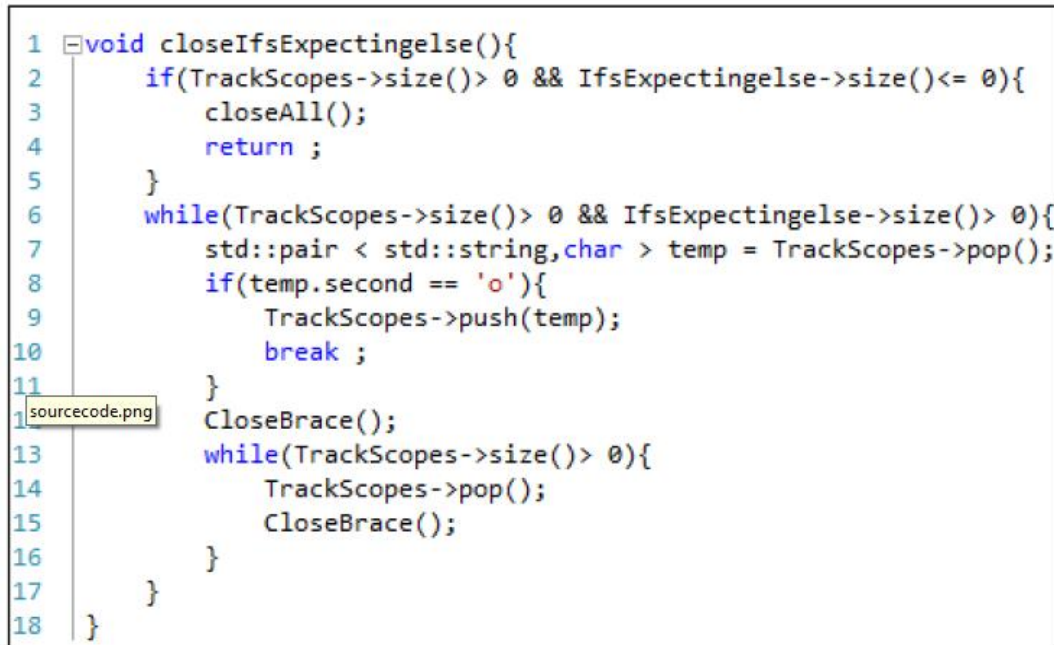
effective refactoring and visualization tools. Table 2.1 shows the scope types that can be considered to construct a placement tree for a given method.

Table 2.1 Scope Types

| Type name | Start Line | End Line |
|-----------|------------|----------|
| Method | Line that includes method name | Line that includes corresponding closing braces |
| For | Line that includes the key word for this scope. The key word is basically the type name itself, e.g. "for, while, do, switch, if" | Line that includes corresponding closing brace to the one following the respective type name. |
| While | | |
| Do-While | | |
| Switch | | |
| If | | |
| If-else | Line that includes the key word "if" | Line that includes corresponding closing brace to the one following "else" keyword. |
| Anonymous | Line that includes open brace that does not have an attached type | Line that includes the closing brace corresponding the anonymous scope opening brace |

An effective refactoring method should be supported by a visualization tool or technique for the developers to observe the suggested refactoring easily in large scale systems. To visualize the structure or placement tree of the input method, a *tree-map* approach is adopted in this research. Tree-map visualization is an effective way of presenting hierarchical information where nodes are represented by nested boxes [14]. Tree-map visualization in this research represents nodes on the placement tree using boxes where a parent includes all its children.

In Figure 2.6, an explanatory method is shown and Figure 2.7 shows placement tree for the given example code where each node is identified by its start and end line numbers. Final visual representation of the placement tree as a tree-map is also shown in Figure 2.7.

```
1  ⊟void closeIfsExpectingelse(){
2        if(TrackScopes->size()> 0 && IfsExpectingelse->size()<= 0){
3            closeAll();
4            return ;
5        }
6        while(TrackScopes->size()> 0 && IfsExpectingelse->size()> 0){
7            std::pair < std::string,char > temp = TrackScopes->pop();
8            if(temp.second == 'o'){
9                TrackScopes->push(temp);
10               break ;
11           }
12           CloseBrace();
13           while(TrackScopes->size()> 0){
14               TrackScopes->pop();
15               CloseBrace();
16           }
17       }
18  }
```

Figure 2.6 Example Method
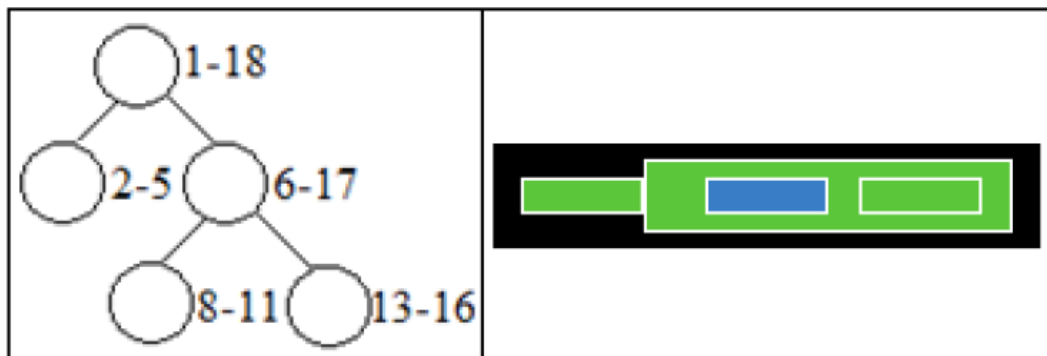


Figure 2.7 Placement Tree and Tree-map Representation

## 2.3.5 Analysis and Visualization Tools

### A. Analysis Tool

In this research, we have generated a static analysis tool to find extract method opportunities based on variable references. This tool creates an intermediate representation of a method for the

tree-map visualization tool that we wrote. This intermediate representation is in XML format and includes all information needed for evaluation of extract method refactoring opportunities.

Our analysis tools are implemented using the Parser facility described in Section 1.3.1. The main analysis tool consists of three fundamental processing units: Brace Insertion, Tree Generation, and Hammock Graph Construction.

**Brace Insertion:**  In the programming languages of interest for this research, scopes are defined with braces. Sometimes, if the scope has only one statement, there is no need to put braces to designate scope borders. Brace insertion unit automatically detects such scopes that include only one statement and places appropriate braces around them. While this process by itself enhances the readability of code, the main purpose of this is to make the code more easily analyzable statically. This is not a very trivial problem when especially if-else statements and other control statements used together hierarchically without having braces around them.

**Tree Generation:** The main task of this unit is to generate an XML representation for the Placement Tree.  Source code is first analyzed to detect all scopes and all the data is collected for each scope. All this information is then saved into an XML file for visualization purposes. This XML file basically contains the following information related to each scope: the entire source code for each particular scope, line numbers that represent where this scope starts and where it ends, and the variables referenced in this scope with the number of times they are used.

**Hammock Graph Constructor:** For each local variable in a given method, we define a code fragment that covers program statements between where the variable is declared and where the last reference of that variable appears.  We call this code fragment a *variable span* and this tool analyzes the code to find variable spans for each variable. Then, based on the interactions

between variable spans and control statements, an XML file is created that represents a graph where each variable span denotes a hammock. We will elaborate on this concept more in the next chapters.

## *B. Visualization Tool*

We have developed a visualization tool that provides different views of the code to support the refactoring process. The first view we have is a tree-map that represents the placement tree for a given method. In this visualization, each scope is represented with a box. This visualization technique is very effective especially to consider the structure of a method, where the outermost box represents the method itself. An example view is shown in Figure 2.8 where the box that represents a parent scope includes all the boxes that represent scopes within the parent scope. Every box has two distinct properties, size and color, that we use to determine refactoring opportunities. While the size of a box refers to the number of statements in the corresponding scope, the color indicates the task or computation carried out in the scope.
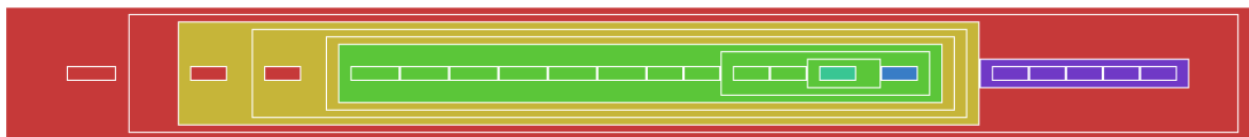


Figure 2.8 An Example View for Refactoring 1

Similarly, in our second view, we visualize the variable spans that represent hammocks to observe extract method refactoring opportunities. In this view, while the size of a box refers to the number of statements in that variables span, the color indicates the number of parameters proposed extracted method would take. This tool allows the user to observe different refactoring opportunities dynamically based on the number of parameters these suggested methods would take. Figure 2.9 shows two different views for method refactoring opportunities based on two

different values we chose arbitrarily for maximum length of the parameter list. These two views show the extract method opportunities for prospective methods taking up to 1 and 2 parameters respectively.
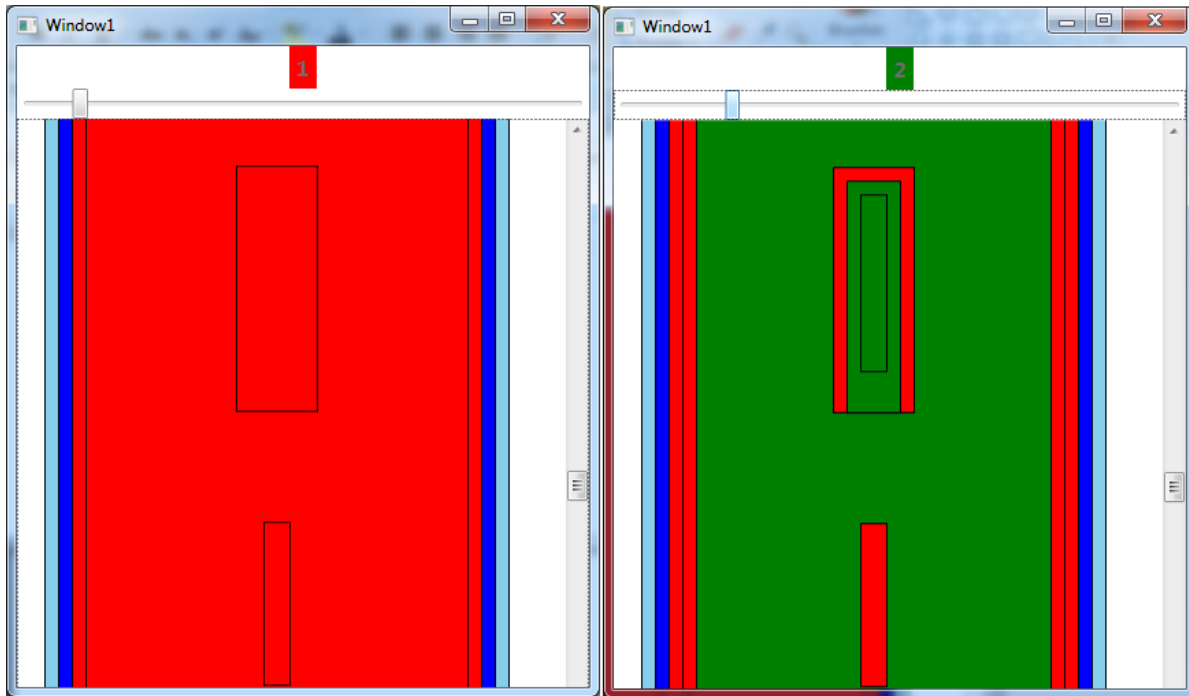


Figure 2.9 Example View for Refactoring 2

Furthermore, one can see the code fragment represented by these boxes at any point of the visualization by simply clicking on a point in the box. An example of such code view is given in Figure 2.10.
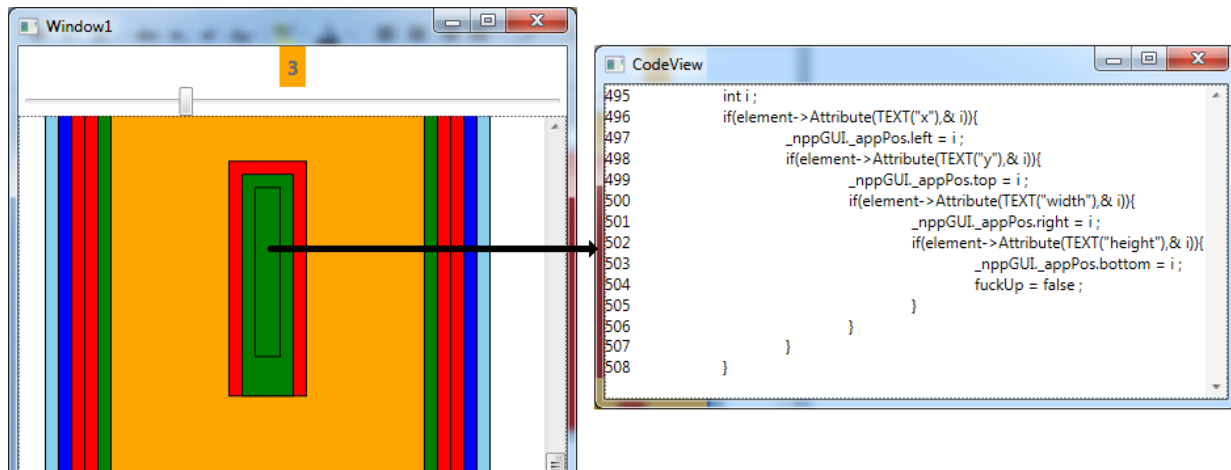
Figure 2.10 An Example of Code View

Last, we visualize the variable spans, control blocks and their interactions with a basic graph-like design where there is a directed edge or arrow from where the variable span or control block starts to where it ends.
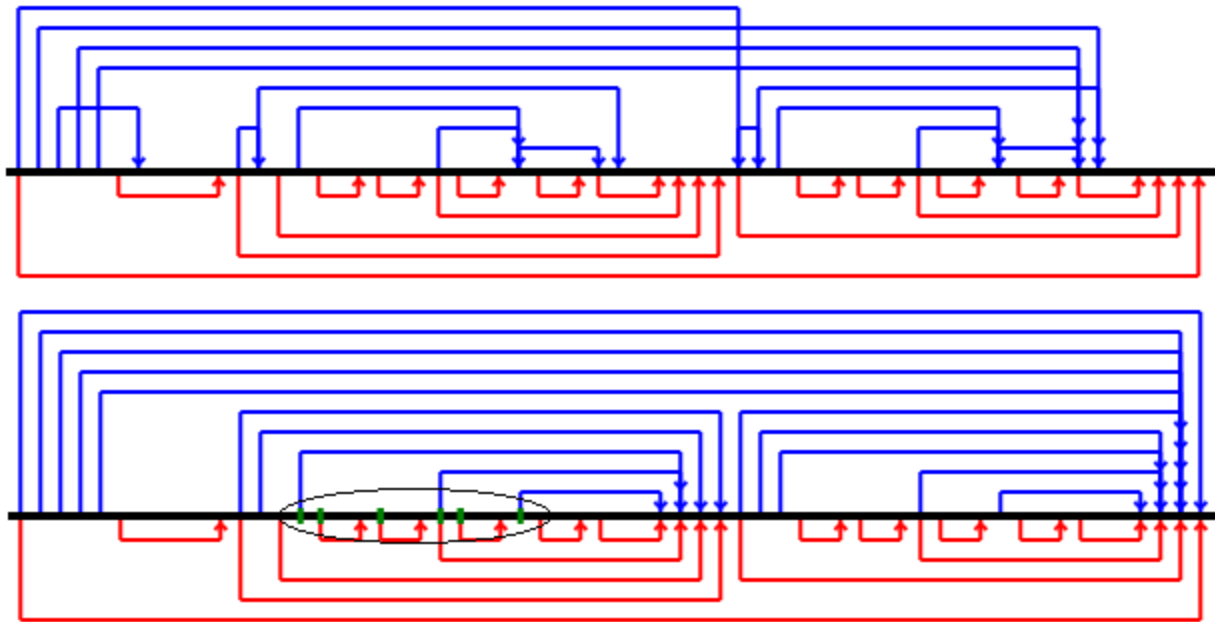


Figure 2.11 An Example View for Variable Span

In Figure 2.11, we see a graph that represents the initial variable spans and control blocks followed by another graph that represents the extended variable spans for hammock construction. In this view, one can observe the points where a particular variable is referenced by clicking on

the line that represents that variable span. The circled part of this figure shows an example of this for an arbitrarily chosen variable.

## 2.4 Related Works

In this research as stated earlier, several different novel techniques and tools are introduced to address software design issues. In this section, we present some of the other works in literature that are related to the techniques used or the problems addressed in this research.

### 2.3.1 Program Slicing

Program slicing has been used in procedural programming extensively since it was first proposed by Weiser. Weiser showed the applications of slicing to debugging and parallelization [83, 84, 52]. An empirical study of some slice-based cohesion metrics can be found in [19]. Slice based cohesion metrics have been proposed for measuring cohesion level of a procedure and used in various studies [15-25]. In [20], reviews on slice-based cohesion measures for procedural design and object-oriented design are discussed. Slice-based cohesion measures for object oriented software discussed in [20] are either an extension of functional cohesion measures [25] or use data member-method interactions for measurement [32], but they are quite different from our approach described in this research as we will see in the next chapters.

An enhanced version of program slicing is given in [74] that is implemented in a C-slicer application supporting the entire C programming language. This techniques is introduced towards handling arrays, structures, unions and expressions with side effect. Application for this particular technique accepts slicing criterion via a mouse click, therefore it offers the user control over the selection of the slicing criterion.

Some studies use program slicing to suggest restructuring for methods by defining the low cohesive parts to be extracted from the procedure. In [24], this technique is used to measure the cohesiveness of each statement in a method and to identify parts of the procedure that cause low cohesion for restructuring purposes. In [36], cohesion level between output variables of a method is determined and a graph (pair-wise cohesion graph) is generated to visualize this relationship. After removing all edges that represent a cohesion below a given threshold level, they suggest restructuring the function based on connected components in the graph.

In [16], the notion of data slices is defined and used for measuring functional cohesion. For each output variable v, the data slice is defined as the set of data tokens, i.e. variable and constant definition and references to them. Therefore, a change made on any of the data tokens in data slice of v will affect the value of output variable v at some point. Cohesion is measured based on the percentage of data tokens that appear in more than one data slice and the ones that appear in all the data slices. In this version of slicing, data tokens are the basic units rather than statements in the program.

## 2.4.2 Object-Oriented Cohesion Metrics and Refactoring

Although there have been a few slice based cohesion metrics proposed for measuring the quality of classes in object oriented design, to our knowledge, none has been proposed for the purpose of restructuring classes. In [25], slice based data cohesion measures for object oriented programs are defined as a modification of slice based functional cohesion measures defined in [16]. In [25], private and protected member variables of a class are considered as the data tokens and data slices are determined based on them. They do not alter the definition of measuring cohesion and use exactly the same measurement technique proposed by [16]. Although this study is one of the

very few that use program slicing for the object oriented paradigm, it does not support restructuring a class into more cohesive form once low cohesion level of the class is determined.

There are some other techniques used to indicate the cohesion level of a class. The majority of them use interactions between data members and methods of the class such as data member usage or sharing of data members [28, 29, 30, 31, 32, and 33]. Some studies represent this method-data member relationship with an undirected graph and suggest that the number of connected components in the graph indicates the cohesion level [31]. In [33] the dependency relationship between data members is also considered when evaluating the usage of a data member by a method. Although using graph theory in this manner is an important technique, none of these works suggest any restructuring method to improve the cohesion level of a class once they quantitatively identified the low cohesion level of the class at hand. A comparative study of graph theory-based class cohesion measures can be found in [67].

## 2.4.3 Clustering Techniques

Clustering Techniques are another preferred area of study for cohesion measurement and restructuring in object oriented programs. The basic idea in this technique is to group entities in a system (these entities are usually data members, methods and classes) based on the similarities between them or relationships that they preserve in order to construct more cohesive groups [34].

Almost all applications of restructuring which use clustering techniques have the idea of moving data members or methods or some other entities that they define from one component to another. For instance, in [34], a clustering approach is given to group entities such as classes, methods and data members to construct more cohesive groupings by moving entities between these groups that could originally be misplaced where they do not actually belong. These kinds of

approaches may not always be very useful, for a method itself may not be cohesive requiring some statement level analysis and refactoring.

### 2.4.4 Extract Method Refactoring

Program slicing has been used extensively to identify code fragments for extract method refactoring [41]. However, implementation can be tedious and time-consuming since this technique usually requires the user to manually select the slicing criterion. Furthermore, there is no a priori approach for choosing the slicing criterion for extract method refactoring.

In [41], a program slicing technique is proposed to extract code fragments related to the computation of a given variable and the state of the given object. A mechanism that uses block-based program slicing to automatically extract methods in object-oriented programs is proposed in [41, 43]. A transformation technique to decompose functions into smaller ones, Tuck, is proposed in [44] based on program slicing.

There are very few studies that aim to identify extract method opportunities. Our extract method opportunities differ from those in [41, 43], in a way that the main concern in [41, 43]  is to compute slices in order to extract the complete computation of a given variable declared inside a method. We argue that, most of the time methods take parameters implying that computation of a particular variable may be split across multiple methods. Further, methods should have a clear intention and as said in [46], the developer is the last authority to decide which refactoring should be applied. With this in mind, in this research we try to find cohesive segments of program statements as refactoring candidates and support this with visualization to help the developer make his/her final decision about the appropriateness of the suggested refactoring.

Control Flow Graphs (CFG) are also used widely as a method extraction tool. In [45], an automatic process for extracting methods based on an input CFG of a method and a set of pre-selected nodes is introduced. To find extract method refactoring candidates, an approach based on Data and Structure Dependency (DSD) graph and longest edge removal algorithm is proposed in [42]. In [39], a methodology is given to extract a set of marked statements that are difficult to extract due to the presence of some certain key words.

Another area where extract method is applied is detection of code duplicates. In [75], an approach based on abstract syntax trees is introduced to detect code duplicates. They provide a visualization technique called star diagram to give hint for extraction of code duplicates.

### 2.4.5 Use of Hammock Graphs in Literature

The concept of Hammock graphs has been used in various research areas such as compiler optimization and software comprehension. In [13], for effective compiler optimization, a technique based on hammock graphs is proposed to eliminate unstructured branches such as *jumps* and *goto* statements by converting them into structured form with well defined scopes. A series of algorithms are provided to convert different branch structures based on their interactions on a control flow graph (CFG).

In [53], a dynamic compiler (Carcal) is introduced to convert unstructured braches into structured forms using hammock graphs in CFGs. This compiler enables the analysis of performance differences between different vendor's GPU programming models that may or may not support unstructured branches.

We came across another application of hammock graphs in software comprehension field. In [54], in order to determine the code that is related to a particular use case, hammock graphs are

induced on call graphs between pre-defined methods (landmark methods) that are known to be a part of that use case.

Hammock graphs have been used for comparing two versions of a program to detect correspondences and differences. In [55], a differencing algorithm is introduced to compare object-oriented programs as an extension of [56]. Control-flow graphs of different versions of a program are first converted to a reduced form by collapsing hammock graphs. The assessment of correspondence or difference is done based on the comparison between matched hammocks.

In this research, we generate hammock graphs based on variable declarations and uses as well as control blocks in a method. We also consider the interactions between these entities to construct hammock graphs out of each variable span. Hammock graphs have never been used this way to find extract method opportunities and this is one of the major contributions of this research.

### 2.4.6 Visualization and Refactoring Tools

Refactoring is a critical process that is practiced frequently to help developers adapt software to the changing demands of customers, markets, and managers [26]. Although Fowler introduces many different kinds of refactoring, the identification of location where to apply these re-factorings is ambiguous. This is, as stated in [46], is one of the main problems when refactoring especially large systems. In [63], discussion is given on the necessity of visualization and how current visualization techniques "can only handle the voluminous aspect of data but fail to address imprecise and incongruent aspects."

Some studies show that, although refactoring is practiced very frequently, 90 percent of refactoring is applied manually and refactoring tools need further improvements [64,65]. In [66], it is stated that lack of tool support makes the developers perform refactoring manually and a

refactoring tool for Smalltalk is provided. This tool, like most other refactoring tools, does not suggest refactoring, but merely helps to apply a particular refactoring on a user-selected code fragment.

Researchers have introduced visualization tools to support different aspects of software engineering processes such as refactoring and software comprehension. In [46], it is stated that although it is believed that the developer is the last authority to decide where to apply the refactoring, tool support is necessary and static structure analysis is the key. They introduce a visualization to locate where to apply the following re-factorings: move method move attribute, extract class and inline class. Yet they are based on moving existing attributes as in clustering techniques.

A visualization designed as an eclipse plug-in is introduced in [23], that helps the programmers see the relationships between class attributes. They aim to use this visualization with clustering algorithms to observe the clustering results more effectively.

A visualization tool is introduced in [27] that uses a tree-map approach for software comprehension. It helps the user detect code defects by visualizing package-class-method structure via tree-map. This tool is not provided to suggest a refactoring specifically. It deals with how modules are organized in packages, classes and methods.

To help  recover evolution of object oriented software systems, a tool is introduced in [62] to visualize the evolution of software systems, using a visualization technique based on what they call Evolution Metrics.

Visualization is also used for studying the release history of a software system and abstract software structure in [68]; and for comprehension and evaluating the fault proneness of a class in

[69]. A study [70] uses visualization also for package dependency and relationships between packages. In [71], visualization of package dependencies based on type dependency is studied for diagnosing software design issues and risk analysis.

There are some tools and IDEs that support execution of certain re-factorings. One such commercial tool, ReSharper [79], support a range of refactoring including extract method refactoring which is defined as a key refactoring [73]. Refactor Pro [80] is another commercial tool for refactoring that supports many different re-factorings. Eclipse [81] and Visual Studio [82] are two most widely used IDEs that provide refactoring support. All these tools have some limitations and rely on the user selection to determine code fragments to be re-factored. If the user selected code fragments do not meet the refactoring preconditions, these tools provide error messages indicating the problem.

Error messages emitted by existing tools are non-specific and unhelpful in diagnosing problems [73]. A study in [73] concludes that programmers often encounter a variety of errors arising from violated refactoring preconditions and the error messages are "insufficiently descriptive, conflated and discouraging programmers from refactoring at all." This work, due to the deficiencies of the error messages in restructuring tools, provides three visualizations to help the user with the selection of candidate code fragments for extraction. The first visualization provided, selection assist, highlights the whole statement that the user partially selected. Second view, box view, helps to display the code in nested box form. In this regard this work is similar to our visualization, yet they do not suggest any box to be selected. By clicking on a box user can select that block of code. The last view, refactoring annotation, helps the user observe the control flow and variable usages by highlighting variables with different colors. Although this work partially targets the selection part in refactoring, it does not help to suggest user a segment of

code for selection. Moreover, all of these views are already supported in different ways by IDEs such as Visual Studio and we do not think that they add any significant contribution towards refactoring suggestion.

# Chapter 3

## Class Cohesion and Refactoring

In this research, one of our main contributions is to introduce a new cohesion metric and a refactoring technique for classes in object oriented programs. This section describes our work to evaluate the cohesiveness of a class and to address the Large Class code defect by transforming the class into a more cohesive form by splitting distinct abstractions.

### 3.1 A Program Slicing Technique for Class Refactoring

Program slicing traditionally has been used to discover program statements that affect the value of a variable at some point in the program. The process of program slicing usually starts by determining a slicing criterion in the form of $C=(s, V)$, where s is a program statement and V is a subset of variables in a program P based on data and control dependencies between statements.

| 1 | int i; |
|---|---|
| 2 | int sum = 0; |
| 3 | int product = 1; |
| 4 | for(i = 0; i < N; ++i) |
| 5 | { |
| 6 | sum = sum + i; |
| 7 | product = product *i; |
| 8 | } |
| 9 | cout<< sum; |
| 10 | cout<< product; |

Figure 3.1 Example Code Fragment P

| 1 | int i; |
|---|---|
| 2 | int sum = 0; |
| 3 | for(i = 0; i < N; ++i) |
| 4 | { |
| 5 | sum = sum + i; |
| 6 | } |
| 7 | cout<< sum; |

Figure 3.2 A Slice of Code Fragment P

Figure 3.2 shows a program slice on the example program fragment P given in Figure 3.1, with respect to the criterion C= (9, sum). Here the number 9 represents the statement in line 9 of program P, i.e. cout << sum;, shown in Figure 3.1.

In this research, we have defined a set of relationships to evaluate relationships between statements for refactoring purposes. The program slices that we find this way, can be used to evaluate cohesiveness of classes and support refactoring as a remedy for Large Class code defect.

### 3.1.1 Selection of Slicing Criteria

The first step in program slicing is usually to determine a slicing criterion. In this section we describe how we choose slicing criteria in a given class and how we identify program slices based on these criteria.

### *3.1.1.1 Slicing Criteria for a Class*

We seek to determine slicing criteria in a given class for evaluating its cohesiveness and proposing refactoring opportunities. To identify slicing criteria and slices in a class C, the following sets have been defined:

- *$DM_C$ is the union of all private data members defined in class C.*

- *$ST_{dxC}$ is the set of all program statements using data member d in C where d $\in$ $DM_C$.*

Each element of the set $ST_{dxC}$ represents a slicing criterion for data member d. For each d1 in $DM_C$, we construct a corresponding set, $ST_{d1xC}$, representing the slicing criteria for data member d1.

This part of our research deals with improving the structure of an existing class without changing its external behavior. To protect the behavior of the whole system and keep the client code in the system unaffected by any changes we make to the class, there are two options to be considered during refactoring regarding protected data members. By default, we include in our slicing criteria only those protected data members that have never been used in a class derived from the class in which they are defined. If this is the case, these protected data members can be treated just like private data members. The second option is to include all protected data members as slicing criteria. In this case, if a protected data member is moved into a newly created class, our refactoring process should be extended to support multiple-inheritance for the derived classes that use the moved data members. We have not done this in the following.

Since a private data member in a class can never be used outside of the class it is defined in, we include all private data members in our slicing criteria.

### 3.1.2 Defining Relationships between Statements

Generally, only control and data dependencies are used to find the dependencies between program statements to identify program slices. Informally speaking, a slice is the combination of backward and forward slices based on a criterion in a given code fragment or a program unit. While a backward slice is defined as all statements that the computation of the slicing criteria may depend on, a forward slice consists of all the statements computation of which may be affected by the slicing criteria [9].

After determining our slicing criteria, we want to find certain relationships between program statements with respect to these criteria. Since we seek program slices for refactoring purposes, rather than using program slicing as it is, constructed based only on control and data dependencies, we define a broader set of rules to define program slices.

Our primary focus is to find program statements which possibly constitute operations in the same abstraction domain. For this reason, we define a set of conditions for the statements to be evaluated in this manner. We say that two statements, S1 and S2, are dependent when one of the following conditions is true:

1.  *Execution of statement S1 is affected by statement S2, or vice versa. An "if" control statement and a "for" loop statement are good examples for this case.*

2.  *A variable defined in S1 is being used in S2*

3.  *A variable, defined in statement S' which uses a variable defined in S1, is being used in S2.*

4.  *A variable defined in statement S' is being used in both S1 and S2.*

5. *Invocation of a method f( ) which includes the statement S1 is affected by statement S2.*

6. *Execution of both S1 and S2 is affected by the statement S'.*

7. *A variable defined in S1 is passed to a method f as an argument and the argument is being used in statement S2 of method f.*

Our definition of relationships between statements takes both direct and indirect interactions between these statements into consideration and generates different slices than other traditional program slicing approaches do as shown in Figure 3.3. After finding a specific slicing criterion, we take all the statements into the slice rather than only the statements which may affect the value of a specific variable in the criterion. We argue that these slices can be considered to be in the same abstraction domain.

| 1 | int i; |
|----|--------|
| 2 | int sum = 0; |
| 3 | int product = 1; |
| 4 | for(i = 0; i < N; ++i) |
| 5 | { |
| 6 | sum = sum + i; |
| 7 | product = product *i; |
| 8 | } |
| 9 | cout<< sum; |
| 10 | cout<< product; |

Figure 3.3 Our Program Slice on P

Figure 3.3 shows the slice we get from program fragment given in Figure 3.1 with respect to the criterion C= (9, sum) considering the dependency conditions stated above. Notice that from Figure 3.3, one can infer that all the statements are related in that program fragment with respect to the given criterion. We claim that considering class cohesion and refactoring, using all the dependency conditions listed above will lead to more accurate results as we seek relationships between statements for extract class refactoring opportunities.

### *3.1.3 Demonstration of Relationship Conditions*

To better explain this concept, we demonstrate the dependency conditions listed above using some fragments of the source code of the original version of Class1 given in [35].

### *3.1.3.1    Conditions 1,5 and 6*

The code fragment given in Figure 3.4, shows a representation of conditions 1, 5 and 6. Let the statements at line numbers 129, 131, 132 and 19 be represented by S1, S2, S3 and S4 respectively.

| Condition 1, 5 and 6 | |
|---|---|
| 16 | void ErrorInSize() |
| 17 | { |
| 18 |   cout<<"Index out of range!\n"; |
| 19 |   cout<<"The Array has "<<top |
| 20 |     <<" elements.\n"; |
| 21 | } |
| 129 | if (top > 0) |
| 130 | { |
| 131 |    top--; |
| 132 |    int temp_int=stk[top]; |
| 133 |    return temp_int; |
| 134 | } |
| 135 | else |
| 136 | { |
| 137 |    ErrorInSize(); |
| 138 |    return -1; |
| 139 | } |

Figure 3.4 Conditions 1, 5 and 6

S1, S2 and S3 are dependent since execution of S2 (131) and S3 (132) is controlled by S1 (129) and moreover S1 (129) and S4 (19) are also dependent as invocation of method *ErrorInSize()*, which includes S4, depends on S1.

### *3.1.3.2    Conditions 2 and 3*

The code fragment given in Figure 3.5, demonstrates conditions 2 and 3. Let the statements at line numbers 64, 65, 66 and 67 be represented by S1, S2, S3 and S4 respectively. S1, S2, S3 and S4

are dependent because of their variable usage. That is S3 uses two variables defined in S1 and S2 respectively and S4, subsequently uses a variable defined in S3.

| Conditions 2 and 3 | |
|---|---|
| 64 | int w=x2-x1; |
| 65 | int h=y2-y1; |
| 66 | int a=w*h; |
| 67 | return a; |

Figure 3.5 Conditions 2 and 3

### *3.1.3.3* *Condition 4*

The code fragment given in Figure 3.6, shows an example of condition 4. Let the statements at line numbers 111, 112 and 113 be represented by S1, S2 and S3 respectively. S1, S2 and S3 are dependent because a variable defined in S1 is used in both S2 and S3.

| Condition 4 | |
|---|---|
| 111 | string temp="Push invoked: "; |
| 112 | temp+=t; |
| 113 | PushFunInvok(temp); |

Figure 3.6 Condition 4

### *3.1.3.4* *Condition 7*

In the code fragment given in Figure 3.7, we show a representation of condition 7. Let the statements at line numbers 29 and 101 be represented by S1 and S2 respectively. S1 and S2 are dependent because a variable defined in S1 is passed to a method, *PushFunInvok(std::string str)*, as an argument and the argument is being used in statement S2 of the method.

| Condition 7 | |
|---|---|
| 29 | string temp="Class1 invoked: "; |
| 30 | temp+=t; |
| 31 | PushFunInvok(temp); |
| 97 | void PushFunInvok(std::string str) |
| 98 | { |
| 99 | if (topInvok < 100) |
| 100 | { |
| 101 | funinvokes[topInvok]=str; |
| 102 | topInvok++; |
| 103 | } |
| 104 | else |
| 105 | ErrorInSizeFunInvok(); |
| 106 | } |

Figure 3.7 Condition 7

### *3.1.4 Determination of Program Slices*

To identify slices for data members defined in a class C, we have defined the following sets in addition to the sets defined in section 3.1.1.1:

- *$SL_{stxC}$ is the set of all program statements which are directly or indirectly related to the statement st based on the conditions listed in section 3.2. In other words, $SL_{stxC}$ is the union of backward and forward slices based on the criterion of statement st.*

- *$SL_{dxC}$ is the union of all $SL_{stxC}$ where st $\in ST_{dxC}$ and d $\in DM_C$.*

$$SL_{dxC}= \bigcup_{st \in STdxC} SLstxC$$

Therefore $SL_{dxC}$ is the slice in our class C which includes all statements that are directly or indirectly related to at least one of the statements that uses data member d in C.

## 3.2 A New Cohesion Metric for a Class and Refactoring Suggestion

Class structure is the key unit of object oriented programming. Therefore, developers aim to design classes with high quality so that they can be reused, maintained, and tested easily. To

reduce maintenance cost, these key units are expected to be simple, understandable, and readable as well.

Cohesion metrics have been studied extensively for the purpose of evaluating the relatedness of the components of a class. Different techniques have been used to quantify this aspect of the quality for a class. Most of the cohesion metrics are not proposed for refactoring classes to improve cohesion; therefore they may not be practical as a refactoring indicator. A cohesion metric based on this program slicing as described in our approach will give more accurate results in object oriented environment as a refactoring criterion, as one can see all possible relationships between program statements that are suitable for extract class refactoring.

This section presents a new cohesion metric based on program slicing and graph theory for units using object oriented design. One can make judgments about clarity of intent of the code using the metric we propose here. By identifying all program statements which constitute operations in the same abstraction domain, our goal is to determine if a class is cohesive handling one specific operation. When a class has more than one abstraction, this technique suggests an extract class refactoring for generating more cohesive units based on this new cohesion metric.

### 3.2.1 Data Slice Graph

After constructing the program slices for a given method, we generate a graph which we call Data-Slice-Graph (DSG) to evaluate cohesiveness of the class. This graph shows the relationships between the data members of the class based on interactions between their slices. Our motivation is to find out which data members contribute to the same operation in the class, in case the class is not cohesive. As we will see in the remainder of this chapter, DSG reveals

this information providing a great help for evaluating cohesion and suggesting refactoring for the class.

In DSG, each node represents a data member of the class. We have the following definitions for DSG:

- *DSG= (V, E) is a undirected graph such that V is the finite set of data members representing vertices in the graph and E is the finite set of relationships between data members representing edges in the graph.*

- *|V| is the number of data members of the class*

- *Let v1v2 represent an edge between two nodes v1 and v2;*

$$v1v2 \in E \text{ iff } SL_{v1xC} \cap SL_{v2xC} \neq \emptyset$$

The description of DSG suggests that two data members, d1 and d2, are related if there is at least one program statement in the class that affects at least one occurrence of both data member d1 and data member d2 based on the dependency conditions given in section 3.1.2. Therefore the vertices, v1 and v2, representing data member d1 and data member d2 respectively, have an edge between them in DSG, i.e. v1v2 is in E. An example DSG is shown below in Figure 3.8 for a class with eight data members.

## 3.2.2 Evaluation of Cohesion Based on DSG

Cohesion level of the class is defined the as the number of connected components, NC, in its DSG. The bigger NC the less cohesive our class is. Each connected component in DSG refers to one abstraction that the class holds.
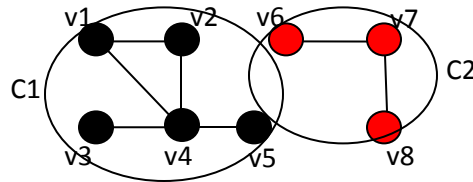
Figure 3.8 Example DSG

To consider restructuring this class, we use DSG and the number of connected components (NC) in DSG. Before discussing the refactoring process, we shall explain what various values of NC mean:

- *NC = 0 means there are no data members defined in the class. That is a class that has no state - a utility class may be an example for this. Note that we do not propose any refactoring on this type of classes as DSG does not reveal any information for this kind of class.*

- *NC = 1 occurs when the class has only one abstraction and when it is most cohesive. Therefore, we do not propose refactoring this kind of class.*

- *NC > 1 occurs when the class has more than one abstraction. DSG reveals this by having more than one connected component and each connected component in this case represents a different abstraction the class is designed to handle. We suggest refactoring the code in this case and generate one cohesive class out of each connected component in DSG.*

### 3.2.3 Suggesting Extract Class Refactoring

In object-oriented programming, a class generally is designed to handle one certain operation (one abstraction). To achieve this, most classes have some data members and methods to handle some part of the operation based on the clients' requests using some of the data members defined

in the class. From this point of view, we think that there is likely to be a relationship between the data members that are used to perform the intended operation of the class.

In DSG each connected component is a candidate to be extracted as a new smaller, yet more cohesive class. In the example DSG given in Figure 3.8, C1 and C2 represent two different abstractions and our approach suggests that they should be extracted as new classes. Therefore data members represented by v1-v5 together with their slices are to be one class and data members represented by v6-v8 together with their slices are to be another class.

Our approach proposes this refactoring by defining independent sets of statements to be components of new classes. We propose to generate a method in the class with each consecutive set of statements in the slice of any data member that construct the connected component of the class. For example, in Figure 3.8 data members v6, v7, and v8 construct the connected component for a new class C2 and each consecutive set of statements in the slices of these data members are suggested to be a method in C2.

### 3.2.4 Edge Cases to be Considered

As we have stated earlier, the main focus of our research is on the first step of extract class or extract method refactoring which is the identification of code fragments for extraction. Although we do not tackle the second step of these refactoring approaches, which is the extraction of the indentified code itself, we find it useful to highlight some of the edge cases that may be encountered during the execution of our techniques.

In most cases extracting statements in a slice as a method into the new class is straightforward. Yet, there might be a few edge cases to handle to prevent undesirable results. For example, we do not want to have any dependency from the classes we generate to the original class as this will

affect the reusability of the classes by having a mutual dependency with the original class. This scenario is possible if a slice is including a method call. We have defined three cases that need particular attention during refactoring related to method calls. Considering all of these cases will eliminate undesirable dependencies that might arise from call-backs to the original class.

### 3.2.4.1 Edge Case 1: Method call in a control block

Description: Our technique always guarantees that the method definition and the method call in this case are in the same slice. The 5th dependency condition listed in section 3.1.2 assures this and code fragment given for this condition in Figure 3.4 demonstrates this case. In that example code, statements at lines 18, 19, 20 and 137 are always guaranteed to be in the same slice.

Action: We suggest changing that method call in the control block with a method call to the corresponding method created in the new class. That will eliminate a call-back to the original class. In other words, during refactoring of the code fragment given in Figure 3.4, statements at line numbers 19, 20 and 21 will be moved into a new class as a method and statements at line numbers between 129 and 139 will be moved to the same class as another method. Let f1 and f2 represent these two methods respectively. We can assure this as all of the statements above will always reside in the same slice. Therefore we replace the method call *ErrorInSize();* in method f2 with a method call to f1.

### 3.2.4.2 Edge Case 2: Method call without an argument

Description: Our technique always guarantees that the method call in this case will not reside in any of the slices defined by our technique as there is not any data being passed in this method call – see Condition 7 [3.1.2].

Action: No special action needs to be taken for this case. This case does not cause any call back to the original class.

### *3.2.4.3 Edge Case 3: Method call with an argument*

Description: Our technique does not guarantee that all of the statements in the definition of the method will always be in the same slice as the method call, although we think that this is a case unlikely to happen. Yet, at least some parts of the method will be in the same slice with the method call, but the action in case 1 would not solve this problem in this case. The 7th dependency condition listed in section 3.1.2 is related to this case and code fragment given for this condition in Figure 3.7 demonstrates it. In that code fragment method call at line number 31 passes an argument to the method *PushFunInvok(std::string str)* and statement at line 101 uses that argument. When we analyze this method we will see that all the statements in its definition are fully related with respect to the criterion C= (101, str) based on our dependency conditions which means that all the statements in the definition of this method are in the same slice and therefore all together they will construct a new method in the extracted class. This makes the case to be just like case 1 and action in case 1 will handle any possible call backs for this program.

Action: One should verify if the definition of the method consists of statements which are fully related based on some previously defined criteria or based on criterion of the statement which uses the argument passed to the method. If the method definition consists of fully dependent statements then this case is same as case 1. Otherwise this method call should be removed from the slice and the argument in the method call should be replaced with a return value from the

method created in the new class corresponding to this consecutive set of statements where the method call resides.

### 3.2.5 Experimental Results for Class 1

We now present our experiments for demonstration of this new cohesion and refactoring approach. The original class that we used for this experiment is given in [35] named as Class1. The remainder of this chapter, discusses the execution of our refactoring approach step by step.

### 3.2.5.1 Determination of Slicing Criteria and Construction of Slices

This class has 9 private data members and is not very well designed. Figure 3.9 below shows all the properties of the class based on our approach for data members *stk* and *top*. Note that in Figure 3.9 all the numbers given as an element of a set are the line numbers of the statements in the program. Sets for other data members of the class and their pair wise comparisons can be found in the appendix.

In Figure 3.9, the set $ST_{stkxClass1}$ is the set of all statements that use data member stk in Class1. This set represents the slicing criteria for that data member. In this case there are two statements which use data member stk in Class1, and they are the statements at line number 116 and 132. The sets $SL_{116xClass1}$ and $SL_{132xClass1}$ represent slices based on the criterion of the statement at line number 116 and the criterion of the statement at line number 132 respectively. And finally the set $SL_{stkxClass1}$ represents the slice of data member stk including all the statements that is related to at least one occurrence of that data member.

$DM_{Class1}$ = {stk, top, funinvokes, topInvok, rawtime, x1, y1, x2, y2}

*stk*

$ST_{stkxClass1}$ ={116, 132}
$SL_{116xClass1}$ ={114, 116, 117, 120, 18, 19}
$SL_{132xClass1}$ ={129, 131, 132, 133, 137, 138, 18, 19}
$SL_{stkxClass1}$ = $SL_{116xClass1}$  U   $SL_{132xClass1}$
$SL_{stkxClass1}$ ={114,116,117,120,18,19,129,131,132,133,137,138}

*top*

$ST_{topxClass1}$={19,32,87,114,116,117,129,
            131,132,148}
$SL_{19xClass1}$ ={19}
$SL_{32xClass1}$ ={32}
$SL_{87xClass1}$ ={87}
$SL_{114xClass1}$ ={114,116,117,120,18,19}
$SL_{116xClass1}$ ={114,116,117,120,18,19}
$SL_{117xClass1}$ ={114,116,117,120,18,19}
$SL_{129xClass1}$ ={129,131,132,133,137,18,19,138}
$SL_{131xClass1}$ ={129,131,132,133,137,18,19,138}
$SL_{132xClass1}$ ={129,131,132,133,137,18,19,138}
$SL_{148xClass1}$ ={148}
$SL_{topxClass1}$ = $\bigcup_{st\in STtopxClass1}$ SLstxClass1
            = {19,32,87,114,116,117,120,18,129,131,132,133,137,138,148}

Figure 3.9 Example Calculations for Class 1

The results of same calculations are shown in the Figure 3.9 for the data member *top* for this demonstration. Calculation for the remaining data member of the class can be found in the appendix.

### 3.2.5.2 Generating DSG

After finding all the properties for each data member of the class as shown in Figure 3.9, now we are ready to construct DSG for the class. Figure 3.10 shows the process of constructing the graph.

Number of vertices

|V|=#of private data members=9

Edges

Let v1 and v2 represent data members *stk* and *top* respectively.
$SL_{stkxClass1}$ ∩ $SL_{topxClass1}$={114,116,117,120,18,19,129,131,132,133,137,138}
$SL_{stkxClass1}$ ∩ $SL_{topxClass1}$ ≠ Ø

Therefore;
                    *v1v2* $\in$ E

Figure 3.10 Construction Process of DSG

In Figure 3.10, we see the intersection of the slices for data member *stk* and data member *top* with 12 elements. Based on the definition of DSG, this means that these two data members are related and therefore there is an edge between the vertices that represent them in the corresponding DSG.

We have analyzed the code for the class given in [35] and generated the data shown in Table 3.1 that demonstrates pair-wise comparisons of the slices of every pair of data members in the class.

Table 3.1 Intersections of Slices for Class 1

| ∩ | stk | top | funinvokes | topInvok | rawtime | x1 | y1 | x2 | y2 |
|---|---|---|---|---|---|---|---|---|---|
| stk | ∩ | ∩ | Ø | Ø | Ø | Ø | Ø | Ø | Ø |
| top | ∩ | ∩ | Ø | Ø | Ø | Ø | Ø | Ø | Ø |
| funinvokes | Ø | Ø | ∩ | ∩ | ∩ | Ø | Ø | Ø | Ø |
| topInvok | Ø | Ø | ∩ | ∩ | ∩ | Ø | Ø | Ø | Ø |
| rawtime | Ø | Ø | ∩ | ∩ | ∩ | Ø | Ø | Ø | Ø |
| x1 | Ø | Ø | Ø | Ø | Ø | ∩ | ∩ | ∩ | ∩ |
| y1 | Ø | Ø | Ø | Ø | Ø | ∩ | ∩ | ∩ | ∩ |
| x2 | Ø | Ø | Ø | Ø | Ø | ∩ | ∩ | ∩ | ∩ |
| y2 | Ø | Ø | Ø | Ø | Ø | ∩ | ∩ | ∩ | ∩ |

In Table 3.1, a "Ø" means that the intersection of the slices is the empty set for the two data members and a "∩" means that there are some elements in the intersection of the slices. In other words, an intersection "∩" represents presence of an edge between the two data members in Table 3.1. From this table, we generate the DSG of the Class 1as shown in Figure 3.11.
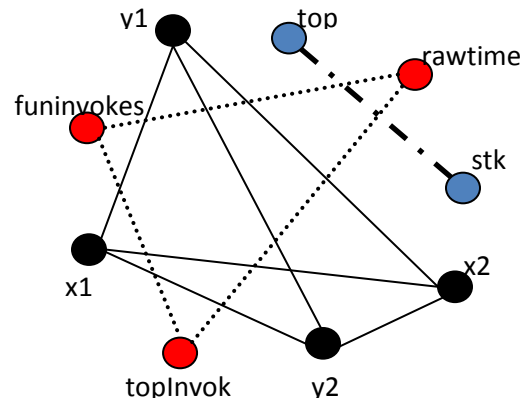
Figure 3.11 DSG of Class1

Each connected component in the DSG in Figure 3.11 is shown with a different line style. In this case our corresponding DSG has three connected components. Which means that the class we have analyzed has three distinct abstractions. Therefore, this class is not in the most cohesive form and it should be re-factored.

Our refactoring approach suggests creating a new class for each one of these three connected components. The first class will include the following data members: *funinvokes, rawtime*, and *topInvok* together with their corresponding slices. The second class will include the following data members: *top* and *stk* together with their corresponding slices. And lastly the third class will include *x1, x2, y1, y2* and their corresponding slices.

### 3.2.5.3 Refactoring and Handling Edge Cases in Class1

Although our research focus is not on the extraction part as stated earlier, to better explain the concept, we show the refactoring process for Class1 in Figure 3.12.

In Figure 3.12, statements between line numbers 124 and 128 in the code fragment before refactoring are in the same slice with respect to the criterion of C=(125, rawtime). On the other hand, statements between line numbers 129 and 139 in the code fragment before refactoring are

in the same slice with respect to the criterion of C=(129, top). As we see in the DSG of this class given in Figure 3.11, data member *rawtime* and data member *top* reside in different connected components. Therefore, these two slices should be in different classes created for the corresponding connected components in DSG.

Figure 3.12 shows that, method fun2_10() in class New2 is created corresponding to the slice with respect to the criterion of C=(125, rawtime) and method fun1_4() in class New1 is created corresponding to the slice with respect to the criterion of C=(129, top) in the code fragment before refactoring. Statements that construct these slices are then replaced with appropriate method calls as shown in Figure 3.12.

The example code fragment given in Figure 3.12 exhibits two of the edge cases described earlier: Case 1 and Case 3. At line number 137, we see a method call to the method ErrorInSize() in a control block. According to the explanation of Case 1, all the statements in the implementation of ErrorInSize() are automatically bounded to the statements in the slice where the method call resides. That means, a method will be created for the statements in the implementation of ErrorInSize() and that method call can be replaced with a method call to the method created corresponding to it. For example in Figure 3.12, method fun1_1() in class New1 is created corresponding to ErrorInSize() and the method call to ErrorInSize() is replaced with a method call to fun1_1() in the restructured version of the code.

| Code Fragment before Refactoring |
|---|
| 122 | int Pop() |
| 123 | { |
| 124 | time ( &rawtime ); |
| 125 | string t=string(ctime(&rawtime)); |
| 126 | string temp="Pop invoked: "; |
| 127 | temp+=t; |
| 128 | PushFunInvok(temp); |
| 129 | if (top > 0) |
| 130 | { |
| 131 | top--; |
| 132 | int temp_int=stk[top]; |
| 133 | return temp_int; |
| 134 | } |
| 135 | else |
| 136 | { |
| 137 | ErrorInSize(); |
| 138 | return -1; |
| 139 | } |
| 140 | } |

| Code Fragment after Refactoring |
|---|
| 62 | int Pop() |
| 63 | { |
| 64 | n2->fun2_10(); |
| 65 | return n1->fun1_4(); |
| 66 | } |
| 1 | class New2 |
| 2 | { |
| : | : |
| 89 | void fun2_10() |
| 90 | { |
| 91 | time ( &rawtime ); |
| 92 | string t=string(ctime(&rawtime)); |
| 93 | string temp="Pop invoked: "; |
| 94 | temp+=t; |
| 95 | fun2_8(temp); |
| 96 | } |
| : | : |
| 105 | }; |
| 1 | class New1 |
| 2 | { |
| : | : |
| 11 | void fun1_1() |
| 12 | { |
| 13 | cout<<"Index out of range!\n"; |
| 14 | cout<<"The Array has "<<top |
| 15 | <<" elements.\n"; |
| 16 | } |
| : | : |
| 31 | int fun1_4() |
| 32 | { |
| 33 | if (top > 0) |
| 34 | { |
| 35 | top--; |
| 36 | int temp_int=stk[top]; |
| 37 | return temp_int; |
| 38 | } |
| 39 | else |
| 40 | { |
| 41 | fun1_1(); |
| 42 | return -1; |
| 43 | } |
| 44 | } |
| : | : |
| 49 | }; |

Figure 3.12 Handling Edge Cases for Class1

The other edge case, Case 3, is observed at line 128 in the code fragment before refactoring with

a method call to *PushFunInvok(temp)*. According to the explanation of Case 3, when we analyze

that method we will see that all the statements are fully related based on our dependency conditions and that allows us to create a new method for the statements in the implementation of PushFunInvok(temp) and replace that method call with a method call to the corresponding method created in the restructured version of the code. As we see in Figure 3.12, method call to PushFunInvok(temp) is replaced with a method call to fun2_8(temp), which is the corresponding new method to PushFunInvok(temp) in the restructured version of the code.

### 3.2.6 Experimental Results for Token

Second class we chose for our experiment is called Token and its source code is given in [35]. We have used this class without any single change on its implementation in many of our projects for source code analysis. It is in fact well designed and has one certain task to accomplish: reading words from an attached file (usually source code files) or string based on some predefined rules on the word boundaries.
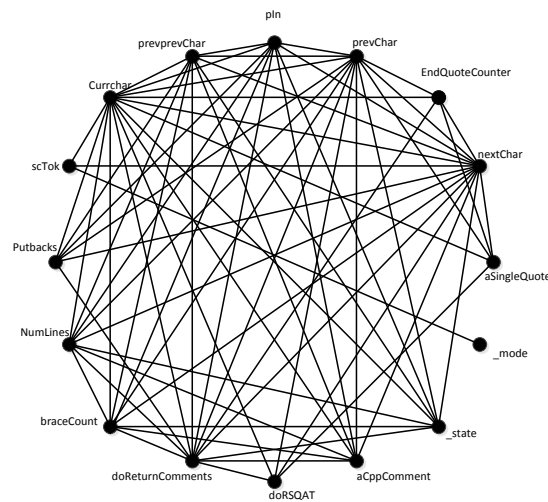


Figure 3.13 DSG of Token Class

From the DSG of the class Token given in Figure 3.13, we see that there is only one connected component. When the number of connected components is one, the class is in its most cohesive form and we do not propose any refactoring for a class holding this property.

A DSG with only one connected component implies a class with one abstraction. What this means is that all data members defined in this class are closely related and because of the common statements that affect the data members, refactoring this class will require a lot of work and any alteration such as class extraction will most likely worsen the design and quality of this code.

## 3.3 Summary

In this part of our research, we have proposed a new cohesion metric and an extract class refactoring technique for classes in object oriented environments using program slicing and graph theory. One can make a judgment on clarity of intent of the code using the metric we propose here. We aim to find out if a class is cohesive, handling one specific operation. We identify all program statements which constitute the operations in the same abstraction domain. When a class has more than one abstraction, this technique suggests extract class refactoring for generating more cohesive units based on this new cohesion metric.

Our approach is different from other related works in that we try to find statements that constitute the same abstraction in a class rather than regrouping existing components of a system.

The contributions of this part of our research can be summarized as follows:

- We have defined a new set of rules to detect extended program slices rather than using traditional definition of program slicing.

- We have defined a new graph, DSG that reveals information regarding the relatedness of the data members in a given class at the statement level.

- We have defined a new cohesion value for a class based on DSG and proposed refactoring if the class is not cohesive.

# Chapter 4

# Identification of Extract Method Opportunities

Long methods are usually difficult to read and comprehend due to the length and complexity of the code. As a result, maintenance can be time consuming and costly. One strategy to lower overall cost of software development for large systems is to produce smaller and less complex methods through extract method refactoring. This chapter presents our novel contribution towards identifying code fragments for method extraction based on the concept of placement tree and variable references.

## 4.1 Extract Method Refactoring

For large software systems, we try to build comprehensible, readable, well-structured, and simple code to minimize cost associated with the maintenance phase. If code fails to meet these goals, refactoring is one strategy to improve the internal structure of the code and thus its

maintainability. Refactoring is an umbrella term to describe any process that enhances the program's internal structure without changing its external behavior. One such process, Extract Method Refactoring, focuses on reducing the complexity of code by decomposing a large method into smaller ones. The re-factored methods are optimal, since they extend the lifetime of programs by making the reader better able to understand the purpose of each method [4,5].

Extract Method refactoring consists of two major activities: identification of the fragment of code to be extracted followed by the extraction of the identified code as a new function and replacement of the original fragment with a function call. The extraction itself is easily achieved using built in functionality within existing IDEs; however, the selection process is done manually. There is not a uniform identification strategy, and various heuristics or guidelines for selection of code fragments have been proposed in the literature [41,42,43,44]. Implementation of a selection process can prove challenging and the addition of visualization to these techniques is crucial for effective execution.

Programming Slicing is the most widely used refactoring technique. However, implementation can be tedious and time-consuming since this technique requires the user to manually select the slicing criterion. Furthermore, there is no a priori approach for choosing the slicing criterion. In this part of our research, we introduce a fully automated selection technique and placement tree visualization tool to detect candidate code fragments in large scale systems for extract method refactoring.

Our approach constructs a placement tree that contains variable reference counts for individual scopes within a given function. The placement tree is then visualized so that each node, which represents a scope on the placement tree, has a color associated with its dominant or most

referenced variable. The goal is to create functions with a single color placement tree or with one dominant variable leading to methods with good cohesion, i.e., with focus on a single task. Therefore, a node that does not match the color of its parent node is identified as a candidate code fragment to be extracted.

## 4.2 Placement Tree

Extract method refactoring activity, in this part of our research, starts with generating the placement tree for a given method to represent the hierarchically nested scopes. As explained in Chapter 2, placement tree is the term we use to refer to the tree of scopes in the input method. The root node of the placement tree represents the method itself and each child node of a specific node represents a scope that is included by their parents. For placement tree construction, we analyze the method to detect the scopes for different types of control blocks given in Table 4.1.

Table 4.1 Scope Types

| Type name | Start Line | End Line |
|---|---|---|
| Function | Line that includes function name | Line that includes corresponding closing braces |
| For | Line that includes the key word for this scope. The key word is basically the type name itself, e.g. "for, while, do, switch, if" | Line that includes corresponding closing brace to the one following the respective type name. |
| While | | |
| Do-While | | |
| Switch | | |
| If | | |
| If-else | Line that includes the key word "if" | Line that includes corresponding closing brace to the one following "else" keyword. |
| Anonymos | Line that includes open brace that does not have an attached type | Line that includes the closing brace corresponding the anonymous scope opening brace |

The basic idea behind this extract method refactoring technique is that each method should have one clear task to accomplish. What this implies is that the method and all scopes subsequently will include references to some variables to accomplish this task. Using the placement tree concept, we determine the variable which dominates the operation carried out in the method and in every node of the placement tree as well. In this sense, scopes that are dominated by different variables perform different sub-tasks in this operation and indicate possible extract method refactoring opportunities.

```
1  void closeIfsExpectingelse(){
2      if(TrackScopes->size()> 0 && IfsExpectingelse->size()<= 0){
3          closeAll();
4          return ;
5      }
6      while(TrackScopes->size()> 0 && IfsExpectingelse->size()> 0){
7          std::pair < std::string,char > temp = TrackScopes->pop();
8          if(temp.second == 'o'){
9              TrackScopes->push(temp);
10             break ;
11         }
12         CloseBrace();
13         while(TrackScopes->size()> 0){
14             TrackScopes->pop();
15             CloseBrace();
16         }
17     }
18 }
```
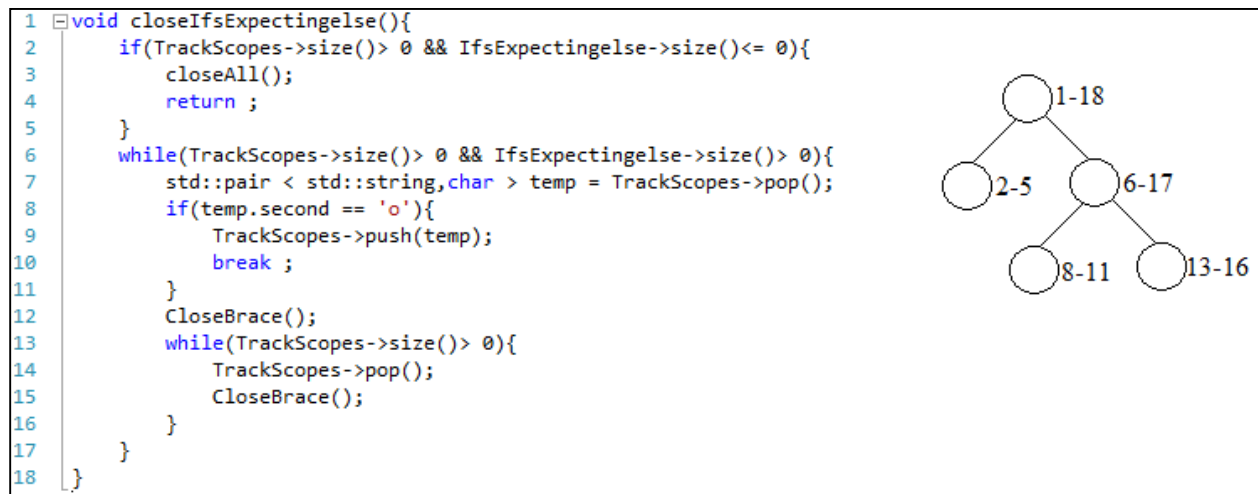
Figure 4.1 Example Method and Placement Tree

We expect that, a method or a scope in the given method that start with one dominant variable, continues the sub-tasks with the same dominant variable within their children scopes for better comprehension. Therefore, the hierarchical placement of scopes - the placement tree itself - helps us detect continuity of the dominant variables in a given method very effectively. An example method and its corresponding placement tree is shown in Figure 4.1.

## 4.3 Dominant Variable

After constructing the placement tree for the input method, this approach proceeds with identifying the dominant variable(s) in each scope. We propose a simple heuristic for selection of the dominant variables based on variable references. Variables with the highest reference counts are identified as the dominant variables for the given scope. Although this heuristic gives promising results and works effectively, improvements for selection of dominant variables remain as a future work of our study to consider the effect of multiple variables on the determination of cohesion.

Initially all variables referenced inside the given method are candidates to be the dominant variable of each scope. Let V(F) represent the set of all variable names that appear anywhere in the method. All global variables and data members of the class that the method under analysis belongs to, may be elements of V(F). Therefore, Long Method code defects can be detected in Object Oriented Programs, Procedural Programs or Legacy Code using the technique proposed here. Local variables declared in the method are also in the set V(F). In other words, all variables accessible from within the given method are possible elements of the set V(F), subject to whether they are referenced in the method at least once.

$$V(F)=\{ v_1, v_2, .., v_n \}$$

We have also defined several notations to express various properties of program statements, variables, scopes and of the method itself. We will use these notations in our explanations throughout this chapter. Table 4.2 shows these notations and their respective input and output values.

Table 4.2 Defined Properties

| Name | Input | Output | Use |
|------|-------|--------|-----|
| LN | A program statement, S | Line # of S | LN(S) |
| RC | A variable name, v; and a scope, B | # of references of v in B | RC(v, B) |
| SLN | A scope, B | Starting line # of B | SLN(B) |
| ELN | A scope, B | Ending line # of B | ELN(B) |

Let F represent the set of all scopes in the given method to be re-factored. And every scope in the given method is represented with a set of statements, B. Let the set V(S) represent the set of variables that are used for execution of statement S and let the set V(B) represent the set of all variables that appear in scope B at least once.

$$V(S) = \{v | v \text{ is used in } S\} \qquad (1)$$

$$B = \{S \,|\, SLN(B) \leq LN(S) \leq ELN(B)\} \qquad (2)$$

$$\forall S \in B: \nexists B': SLN(B) \leq SLN(B') \leq LN(S) \leq ELN(B') \qquad (3)$$

$$V(B) = \{v' \,|\, \exists S \in B: v' \in V(S)\} \qquad (4)$$

$$F = \{B \,|\, \forall S \in B, B' \in F, B \neq B': S \notin B'\} \qquad (5)$$

$$\forall S, B: V(S) \subseteq V(F) \text{ and } V(B) \subseteq V(F) \qquad (6)$$

From formulas 2 and 3, one can conclude that every statement belongs to only one scope or node in the placement tree. A statement S' is and can only be an element of one scope B that encloses statement S'; but S' is not considered as an element of an ancestor node of B, AB, at the same time.

After defining our scopes and elements of them, now we find the variable(s) that dominate(s) the computations in every scope. We determine the dominant variable(s) based on their respective reference counts in the subject scope. The reference count of a variable refers to the number of

times that variable appears in the given scope. Let D(B) represent the dominant variables in scope B,

$$D(B) \subseteq V(B) \; such \; that$$

$$\forall v \in V(B), v' \in D(B), v'' \in D(B) : \; RC(v', B) > RC(v, B) \; and \; RC(v', B) = RC(v'', B)$$

The set D(B) therefore includes only those variables whose reference counts are the highest in block B. Every dominant variable name in the whole method is assigned a unique color to represent its power, e.g., reference count relative to the other variable reference counts in the scope that it dominates. In our visual representation of the code, these colors are used to distinguish the nodes that may be extracted.

Although it is usually very rare, we might have more than one variable as candidates to be dominant variables or a given scope may include no variable reference. When the number of dominant variables is zero for the scope B, that is |D(B)|=0, in the placement tree that node is represented with the color of black. On the other hand, when the number of dominant variables is greater than one for the scope B, that is |D(B)|>1, we propose two approaches for this case that we call *Parent Protection* and *Sibling Collaboration*. The idea behind these approaches is to keep those blocks that are dominated by the same variable together as much as possible to come up with larger code fragments as candidates for extract method refactoring.

### 4.3.1 Parent Protection

When scope B is dominated by more than one variable, the Parent Protection approach checks the dominant variable of B's parent node, BP. If the dominant variable of the parent node is an element of D(B), then this dominant variable is assigned to be the dominant variable of node B.

Otherwise, one of the dominant variable from D(B) is randomly selected to be the dominant variable of this scope. Let dB and dBP represent the dominant variables of the nodes B and BP respectively.

$$if\ dBP \in D(B) \qquad\qquad dB = dBP,$$

$$if\ dBP \notin D(B) \qquad\qquad dB = v, v \in D(B)$$

### 4.3.2 Sibling Collaboration

When scope B is dominated by more than one variable, in the Sibling Collaboration approach, dominant variables of its sibling nodes, SB and SA, are checked. SB and SA are those nodes that come right before, and after scope B in source code (left and right nodes in placement tree respectively). The dominant variable of the sibling SB is evaluated first. If the dominant variable of sibling node SB is an element of D(B), then we say that sibling nodes, B and SB, collaborate, meaning they share in executing the operations of a cohesive activity. In this case, the dominant variable of SB is assigned to be the dominant variable of node B.

If the dominant variable of the sibling node SB is not an element of D(B), dominant variable(s) of the other sibling node, SA, is evaluated. If SA has only one dominant variable, that is |D(SA)|=1, and this dominant variable of SA is an element of D(B), then sibling nodes, BP and SA are collaborators, and this dominant variable is assigned to be the dominant variable of node B as well.

If SA is dominated by more than one variable, that is |D(SA)|>1, a random variable from D(B)∩ D(SA) is chosen and assigned to be the dominant variable of both B and SA. If |D(B)∩D(SA)|=0, a random variable from D(B) is chosen and assigned to be the dominant

variable of node B. Let dB, dSB and dSA represent the dominant variables of the nodes B, SB and SA respectively.

$$if \ dSB \ \in \ D(B) \qquad\qquad dB = dSB,$$

$$if \ dSB \ \notin \ D(B) and \ |D(B) \cap D(SA)| > 0 \qquad dB = \ dSA = v, v \in D(B) \cap D(SA)$$

$$if \ dSB \ \notin \ D(B) \ and \ |D(B) \cap D(SA)| = 0 \qquad dB = v, v \in D(B)$$

Adopting one of these two approaches, only one dominant variable is assigned to each node in our placement tree. Usually, this variable will be the one that dominates in the computation of that scope. The following section offers an explanatory example and discusses how we select scopes to be extracted.

## 4.4 Identifying Candidate Code Fragments

We have implemented a tool to visualize the placement tree of a given method based on the variables that dominate the computations in every node. This tool visualizes the placement tree as nested boxes, referred to as a *tree-map*, to effectively present hierarchical placement of the control blocks inside a given method [47]. Each box in our visualization corresponds to a scope in the method, the outermost scope being the method itself.

Each box in this visualization has two attributes that show properties of the corresponding scope: color and size. Boxes in the *tree-map-visualization* of placement tree are colored according to the dominant variable of their corresponding control blocks in the method. Therefore, two boxes corresponding to control blocks which are dominated by the same variable, are given the same color. The second attribute of a box in *tree-map*, size, is determined based on the number of statements in the scope represented by this box.

Our visualization tool accepts an XML file as an input generated by our static analysis tool. Thus, we first analyze the input method and generate an XML file that represents the placement tree for the method. Analysis and visualization of the scopes involve the following computational steps:

1) Put appropriate "{", "}" to mark scopes and place every statement in consecutive separate lines with proper indentation.

2) Find scopes and reference counts of every variable in each scope and generate in-memory representation of placement tree

3) Write the tree into an XML file

4) Read the XML file and find dominant variables

5) Assign a distinct color for each dominant variable and visualize refactoring opportunities.

Figure 4.2 below shows the activity diagrams for analysis and visualization phases in this extract method refactoring technique. Notice that refactoring with this technique is a cyclic process that involves three main phases as shown in Figure 4.2: analyze/visualize, observe refactoring suggestions, and carry out the suggested refactoring. Therefore, by applying a refactoring suggested by this tool, the user will extract one or more methods, producing a new version of the code (re-factored version). Following this, using the re-factored code as input, user may seek additional extract method opportunities on the re-factored version of the code.
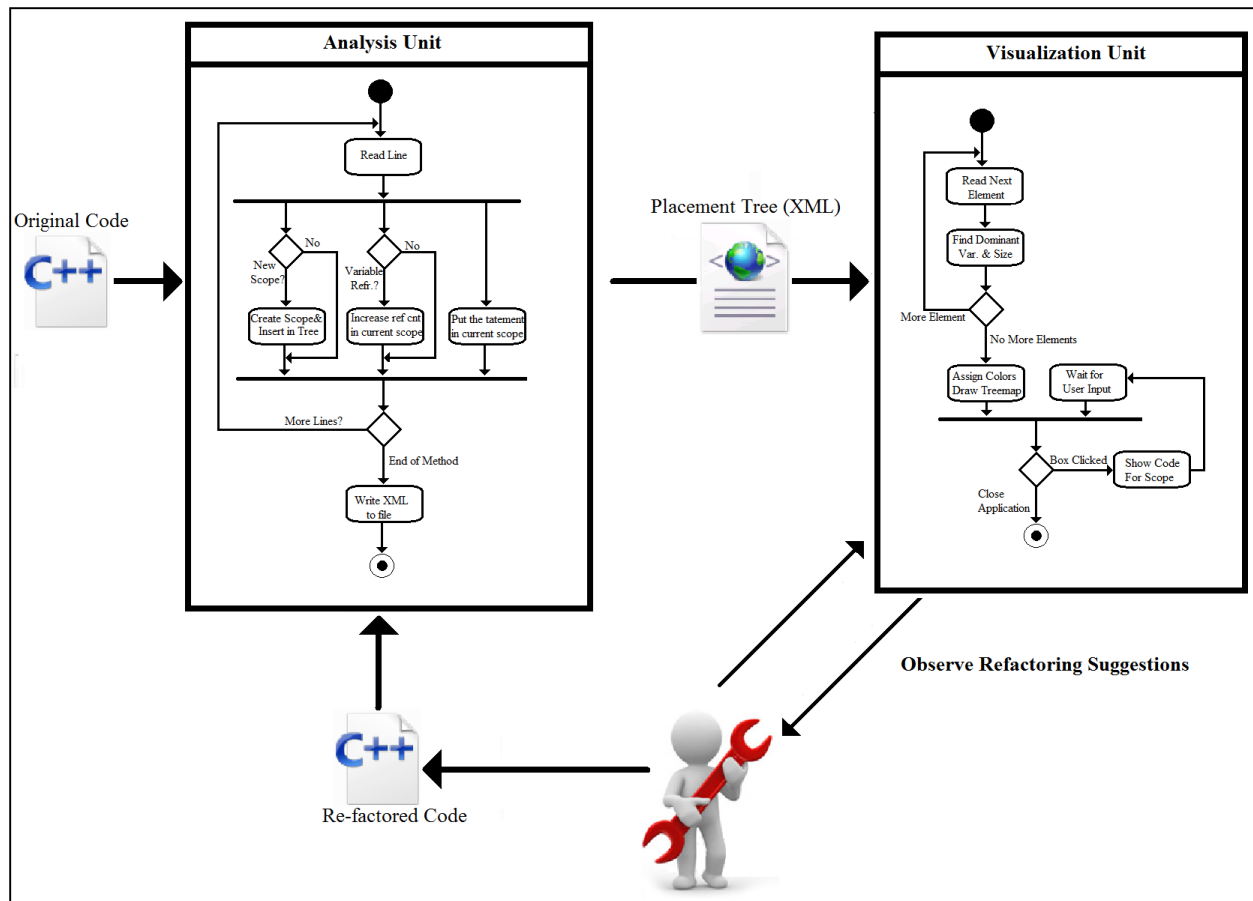
Figure 4.2 Activity Diagram for Method refactoring

There are two types of scopes that we suggest to be extracted from original code as new methods.

1) Large code fragments with a color different from parent's color. In Figure 4.3, we show an example for this case.

2) Consecutive sibling nodes with the same color. In Figure 4.4, we show an example for this case

Our refactoring suggestions aim to generate methods with minimum color diversity. We suggest to extract first the out most scopes with a color different from their parent's color. After refactoring, resulting code should be analyzed again for further refactoring until possibly all generated methods and the original method have only one dominant variable for every scope.

Therefore; the resulting code after refactoring will have methods that handle only one smaller and less complex task. Figure 4.3 and Figure 4.4 show some of the possible scopes that suit our refactoring suggestions for demonstration.
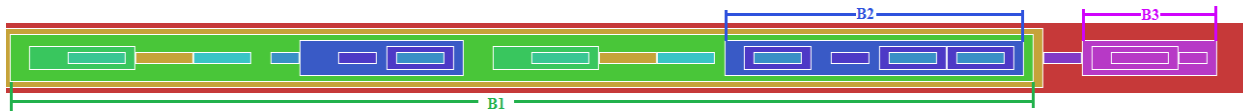

Figure 4.3 Refactoring Suggestion Example 1
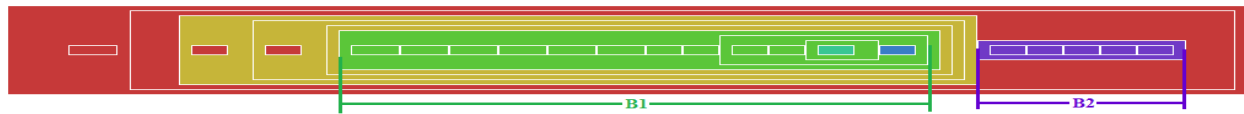

Figure 4.4 Refactoring Suggestion Example 2

## 4.5 Extraction of Code Fragments

After determining the code fragments for refactoring using our tools, developers extract those fragments as new methods and replace the fragments with function calls to the new methods. Once the code fragments for refactoring are identified, extracting them is usually trivial except a few points that require attention. This section explains how the extracting process should be carried out considering some important cases that, without careful handling, might cause compilation errors or alteration in the behavior of the system.

### 4.5.1 Parameters of Extracted Methods

Analysis and visualization tools introduced in this research have been tested on several methods with different sizes. Following this, extract method refactoring is applied on the identified code fragments. As mentioned earlier, main motivation of this work is to effectively detect possible code fragments for extraction as new methods. Number of arguments that need to be passed to the extracted methods, has not been considered for this part of our research.

C++ is one of the most difficult programming languages for static code analysis because of its complex syntax. For this reason, methods that are written in C++ programming language are chosen for experiments. In C++ language, there are various ways of passing arguments to methods. By default, arguments to methods are passed by value. When an argument is passed to a method by value, changes that the method does on this argument never affect the value of the argument in the calling method. C++ also provides an option to pass arguments by reference. A reference to a variable is simply an alias for that variable. When an argument is passed by reference, the changes made on the argument within the method are also reflected to the calling method.

During the execution of refactoring suggested by our tools, one can pass all arguments by reference. Yet, if a local variable is never used after the code fragment to be extracted, this variable can be passed to the new method by value. Variables, that have been used after the fragment to be extracted in the original method, need to be passed to the extracted method by reference to preserve the behavior of the program. Therefore, we suggest to use "pass by reference" whenever the programming language allows, to simplify the process of extract method refactoring because of the following advantages of this approach.

- Passing arguments by reference, compared to pointers, has exactly the same syntax as the original name for a variable. Therefore, passing arguments by reference, prevents compilation error due to syntax errors and simplifies the extraction process.

- Passing arguments by reference, compared to "pass by value", creates an alias for the variable to be passed as an argument. Therefore, passing arguments by reference, assures that everything that has been done in the new method preserves the behavior of the original system.

- When arguments are passed by value, the only way to return a value to the caller is through the return value of the method. Furthermore a method can only return one value in programming languages that we are interested in. If a fragment is modifying values of more than one variable, we will not be able to return all the modified values to the caller method without any change on the syntax. Unlike "pass by value", when we pass the arguments by reference, we no longer have to worry about the return values from methods that we generate, as all the modifications are automatically reflected to the caller method.

- When arguments are passed by value, a copy of the argument is passed to the method. In contrast to this, passing arguments by reference does not make a copy of the arguments, and therefore it is usually faster.

### 4.5.2 Return Values From Extracted Method

As stated earlier, IDEs like Eclipse and Visual Studio 2012 support extraction of a set of preselected statements as a new method, though, they have some limitations when extracting certain types of code. First of all, Visual studio does not even include a refactoring menu for C++ projects. Eclipse, on the other hand, requires the selected code fragments not to include any return statement. Visual Studio, similarly, puts some limitation on the code with return statements for C# programs. When selection contains return statements, all paths are expected to be terminated by a return statement too. Refactoring preconditions that selected code fragments need to fulfill are discussed in [73] in more detail.

In this section, we explain how we suggest to handle the situation where a code fragment identified by our tool contains return statements. To avoid the aforementioned limitations, we explain how this extraction can be carried out manually. Unlike Visual studio and other such

tools, we do not expect all paths in the identified code fragment to be terminated by a return statement. We evaluate the fragments with return key word in two different ways depending on whether the return statements do return a value.

### 4.5.2.1 Return without a Value

In Figure 4.5, a fragment of code with a return statement (from line 19 to line 29), identified by our tool, is shown. This code fragment includes a possible exit point, which is the return statement at line 27, for the method that it resides in. This statement, as seen in Figure 4.5, exits the method without returning a value to its caller.

When we have a return statement without returning an object, we suggest all the return statements in the extracted method be replaced with "return true". And at the end of the method we place an additional statement that return the value of "false". This way, we can easily determine whether the caller method (original method after refactoring) should exit or continue with the execution of the next statement after this method call. Figure 4.6 shows the extracted method for this identified code fragment. This method returns the value of "true" from all the places where a return statement existed in the original version; and at the end of the method the value of "false" is returned in this new method.

```
15              int _for = tc.find("for");
16              int _eq = tc.find("=");
17              int _sc = tc.find(";");
18              int StartIndex ;
19              if(_eq < _sc && _eq < tc.length()){
20                  StartIndex = _eq ;
21              }
22              else {
23                  if(_sc < _eq && _sc < tc.length()){
24                      StartIndex = _sc ;
25                  }
26                  else {
27                      return ;
28                  }
29              }
30              if(StartIndex == _for+3){
31                  return ;
32              }
33              for(int i = StartIndex-1 ; i >= 0 ; i--){
34                  if(! isSpecialChar(tc[i])){
35                      std::string type :
```

Figure 4.5 Code Fragment with Return Statement

After extracting the method, the identified code fragment in the original method is replaced with a method call to this new method. The function call in the original method is shown in Figure 4.7.

```
1  bool FindStartIndex(int &StartIndex, int& _eq, int& _sc, ITokCollection & tc)
2  {
3    if(_eq < _sc && _eq < tc.length()){
4          StartIndex = _eq ;
5      }
6      else {
7          if(_sc < _eq && _sc < tc.length()){
8              StartIndex = _sc ;
9          }
10         else {
11             return true;
12         }
13     }
14   return false;
15 }
```

Figure 4.6 Extracted Method with Return Statement

As shown in Figure 4.7, the re-factored version of the original method exits based on the return value of the extracted method.

```
134        int _sc = tc.find( ; );
135        int StartIndex ;
136        //
137        if(FindStartIndex(StartIndex,_eq,_sc,tc))
138            return;
139        //
```

Figure 4.7 Method Call with Return

### 4.5.2.2 Return with a Value

Extracting code fragments that contain return statements is one of the greatest challenges in method refactoring. This section of our research specifically targets a systematic way to handle the problem where only some of the paths in the identified code fragments are terminated by a return statement. Figure 4.8 shows an example of such fragment identified using our analysis and visualization tools. Users of our technique and tool are advised to carry out the extraction process as described here for these exceptions. Note that experiencing any of these cases at any point in refactoring does not hinder execution of other method refactoring opportunities suggested by our tool. Therefore, one may prefer not to carry out a refactoring suggested as a candidate by our tool that exhibits these special cases and continue with the remaining refactoring opportunities.

To extract such code fragments successfully, a new Boolean variable is introduced that we call "return control" variable. In the original method, return control variable, rc, is defined assigning to it the value of "false". When the identified code fragment is extracted as a new method in this particular case, an extra argument of Boolean reference type is passed to this method together with the other parameters that the method normally requires. This new extra parameter helps us evaluate whether the new extracted method exits with returning a value.
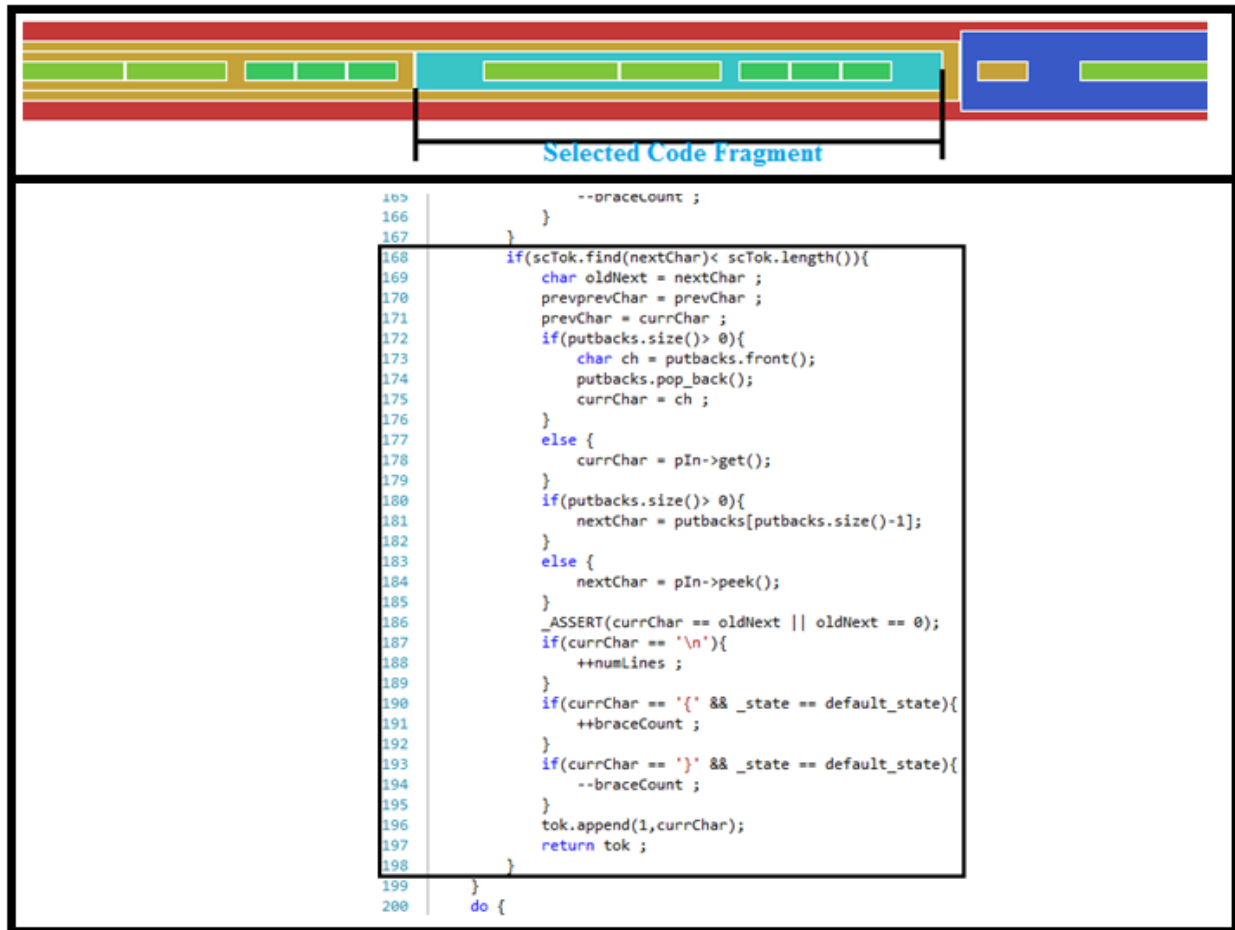
Figure 4.8 Code Fragment Returning Object Value

In the identified code fragment, a statement "rc=true;" is placed before every return statement. Another statement is also placed at the end of the extracted method to return an arbitrary value of the return type of the method. After invocation of the extracted method, depending on the value of rc, the original method returns the value returned by the extracted method. If rc is still holding the value of false after invocation of the extracted method, execution continues with the next statement inside the original method. Method call to the extracted method after refactoring is shown in Figure 4.9 together with the extracted method itself.

Figure 4.9 Original Method after Refactoring and Extracted Method

### 4.5.2.3 Bound Blocks

There are two key words the presence of which precludes extraction of the code fragments that they reside in. These key words are "continue" and "break" that respectively carries a loop to next iteration or terminates the loop. If a code fragment contains one of these two key words and the corresponding loop is not included in the fragment, then this code fragment cannot be considered for extract method refactoring. In other words, if a code fragment or node in the placement tree contains one of the key words, "continue" and "break", then this node is bound to its ancestor node that represents the loop associated with these keywords.

```
590    int tri=1;
591    for(int j=i+1; j<tc.length(); j++)
592    {
593        if(tc[j]==">")
594        {
595            tri--;
596        }
597        if(tc[j]=="<")
598        {
599            tri++;
600        }
601        if(tri==0)
602        {
603            i=j;
604            break;
605        }
606    }
```

Figure 4.10 Bound Blocks Example

In Figure 4.10, an example code fragment that cannot be considered for extraction as a new method is shown due to the presence of key word "break". The code fragment between lines 601 and 605 is bound to the code fragment between line numbers 591 and 606. Whenever two such scopes are bound, they have to be moved together in case of refactoring to protect syntax and behavior of the program.

## 4.6 Experimental Results

We have run our analysis and visualization tools on several methods from different systems. Identified code fragments in these methods are then extracted as new methods following the process explained in Section 4.5. Throughout the experiments, the parent protection approach explained in Section 4.3 is adopted. Some of the programs and restructured versions used in experiments can be found in appendix.

### 4.6.1 Experiments from Our Analysis Tool

We first applied this technique to a method from our analysis tool that we thought, may need refactoring. This method basically analyses a statement to find variable declaration and references in the statement. Figure 4.11 shows the placement tree of the original version of the analyzed method.
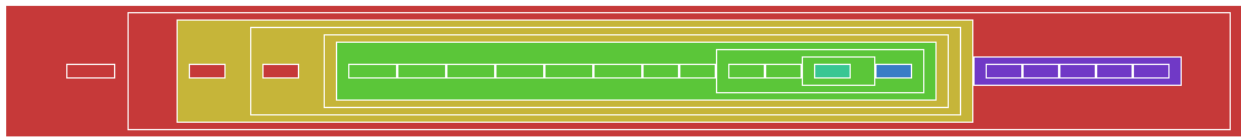


Figure 4.11 Experiment 1

We extracted three new methods and ran our tool again after refactoring. Figure 4.12 shows the placement trees of the methods we generated. During refactoring, code fragments with small lengths (usually two lines of codes) were ignored, as our main focus is on extracting large code fragments.
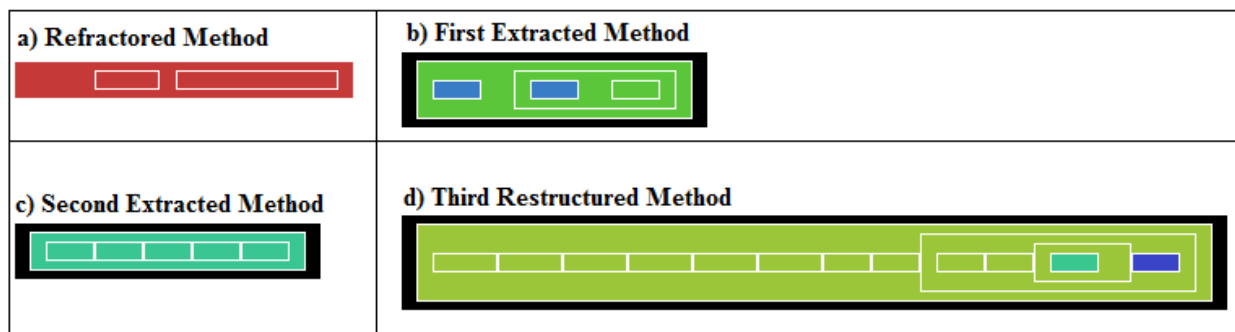


Figure 4.12 Refactoring Result for Experiment 1

For all the test cases that we used from our analysis tool, we could easily come up with meaningful names for the extracted methods that reflect their respective purposes. This demonstrates that our approach, with a high probability, will identify fragments that have a distinct task in the larger operation of the whole method based on the heuristic we chose. That is,

suggested refactoring approach successfully identifies candidate code fragments that actually should have been separate methods.

### 4.6.2 Experiments from Other Research and Open Source Projects

One method used for this experiment comes from code in a research project written by a group with which we occasionally collaborate. This method implements part of a reconstruction process of medical images obtained using cone beam and/or parallel beam collimators. The original function before any processing or refactoring has nearly 400 lines of codes with comments and white spaces. Figure 4.13 shows a portion of placement tree for the original version of this method.
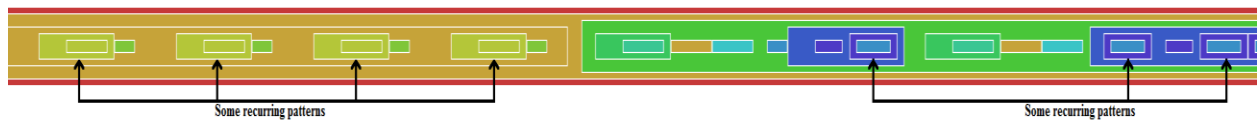


Figure 4.13 Experiment 2

We extracted nine methods and ran our tool again on these methods after refactoring. Figure 4.14 shows the placement trees for all of these methods after refactoring. The original method after refactoring includes less than 40 lines of code. This improves the readability of the code dramatically and makes the code more comprehensible reducing its complexity. Hence, the overall maintainability of the whole system is improved as the developer has the chance to work on smaller and less complex methods after refactoring.
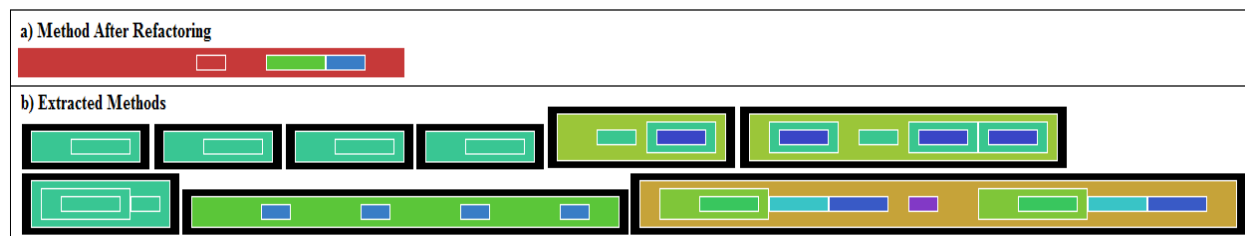


Figure 4.14 Refactoring Results for Experiment 2

The second method used for our experiment is one of the longest methods from this research project with nearly 4000 lines of codes with comments and white spaces. Our tool clearly detects many code fragments as extract method refactoring candidates as shown in Figure 4.15 and one can easily observe these candidate code fragments for extraction through this visual representation. Although Figure 4.15 shows just a portion of the placement tree for this method, we should note that, other parts of the placement tree are not any less diverse in terms of colors of nodes or their dominant variables.
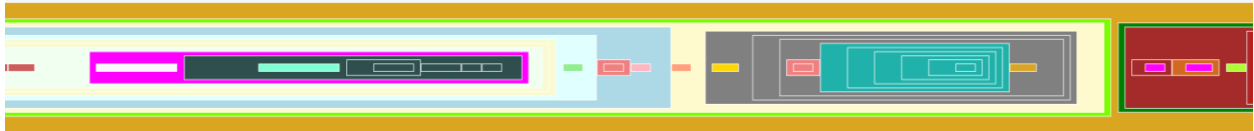


Figure 4.15 Experiment 3

Another software that we used for our experiments is called Notepad++. Notepad++ is an open source code editor and Notepad replacement that supports several programming languages and natural languages [48]. An analyzed method, *feedGUIParameters*, has more than 800 lines of code. Figure 4.16 shows a part of the placement tree for this method. As shown in Figure 4.16, our tool was able to indentify large code fragments that are candidates for extract method refactoring successfully in this software as well.
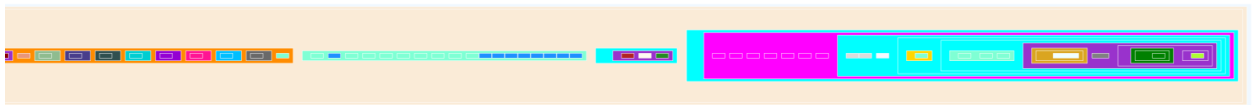


Figure 4.16 Experiment 4

## 4.7 Summary

In this part of our research, we mainly focused on identification of code fragments for extract method refactoring. Our identification process, as stated earlier, is based on placement trees and

variable reference counts in each node of the placement tree. This approach is straightforward to implement and it works effectively in real software systems as shown in the experiments.

In this work, initially we did not target refactoring to reduce the total number of statements in systems by detecting and removing duplicated code. Yet, our visualization reveals that variable reference counts can also be used for this purpose. As shown in Figure 4.13, we have encountered quite a lot recurring patterns in our placement trees. Such recurring patterns can be found in Figure 4.14 as well in the placement trees of the new methods. When we compared the corresponding code fragments of these nodes with the same pattern, we realized that some of these code fragments were identical to each other, while for some, there was a tremendous similarity between corresponding code fragments. This shows that our approach with some improvement can be used to detect duplicated code as well in a given large method.

The main contributions of this part of our research can be summarized as follows:

1) We introduced a novel technique to detect code fragments or blocks that handle similar operations.

2)  Our technique is based on variable reference counts which is straightforward to detect even in complex languages such as C++. We introduced two approaches: Parent Protection and Sibling Collaboration to improve refactoring suggestion results in terms of size of the candidate code fragments.

3) Our technique is supported by analysis and visualization tools, two crucial elements for an effective refactoring technique. Our final visual representation is very clear using color coded boxes with different sizes and can be used on real large software effectively as shown in the experiments.

# Chapter 5

## Extract Method Opportunities Using Hammock Graphs

As stated earlier, for many cases, problems in large software systems stem from large methods [3]. In Chapter 4, we have introduced a novel technique to tackle the problem of long methods by decomposing them into smaller ones based on the placement tree and variable references. When a code fragment or a scope is extracted as a separate method, all the local variables used inside but declared outside of the scope need to be passed as arguments. Sometimes, this may cause a method to have an undesirably long parameter list. A long parameter list usually increases the complexity of a method and makes it difficult to maintain and to comprehend its intent [5]. To overcome this issue, we introduce a novel extract method refactoring technique to confine code regions for extraction using hammock graphs, based on the number of arguments the resulting methods would require.

## 5.1 Some Limitations for Extract Method Refactoring

Automating the selection process is a very challenging task in extract method refactoring. Once an appropriate segment of code is selected by the user, extraction of this code can be very trivial using renowned tools such as Resharper [79] and Refactor Pro [80] or popular IDEs such as Eclipse [81] and Visual Studio [82]. Due to the limitations imposed by these tools, users have to inspect the code manually for selection of candidate code fragments for extraction. We should also note that most of refactoring tools and IDEs do not support refactoring C++ code even for only the extraction part.

When selecting a code fragment, users have to carefully analyze the code to make sure that their selection is extractable. In other words, extraction of the selected part must preserve not only the semantics of the program, but its syntactic correctness as well by meeting some refactoring preconditions [73]. Two important points regarding variable declaration and use that need attention when using refactoring with Eclipse are listed in the following link [58] as follows:

- *Selected block references a local type declared outside the selection: A local type declaration is not part of the selection but is referenced by one of the statements selected for extraction. Either extend the selection so that it includes the local type declaration or reduce the selection so that no reference to the local type declaration is selected.*

- *A local type declared in the selected block is referenced outside the selection: The selection covers a local type declaration but the type is also referenced outside the selected statements. Either extend the selection such that it includes all references to the local type or reduce the selection such that the local type declaration isn't selected.*

What this means is that, users need to find all variable declarations and their references to make sure that they do not violate rules imposed by the tool they are using. This may be feasible for small methods, but for large methods with thousands of lines of code, manual selection and investigation of code regions for such violations will not be very practical.

## 5.2  Hammock Graph for Method Extraction

When the use of hammock graphs is analyzed in literature, one realizes that hammock graphs have been defined on control flow graphs mostly for structuring or optimization purposes by eliminating unstructured branches and jumps. Yet, when we consider the restrictions mentioned in section 5.1 on selecting code fragments for method extraction, we can see how effectively hammock graphs may work for that purpose, when they are defined based on variable declarations and uses. Because for their most fundamental property, having a single entry and exit points, we have discovered that hammock graph is an effective tool to automate the process of selecting code regions that are suitable to form a method.

In this part of our research, we focus on methods in two different ways. First, a method always has a single entry and single exit point which are respectively the first statement in the method definition and the next statement to be executed in the caller context after the method invocation. Second, a method usually starts with one or more variable declarations and carries out a set of operations around these variables and ends with or without returning one of the variables. By defining hammock graphs on variable declarations and using that together with their interactions with the control scopes, the main contribution of this part of our research is to automatically select appropriate code regions that do not violate any syntactic or semantic rules related to

variables for method extraction, providing a visual aid and a control over the number of parameter resulting methods will take.

Before proceeding further, we shall restate the definition and properties of a hammock graphs given in Chapter 2. Hammock graphs are defined as a special kind of graph with one entry and one exit points. A definition of hammock graph is given in [57] as follows:

*Definition: Let G be a control flow graph for program P. A hammock H is an induced sub-graph of G with a distinguished node V in H called the entry node and a distinguished node W not in H called the exit node such that*

    *(1) All edges from (G - H) to H go to V.*

    *(2) All edges from H to (G - H) go to W.*

Researchers define hammock sub-graphs on control flow graphs for different purposes such as optimization, eliminating unstructured jumps and goto statements [13, 53]. The following example of a hammock is given in [13] to demonstrate its properties.



```
1:      S0
2:      if (X) then
3:10       S1
4:      else
5:         S2
6:         if (Y) goto 10
7:      endif
8:      S3
```
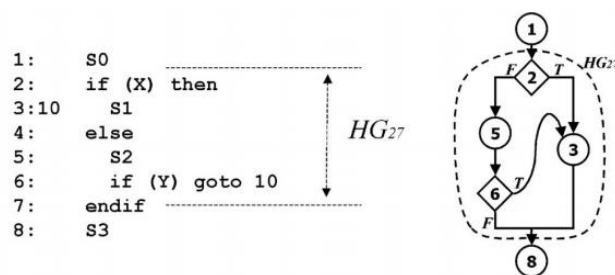
Figure 5.1 Example  program, its control flow graph and a hammock graph.

## 5.3 Construction of Hammocks

The control flow graphs of many methods in a program inherently have the properties of hammock graphs. All methods have one entry point which is the first program statement inside a method to be executed upon calling the method. Most also have one exit point which is the program statement in the calling method to be executed upon exiting the method.

In this part of our research, we aim to decompose large methods creating smaller ones with one clear intention providing an opportunity to dynamically observe only those code fragments that, when extracted as a method, would take a pre-determined number of parameters or less. We suggest that methods that hold this simple yet very powerful property, in most cases, should have declared at least one local variable and have references to that variable in its definition towards achieving its intention. Therefore, we introduce a technique to segregate code fragments that declare and use some local variables to implement a specific operation. For that, our technique relies on a graph that we generate from variable declarations and uses as well as control blocks rather that the control flow graph of the method.

Our technique proceeds in following steps:

1) Generate the initial graph of variable declarations and references together with  control blocks

2) Convert all variable declaration-use sub-graphs into hammocks

3) For each hammock find the number of variables referenced in the hammock

4) Visualize the candidates based on a selected number of parameters dynamically

5) Observe refactoring opportunities, re-factor the code and start over if necessary

### *5.3.1 Generating the Initial Graph*

The initial graph we generate has two types of edges: reference edges and control edges. Reference edges are due to local variable declarations and references. For each local variable declared in the method, there is an edge in the graph that goes from the line where variable is declared to the line that includes the last reference of that variable in the method. We refer to the code fragment confined by this edge as the variable span of this particular variable.

Control edges are due to the control statements such as for, while and if statements. For each such statement, the graph includes an edge that goes from the starting line of the control block to its end. Note that if an if control statement has an associated else statement, then the edge representing this control block goes until the end of the else statement.

A more formal definition of the initial graph G is given below.

- *G= (V, E) is a directed graph such that V is the set of program statements in a given method representing vertices in the graph and E representing edges in the graph, includes all reference edges and control edges.*

- *Let the set L, line number D(l), and line number LR(l) represent the set of all local variables declared in the given method, the line number where variable l is declared and the line number of the statement that includes the last reference of the variable l respectively.*

- *Therefore:*

$$\forall l \in L: D(l)LR(l) \in E$$

- *Furthermore, let the set C, line number S(c), and line number E(c) represent the set of all control statements in the given method, the line number where the*

> *control block starts and the line number where the control block ends for the*
>
> *control statement c  respectively.*

- *Therefore:*

$$\forall c \in C : S(c)E(c) \in E$$

```
1  ⊟void example(){
2        int a=0;
3        int i;
4        for(i=0; i<10; i++){
5            a+=i*i;
6            std::cout<<a<<"\n";
7        }
8        int b;
9        b=a;
10       std::cout<<"Current value of b="<<b<<"\n";
11       if(b>100){
12           std::cout<<"b is greater than 100\n";
13       }
14       else{
15           std::cout<<"b is not greater than 100\n";
16       }
17   └}
```
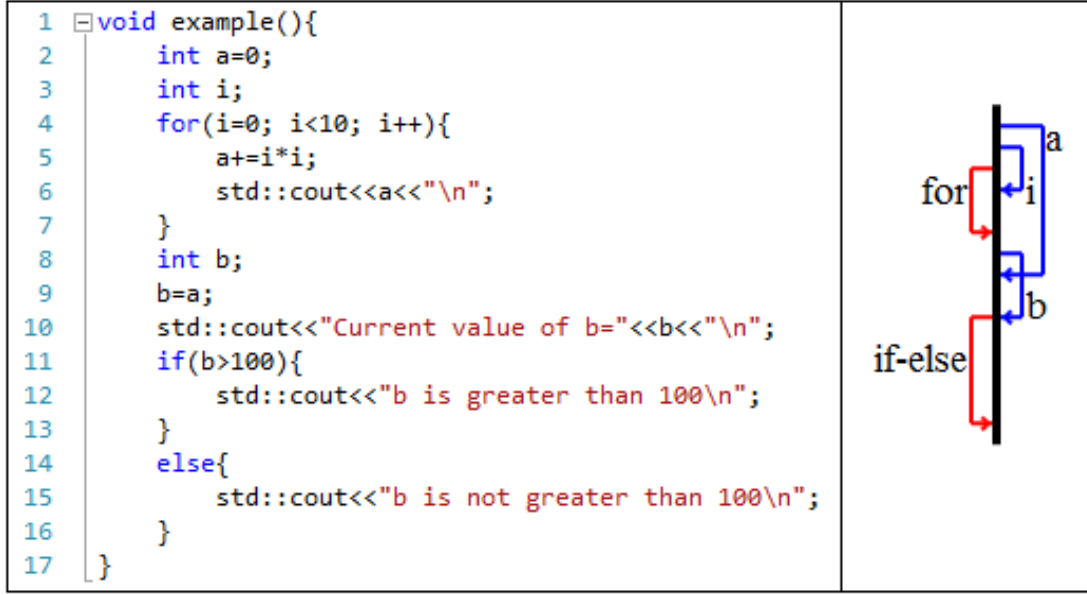
Figure 5.2 Example Initial Graph

In Figure 5.2, an example method and its corresponding initial graph are shown. Red-colored edges represent control blocks while blue-colored ones show the reference edges that stem from variable declarations and uses. For the sake of clarity, we labeled edges with the variable name or the type of control statement that they represent for this example method.

### 5.3.2 Generating Hammocks on the Initial Graph

The main goal of this part of our research is to come up with an effective way to observe all variable spans as candidates for method extraction with a selected number of parameters or less. Most of the time, a variable span denoted in the initial graph cannot be extracted as a method because of two reasons:

1) Extraction of a variable span from the initial graph may split a control block.

2) Extraction of a variable span from the initial graph into a new method may move the declaration of another local variable to a new scope leaving references of that variable in the original method.

For example, in Figure 5.2, the variable span for variable *i* cannot be extracted as a new method because doing so would split a control block e.g. the *for* loop. Likewise, the variable span for variable *a* cannot be extracted as a new method, as doing so, while removing the declaration of variable *b*, leaves some of its references in the original method.

To resolve this dilemma, we convert the initial graph of the given method into another graph where all reference edges are hammocks. This conversion eliminates any interaction between variable spans or reference edges and control edges that would otherwise preclude these variable spans from being extracted as new methods.

This conversion is done in two steps by comparing first all reference edges with each other and next each reference edge with every control edge.

That is, let *l1* and *l2* be two distinct local variables in the given method. If the variable span for variable *l1* contains the declaration of variable *l2* excluding some of its references, the variable span for variable *l1* is extended to cover the entire code fragment denoted by the variable span of *l2*.

$$\forall l1, l2 \in L:$$

$$if\ D(l1) \leq D(l2) \leq LR(l1)\ and\ LR(l1) \leq LR(l2)$$

$$then\ LR(l1) = LR(l2)$$

| Example Method | Initial Graph | Extended Graph |
|---|---|---|

```
1  □void example(){
2      int a=0;
3      int i;
4      for(i=0; i<10; i++){
5          a+=i*i;
6          std::cout<<a<<"\n";
7      }
8      int b;
9      b=a;
10     std::cout<<"Current value of b="<<b<<"\n";
11     if(b>100){
12         std::cout<<"b is greater than 100\n";
13     }
14     else{
15         std::cout<<"b is not greater than 100\n";
16     }
17 }
```
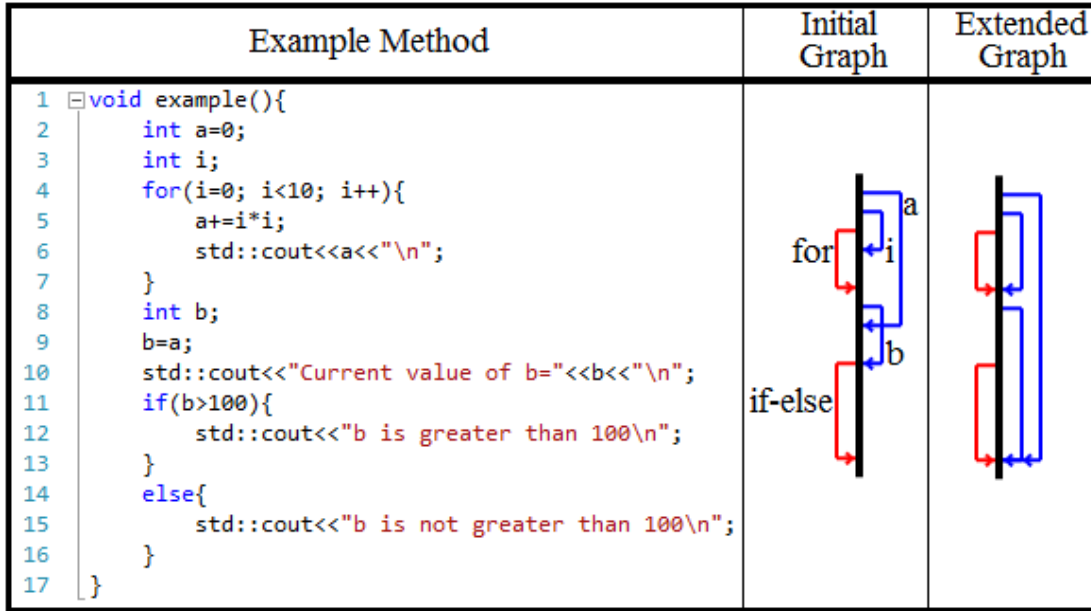
Figure 5.3 Example Method, Initial Graph and Extended Graph

Furthermore, let $l$ and $c$ represent a local variable and a control block respectively in a given method. If the declaration of the variable $l$ is outside of the control block $c$ and its variable span partially contains this control block, variable span for $l$ is extended to cover the entire control block.

$$\forall l \in L \; and \; \forall c \in C:$$

$$if \; D(l) \leq S(c) \leq LR(l) \leq E(c) \; then \; LR(l) = E(c)$$

Figure 5.3 shows the same example method presented in Figure 5.2 together with its extended graph after these conversions.

## 5.4 Restructuring Opportunities

In the extended graph that we generate, each reference edge represents a hammock having one entry and one exit point. Therefore they are extractable as a new method.

The main contribution of this technique over our earlier work presented in Chapter 4 is the fact that one can observe all possible extract method refactoring opportunities with a selected number of arguments dynamically through a visual representation. The number of arguments that a candidate code fragment (or reference edge on the extended graph) requires is equal to the number of local variables that are referenced within this code fragment but declared outside of it.
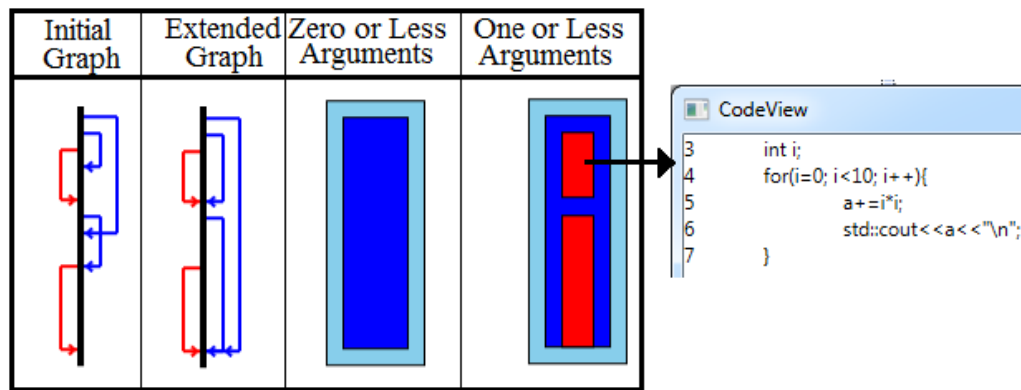


Figure 5.4 Observing Refactoring Opportunities For Example Method

Figure 5.4 shows the refactoring opportunities for the example method given in Figure 5.2 based on a user-selected value as the maximum number of arguments. Code fragments that would require the same number of arguments are given the same color in the visual representation. This tool enables the user to see the source code, represented by each box simply by clicking on the box. An example code fragment that would appear when clicked on the small red box taking one parameter is also shown in Figure 5.4.

## 5.5 Experiments

This refactoring suggestion technique and the tool are designed to help program developers to eliminate long method code defects by extracting new methods with a number of arguments less than a threshold.
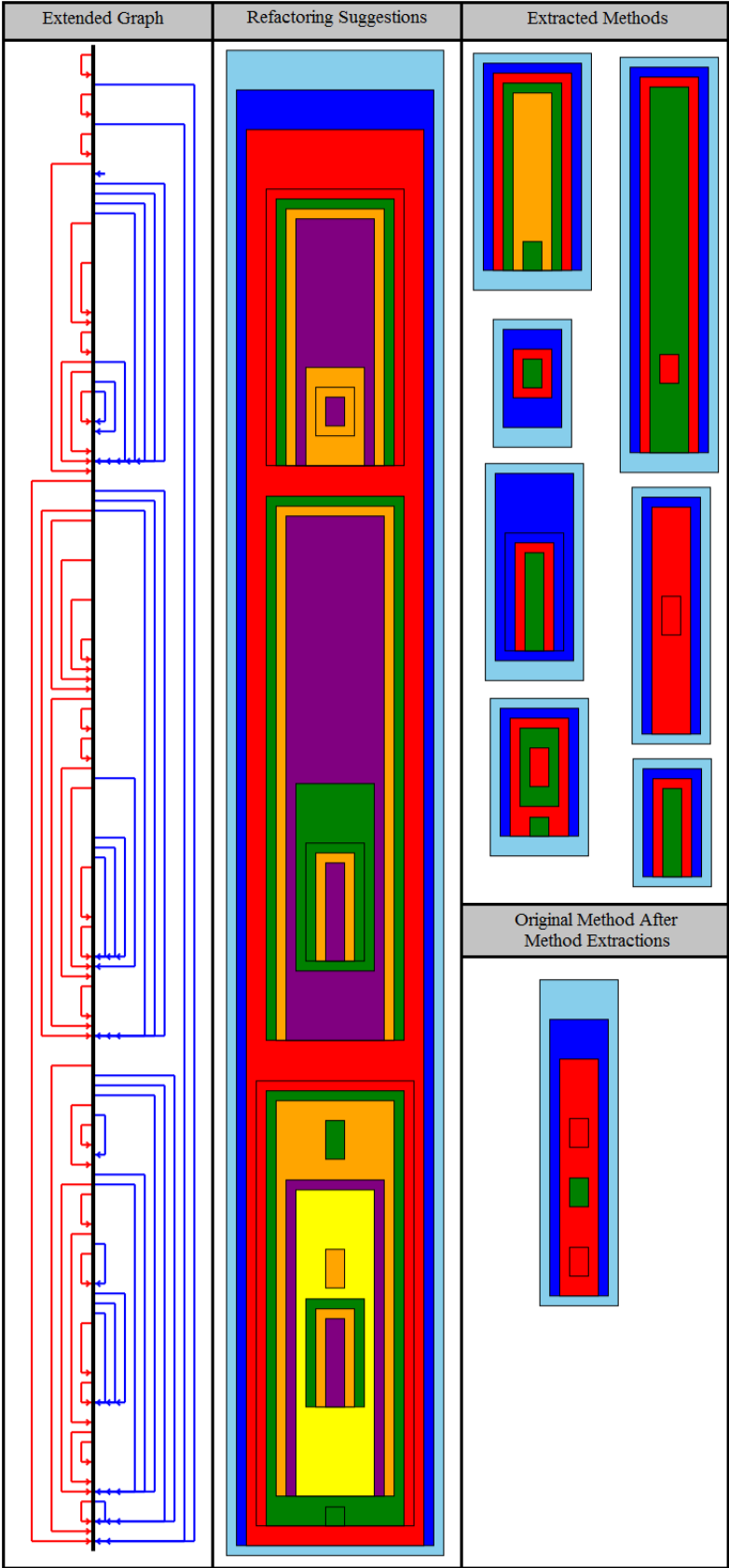
Figure 5.5 Experiment 1

Using this tool, developers are able to observe all possible refactoring opportunities or suggestions. They can also see the code fragments suggested as prospective methods by clicking on the boxes that represent these code fragments. The size of these boxes shows the length of the method and the color is determined based on the number of arguments method requires when the pointed code fragment is extracted as a method.

Developers may not want to extract all code fragments suggested by this tool at all times. For the given example method, Figure 4 shows all possible extract method refactoring opportunities with different number-of-argument thresholds e.g. zero and one. The tool points out a code region that covers the entire method as a candidate method that takes zero argument which would not be logical to extract. But, when the threshold is increased to one, new code fragments appear as refactoring suggestions.

Therefore, the final decision about which candidate code fragments are going to be extracted as new methods are made by the developers or the users of this tool. This evaluation is of course will be more effective, if the user has at least some knowledge about the original method before executing the extraction. That is why, we are using some methods from our own source code, to demonstrate the effectiveness and use of this technique and tool.

In Figure 5.5, an example method that has been diagnosed to be long and complex by our team is shown. We extract seven methods out of this long method and show what the original method looks like after the refactoring. We could easily give meaningful names to these extracted methods that reflected what each method's intended operation was.

```
Utf8_16_Write UnicodeConvertor ;
UINT length = 0 ;
char * buffer = NULL ;
ScintillaEditView * pSci ;
if(wParam == MAIN_VIEW){
        pSci = & _mainEditView ;
}
else {
        if(wParam == SUB_VIEW){
                pSci = & _subEditView ;
        }
        else {
                return-1 ;
        }
}
length = pSci->execute(SCI_GETTEXTLENGTH,0,0)+1 ;
buffer = new char[length];
pSci->execute(SCI_GETTEXT,length,(LPARAM)buffer);
UniMode unicodeMode = pSci->getCurrentBuffer()->getUnicodeMode();
UnicodeConvertor.setEncoding(unicodeMode);
length = UnicodeConvertor.convert(buffer,length-1);
pSci->execute(SCI_CLEARALL);
pSci->addText(length,UnicodeConvertor.getNewBuf());
pSci->execute(SCI_EMPTYUNDOBUFFER);
pSci->execute(SCI_SETCODEPAGE);
pSci->execute(SCI_GOTOPOS);
delete[]buffer ;
return unicodeMode ;
```

```
TCHAR ** fileNames =(TCHAR **)wParam ;
int nbFileNames = lParam ;
int j = 0 ;
if(Message != NPPM_GETOPENFILENAMESSECOND){
        for(int i = 0 ; i < _mainDocTab.nbItem()&& j < nbFileNames ; i++){
                BufferID id = _mainDocTab.getBufferByIndex(i);
                Buffer * buf = MainFileManager->getBufferByID(id);
                lstrcpy(fileNames[j++],buf->getFullPathName());
        }
}
if(Message != NPPM_GETOPENFILENAMESPRIMARY){
        for(int i = 0 ; i < _subDocTab.nbItem()&& j < nbFileNames ; i++){
                BufferID id = _subDocTab.getBufferByIndex(i);
                Buffer * buf = MainFileManager->getBufferByID(id);
                lstrcpy(fileNames[j++],buf->getFullPathName());
        }
}
return j ;
```

Figure 5.6 Experiment 2: Notepad++

Another method that we use for our experiments belongs to Notepad++ which is an open source code editor and Notepad replacement that supports several programming languages and natural languages [48]. This method has nearly 2000 lines of code which shows that it needs to be decomposed into smaller ones.

Figure 5.6 shows some of extract method opportunities with less than four arguments for aforementioned Notepad++ method. Two boxes that represent candidate methods taking zero

arguments are shown and marked as A and B respectively and their corresponding code is also shown.

Although the code for this example has not been written in our research group, when we read the code given in Figure 5.6, we can still get a sense of what these designated code fragments do. For example code fragment represented by A, does some sort of text conversion and subsequently calls some methods. This much information would be enough to draw the conclusion that extracted method out of this code fragment will have a well-defined clear intention and therefore improve the quality of this code.



```
TCHAR * extFilterName = TEXT("Notepad++ plugin");
TCHAR * extFilter = TEXT(".dll");
TCHAR * destDir = TEXT("plugins");
vector < generic_string > copiedFiles = addNppComponents(destDir,extFilterName,extFilter);
vector < generic_string > dll2Remove ;
for(size_t i = 0 ; i < copiedFiles.size(); i++){
        int index = _pluginsManager.loadPlugin(copiedFiles[i].c_str(),dll2Remove);
        if(_pluginsManager.getMenuHandle()){
                _pluginsManager.addInMenuFromPMIndex(index);
        }
}
```

```
generic_string tmp((NppParameters::getInstance())->getNppPath());
generic_string nppHelpPath = tmp.c_str();
nppHelpPath+=TEXT("\\user.manual\\documentation\\notepad-online-document.html");
if(::PathFileExists(nppHelpPath.c_str())){
        ::ShellExecute(NULL,TEXT("open"),nppHelpPath.c_str(),NULL,NULL,SW_SHOWNORMAL);
}
else {
        generic_string msg = nppHelpPath ;
        generic_string warning,title ;
        if(! _nativeLangSpeaker.getMsgBoxLang("NppHelpAbsentWarning",title,warning)){
                title = TEXT("File does not exist");
                warning = TEXT("\rdoesn't exist. Please download it on Notepad++ site.");
        }
        msg+=warning ;
        ::MessageBox(_pPublicInterface->getHSelf(),msg.c_str(),title.c_str(),MB_OK);
}
```
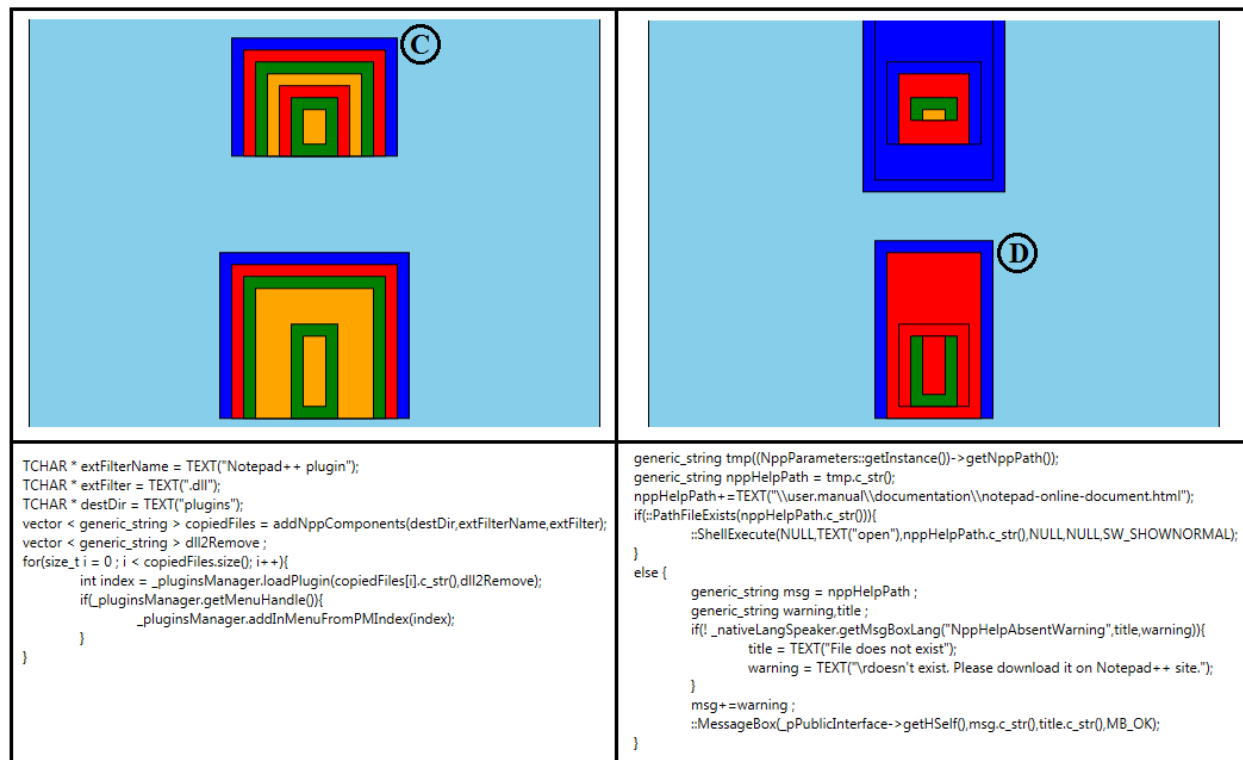
Figure 5.7 Experiment 3: Notepad++

Another method that belongs to the Notepad++ and some extraction suggestions taking less than four arguments are shown in Figure 5.7. Likewise, we arbitrarily chose two of those suggestions taking zero arguments and showed their corresponding code fragments. When we read the

comments within the original source code, we see that these two code fragments have their own well defined sub-tasks to accomplish and therefore can be made separate methods.

## 5.6 Summary

In this part of our research, a new technique and tool are introduced to identify code fragments for method extraction. Using our tool, developers have the opportunity to observe different code fragments suggested as candidates for method extraction based on a desired number of arguments. Our technique uses a special kind of graph, the hammock graph, which has one entry and one exit point. Hammock graphs are generated based on local variable declarations and their references within a given method. Providing different visual representations of suggested code fragments, developers get more detailed information about their code using this tool in addition to method extraction suggestions. The main contribution here is, therefore, to reduce the effort required to re-factor software source code towards making the code more readable and maintainable and less complex.

Our technique differs from existing techniques in several ways. First of all in this research we identify hammocks on a graph that is constructed based on control blocks, local variable references and their interactions. This ensures that all local variable declarations designates a sub graph or a code region that is a hammock and therefore extractable. Furthermore, the visualization support for this technique provides significant help and control for observing all possible method extraction opportunities taking parameters no more than a user defined threshold.

We have shown the effectiveness of the proposed technique and tool on several real methods. When we extracted methods from our own analysis code, we could easily come up with meaningful names that reflected the intent of the extracted method. We could also effectively identify code regions as candidate methods with varying number of arguments on large methods taken from open-source software such as Notepad++.

The main contributions of this part of our research can be summarized as follows:

1. We have introduced a novel technique to identify extract method opportunities limiting the number of parameters that generated method after refactoring would take.

2. We have defined hammock graphs based on variable declaration-use, control blocks and the interactions between them to make every variable span extractable without any restrictions.

3. We support this technique with analysis and visualization tools to effectively observe the extract method opportunities dynamically based on varying number of arguments.

4. We show the soundness of this technique both on our own source code and open-source software.

# Chapter 6

## Conclusions and Future Work

In this research, we explored three main software code defects and proposed solutions with novel techniques and tools. These code defects are Large Class, Long Method and Long Parameter List code defects and the techniques we propose are to achieve Extract Class, Extract Method and Extract Method with Shorter Parameter List refactoring respectively for each code defect we addressed.

Extraction of units from an existing software is usually carried out in two steps. First the code needs to be inspected to find appropriate code fragments for extraction and then these selected code fragments are extracted as separate units making the required changes on the source code such as addition of method invocations and object declarations etc.

In this research, our concern is to introduce novel techniques to identify refactoring opportunities automatically, since the extraction step, especially for method extraction, is supported by popular industrial tools as discussed.

Techniques and tools we provide enable users to identify code fragments without having to inspect code manually. Therefore, they can be applied to foreign software code that the user wants to re-factor. Moreover these techniques and tools rely mostly on variable declarations and references with the integration of other concepts such as placement trees and hammock graphs, which can be automated for detection regardless of the programming language. Hence, these techniques can easily be applied to C++ software code that traditionally has weak refactoring tools, compared to other programming languages such as C# and Java.  Our methods improve refactoring for these languages as well.

## 6.1 General Summary of Contributions

Our contributions can be grouped in three different categories. With our first contribution, we tackle the Large Class code defect and developed a new cohesion metric suitable for Extract Class refactoring. Our second contribution solves the Large Method code defect by providing a novel technique that represents the method with a placement tree and identifies refactoring opportunities based on the properties of the placement tree. With visualization support we provided, one can observe these refactoring opportunities on large scale software very efficiently. Last, we introduced an Extract Method refactoring to confine code regions to those with one entry and one exit point based on the variables declarations and uses. This technique together with our visualization tool provides an opportunity to observe Extract Method refactoring opportunities limiting the number of parameters the extracted methods would take.

Since long parameter list is a code defect that increases complexity and understandability of methods [5], this technique ensures that each extracted method will have a parameter list no longer than that specified by the user.

## 6.2 Extract Class Refactoring Technique

This section of our research is published as an academic paper in the 6th IEEE International Workshop on Quality Oriented Reuse of Software [40] and is summarized in the following section.

### *6.2.1 Summary*

As our first contribution, the extract class re-factoring technique we introduced in this study, evaluates the cohesiveness of a given class and suggests re-factoring the class by extracting new classes. We define a new cohesion metric that can be used to indicate the need for refactoring for a given class. The cohesion level measured this way shows how many different abstractions the class includes, suggesting creation of a separate class corresponding to each abstraction.

We introduced an algorithm that was inspired from the concept of program slicing. Our algorithm, in contrast to the traditional program slicing, provides a clear definition of slicing criteria. For each data member of the given class, we simply take all the program statements that use this data member as slicing criteria. Using the relationship conditions given in Chapter 2, we find all program statements or "program slice" for each slicing criterion we identified. Combination of all program slices with respect to every slicing criterion for a given data member, constructs the final program slice for that data member of the class. By comparing the program slices of every pair of data members we construct a graph that we call the data slice

graph. Each data member in this graph is represented by a node and the fact that program slices for two data members intersect introduces an edge to this graph between the nodes that represent these two data members.

Finally, we interpret the cohesiveness for the given class based on the number of connected components in the data slice graph. We suggest that the class should be restructured by extracting separate classes for each connected component, if the number of connected component is greater than one. Figure 6.1 shows the data slice graphs for two classes that we analyzed. The first one on the left, as seen in Figure 6.1 has three connected components and therefore is less than ideally cohesive and should be restructured. The other one on the other hand has only one connected component which means that this class is cohesive and needs no restructuring.
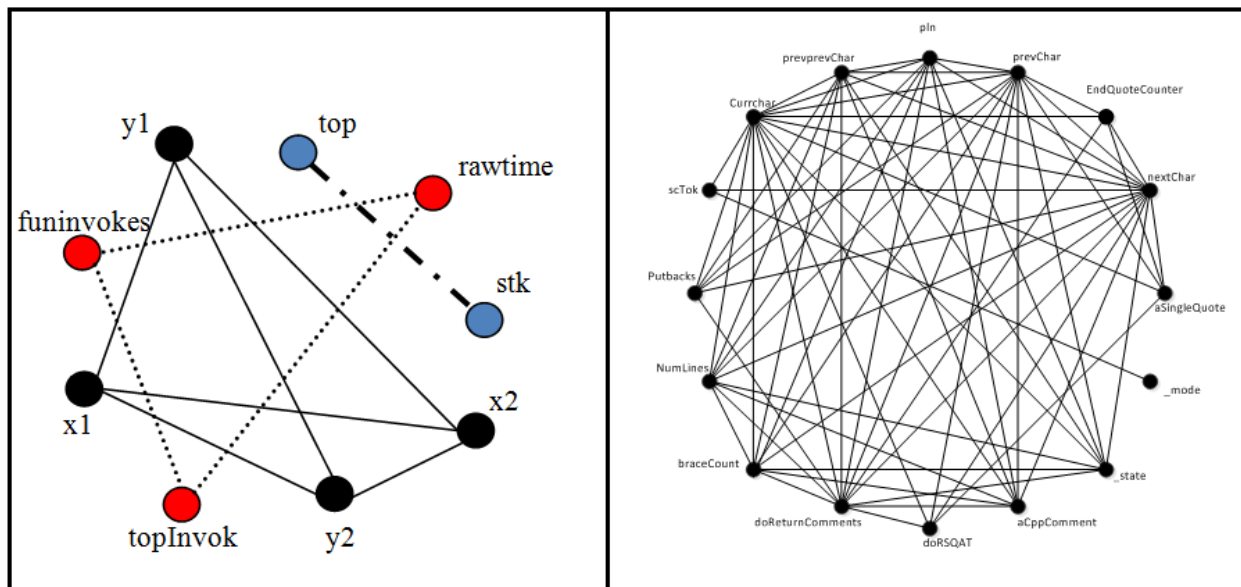


Figure 6.1 Data Slice Graphs

Table 6.1 shows the refactoring results quantitatively in terms of total number of methods, data members and lines of code for class shown in Figure 6.1. When we compare the original class before and after refactoring, we notice that while the interface of the original class is preserved

with 13 methods, number of data members is reduced to 3 and number of lines of code is reduced to 72. These three data members in the re-factored version of the original class are due to the objects of the extracted three classes to access these classes.

Table 6.1 Refactoring Summary for Class 1

|  | # of Methods | # of Data Members | # of Lines |
|---|---|---|---|
| **Original Class** | 13 | 9 | 150 |
| **Class After Refactoring** | 13 | 3 | 72 |
| **Extracted Class 1** | 6 | 2 | 49 |
| **Extracted Class 2** | 12 | 3 | 105 |
| **Extracted Class 3** | 5 | 4 | 35 |

## *6.2.2 Contributions and Future Work*

Automated analysis of the C++ code is a very challenging task due to its complex syntax and semantics. In this research, we would like to be able to apply these refactoring techniques to a wide range of software systems including especially those written in C++. Therefore, the techniques we introduced needed to be simple enough for implementation for analysis of different programming languages and at the same time powerful enough to effectively identify possible refactoring opportunities. With the help of analysis tools we implemented, we realized that we can easily automatically detect variable declarations, references and scopes in a given method with a tool much simpler than a compiler front-end. This first contribution was mainly to justify that, in conjunction with the other concepts we used, variable references are an important tool to detect refactoring opportunities.

When we want to evaluate the novelty of this contribution, we should think of cohesion in two different ways as addressed in literature. Cohesion as a software quality metric is used to refer the relatedness of program segments and output variables in a method (functional cohesion) [24, 36], or it is used to evaluate the relatedness of class entities such as data members and methods in an object oriented program [67]. In both cases, one attempts to discover whether the method or class is designed to handle one specific activity - an important feature for a program towards its comprehension and maintainability.

In [36] for example, a technique is suggested to find cohesion level between every pair of output variables for a given method. Using program slicing, all the statements that affect values of these variables are explored and depending on the cohesion between these output variables, corresponding statements are extracted and replaced with a method call. These studies [24,36], imply that methods may be not cohesive within themselves by including unrelated code segments.

Although different cohesion metrics aim to detect different aspects of the software units to evaluate their cohesiveness [59], in object oriented programs, cohesion level of a class is usually measured by the interactions between the data members and methods of the class [28, 29, 30, 31, 32, and 33]. In this case, when two methods share the same variables or two variables are used by the same methods, they are considered cohesive. As per intuition, one may argue that this conflicts with the fact that methods may include unrelated parts. In other words, two variables, even though they are used in the same method, may still be not cohesive or related because of the fact that they may reside in two unrelated segments of code within the method. A discussion on some cohesion metrics and their anomalies are given in [59].

Furthermore, studies in [60, 61] criticize some of the existing cohesion metrics for including in their computations the special methods in a class such as constructors and destructors, reporting that these metrics do not satisfactorily capture the concept of cohesion. In conclusion, we found it desirable to introduce a new cohesion metric that serves as a tool to indicate when a class needs refactoring and how to apply the refactoring, rather than using an existing method.

Program Slicing and Clustering are two techniques that are related to what has been proposed in this part of our research in terms of either the technique that has been adopted or problem that has been addressed to.

A discussion on program slicing has been given in Chapter 2 and Chapter 3. In fact, by introducing a set of rules to evaluate the relatedness between program statements, we produce different "program slices" than the traditional program slicing method would yield as shown in Figure 6.2. For class refactoring, rather than seeking program statements that affect value of a certain variable at a specific location in the program, we aim to find the program statements that should be in the same abstraction. Hence, using the traditional program slicing method would not be a proper technique for this purpose.

| Line# | Original Program Fragment | Program Slicing Result | Our Result |
|---|---|---|---|
| 1 | int i; | int i; | int i; |
| 2 | int sum = 0; | int sum = 0; | int sum = 0; |
| 3 | int product = 1; | | int product = 1; |
| 4 | for(i = 0; i < N; ++i) | for(i = 0; i < N; ++i) | for(i = 0; i < N; ++i) |
| 5 | { | { | { |
| 6 | sum = sum + i; | sum = sum + i; | sum = sum + i; |
| 7 | product = product *i; | | product = product *i; |
| 8 | } | } | } |
| 9 | cout<< sum; | cout<< sum; | cout<< sum; |
| 10 | cout<< product; | | cout<< product; |

Figure 6.2 Program Slicing vs. Our Result

On the other hand, the program fragment that we end up inducing as a product of this slicing should be extractable as a separate unit. In Figure 6.2, we see the results of traditional program

slicing method and our slicing result. It is important to note that, if we used the program slicing as it is without defining our own relationship conditions as we did in Chapter 3, the resulting code fragment as shown in Figure 6.2 would not be possible to extract as a separate unit as it breaks a loop with including only some of the statements in the loop. On the other hand, our result is suitable for restructuring, as it takes the entire loop within the slice making the resulting code fragment extractable.

Clustering techniques are related to the problem we are addressing in this research, being used as a tool to group entities in object oriented programs such as classes, methods and attributes, in order to construct more cohesive units. For example in [34], a k-medoids algorithm is introduced to re-group the entities in a given software to form a more cohesive design. This is accomplished by moving existing entities between classes, in case they are misplaced. Yet, although a method may not be cohesive internally within itself, this approach does not suggest decomposing methods into smaller ones or it is not at the statement level. Moreover, moving a public attribute or a method from one class to another will change the public interface of the class and this will break all the client code that depends on this class. We think that this may result in undesirable results which need to be handled very carefully.

The main advantages and contributions of our work can be summarized as follows:

1. We introduced a new cohesion metric that can be used to indicate the necessity of refactoring in a class.

2. Our slicing result are fully extractable as we identify code blocks as slices.

3. Our refactoring suggestion does not alter the public interface of the class, but requires to extract new classes from it.

4. Our analysis depends on the statements rather than already existing program units such as methods. Therefore, rather than moving these units between different classes, we identify program statements to construct new units.

Several future extensions are possible to enhance the results for this extract class refactoring technique. We can enhance the conditions that establish the relationships between statements to preclude undesirably long code chunks in one slice which may increase the chances of containing non-cohesive statements. An improvement on the Data-Slice-Graph (DSG) also may be to convert the graph into a weighted graph. Currently the fact that intersection of slices of two data members is the empty set or not determines the existence of an edge between the nodes that represent these two data members. This technique as described in this research results in unrelated classes which, we argue, include different abstractions. Yet, dividing the DSG based on the weights of edges and a threshold may help to construct classes that aggregate or composite other extracted class objects.

## 6.3 Extract Method Refactoring Using Placement Trees

This part of our research is published in the 25th International Conference on Software Engineering and Knowledge Engineering (SEKE, 2013) [38] and a summary is given in the next section.

### *6.3.1 Summary*

In this part of our research we introduced a novel extract method refactoring technique and tool that constitutes our second major contribution. This technique, using the concept of placement trees, identifies possible candidate code fragments for method extraction. We have implemented

this technique and provided a visualization tool that helps the user observe this refactoring opportunities effectively. This visualization uses the tree-map concept where hierarchical information is represented with nested box, a technique that suits our purpose very well.

This technique is based on variable reference counts and very straightforward to implement for any programming languages including C++. By defining a dominant variable in every node of the placement tree, we aim to find the variable that represents the operation in that node best. By comparing the dominant variable of a node with those of its parent and sibling nodes, refactoring opportunities are sought capturing sub-trees whose nodes are dominated by the same variable.
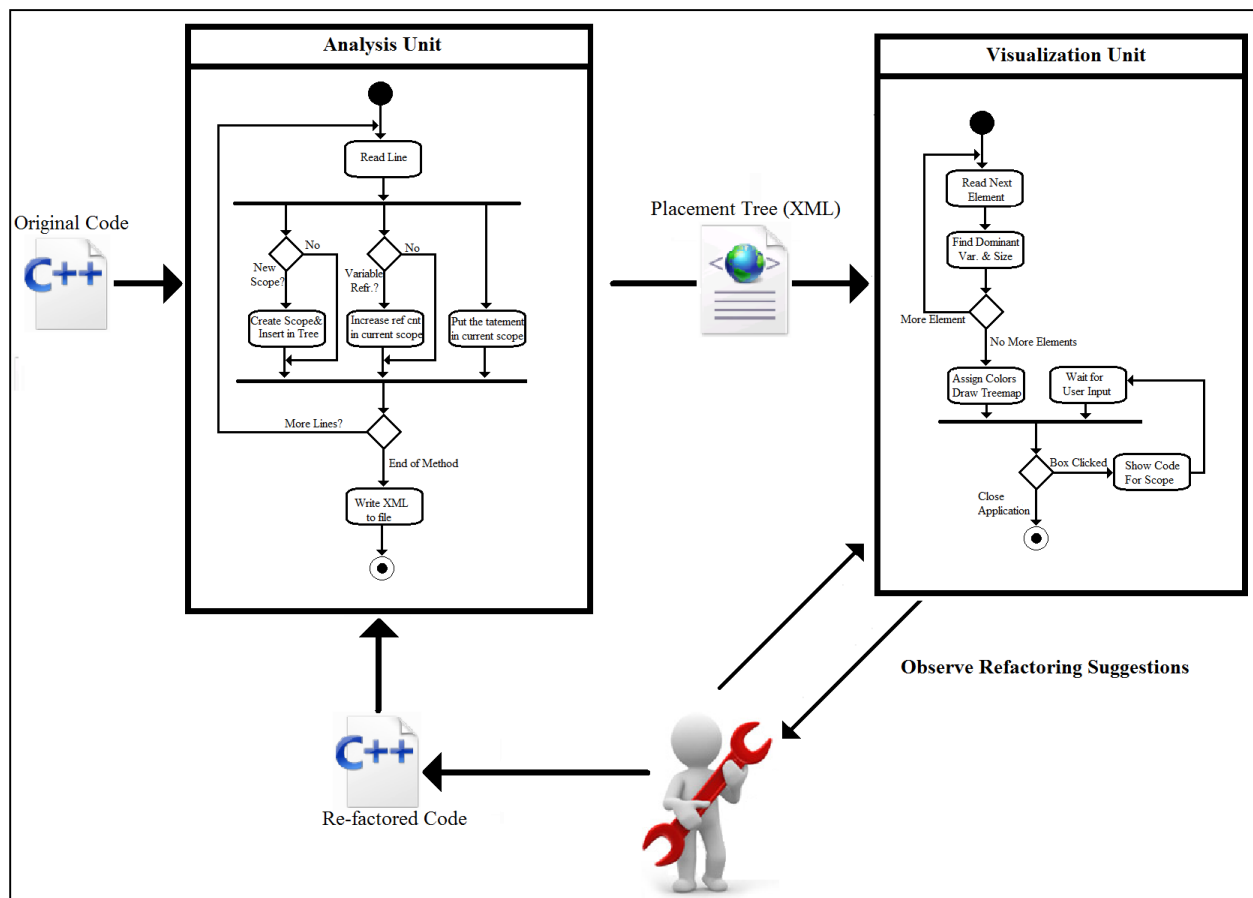


Figure 6.3 Refactoring Process

Figure 6.3 summarizes how this extract method refactoring technique proceeds. The input method is first reformatted to put appropriate braces around the control blocks and indentations

of statements are organized. This first step not only makes the code simpler to analyze, it also enhances the readability of the code dramatically as shown in the example in Figure 6.4. After this, the input method is analyzed to find all scopes in the method as well as the variable reference counts for each scope. Dominant variables are determined based on their respective reference counts and an XML file is generated that contains all the information regarding the scopes and variables that the visualization unit needs.

| Before Brace Insertaion | After Brace Insertion |
|---|---|
| ```
1    #include <iostream>
2  ⊟void main()
3   {
4       int odd=0;int even=1;
5       for (int i = 0; i < 100; i++) if(i%2)odd+=i;else even+=i;
6       std::cout<<"Accumulation of odd="<<odd;std::cout<<"Accumulation of even="<<even;
7
8
9   }
``` | ```
1    # include < iostream >
2  ⊟void main(){
3       int odd = 0 ;
4       int even = 1 ;
5       for(int i = 0 ; i < 100 ; i++){
6           if(i%2){
7               odd+=i ;
8           }
9           else {
10              even+=i ;
11          }
12      }
13      std::cout << "Accumulation of odd=" << odd ;
14      std::cout << "Accumulation of even=" << even ;
15  }
``` |

Figure 6.4 Brace Insertion Example Result

The visualization tool represents the placement tree using *tree-map* with nested boxes. Size and color of a particular box are determined based on the length and the dominant variable of the scope it represents respectively. Refactoring opportunities are sought to create methods with possibly a single dominant variable. This is accomplished by extracting sub-trees of the placement tree in which every node has the same color as separate methods.

Table 6.2 shows some extract method refactoring results quantitatively. The first method, *doAction*, is taken from earlier version of our analysis tool and the second one comes from the medical imaging application that we used for some part of our experiments. For each method, Table 6.2 shows the length and cyclomatic complexity of the method before and after refactoring together with the length and cyclomatic complexity of each extracted method. Complexity of a method is measured by an analysis tool that was developed Dr. Fawcett and has been used in our

research group. This tool relies on the number of control statements such as if, else and for to measure the cyclomatic complexity of a method.

Table 6.2 Some Quantitative Results for Extract Method Refactoring

| **Method:** doAction **Domain:** Analyzer **# of Extraction:** 3 | | |
|---|---|---|
| | *LOC* | *Cyclomatic Complexity* |
| *Before Refactoring* | 101 | 44 |
| *After Refactoring* | 21 | 3 |
| *Extracted Method 1* | 52 | 27 |
| *Extracted Method 2* | 20 | 11 |
| *Extracted Method 3* | 21 | 6 |
| **Method:** W_Calculate **Domain:** Medical **# of Extraction:** 9 | | |
| | *LOC* | *Cyclomatic Complexity* |
| *Before Refactoring* | 379 | 46 |
| *After Refactoring* | 39 | 4 |
| *Extracted Method 1* | 13 | 3 |
| *Extracted Method 2* | 13 | 3 |
| *Extracted Method 3* | 13 | 3 |
| *Extracted Method 4* | 13 | 3 |
| *Extracted Method 5* | 19 | 5 |
| *Extracted Method 6* | 33 | 9 |
| *Extracted Method 7* | 16 | 5 |
| *Extracted Method 8* | 45 | 9 |
| *Extracted Method 9* | 62 | 11 |

### *6.3.2 Contributions and Future Work*

Selection of appropriate code fragments for method extraction is a very challenging process that requires tedious inspection of code. On the other hand, once the candidate code is selected, it can be extracted as a separate method easily using available tools or IDEs. While there still exist some studies that target the extraction step in refactoring [66], our focus is on the automation of identification of code fragments. A random selection of code fragments may lead to two major errors that hinders the refactoring process.

1. Selection may include a local variable declaration that has references outside of the selection.
2. Selection may break a control block such as a for loop or if control block.

We have two concerns during the identification of code fragments. We have to make sure that our automated selection does not induce the two cases given above, and identifies scopes that carry out distinct operations.

Program slicing as mentioned earlier is one of the most widely used techniques for extract method refactoring. Yet, there is not an a priori approach that can identify different code regions that carry out possibly distinct operations. Program slicing can be used to find statements that affect a certain variable in a method, but, unlike our method, this will not help to group code segments based on operations for method extraction.

We introduce a technique and visualization tool that can be applied to any method without requiring the user to have any knowledge of it to identify code fragments. Some program slicing methods such as [41, 43] are introduced to find code fragments related to computation of a given variable and in [45], an automatic process for extracting methods based on an input

Control Flow Graphs (CFG) of a function and a set of pre-selected nodes is introduced. In our approach, we do not have to choose a criterion or seed statements in code which would otherwise require the user to inspect the code and have a fair amount of knowledge of the code. The novelty of our work therefore can be listed as follows:

1. We introduced an extract method refactoring based on the concept of placement tree. This technique eliminates any possibility of breaking a control scope or including variable declarations without some of their references.

2. Our technique and tool do not require any prior knowledge of the code and therefore can be applied to foreign software easily.

3. The results can be effectively visualized with tree map due to the structure of the placement tree.

4. We support the technique with a novel visualization tool that helps the user to observe all candidate refactoring opportunities.

5. The combination of the concept of dominant variables with the placement tree and tree map visualization, makes our technique applicable to C++ language easily which currently lacks such automation tool support.

In the current version of our work as described in this dissertations, code regions are differentiated from each other based on their dominant variables. Two approaches, sibling collaboration and parent protection, are introduced to solve issues with the selection of dominant variables if two different variables have the same number of references in a scope. In some rare cases, if a dominant variable cannot be appointed for a node by these two approaches, it is selected randomly amongst the variables with the same highest reference counts.

Rather than adopting Sibling Collaboration and Parent Protection approaches, this work can be expanded to show the power of each variable on individual scopes. Several different approaches may produce promising results which will be the future extensions of our work.

1. We can determine the color of an individual scope based on more than one variable. If a scope is dominated by more than one variable, than it can be given a color or pattern that reflects this case i.e. a striped box with the colors of its dominant variables.

2. We can come up with a function that gives a unique value based on the reference counts of all variables referenced in a scope. This value can be used to assign a unique color to the box to represent the effect of all variables. In this case, only those boxes that have the exact same number of references of each variable will have the same color.

3. We can develop a centrality based approach to determine the dominant variable. For this, the relationships between variables in a scope need to be represented as a graph. Centrality analysis in graph theory and network analysis is used to find the relative importance of nodes and therefore the most important node in a graph. We can use this approach to find the most important variable to be selected as the dominant variable in the scope and color it accordingly.

4. For large methods, oftentimes it is almost impossible to fit the tree-map representation of the placement trees in the screen with its current form. As a future work, we may adopt the more traditional form of tree-map described in [77] for our visualization to make the tool more user friendly by enabling the user to see the entire placement tree on one screen.

## 6.4 Extract Method Refactoring Using Hammock Graphs

### *6.4.1 Summary*

The third and last major contribution of our research is related to the Long Method code defect and Long Parameter List code defect indirectly. In our previous extract method refactoring technique and tool, our concern was to identify appropriate code fragments for extraction without paying attention to how many arguments each code fragment would require when extracted as a method. In some cases, this may result in identifying code fragments as possible candidates for extraction, that would have an undesirably long parameter list after refactoring. Since a long parameter list makes the code more complex and less maintainable, selection of code fragments considering the lengths of their argument lists is a necessity for a more effective method extraction technique and tool.

We introduced a technique based on variable declaration and uses and the concept of hammock graphs. There are two main differences between our previous extract method refactoring technique and this one. First, while in our placement tree based approach, the extractions were block based, here we use variable spans as possible code fragments for extraction. Second, in the placement tree based approach, the number of parameters required by code fragments is omitted, whereas in this hammock graph based approach users can observe code fragments dynamically through our visualization tool based on a limit set on the number of arguments.

This technique starts with the analysis of the code for brace insertion and indentation as we described earlier. After that, the analysis unit finds all local variable declarations and their respective variable spans together with all control blocks. The result of this step is what we call the initial graph.

All methods have one entry and one exit point when their control flow graphs are considered. They often declare one or more local variables and after some operations on them, return a value of one of those variables. We consider each variable span as a candidate code fragment for extraction, but not all variable spans on our initial graph are extractable due to their interaction with other variable spans and control blocks. Therefore, the analysis unit converts the initial graph into what we call an extended graph where each variable span is extractable. This is accomplished by

a) Extending each variable span to include the entire variable span of any variable that is declared in it

b) Extending each variable span to cover an entire control block that starts in it.

As a result of this conversion, we make sure that all variable spans are hammock meaning there does not exist any outgoing or incoming edges from or inside the variable span. We argue that when the interactions between the variable spans and control blocks are inspected in a cohesive method, they will construct a hammock as well. Figure 6.5 shows an example method and its corresponding initial and extended graphs.
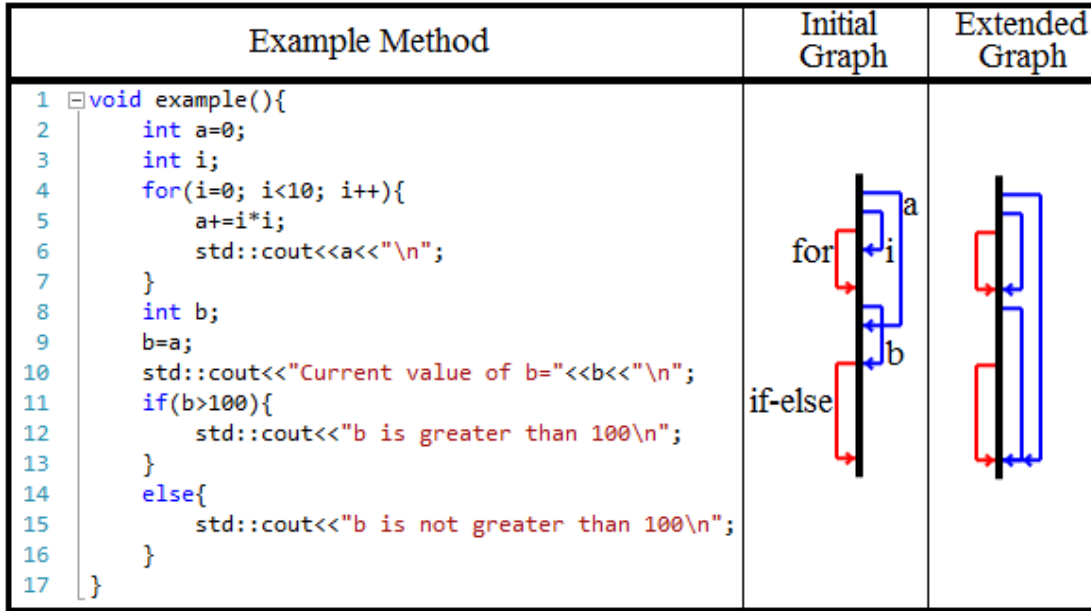
| Example Method | Initial Graph | Extended Graph |
|---|---|---|
| ```
 1  void example(){
 2      int a=0;
 3      int i;
 4      for(i=0; i<10; i++){
 5          a+=i*i;
 6          std::cout<<a<<"\n";
 7      }
 8      int b;
 9      b=a;
10      std::cout<<"Current value of b="<<b<<"\n";
11      if(b>100){
12          std::cout<<"b is greater than 100\n";
13      }
14      else{
15          std::cout<<"b is not greater than 100\n";
16      }
17  }
``` |  |  |

Figure 6.5 Initial Graph and Extended Graph Example

After this conversion, an XML file is generated that includes both the initial graph and extended graph information to be used as an input to the visualization unit. Each variable span is extractable requiring varying number of arguments to be passed once they are extracted.

Since the variable spans are in a nested form, using the tree-map as we did for placement tree works for this visualization technique perfectly as well. In this visualization, each box represents a variable span or a candidate code fragment. Two properties of a box, size and color, are determined based on the length of the variable span and the number of argument that the variable span requires when extracted as a separate method respectively.

Figure 6.6 Example Visualization

In Figure 6.6, an example visualization is shown for a randomly selected method. We see different views of refactoring suggestions for different number of arguments selected by the user.

Notice that using the slider placed on top of the window, user can change the value of the length of parameter list threshold to observe more or less refactoring opportunities.

Another important feature of this visualization is that one can see the actual code fragment represented by each box by clicking on the box as demonstrated in Figure 6.6. Therefore, the user can decide, after a quick inspection of the code, if s/he needs to extract that particular code fragment.

Table 6.3 Quantitative Results for Hammock Graph Refactoring

| **Method:** FindArgumentsInVarSpand **Domain:** Analyzer **# of Extraction:** 2 | | |
|---|---|---|
| | *LOC* | *Cyclomatic Complexity* |
| *Before Refactoring* | 62 | 28 |
| *After Refactoring* | 16 | 3 |
| *Extracted Method 1* | 27 | 14 |
| *Extracted Method 2* | 25 | 13 |
| **Method:** doAction **Domain:** Analyzer **# of Extraction:** 7 | | |
| | *LOC* | *Cyclomatic Complexity* |
| *Before Refactoring* | 156 | 45 |
| *After Refactoring* | 34 | 10 |
| *Extracted Method 1* | 25 | 5 |
| *Extracted Method 2* | 14 | 4 |
| *Extracted Method 3* | 43 | 14 |
| *Extracted Method 4* | 23 | 4 |
| *Extracted Method 5* | 17 | 4 |
| *Extracted Method 6* | 27 | 12 |
| *Extracted Method 7* | 14 | 3 |

Table 6.3 shows how some re-factorings that we performed impact size and complexity of two methods that we chose from our analysis code. For the first method called *FindArgumentsInVarSpand*, we applied two extract method re-factorings, while from the second one seven methods have been extracted.

In Table 6.4, we provide the results for a method that we arbitrarily chose from Notepad++ due to its relatively long implementation. As stated earlier, we suggest performing extract method refactoring step by step, increasing the desired length of prospective methods' parameter lists to observe more refactoring opportunities. In this case, for the method called *run_dlgProc,* we limit the number of arguments that a method may require to three and successfully extracted 25 new methods with different size.

Table 6.4 Quantitative Refactoring Results for Notepad++

| Method: run_dlgProc Domain: Notepad++ # of Extraction: 25 | | | |
|---|---|---|---|
| | *LOC* | *Cyclomatic Complexity* | *# of Parameters* |
| *Before Refactoring* | 560 | 54 | 3 |
| *After Refactoring* | 269 | 35 | 3 |
| *Extracted Method 1* | 19 | 1 | 0 |
| *Extracted Method 2* | 9 | 2 | 0 |
| *Extracted Method 3* | 13 | 3 | 0 |
| *Extracted Method 4* | 28 | 5 | 0 |
| *Extracted Method 5* | 5 | 1 | 0 |
| *Extracted Method 6* | 6 | 1 | 0 |
| *Extracted Method 7* | 8 | 1 | 0 |
| *Extracted Method 8* | 6 | 1 | 0 |
| *Extracted Method 9* | 6 | 1 | 0 |
| *Extracted Method 10* | 15 | 2 | 0 |
| *Extracted Method 11* | 6 | 1 | 0 |
| *Extracted Method 12* | 14 | 2 | 0 |
| *Extracted Method 13* | 7 | 1 | 0 |
| *Extracted Method 14* | 7 | 1 | 0 |
| *Extracted Method 15* | 7 | 1 | 0 |
| *Extracted Method 16* | 6 | 1 | 0 |
| *Extracted Method 17* | 5 | 1 | 0 |
| *Extracted Method 18* | 8 | 2 | 0 |
| *Extracted Method 19* | 4 | 1 | 0 |
| *Extracted Method 20* | 5 | 1 | 0 |
| *Extracted Method 21* | 20 | 3 | 1 |
| *Extracted Method 22* | 21 | 4 | 1 |
| *Extracted Method 23* | 19 | 3 | 1 |
| *Extracted Method 24* | 17 | 2 | 1 |
| *Extracted Method 25* | 17 | 3 | 2 |

As shown in Table 6.4, out of these 25 methods, 20 of them do not take any arguments. While four of the remaining methods take one argument, we extracted one method that takes two arguments. Note that we analyze the local variables in a method when we determine the number of arguments extracted methods take, but not the arguments that are passed to the original method.

Another important point regarding this refactoring is in its ability to detect duplicate or very similar code. For example when we inspected these resulting 25 methods, we found that some of these method were nearly the same. For future work, our technique can be employed to detect duplicate code.

### 6.4.2 Contributions and Future Work

In this part of our research, we have introduced another extract method refactoring technique based on the concept of hammock graphs. Hammock graphs have been defined on control flow graphs and used to eliminate unconditional jumps and *goto* statements. To our knowledge, ours is the first study that defines hammocks on variable spans and control block interaction and identifies extract method refactoring opportunities. Our contributions may be summarized as follows:

1) We have introduced a novel technique for extract method refactoring that takes variable spans as candidate code fragments

2) We defined hammocks on a graph generated based on variable declaration and uses together with their interactions with control blocks.

3) We supported this approach with a novel visualization technique that enables users to observe refactoring opportunities dynamically based on a desired length of parameter list.

4) We have shown the effectiveness of this technique both on our own analysis code and other professional software.

In software refactoring, there is a need for an industrial quality code analysis tool that can suggest code fragments for method extraction especially for C++ code. We think that combination of the analysis and visualization tools we presented in this research, after some enhancement and addition of the features discussed as future work, can be a strong candidate to fill this gap in refactoring field. Thus, reconstruction of these tools with planned extensions in one application remains as a future work as well.

# Chapter 7

## Appendix

In this section, we provide details of some computations discussed in the content of our research. For some parts of our research, we present more detailed results of the experiments that we conducted. We also provide the source code of some programs that we used in our experiments.

## A.1 Data for Class Cohesion and Refactoring

In this section, we provide the source code for the class used in Chapter 3 for our extract refactoring technique. We give the original version of the code, provide all the computational results and the final version of the code after refactoring.

### A.1.1 Original Source Code

Table A.1 shows the source code of the example class, Class 1, that we used in Chapter 3 before any refactoring is applied.

Table 7.1 Class 1

```
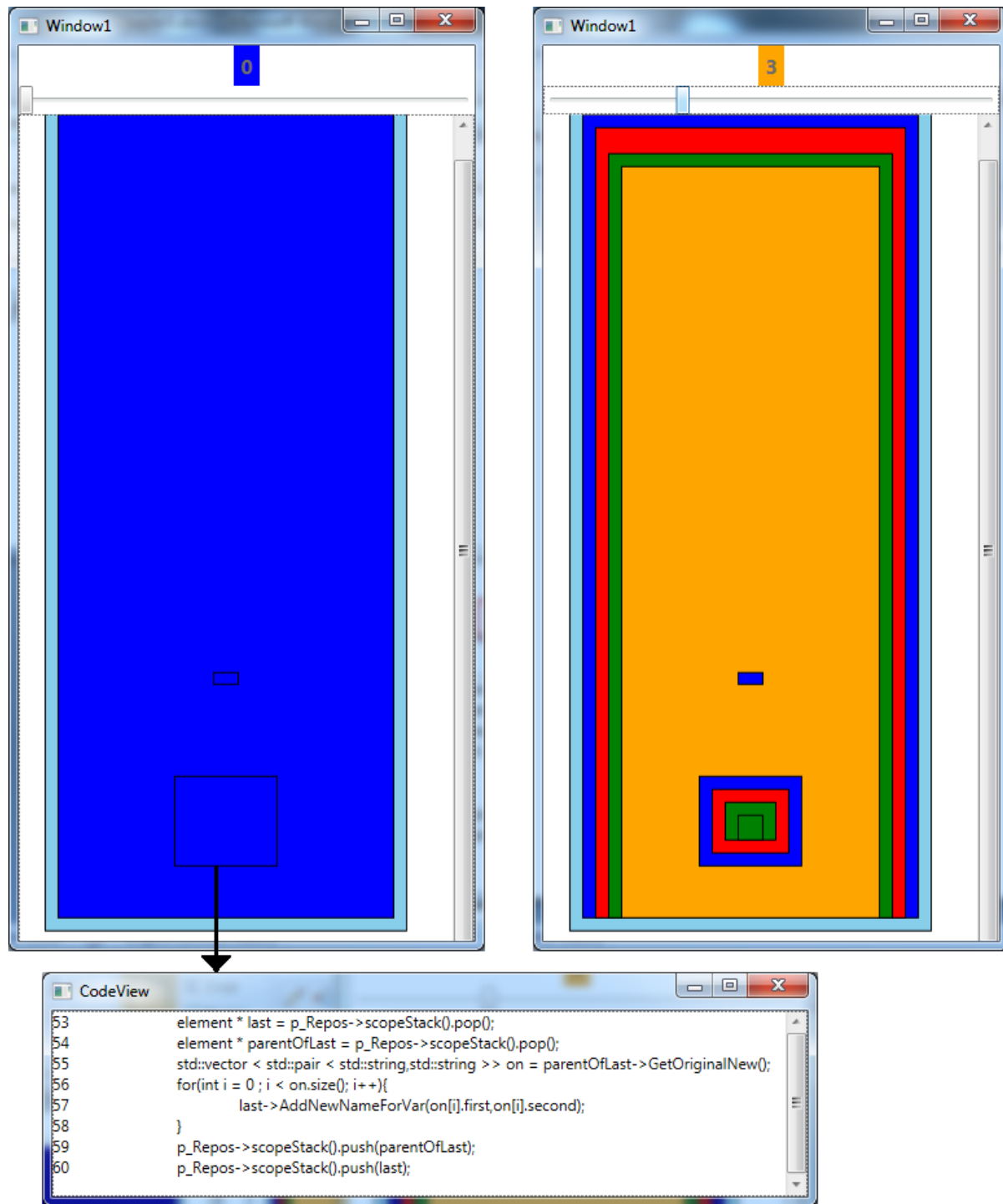1    class Class1
2    {
3    private:
4      int stk[100];
5      int top;
6      string funinvokes[100];
7      int topInvok;
8      time_t rawtime;
9      int x1, y1, x2, y2;
10     void ErrorInSizeFunInvok()
11     {
12       cout<<"Index out of range!\n";
13       cout<<"The Array has "<<topInvok
14           <<" elements.\n";
15     }
16     void ErrorInSize()
17     {
18       cout<<"Index out of range!\n";
19       cout<<"The Array has "<<top
20           <<" elements.\n";
21     }
22   public:
23     Class1(int left=0,int up=0,
24             int right=0,int bottom=0)
25     {
26       topInvok=0;
27       time ( &rawtime );
28       string t=string(ctime(&rawtime));
29       string temp="Class1 invoked: ";
30       temp+=t;
31       PushFunInvok(temp);
32       top=0;
33       x1=left;
34       y1=up;
35       x2=right;
36       y2=bottom;
37     }
38     ~Class1() {}
39     int Height()
40     {
41       time ( &rawtime );
42       string t=string(ctime(&rawtime));
43       string temp="Height invoked: ";
44       temp+=t;
45       PushFunInvok(temp);
46       return (y2-y1);
47     }
48     int Width()
49     {
50       time ( &rawtime );
51       string t=string(ctime(&rawtime));
52       string temp="Width invoked: ";
53       temp+=t;
54       PushFunInvok(temp);
55       return (x2-x1);
56     }
57   int Area()
58   {
59     time ( &rawtime );
60     string t=string(ctime(&rawtime));
61     string temp="Area invoked: ";
62     temp+=t;
63     PushFunInvok(temp);
64     int w=x2-x1;
65     int h=y2-y1;
66     int a=w*h;
67     return a;
68   }
69   int Perimeter()
70   {
71     time ( &rawtime );
72     string t=string(ctime(&rawtime));
73     string temp="Perimeter invoked:";
74     temp+=t;
75     PushFunInvok(temp);
76     int w=x2-x1;
77     int h=y2-y1;
78     return 2*w+2*h;
79   }
80   void Clear()
81   {
82     time ( &rawtime );
83     string t=string(ctime(&rawtime));
84     string temp="Clear invoked: ";
85     temp+=t;
86     PushFunInvok(temp);
87     top=0;
88   }
89   void printAllInvoks()
90   {
91     for(int i=0; i<topInvok; i++)
92     {
93       string temp=funinvokes[i];
94       cout<<temp;
95     }
96   }
97   void PushFunInvok(std::string str)
98   {
99     if (topInvok < 100)
100    {
101      funinvokes[topInvok]=str;
102      topInvok++;
103    }
104    else
105      ErrorInSizeFunInvok();
106  }
107  void Push(int i)
108  {
109    time ( &rawtime );
110    string t=string(ctime(&rawtime));
111    string temp="Push invoked: ";
112    temp+=t;
```

```
113        PushFunInvok(temp);
114        if (top < 100)
115        {
116            stk[top]=i;
117            top++;
118        }
119        else
120            ErrorInSize();
121    }
122    int Pop()
123    {
124      time ( &rawtime );
125      string t=string(ctime(&rawtime));
126      string temp="Pop invoked: ";
127      temp+=t;
128      PushFunInvok(temp);
129      if (top > 0)
130      {
131          top--;
132          int temp_int=stk[top];
133          return temp_int;
134      }
135      else
136      {
137          ErrorInSize();
138          return -1;
139      }
140    }
141    int Size()
142    {
143      time ( &rawtime );
144      string t=string(ctime(&rawtime));
145      string temp="Size invoked: ";
146      temp+=t;
147      PushFunInvok(temp);
148      return top;
149    }
150 };
```

### A.1.2 Refactoring Process

Table A.2 shows the computations for how the slices for each data member is acquired.

Table 7.2 Computation of slices

| $DM_{Class1} = \{stk, top, funinvokes, topInvok, rawtime, x1, y1, x2, y2\}$ |
|---|
| **stk** |
| $ST_{stkxClass1} = \{116, 132\}$ |
| $SL_{116xClass1} = \{114, 116, 117, 120, 18, 19\}$ |
| $SL_{132xClass1} = \{129, 131, 132, 133, 137, 138, 18, 19\}$ |
| $SL_{stkxClass1} = SL_{116xClass1} \cup SL_{132xClass1}$<br>$= \{114,116,117,120,18,19,129,$<br>$131,132,133,137,138\}$ |

| **top** |
|---|
| $ST_{topxClass1} = \{19,32,87,114,116,117,129,$<br>$131,132,148\}$ |
| $SL_{19xClass1} = \{19\}$ |
| $SL_{32xClass1} = \{32\}$ |
| $SL_{87xClass1} = \{87\}$ |
| $SL_{114xClass1} = \{114,116,117,120,18,19\}$ |
| $SL_{116xClass1} = \{114,116,117,120,18,19\}$ |
| $SL_{117xClass1} = \{114,116,117,120,18,19\}$ |

| | |
|---|---|
| $SL_{129xClass1}$ ={129,131,132,133,137,18,19,138}<br>$SL_{131xClass1}$ ={129,131,132,133,137,18,19,138}<br>$SL_{132xClass1}$ ={129,131,132,133,137,18,19,138}<br>$SL_{148xClass1}$ ={148}<br>$SL_{topxClass1}$ = $\bigcup_{s \in STtopxClass1} SLsxClass1$<br><br>  = {19,32,87,114,116,117,120,18,<br>    129,131,132,133,137,138,148} | $SL_{144xClass1}$ ={144,145,146,147,101,<br>      102,99,105,12,13}<br>$SL_{rawtimexClass1}$ ={27,28,29,30,31, 41,42,43,44,45,50,<br>      51,52,53,54,59, 60,61,62,63,71,72,<br>      73,74,75,82, 83,84,85,86, 109, 110,<br>      111,112,113, 124,125,126,127,128,<br>      143, 144,145,146,147, 101,102,99,<br>      105,12,13} |
| **funinvok** | **x1** |
| $ST_{funinvokesxClass1}$ ={93, 101}<br>$SL_{93xClass1}$ ={91,93,94}<br>$SL_{101xClass1}$ ={99,101,102,105,12,13}<br>$SL_{funinvokxClass1}$ = {91,93,94,99,101,102,105,12,13} | $ST_{x1xClass1}$ ={33,55,64,76}<br>$SL_{33xClass1}$ ={33}<br>$SL_{55xClass1}$ ={55}<br>$SL_{64xClass1}$ ={64,65,66,67}<br>$SL_{76xClass1}$ ={76,77,78}<br>$SL_{x1xClass1}$ =={33,55,64,65,66,67,76,77,78} |
| **topInvok** | **x2** |
| $ST_{topInvokxClass1}$ ={13,26,91,99,101,102}<br>$SL_{13xClass1}$ ={13}<br>$SL_{26xClass1}$ ={26}<br>$SL_{91xClass1}$ ={91,93,94}<br>$SL_{99xClass1}$ ={99,101,102,105,12,13}<br>$SL_{101xClass1}$ ={99,101,102,105,12,13}<br>$SL_{102xClass1}$ ={99,101,102,105,12,13}<br>$SL_{topInvokxClass1}$ = {13,26,91,93,94,<br>        99,101,102,105,12} | $ST_{x2xClass1}$ ={35,55,64,76}<br>$SL_{35xClass1}$ ={35}<br>$SL_{55xClass1}$ ={55}<br>$SL_{64xClass1}$ ={64,65,66,67}<br>$SL_{76xClass1}$ ={76,77,78}<br>$SL_{x2xClass1}$ =={35,55,64,65,66,67,76,77,78} |
| **rawtime** | **y1** |
| $ST_{rawtimexClass1}$ ={27,28,41,42,50,51,59,60,71,72,<br>         82,83,109,110,124,125,143,144}<br>$SL_{27xClass1}$ ={27}<br>$SL_{28xClass1}$ ={28,29,30,31,101,102,99,105,12,13}<br>$SL_{41xClass1}$ ={41}<br>$SL_{42xClass1}$ ={42,43,44,45,101,102,99,105,12,13}<br>$SL_{50xClass1}$ ={50}<br>$SL_{51xClass1}$ ={51,52,53,54,101,102,99,105,12,13}<br>$SL_{59xClass1}$ ={59}<br>$SL_{60xClass1}$ ={60,61,62,63,101,102,99,105,12,13}<br>$SL_{71xClass1}$ ={71}<br>$SL_{72xClass1}$ ={72,73,74,75,101,102,99,105,12,13}<br>$SL_{82xClass1}$ ={82}<br>$SL_{83xClass1}$ ={83,84,85,86,101,102,99,105,12,13}<br>$SL_{109xClass1}$ ={109}<br>$SL_{110xClass1}$ ={110,111,112,113,101,<br>          102,99,105,12,13}<br>$SL_{124xClass1}$ ={124}<br>$SL_{125xClass1}$ ={125,126,127,128,101,<br>          102,99,105,12,13}<br>$SL_{143xClass1}$ ={143} | $ST_{y1xClass1}$ ={34,46,65,77}<br>$SL_{34xClass1}$ ={34}<br>$SL_{46xClass1}$ ={46}<br>$SL_{65xClass1}$ ={64,65,66,67}<br>$SL_{77xClass1}$ ={76,77,78}<br>$SL_{y1xClass1}$ =={34,46,64,65,66,67,76,77,78} |
| | **y2** |
| | $ST_{y2xClass1}$ ={36,46,65,77}<br>$SL_{36xClass1}$ ={36}<br>$SL_{46xClass1}$ ={46}<br>$SL_{65xClass1}$ ={64,65,66,67}<br>$SL_{77xClass1}$ ={76,77,78}<br>$SL_{y2xClass1}$ =={36,46,64,65,66,67,76,77,78} |

And in Table A.3, we see the intersections for the slices of each data member in Class 1. Based

on this data, we construct our final DSG for this class.

Table 7.3 Intersections of Slices

$SL_{stkxClass1}$ ={114,116,117,120,18,19,129,131,132,133,137,138}
$SL_{topxClass1}$={19,32,87,114,116,117,120,18,129,131,132,133,137,138,148}
$SL_{funinvokxClass1}$ ={91,93,94,99,101,102,105,12,13}
$SL_{topInvokxClass1}$ ={13,26,91,93,94,99,101,102,105,12}
$SL_{rawtimexClass1}$ ={27,28,29,30,31, 41,42,43,44,45, 50,51,52,53,54,59,60,61,62,63,71,72,73,74,75,82,83,84,
85,86,109, 110,111,112,113, 124,125,126,127,128,143, 144,145,146,147,101,102,99,105,
12,13}
$SL_{x1xClass1}$ ={33,55,64,65,66,67,76,77,78}
$SL_{x2xClass1}$ ={35,55,64,65,66,67,76,77,78}
$SL_{y1xClass1}$ ={34,46,64,65,66,67,76,77,78}
$SL_{y2xClass1}$ ={36,46,64,65,66,67,76,77,78}

| ∩ | stk | top | funinvok | topInvok | rawtime | x1 | y1 | x2 | y2 |
|---|---|---|---|---|---|---|---|---|---|
| stk | ∩ | ∩ | Ø | Ø | Ø | Ø | Ø | Ø | Ø |
| top | ∩ | ∩ | Ø | Ø | Ø | Ø | Ø | Ø | Ø |
| funinvok | Ø | Ø | ∩ | ∩ | ∩ | Ø | Ø | Ø | Ø |
| topInvok | Ø | Ø | ∩ | ∩ | ∩ | Ø | Ø | Ø | Ø |
| rawtime | Ø | Ø | ∩ | ∩ | ∩ | Ø | Ø | Ø | Ø |
| x1 | Ø | Ø | Ø | Ø | Ø | ∩ | ∩ | ∩ | ∩ |
| y1 | Ø | Ø | Ø | Ø | Ø | ∩ | ∩ | ∩ | ∩ |
| x2 | Ø | Ø | Ø | Ø | Ø | ∩ | ∩ | ∩ | ∩ |
| y2 | Ø | Ø | Ø | Ø | Ø | ∩ | ∩ | ∩ | ∩ |

## A.1.3 Refactoring Result

Table A.4 shows the original class and the new classes that we generated after the suggested

refactoring is applied to this class as explained in Chapter 3.

Table 7.4 Refactoring Result

### Restructured Original Class - Class1

```
1    class Class1
2    {
3     private:
4      New1* n1;
5      New2* n2;
6      New3* n3;
7      void ErrorInSizeFunInvok()
8      {
9            n2->fun2_1();
10     }
11     void ErrorInSize()
12     {
13           n1->fun1_1();
14     }
15    public:
16     Class1(int left=0,int up=0,
17             int right=0,int bottom=0)
18     {
19      n1=new New1();
20      n2=new New2();
21      n3=new New3(left,up,right,bottom);
22     }
23     ~Class1() {}
24     int Height()
25     {
26       n2->fun2_2();
27       return n3->fun3_1();
28     }
29     int Width()
30     {
31       n2->fun2_3();
32       return n3->fun3_2();
33     }
34     int Area()
35     {
36       n2->fun2_4();
37       return n3->fun3_3();
38     }
39     int Perimeter()
40     {
41       n2->fun2_5();
42       return n3->fun3_4();
43     }
44     void Clear()
45     {
46       n2->fun2_6();
47       n1->fun1_2();
48     }
49     void printAllInvoks()
50     {
51       n2->fun2_7();
52     }
53     void PushFunInvok(std::string str)
54     {
55       n2->fun2_8(str);
56     }
57     void Push(int i)
58     {
59       n2->fun2_9();
60       n1->fun1_3(i);
61     }
62     int Pop()
63     {
64       n2->fun2_10();
65       return n1->fun1_4();
66     }
67     int Size()
68     {
69       n2->fun2_11();
70       return n1->fun1_5();
71     }
72   };
```

### Extracted Class 1

```
1    class New1
2    {
3     private:
4      int stk[100];
5      int top;
6     public:
7      New1()
8      {
9        top=0;
10     }
11     void fun1_1()
12     {
13       cout<<"Index out of range!\n";
14       cout<<"The Array has "<<top
15             <<" elements.\n";
16     }
17     void fun1_2()
18     {
19         top=0;
20     }
21     void fun1_3(int i)
22     {
23         if (top < 100)
24         {
25             stk[top]=i;
26             top++;
27         }
28         else
29             fun1_1();
30     }
31     int fun1_4()
32     {
33       if (top > 0)
34       {
35         top--;
36         int temp_int=stk[top];
```

```cpp
37          return temp_int;
38        }
39      else
40      {
41          fun1_1();
42          return -1;
43      }
44    }
45    int fun1_5()
46    {
47      return top;
48    }
49  };
```

### Extracted Class 2

```cpp
1   class New2
2   {
3     private:
4       string funinvokes[100];
5       int topInvok;
6       time_t rawtime;
7     public:
8     New2()
9     {
10      topInvok=0;
11      time ( &rawtime );
12      string t=string(ctime(&rawtime));
13      string temp="Class1 invoked: ";
14      temp+=t;
15      fun2_8(temp);
16    }
17    void fun2_1()
18    {
19        cout<<"Index out of range!\n";
20        cout<<"The Array has "<<topInvok
21            <<" elements.\n";
22    }
23    void fun2_2()
24    {
25      time ( &rawtime );
26      string t=string(ctime(&rawtime));
27      string temp="Height invoked: ";
28      temp+=t;
29      fun2_8(temp);
30    }
31    void fun2_3()
32    {
33      time ( &rawtime );
34      string t=string(ctime(&rawtime));
35      string temp="Width invoked: ";
36      temp+=t;
37      fun2_8(temp);
38    }
39    void fun2_4()
40    {
41      time ( &rawtime );
42      string t=string(ctime(&rawtime));
43      string temp="Area invoked: ";
44      temp+=t;
45      fun2_8(temp);
46    }
47    void fun2_5()
48    {
49      time ( &rawtime );
50      string t=string(ctime(&rawtime));
51      string temp="Perimeter invoked:";
52      temp+=t;
53      fun2_8(temp);
54    }
55    void fun2_6()
56    {
57      time ( &rawtime );
58      string t=string(ctime(&rawtime));
59      string temp="Clear invoked: ";
60      temp+=t;
61      fun2_8(temp);
62    }
63    void fun2_7()
64    {
65      for(int i=0; i<topInvok; i++)
66      {
67          string temp=funinvokes[i];
68          cout<<temp;
69      }
70    }
71    void fun2_8(string str)
72    {
73      if (topInvok < 100)
74      {
75          funinvokes[topInvok]=str;
76          topInvok++;
77      }
78      else
79          fun2_1();
80    }
81    void fun2_9()
82    {
83      time ( &rawtime );
84      string t=string(ctime(&rawtime));
85      string temp="Push invoked: ";
86      temp+=t;
87      fun2_8(temp);
88    }
89    void fun2_10()
90    {
91      time ( &rawtime );
92      string t=string(ctime(&rawtime));
93      string temp="Pop invoked: ";
94      temp+=t;
95      fun2_8(temp);
96    }
97    void fun2_11()
98    {
99      time ( &rawtime );
100     string t=string(ctime(&rawtime));
101     string temp="Size invoked: ";
```

| | |
|---|---|
| 102 | `    temp+=t;` |
| 103 | `    fun2_8(temp);` |
| 104 | `  }` |
| 105 | `};` |

<div align="center">Extracted Class 3</div>

```
1    class New3
2    {
3      private:
4        int x1, y1, x2, y2;
5      public:
6      New3(int left,int up,
7            int right,int bottom)
8      {
9        x1=left;
10       y1=up;
11       x2=right;
12       y2=bottom;
13     }
14     int fun3_1()
15     {
16        return (y2-y1);
17     }
18     int fun3_2()
19     {
20        return (x2-x1);
21     }
22     int fun3_3()
23     {
24        int w=x2-x1;
25        int h=y2-y1;
26        int a=w*h;
27        return a;
28     }
29     int fun3_4()
30     {
31        int w=x2-x1;
32        int h=y2-y1;
33        return 2*w+2*h;
34     }
35   };
```

## A.2 Hammock Graph Based Refactoring Example

To demonstrate we refractor using our analysis and visualization tool, in this section we provide step by step guide to the application of this refactoring on an example method. The initial graph, extended graph and different refactoring opportunities are shown in Figure A.1 for this example method.
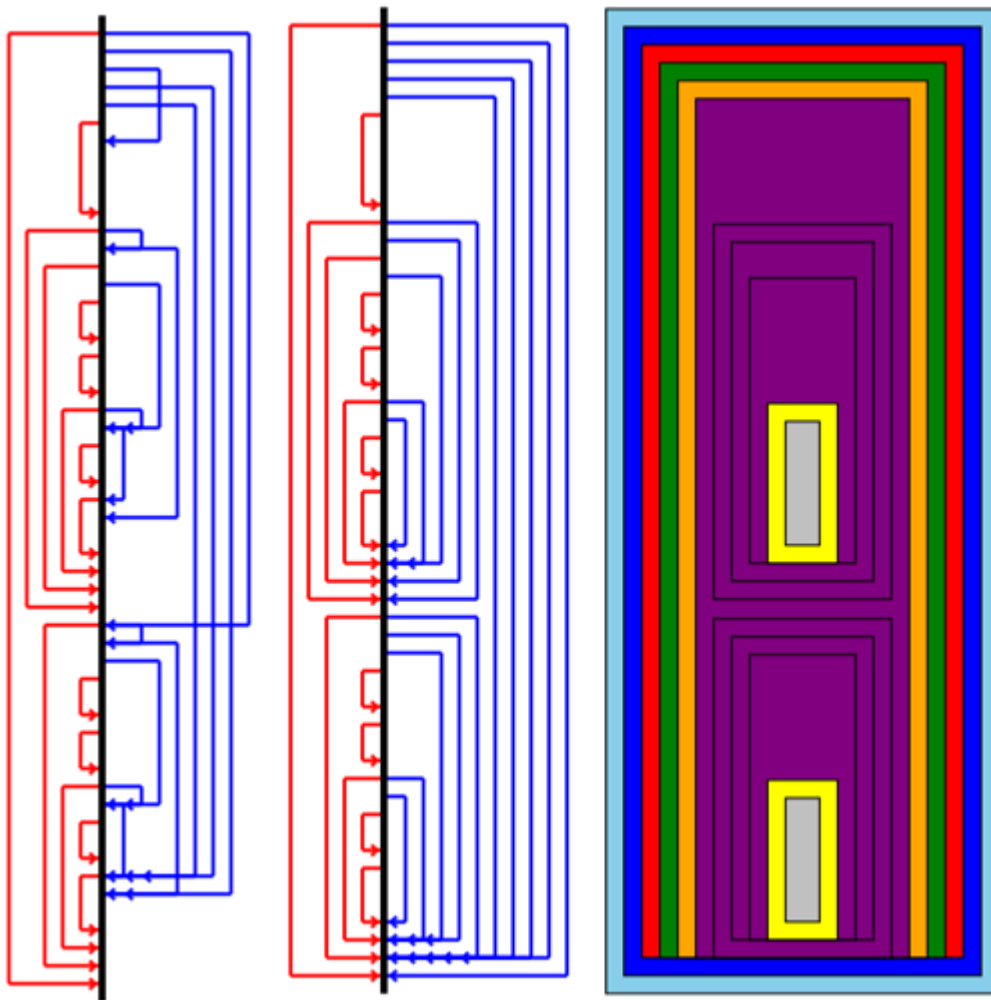


Figure 7.1 Original Method

This figure shows all refactoring opportunities that require at most 6 arguments, a number we arbitrarily chose. As stated earlier, the user may not decide to apply all the refactoring suggested

by our tool. For that, using the code view user can inspect each one of these suggestion quickly by clicking on the box for which he/she wants to see the code fragment and evaluate the feasibility of the extraction from code view. One such view is shown in Figure A.2.



```
CodeView

273    int nbCounted = processAll(ProcessCountAll,& _options);
274    generic_string result = TEXT("");
275    if(nbCounted < 0){
276            result = TEXT("The regular expression to search is malformed.\r\nDoes it result in nothing?");
277    }
278    else {
279            TCHAR moreInfo[128];
280            if(nbCounted == 1){
281                    wsprintf(moreInfo,TEXT("1 match."));
282            }
283            else {
284                    wsprintf(moreInfo,TEXT("%d matches."),nbCounted);
285            }
286            result = moreInfo ;
287    }
288    if(isMacroRecording){
289            saveInMacro(wParam,FR_OP_FIND);
290    }
291    ::MessageBox(_hParent,result.c_str(),TEXT("Count"),MB_OK);
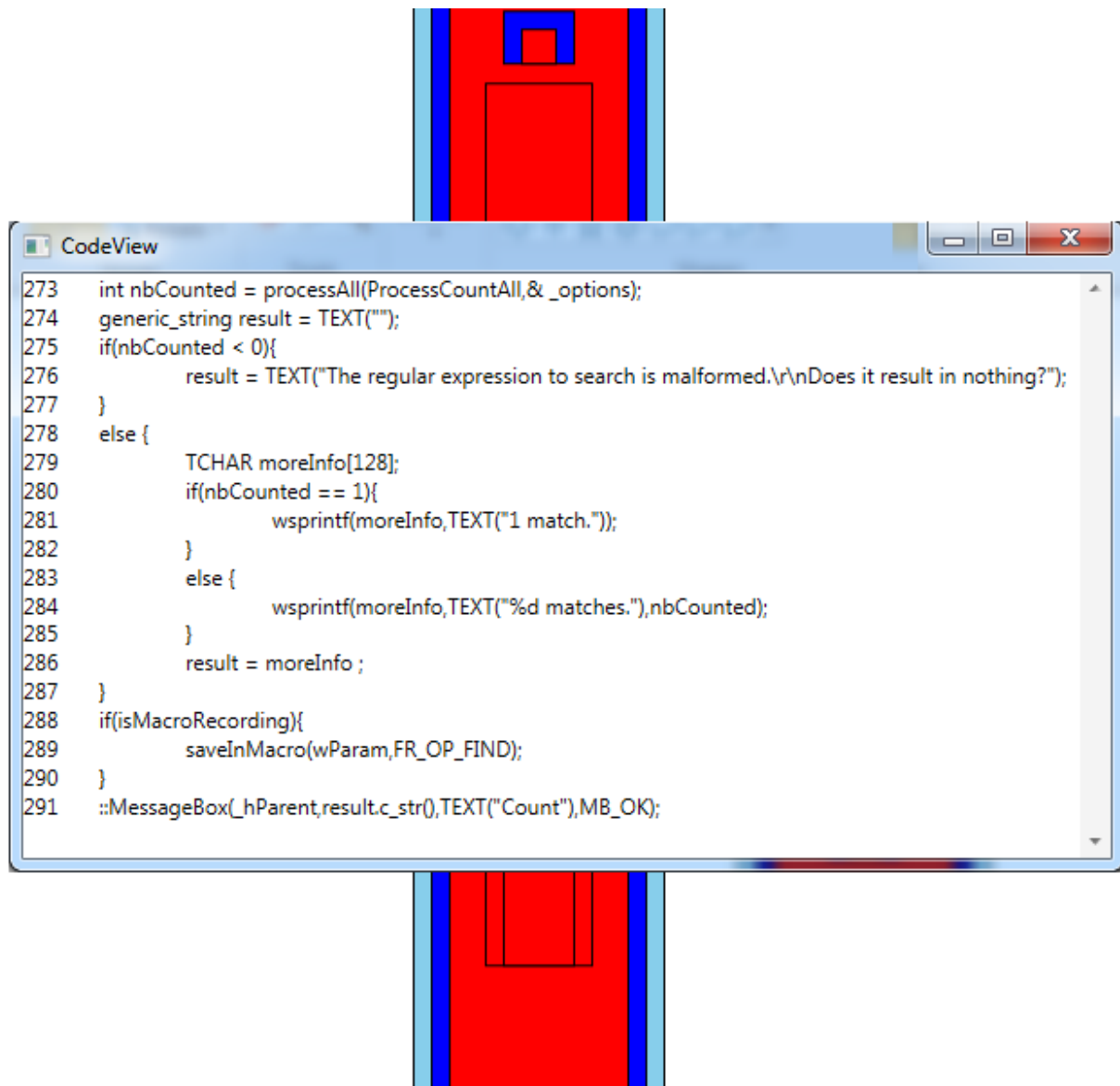```

Figure 7.2 Code View for Refactoring Suggestion

We evaluated the suggested re-factorings and applied two extract method re-factorings for this particular method. We analyzed the resulting methods to seek further refactoring opportunities and provided the views for new methods and the original method after refactoring in Figure A.3.
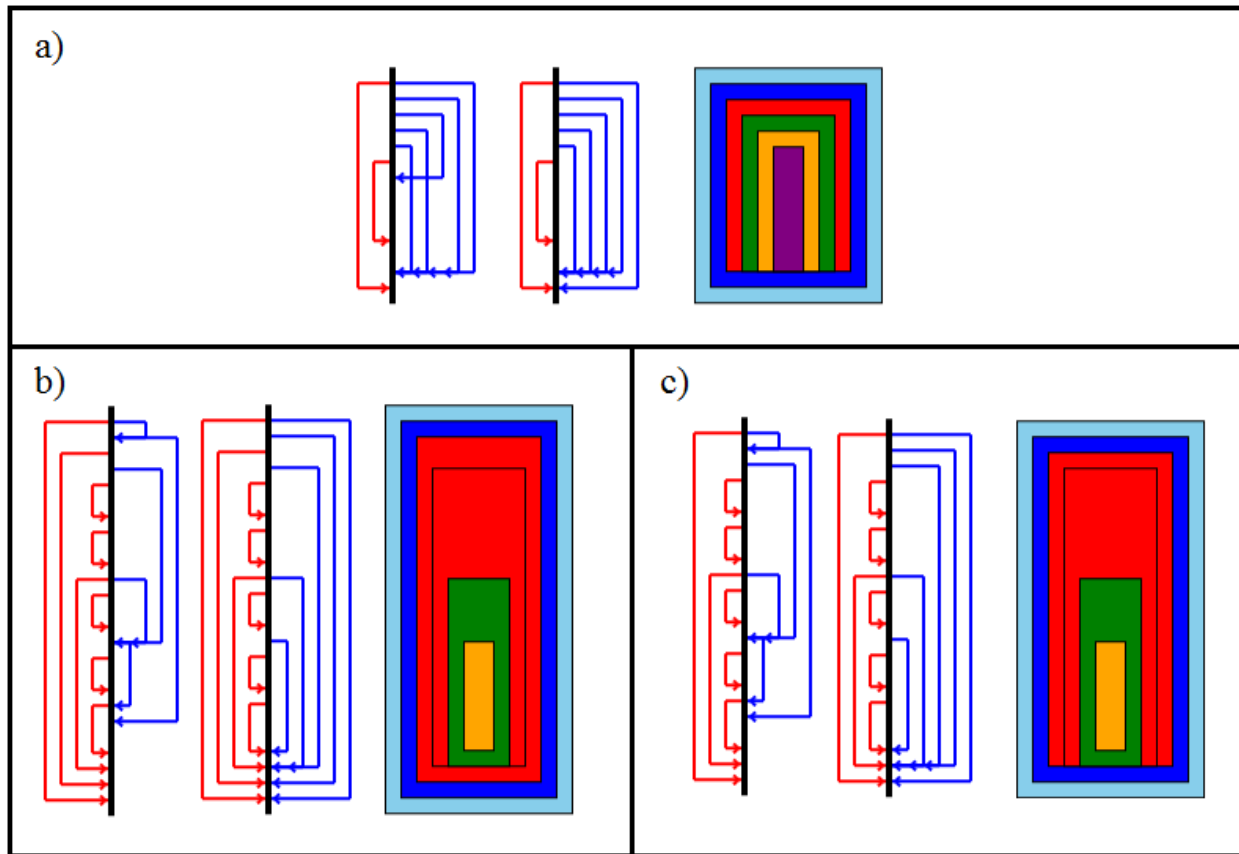
Figure 7.3 a) Original Method b-c) Extracted Methods

# Bibliography

[1] IEEE Standard for Software Maintenance," IEEE Std 1219-1998 , 1998

[2] Grady, Robert B, "Practical Software Metrics For Project Management and Process Improvement," Prentice Hall, Englewood Cliffs, NJ (1992)

[3] Hunt, B.; Turner, B.; McRitchie, K., "Software Maintenance Implications on Cost and Schedule," Aerospace Conference, 2008 IEEE , vol., no., pp.1,6, 1-8 March 2008

[4] Tu Honglei; Sun Wei; Zhang Yanan, "The Research on Software Metrics and Software Complexity Metrics," Computer Science-Technology and Applications, 2009. IFCSTA '09. International Forum on , vol.1, no., pp.131,136, 25-27 Dec. 2009

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the Design of Existing Code," Addison Wesley, Boston, MA, 1999.

[6] Parnas, D.L., "Software aging," Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on , vol., no., pp.279,287, 16-21 May 1994

[7] Marcus, A.; Poshyvanyk, D.; Ferenc, R., "Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems," Software Engineering, IEEE Transactions on , vol.34, no.2, pp.287,300, March-April 2008

[8] Yourdon, E.; Constantine, L L., "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design," Yourdon Press, 1979

[9] M. Weiser, "Program slices: formal, psychological, and practical investigations of an automatic program abstraction method," PhD thesis, University of Michigan, Ann Arbor, 1979.

[10] M. Weiser, "Program Slicing," IEEE Transactions on Software Engineering, vol. 10, no. 4, pp. 352-357, July 1984

[11] M. Weiser, "Program Slicing," Proceedings of the 5th International Conference on Software Engineering, pp. 439-449, 1981

[12] S.G.Shirinivas, S.Vetrivel and N.M.Elango,  "APPLICATIONS OF GRAPH THEORY IN COMPUTER SCIENCE AN OVERVIEW". International journal of engineering science and technology (0975-5462), 2 (9), p. 4610

[13] Fubo Zhang , Erik H. D'Hollander, "Using Hammock Graphs to Structure Programs," IEEE Transactions on Software Engineering, v.30 n.4, p.231-245, April 2004

[14] D. Turo and B. Johnson, "Improving the visualization of hierarchies with treemaps: design issues and experimentation," Visualization, 1992. Visualization '92, Proceedings., IEEE Conference on, pp.124-131, 19-23 Oct 1992.

[15]    M. Harman, S. Danicic, B. Sivagurunathan, B. Jones, and Y. Sivagurunathan. "Cohesion metrics," In Proceedings of the 8th International Quality Week, San Francisco CA, May 1995.

[16]    J. M. Bieman, and L. M. Ott, "Measuring functional cohesion," IEEE Trans. Softw. Eng., vol. 20, no. 8, pp. 644–657, Aug 1994.

[17]    S. Karstu, "An examination of the behavior of slice based cohesion measures," Master's thesis, Department of Computer Science, Michigan Technological University, 1994.

[18]    H. D. Longworth, "Slice based program metrics," Master's thesis, Michigan Technological University, 1985.

[19]    T. Meyers, and D. W. Binkley, "Slice-based cohesion metrics and software intervention," In 11th IEEE Working Conference on Reverse Engineering (WCRE 2004), pp. 256–265, 8-12 Nov 2004

[20]    L. M. Ott, and J. M. Bieman, "Program slices as an abstraction for cohesion measurement," Information and Software Technology, vol. 40, no. 11/12, pp. 691–700, 1998.

[21]    L. M. Ott, and J. J. Thuss, "The relationship between slices and module cohesion," In Proceedings of the 11th ACM conference on Software Engineering, pp.198–204, 1989.

[22]    L. M. Ott, and J. J. Thuss, "Slice based metrics for estimating cohesion," In Proceedings of the IEEE-CS International Software Metrics Symposium, pp. 71–81,  21-22 May 1993.

[23]    Keith Cassell, Craig Anslow, Lindsay Groves, and Peter Andreae. 2011. Visualizing the refactoring of classes via clustering. In Proceedings of the Thirty-Fourth Australasian Computer Science Conference - Volume 113 (ACSC '11), Mark Reynolds (Ed.), Vol. 113. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 63-72.

[24]    J. Krinke, "Statement-Level Cohesion Metrics and their Visualization," Seventh IEEE Int. Working conference on Source Code Analysis and Manipulation, pp. 37-48, Sept. 30 2007-Oct. 1 2007

[25]    B. Gupta, "A critique of cohesion measures in the object oriented paradigm," Master's thesis, Department of Computer Science, Michigan Technological University, 1997.

[26]    Murphy-Hill, E.; Ayazifar, M.; Black, A.P., "Restructuring software with gestures," Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on , pp.165,172, 18-22 Sept. 2011

[27]    de F Carneiro, G.; Silva, M.; Mara, L.; Figueiredo, E.; Sant'Anna, C.; Garcia, A.; Mendonca, M., "Identifying Code Smells with Multiple Concern Views," Software Engineering (SBES), 2010 Brazilian Symposium on, pp.128,137, Sept. 27 2010-Oct. 1 2010

[28]    S. R. Chidamber, and C. F. Kemerer, "Towards a Metrics Suite for Object-Oriented Design," Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, Vol. 26, No. 10, pp. 197-211, October 1991.

[29]    S. R. Chidamber, and C. F. Kemerer, "A Metrics suite for object Oriented Design," IEEE Transactions on Software Engineering, Vol. 20, No. 6, pp. 476-493, June 1994.

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

[31]    M. Hitz, and B. Montazeri, "Measuring coupling and cohesion in object oriented systems," Proceedings of the Int. Symposium on Applied Corporate Computing, October 1995.

[32]    J. M. Bieman and B.-K. Kang, "Cohesion and Reuse in an Object-Oriented System," Proc. Symp. Software Reusability, pp. 259-262, 1995.

[33]    H. S. Chae, Y. R. Kwon, and D. H. Bae, "Improving Cohesion Metrics for Classes by Considering Dependent Instance Variables," IEEE Transaction on Software Engineering, vol. 30, no. 11, pp. 826-832, November 2004,

[34]    G. Şerban, I-G. Czibula, "Restructuring Software Systems Using Clustering," 22nd Int. Symp. on Computer and Information Sciences (ISCIS), pp. 1-6, 7-9 Nov 2007

[35] http://www.ecs.syr.edu/faculty/fawcett/handouts/research/kaya/files/appendix.pdf

[36]    A. Lakhotia, and J.-C. Deprez, "Restructuring Functions with Low Cohesion," Proc. Sixth Working Conf. Reverse Eng. (WCRE), pp. 36-46, 6-8 Oct 1999.

[37] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Trans. Programming Lang. and Syst. (TOPLAS),vol. 12, no. 1, pp. 26-60, Jan. 1990

[38] M. Kaya and J. W. Fawcett, "Identifying Extract Method Opportunities Based on Variable Reference," 25th International Conference on Software Engineering and Knowledge Engineering (SEKE, 2013), Boston,USA,  June 27 - 29, 2013

[39] R. Komondoor and S. Horwitz, "Effective, Automatic Procedure Extraction," Proceedings of the 11th IEEE International Workshop on Program Comprehension, pp.33, May 10-11, 2003.

[40] M. Kaya and J. W. Fawcett, "A New Cohesion Metric and Restructuring Technique for Object Oriented Paradigm," Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual , pp.296-301, 16-20 July 2012.

[41] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method Refactoring Opportunities," Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on , pp.119-128, 24-27 March 2009.

[42] T. Sharma, "Identifying extract-method refactoring candidates automatically," In Proceedings of the Fifth Workshop on Refactoring Tools (WRT '12). ACM, New York, NY, USA, pp.50-53, 2012.

[43] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing," SIGSOFT Softw. Eng. Notes 26, pp.31-40, May 2001.

[44] A. Lakhotia and J.-C. Deprez, "Restructuring Programs by Tucking Statements into Functions," Information and Software Technology, vol. 40, no. 11-12, pp. 677-690,1998.

[45] R. Komondoor and S. Horwitz, "Semantics-preserving procedure extraction," In Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '00). ACM, New York, NY, USA, pp.155-169, 2000.

[46] Simon, F.; Steinbruckner, F.; Lewerentz, C., "Metrics based refactoring," Software Maintenance and Reengineering, 2001. Fifth European Conference on , vol., no., pp.30,38, 2001

[47] D. Turo and B. Johnson, "Improving the visualization of hierarchies with treemaps: design issues and experimentation," Visualization, 1992. Visualization '92, Proceedings., IEEE Conference on, pp.124-131, 19-23 Oct 1992.

[48] http://notepad-plus-plus.org/

[49]  IEEE Standard for Developing Software Life Cycle Processes," IEEE Std 1074-1995 , 1996

[50]  Draft Standard for Software engineering - Software life cycle processes - maintenance," IEEE Std P14764, Nov 2004

[51]  Mealy, E.; Strooper, P., "Evaluating software refactoring tool support," Software Engineering Conference, 2006. Australian , pp.10 pp.,, 18-21 April 2006

[52]  Mark Weiser, Reconstructing sequential behavior from parallel behavior projections, Information Processing Letters, Volume 17, Issue 3, 5 October 1983, Pages 129-135

[53]  Dominguez, R.; Kaeli, D.R., "Unstructured Control Flow in GPGPU," Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International , vol., no., pp.1194,1202, 20-24 May 2013

[54]  Walkinshaw, N.; Roper, M.; Wood, M., "Understanding object-oriented source code from the behavioural perspective," Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on , vol., no., pp.215,224, 15-16 May 2005

[55]  Apiwattanapong, T.; Orso, A.; Harrold, M.J., "A differencing algorithm for object-oriented programs," Automated Software Engineering, 2004. Proceedings. 19th International Conference on , vol., no., pp.2,13, 20-24 Sept. 2004

[56]  J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In ICSM, 1992.

[57]  Jeanne Ferrante , Karl J. Ottenstein , Joe D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems (TOPLAS), v.9 n.3, p.319-349, July 1987

[58]

http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm

[59] Joshi, P.; Joshi, R.K., "Quality Analysis of Object Oriented Cohesion Metrics," 2010 Seventh International Conference on the Quality of Information and Communications Technology (QUATIC), pp.319,324, Sept. 29 2010-Oct. 2 2010

[60] H. S. Chae and Y. R. Kwon, "A cohesion measure for classes in object-oriented systems," in Proceedings of Fifth International Software Metrics Symposium, Nov 1998, pp. 158–166.

[61] H. S. Chae, Y. R. Kwon, and D. H. Bae, "A cohesion measure for object-oriented classes," Software- Practice and Experience, vol. 30, pp. 1405–1431, 2000.

[62] M. Lanza, " Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques" Proceedings of the 4th International Workshop on Principles of Software Evolution, pp. 37-42, 2001.

[63] Chris Parnin, Carsten Görg, and Ogechi Nnadi. 2008. A catalogue of lightweight visualizations to support code smell inspection. In Proceedings of the 4th ACM symposium on Software visualization (SoftVis '08). ACM, New York, NY, USA

[64] Zhenchang Xing; Stroulia, E., "Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study," Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on , vol., no., pp.458,468, 24-27 Sept. 2006

[65] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How we refactor, and how we know it. In Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, Washington, DC, USA, 287-297

[66] Don Roberts, John Brant, and Ralph Johnson. 1997. A refactoring tool for Smalltalk. Theor. Pract. Object Syst. 3, 4 (October 1997), 253-263

[67] Yuming Zhou, Jiangtao Lu, and Hongmin Lu Baowen Xu. 2004. A comparative study of graph theory-based class cohesion measures. SIGSOFT Softw. Eng. Notes 29, 2 (March 2004), 13-13.

[68] Gall, H.; Jazayeri, M.; Riva, C., "Visualizing software release histories: the use of color and third dimension," Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on , pp.99,108, 1999

[69] Michele Risi and Giuseppe Scanniello. 2012. MetricAttitude: a visualization tool for the reverse engineering of object oriented software. In Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI '12), Genny Tortora, Stefano Levialdi, and Maurizio Tucci (Eds.). ACM, New York, NY, USA, 449-456

[70] Ducasse, S.; Pollet, D.; Suen, M.; Abdeen, H.; Alloui, I., "Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships," Software Maintenance, 2007. ICSM 2007. IEEE International Conference on , pp.94,103, 2-5 Oct. 2007

[71] M. K. Gungor, " Structural models for large software systems " PhD Dissertation, Syracuse Uiveristy, 2006

[72] Yip, S.W.L.; Lam, T., "A software maintenance survey," Software Engineering Conference, 1994. Proceedings., 1994 First Asia-Pacific , vol., no., pp.70,79, 7-9 Dec 1994

[73] Emerson Murphy-Hill and Andrew P. Black. 2008. Breaking the barriers to successful refactoring: observations and tools for extract method. In Proceedings of the 30th international conference on Software engineering (ICSE '08). ACM, New York, NY, USA, 421-430.

[74] M. D. Ernst. "Practical fine-grained static slicing of optimized code." Technical Report MSR-TR-94-14, 1994.

[75] Alexis O'Connor, Macneil Shonle, and William Griswold. 2005. Star diagram with automated refactorings for Eclipse. In Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange (eclipse '05). ACM, New York, NY, USA, 16-20.

[76] Norman E. Fenton and Shari Lawrence Pfleeger. 1998. Software Metrics: A Rigorous and Practical Approach (2nd ed.). PWS Pub. Co., Boston, MA, USA.

[77] Ben Shneiderman. 1992. Tree visualization with tree-maps: 2-d space-filling approach. ACM Trans. Graph. 11, 1 (January 1992), 92-99.

[78] Riel, A. Object-Oriented Design Heuristics. Addison-Wesley Professional, 1996.

[79]  http://www.jetbrains.com/resharper/

[80] https://www.devexpress.com/Products/CodeRush/refactor_pro.xml

[81] https://www.eclipse.org/

[82] http://msdn.microsoft.com/en-US/vstudio/

[83] Mark Weiser. 1982. Programmers use slices when debugging. Commun. ACM 25, 7 (July 1982), 446-452.

[84] Mark Weiser and Jim Lyle. 1986. Experiments on slicing-based debugging aids. In Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers, Elliot Soloway and Sitharama Iyengar (Eds.). Ablex Publishing Corp., Norwood, NJ, USA, 187-197.

# Vita

**NAME OF AUTHOR**     Mehmet Kaya

**PLACE OF BIRTH**     Sanliurfa, Turkey

**DATE OF BIRTH**     February 9, 1986

**EDUCATION**

June 2014     Ph.D. in Electrical and Computer Engineering,

Department of Electrical Engineering and Computer Science,

Syracuse University,

Syracuse, NY, USA

August 2010     M.S. in Computer Science,

Department of Electrical Engineering and Computer Science,

Syracuse University,

Syracuse, NY, USA

May 2007     B.S. in Electronic and Computer Education,

Faculty of Technical Education ,

Suleyman Demirel University,

Isparta, Turkey

**EXPERIENCE**

Teaching     Bozova Vocational High School, Sanliurfa, Turkey, Fall 2007

Teaching Assistant     CIS 681 - Software Modeling And Analysis - Fall 2012, Fall 2013

CIS 776 - Design Patterns - Fall 2012

CSE 775 - Distributed Objects - Spring 2012

CSE 687 - Object Oriented Design - Spring 20122, Spring 2013