FUNCTIONALITY BASED REFACTORING: IMPROVING SOURCE CODE COMPREHENSION

by

JEFFREY LEE BEIKO

A thesis submitted to The School of Computing

in conformity with requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

September, 2007

Abstract

Software maintenance is the lifecycle activity that consumes the greatest amount of resources. Maintenance is a difficult task because of the size of software systems. Much of the time spent on maintenance is spent trying to understand source code. Refactoring offers a way to improve source code design and quality. We present an approach to refactoring that is based on the functionality of source code. Sets of heuristics are captured as patterns of source code. Refactoring opportunities are located using these patterns, and dependencies are verified to check if the located refactorings preserve the dependencies in the source code. Our automated tool performs the functional-based refactoring opportunities detection process, verifies dependencies, and performs the refactorings that preserve dependencies. These refactorings transform the source code into a series of functional regions of code, which makes it easier for developers to locate code they are searching for. This also creates a chunked structure in the source code, which helps with bottom-up program comprehension. Thus, this process reduces the amount of time required for maintenance by reducing the amount of time spent on program comprehension. We perform case studies to demonstrate the effectiveness of our automated approach on two open source applications.

Acknowledgement

First, I gratefully acknowledge my supervisor, Dr. Ying Zou, for her constructive comments and suggestions that led me through my research. I also thank you for your tremendous support. It has been a great experience to work with you and I have learned much.

I would like to express my gratitude to my committee members: Professor Suning Wang, Professor Parvin Mousavi, Professor Nick Graham and Professor Diane Kelly, for their valuable feedback on this thesis.

I also thank Debby Robertson for her continual help during my two years at Queen's University.

Many thanks go to the members of SERL laboratory: Qi Zhang, Alex Hung, Rongchao Chen, Xulin Zhao and Ben Kam, for making the SERL laboratory a great experience.

I would also like to thank my mother and brother John, and all of my family and friends for their encouragement and support.

Finally, I would like to thank my wife Lindsay, and my daughters Stephanie and Sydney, for being such a great family. This thesis is dedicated to my wife and children. I also dedicate this thesis to Molly, who touched the lives of many people and whom is greatly missed, every day.

Table of Contents

List of Tables

List of Figures and Illustrations

Chapter 1

Introduction

1.1 State of the Practice

Refactoring changes software in such a way that it does not alter the external behaviour of the code, but may improve its internal code structure [28]. Most notably, refactoring may enhance software quality by making source code easier to comprehend, maintain and reuse. Therefore, refactorings can reduce the cost associated with the software maintenance process. For example, the "pull-up" refactoring can pull behaviour or attributes that are common to multiple children classes to the parent class, centralizing duplicate behaviour to a single location [28]. As software systems continue to grow, maintenance becomes increasingly difficult. Studies indicate that up to 50% of the time spent on maintenance is spent understanding the source code [19][52]. This requires that we develop a method to reduce the amount of time required for maintenance. For large software systems we could employ automatic approaches to the help speed up the process of identifying refactoring opportunities in the source code.

Previous research utilizes metrics to detect problematic code blocks or design flaws such as long methods and dead code, through automation [20][27][48]. However, it is

difficult to determine appropriate thresholds for each of the selected metrics, for the reason that the threshold selection is very subjective and lacks empirical support [29][46][59][23]. For example, long methods (methods that have excessive size or functionality [28]) are considered as problematic code for a variety of reasons, such as excessive numbers of objects in a method, too many lines of code, or performing too many tasks. Lines of Code (LOC) is a common metric that is used to detect long methods. For instance, if a threshold of thirty lines of code is set, then any method with more than thirty lines of code is labelled a long method. However, there is no evidence which supports the selection of the threshold values, as is often the case with metrics based refactoring. As there are no automatic techniques that can signal poor structure in a method, the developer is often left to rely on their intuition of how the method structure might be improved.

1.2 Problem Definition

Although research yields many techniques for locating refactoring opportunities, very few of these avoid the subjectivity of choosing metrics thresholds. The developers must be attentive to and manage dependencies in the source code, while manually applying the refactorings. While refactoring usually improves the design of source code, there is no guarantee that the refactorings will improve the actual readability of source code, or help developers locate functionality they are searching for.

In addition, because of the large size of software applications, there needs to be an automated approach to identify and perform refactorings, as manually refactoring a software system whose size is in the millions of lines of code is time-consuming. When

automatically refactoring software systems, we need to have additional safety mechanisms in place in order to ensure that the refactorings being performed won't cause the software to behave incorrectly. We summarize the problems associated with automated refactoring as follows:

(1) *Lack of automated methods to refactor source code into a more comprehensible form:* Most automated approaches to locating problems in source code use a threshold for a set of given metrics. For example, to locate God Classes (classes which contain excessive data or functionality [31]), a threshold of ten data members could be set. Thus, any class with more than ten data members would be classified as a God Class. However, there is a need for empirical evidence that can support the threshold selection. In addition, refactorings usually address the design of source code. They do not always directly address improving source readability

(2) *Lack of dependency verification to ensure that automated refactorings preserve the behaviour of the source code:* Various types of dependencies exist in source code. These dependencies must be identified and appropriately preserved.

(3) *Lack of automated techniques to generate comments for source code:* Comments are often not included by developers. Having a technique to automatically generate comments for developers could help improve readability of source code.

1.3 Motivation

Comprehending source code is a time consuming task. Research has shown that the process of comprehending source code can happen in different ways. There are two main divisions among researchers [11][12][45][54][62][65][66]. One division states that

comprehension happens by developers focusing on what the program does. This is known as a functional approach. The other states that comprehension happens by focusing on how the program works. This is known as the control-flow approach.

Much research examines what is known as the bottom-up theory of program comprehension [53][63][67]. This theory states that programmers first read statements of code and then mentally group these statements into higher level abstractions. This is a process known as *chunking*. These chunks, or higher level abstractions of statements of code, are repeatedly combined until a high-level understanding of the program is achieved.

Although there are other methods of program comprehension which are as equally valid as the bottom-up theory, we focus on improving program comprehension through a bottom-up functional approach. (It is worth nothing that it has been suggested developers prefer a bottom-up approach to program comprehension [70]). To this end, we focus on developing a framework to refactor source code into functionally related regions of code. These regions of code represent statements that are effectively *chunked* together into a single location, based on the functionality of the code. Thus, our framework aims to automatically perform the process of chunking statements of code together for developers. This abstracts a section of code to the level of a functional feature, which exists at a higher abstraction than a statement of code.

In an effort to improve the readability of source code and reduce the amount of time it takes to comprehend source code, we have developed a set of strategies that search for

code patterns to perform this chunking process. Such code patterns identify groups of statements that deliver a specific functionality.

Definition 1-1: We define a *functional-based refactoring* as the result of executing a detection strategy. This means that a *functional-based refactoring* is a collection of statements that are grouped together (chunked) using one of the detection strategies. Each *functional-based refactoring* has its dependencies analyzed before the actual refactoring is performed. If performing the refactoring preserves the original behaviour of the code and does not violate any of the data and control dependencies which exist in the code, the *functional-based refactoring* opportunity is recognized as being valid.

We strive to group source code into regions of statements (chunks) that contribute to a single functional feature that can be re-structured into its own section of code. This process restructures blocks of code into functional regions of code and improves readability of source code. The objectives of the thesis are summarized as follows:

- *Identifying functional features*: We search source code to identify groups of statements of code which contribute to a single functional feature. These groups of code which are identified by our detection processes are recognized as groups of statements that could be refactored into functional regions of code.
- *Verifying dependencies*: With any refactoring, statements of code are modified and/or moved. It is important to ensure that the dependencies in the source code are respected to preserve the intended behaviour of the software. Our framework ensures that in

order for a given functional-based refactoring to be performed, the dependencies which exist in the code are not violated.

- *Restructuring blocks into functionally-related regions of code*: Our approach to refactoring centers on identifying functional regions of source code. After a functionality-based refactoring is located, it is created by locating all of the statements of code from the refactoring into a single continuous region (chunk) or segment of code. This results in source code which consists of functionally-related regions of code, where each functional region created provides a single feature. This new structure makes it easier for developers to find and chunk statements of code related to a feature by having the statements be grouped into their own region.

1.4 Thesis Statement

We aim to provide an automatic framework which safely reorders code into regions of code, where the statements share common functionality. This is in an effort to make it easier for developers to locate source code for a specific feature that they are searching for, by having the code for that feature exist in a single location. This also helps the program comprehension process by having the code automatically chunked for developers, in an effort to help them understand the program better. Our approach to verifying dependencies aims to preserve the original behaviour of the source code by ensuring that the reordering of statements is being safely done and does not violate any of the dependencies in the source code.

Traditional refactorings are loose heuristics which address a wide variety of problems in source code. For example, they deal with managing inheritance hierarchies,

effectively designing interaction between classes and properly distributing class and method responsibilities. Developers have many different alternatives to choose from when implementing different refactorings. The refactorings serve as loose guidelines to improve source code design.

Our research is different from traditional refactorings in two main ways: Firstly, traditional refactorings address a wide variety of problems in source code while our functionality-based refactorings center on identifying functionally-related statements of code, and creating groupings of these statements, which locates the statements into a continuous segment of code. Our overall goal is to structure source code at the block level and make it easier for developers to read source code and find code they are searching for. Traditional refactorings address problems in source code at all levels (block, method, class, module, etc…) and have the goal of improving the overall design of software applications. Secondly, traditional refactorings use general heuristics that serve to guide developers in applying the refactorings to their source code. Our strategies for detecting refactoring opportunities are specific templates that are automatically applied.

1.5 Contribution

Many factors contribute to the ease with which a human can comprehend source code. One area is the actual documentary structure of source code. This area is of great importance because reading source code is central to the development and maintenance process [32]

Many factors contribute to improving the actual readability of source code such as indentation, comments, inter-token spacing and line breaks [50][51][6][69]. We apply

these principles during our refactoring process. In addition, we implement a framework to locate functional-based refactorings and verify dependencies in source code. We also automatically perform our refactoring process and generate comments for the code we refactor.

The main contributions of our research are the following:

- *Division of source code into functional regions*: We focus on identifying and creating functional regions of source code. This can improve the overall readability of source code by locating into a continuous functional region, statements of code which share common properties. Various heuristics are captured as detection strategies, which produce groups of statements that share common functionality. These groups of statements, or functional regions, make it easier for developers to locate statements of code that are related and that work together since the statements are together in a region of code.
- *Dependency verification*: We implement a methodology which verifies the dependencies for the lines of code that constitute a detected functional-based refactoring opportunity. This ensures that the refactorings being performed preserve the original behaviour of the source code.
- *Comment generation*: Comments can be valuable prefixes to statements and sections of code. Often times, due to time constraints, the developers may not comment their source code. We address this problem by automatically generating comments for the source code we refactor. We prefix all functional regions of code that we create, with automatically generated comments.

Having code in well organized, commented, and spaced regions improves the overall source structure. Our research is designed to be used during the development and maintenance phases of the software development lifecycle. Moreover, since our proposed technique improves the readability of source code by organizing source code into functional regions, it becomes easier for developers to locate statements of code being searched for since the statements will reside in the same region. This technique also assists in the bottom-up program comprehension process.

1.6 Thesis Overview

The remaining chapters of this thesis are organized as follows:

Chapter 2: Reviews related work. We review past research which relates to refactoring.

Chapter 3: Presents the overall framework of our refactoring process.

Chapter 4: Discusses the detection strategies. The detection strategies are the processes used to identify functional-based refactorings.

Chapter 5: Describes the exceptions to the detection strategies. These exceptions include patterns of source code which could be identified as functional-based refactorings, but should not be, due to special properties which exists within the source code.

Chapter 6: Discusses the actual refactoring process that our prototype implements.

Chapter 7: Presents our case studies and reviews the performance of our prototype.

Chapter 8: Concludes our work and discusses future work.

Chapter 2

Related Work

This chapter presents research which is related to functionality-based refactoring. We review many topics, including metrics-based refactoring, static analysis, parallelization and compiler optimizations.

2.1 Metrics Based Refactoring

Metrics can serve as indicators of software quality. For this reason, areas of research have tried to incorporate software metrics into the refactoring process. Metrics are frequently used as guides in detecting problems in source code, choosing between multiple refactoring opportunities and even evaluating post-refactoring software quality.

2.1.1 Refactoring Techniques using Metrics

Munro addresses the issue of trying to quantify when code contains code smells by using formulae which incorporate software metrics [48]. Code smell is the term used for a design problem in source code, indicative of needing to be refactored [18]. The two code smells he addresses are *temporary field* and *lazy class*. A temporary field is a variable

which is used to hold an intermediate result from a calculation and a lazy class is a class that does not have enough responsibility assigned to it [28].

For identifying a temporary field, he uses the metric IVMC, which measures the number of methods that reference each instance variable defined in a class. The reasoning for his detection then states that if the value for IVMC is less than, or equal to 1 for a given field, then that field is a temporary field. He states: "A zero value of IVMC means that an instance variable is not used within any method in a class. An IVMC result of one means that the instance variable is only referenced in one method in a class, and should be considered to be a temporary field" [48].

To detect the lazy class smell, he develops an algorithm that has three criteria to see if a class can be classified as a lazy class. First, he checks if the number of methods for a given class is equal to zero. If it is, then that class is a lazy class. Second, a check is done to see if the class' number of lines of code is less than the median lines of code for all classes in the application, and if the average value for the class' methods' complexity is less than two. If both conditions are satisfied for a given class, then it is a lazy class. This second rule checks not only the size of the class, but its functionality also. The justification is that lazy classes do not have a lot of functionality. Lastly, the third criteria verifies if the class' coupling is less than the median value for all classes in the application, and its depth of inheritance tree value is greater than one. This is because a lazy class would not have many connections with other classes, and also, lazy classes are often subclasses.

Munro then ran a case study using the above criteria on the source code for two different production quality software packages; one for hotel management and one for

graphing. Since the hotel software's size was 1,500 lines of code, he was able to manually verify if the code smells were being correctly detected. Each class was classified as either being, or not being a lazy class. Only one class was incorrectly classified. For the detection of temporary fields, all of the fields were classified correctly. However, the process of setting thresholds for the metrics formulae introduces subjectivity into the detection process.

In another study, researchers developed an Eclipse plug-in to detect various code smells [27]. They also implemented functionality to detect long methods. Their metrics based detection strategy consisted of classifying methods as being long if their *Cyclomatic Complexity* was greater than 10 and the number of lines of code in the method was greater than twenty. The study had the tool detect the code smells, and then human participants reviewed the source and located code smells. Only 44% of the cases detected by the tool and by the humans matched. 22% of the total cases detected were detected by the tool, but not the humans and 33% of the cases were detected by the humans, but not the tool. The tool and the humans agreed on almost half of the code smells detected.

Simon et al. also use metrics to identify code smells [64]. For their approach, they focus on structural refactorings which are aimed at moving methods and attributes to their appropriate classes, and extracting and inlining classes.

Moving methods and attributes is done in response to the *feature envy* code smell (e.g.: a class is more interested in another class' members, than it is in its own members). For identifying methods and attributes which perform feature envy, the authors present a distance-based cohesion metric to calculate the distance between two entities. The theory behind this is that methods which are using methods or attributes from other classes will

have a smaller distance between them. This helps in locating methods that would be candidates for relocation to a class that they are using more frequently than their home class. The tool they developed creates a visualization which aids the human in determining a cohesive class structure.

Instead of using metrics to detect refactoring opportunities, our work uses sets of heuristics to locate refactoring opportunities. We use the functionality of statements of code as heuristics.

2.1.2 Change Trend Analysis

In a study which examines fifty-two Java library classes, Counsell et al. track the changes made to those classes over a three year period [20]. They planned to track changes to the source code, and identify categories that the changes would fall into. This would aid in finding locations where refactorings could be applied, as it would locate sections of code that are frequently modified. They found that many of the changes performed to the source could be placed into a category, which corresponds with a refactoring. Further, it was discovered that locations in the code which are modified frequently are candidates for refactoring.

In another study, Korn et al. looked at tracking changes between two different versions of Java applets [41]. They analyzed the applets at different levels, including class, method and field. Their automated tool extracted all of the information related to changes between versions of the applets into a database. Their research work then provided methods to visualize and track data related to changes between versions.

In a study by Antoniol et al. changes to a software application were tracked over time. A set of metrics was used to analyze software at different releases. The data was then used to predicatively decide when to apply refactorings to the software's source code [4].

Our research differs in that we use the functional properties of statements of code to locate refactoring opportunities, as opposed to using heuristics which analyze frequency of change or relationships between changing components.

2.2 Static Analysis

Static analysis is defined as the process of evaluating a system or component based on its form, structure, content or documentation [36]. It can be used to find many different types of problems in source code, such as coding standards non-compliance, uncaught run-time exceptions, redundant code, division by zero and memory problems.

One study contrasted manual and automatic static analysis to detect faults in software [74]. A motivation for this challenge is that the longer a defect resides in software, the more expensive it becomes to fix, because as the software continues to grow, so does the number of software components that would be involved in the fix of the fault [8]. The authors found that the cost of automatic static analysis is relatively the same compared to manual static analysis. They also found that the defect removal rates were not significantly different between the two types of analyses.

In work done by Zou et al. static tracing is performed on single files of source code to recover business processes implemented in e-commerce systems [34][75][76]. To perform the static analysis, abstract syntax trees were used as the primary representation

of the source code. The analysis gained from the static analysis was then compared with the formal descriptions of the business processes. In subsequent work, they discuss how analyzing single files of source code was not sufficient, as many of the business processes being recovered quite often exist across many class files or modules [35]. To expand their initial work, they expand their recovery process to include different components in the e-commerce systems such as user interfaces, application logic and database logic. Analyzing multiple source files and multiple tiers of the e-commerce applications allowed the recovery process to recover more accurate and complete business process models.

Another study used static analysis to predict pre-release defect density in Windows Server 2003 [49]. The authors performed a static analysis of the code base. They ran two different tools to look for defects in the source code. They found that static analysis is very reliable in predicting pre-release defect density. They found a strong correlation between the predicted defect rate and the actual defect rate. This demonstrates the valuable role that static analysis has detecting defects.

Static analysis has even been used in areas such as intrusion detection [25][26][30][73]. The approaches typically analyze program control flow and incorporate dynamic data such as stack states, while using automata to model software execution. Static analysis has even been used to optimize queries [55], analyze real-time distributed systems [47], and has been combined with model checking [10].

Our work relates to static analysis in that we perform a static analysis, or tracing of source code. There are many ways to perform this tracing, from a straight textual analysis of source files, to using tree structures. We use abstract syntax trees to perform our tracing of source code.

2.3 Parallelization

Parallelization looks at converting sequential code into multi-threaded code, to make use of a multi processor environment. Most commonly, it is the bodies of loops which are analyzed to verify if they can be parallelized.

To perform the parallelization process, two questions must usually be answered: Firstly, is it safe to perform the parallelization? This question is answered by performing an analysis of the dependencies which exist in the source code and also an alias analysis. The second question which is usually posed is if it is worthwhile to perform the parallelization.

There are three basic techniques of parallelization: Instruction level parallelization, block level parallelization and function level parallelization. Recently, new approaches to parallelization have taken a hybrid approach by using more than one parallelization technique. One approach explored combining block level parallelization with function level parallelization [43]. Another approach looked at combining loop level parallelization with instruction level parallelization [13]. Both studies found that combining two parallelization techniques produced results that were significantly better than just one parallelization technique alone.

Compilers are often used to perform the parallelization process [13]. However, one approach made use of neural networks to perform the parallelization process [58]. The authors use neural networks to address the task of finding the optimum sequence of transformations.

Our work relates to parallelization in that we also recognize the importance of verifying dependencies in source code. We work at the instruction level in our refactoring process.

2.4 Compiler Optimization Techniques

Compilers provide many optimization techniques. One of these is known as “Code motion”. Code motion consists of moving invariants, which are variables which have unchanging values, outside of the body of the loop. In our work, we detect loop-invariants and move these statements outside of the loop.

There are many compiler optimization techniques which exist. Compiler optimizations have traditionally been applied using static schemes. However, many iterative schemes (which use many versions of the program being compiled) are now being used for compilation. This is to decide between multiple pathways which exist in applying optimizations to source code. It has been shown that this iterative scheme yields superior performance over the static schemes [2][7][42].

A drawback to iterative optimization is the cost of creating and analyzing the many versions of the program. A study looked at developing a new methodology which could speed up this iterative process [1]. The authors use a variant of predictive modeling which they use to target areas which are likely to give the greatest performance from having optimizations applied to them.

Recently, compiler optimization techniques have included optimizations such as reducing the overhead associated with application security [72]. The approach involves an algorithm which decides whether data they receive is trusted or non-trusted. For trusted

values, the security overhead can be ignored. This is accomplished by having a segment of memory store this trusted data. This trusted data can then be stored in this memory without any security overhead. Data which is non-trusted is stored in groupings. These groups then have security operations applied on them as a group as opposed to individually.

Our work relates to compiler optimizations in that we perform an actual technique used by compiler. As opposed to having the compiler perform this optimization on the executable being generated, we perform the optimization on the actual source code.

2.5 Statement Reordering

A study looked at reordering statements in DOACROSS loops to maximize the parallelization of loops, while minimizing synchronization requirements [15]. The authors develop an algorithm which schedules statements together which are strongly connected in a dependence graph. Previously, a two-phase process was used for ordering statements, which first, would partition the dependence graphs and then order the partitions, and then second, would order the statements within the partitions [3][21]. The authors' algorithm avoids the problem of having to determine the directions of loop carried dependencies. It is avoided by postponing these decisions until they are necessary. This new approach, which locates strongly dependant statement groups, was shown to perform more effectively by generating larger and faster parallelized operations. Related work, which optimizes the parallel execution of Do-Loops by reordering statements, can also be found in [16].

Statement reordering has also been applied to assist anti-virus scanners in locating viruses [44]. This study addresses the issue of how metamorphic viruses [9][68] can perform transformations on their code, which preserve the semantics of the code. A single virus could have many different variations of its source code, where all the variations execute in a manner identical to the original virus. The metamorphs can then escape virus scanners simply because their structure is different [17]. The authors present a method to impose an order on statements which reduces the number of variants that can be generated by reordering a virus' statements of code. In generating order on the statements they process, they also track the dependencies associated with the code. By applying their statement reordering technique, the authors are able to dramatically reduce the number of variants which need to be processed to identify a metamorphic virus. Currently, they are looking at if statements can be reordered in such a way to produce a single canonical version of a virus program. As in our work, statement reordering requires that control and data dependencies be managed to preserve the original behaviour of the source code [40].

2.6 Feature Identification

Feature identification aims to identify statements of code which relate to a specific feature [56]. For example, a developer might want to locate the behaviour associated with adding a record in a record management software system. In this case, a trace of the source code would be performed to trace all of the source code which is executed from the moment a user presses a button to add the record, to when the add operation is finished. This process would yield a trace log, which would indicate which statements

were executed during the add operation. This trace record identifies the add feature in this instance.

In feature identification, the source code can be processed statically or dynamically [71]. In a static analysis, a form of graph is usually constructed which shows how the statements of code, or various components relate to each other [14]. Features can then be located as sub trees within the graph. The sub trees show the execution order and hierarchy of statements which are executed in order to deliver the specified feature.

To perform feature location dynamically, a trace execution is performed at the actual run time of the application being inspected [60]. For example, a test case could be set up that only tests the add operation referred to above. The trace log would contain the statements which are executed when the test case is run. The feature has then been located using the dynamic information collected at run time.

Currently, there is a strong movement in the research community to use both static and dynamic data in feature identification and program comprehension [5][22][24][33][61].

In our feature identification, we only use static tracing of source code. We have developed a set of detection strategies that isolate specific features, based on functionality of the source code.

Chapter 3

Detection Strategies

The goal of this research is to restructure blocks of code into functionally-related regions of code, where each region delivers a single functional feature. To support this process, we have developed detection strategies which reorganize statements of code into functional regions. These regions represent the implementation of a single functional feature. To support this goal, we have developed the following detection strategies:

- *Declaration Region*: this strategy looks for statements which declare variables/objects. It aims to group as many of these statements as possible, at the beginning of a block of code.
- *Initialization Region*: this strategy creates regions of code where each statement in the region provides initialization functionality for objects instantiated from the same class.
- *Functional Feature Region*: this strategy collects all of the statements which invoke a method (except initialization methods) and groups these statements into subgroups based on the name of the method being invoked.

- *Leading Duplicate Code Fragment in Conditional Region*: this strategy locates duplicate fragments of code which appear in all branches of a conditional. These fragments lead each branch in the conditional structure. The duplicate fragments have a single copy situated immediately before the conditional structure.
- *Trailing Duplicate Code Fragment in Conditional Region*: this strategy is similar to the strategy above, except that it searches for duplicate fragments of code which appear at the end of each branch in a conditional structure.
- *Loop Constant*: this strategy searches for variables and objects that are declared in a loop structure, and whose value never changes through the iterations of the loop. The loop constants are moved to be immediately before the loop structure.

All of the detection strategies work at the level of a *Block* within the source code. A *Block* is defined as a unit of source code where the statements are grouped within braces or grouped by a control structure, and all of the statements exist at the same lexical level and within the same scope. This means that the statements which are involved in a single functional-based refactoring opportunity will always be from the same scope. Consider the code found in Figure 3-1 below:

The code that exists within the If Statement exists in a different scope than the code before the If Statement. The code in the If Statement is at a higher lexical level. Statements of code from within the If Statement would never be grouped with code from another scope, and would never leave its home scope. The only exceptions to this are the two detection strategies which work within conditional structures (*Leading/Trailing Duplicate Code Fragment in Conditional Region*) and the *Loop Constant* detection

strategy. These three strategies move statements from within a conditional branch or loop, to the parent scope, or in other words, down one lexical level.

```
private void process( ) {
   1 setLayout(boxLayout);
   2 JPanel jPanel;
   3 jPanel = new JPanel( );
   4 jPanel.setLayout(borderLayout);
   5 jPanel.setBackground(Color.white);
   6 jPanel.setAlignmentX((float) 0.0);
   7 if ( state == 1 ) {
   8       jPanel.setBackground(Color.blue);
   9       processLast();
   10 }
}
```

Figure 3-1: Scope example source code.

3.1 Declaration Region

A variable or object can be declared in many places within a block of code. The motivation for this strategy is to create a region of code at the beginning of a block of code, which declares objects and variables used within the region, into one location.

3.1.1 Example

To illustrate how this strategy works, consider the code below in Figure 3-2. Line 2 is a declaration statement. This statement has no dependencies, so it could be moved to the beginning of the block of code, and it would not affect the original behaviour of the code. Line 5 is also a declaration. This line of code *does* have a dependency. It uses the

variable *title* as a declaration parameter. Therefore, line 5 has a dependency on line 4. This means that line 5 could not be moved to the beginning of the block, as it would then be located before line 4, which it depends on. Moving line 5 to the beginning of the block would violate this dependency, and would not preserve the original behaviour of the source code.

```
1    initializeApplication();
2    String title = "Application";
3    startApplication(title);
4    title = "enter text here…";
5    JLabel myLabel = new JLabel(title);
```

Figure 3-2: Declaration dependency example.

Consider another example found in the code in Figure 3-3. Notice that a declaration for an object of type JPanel appears on line 2 and a declaration of an object of type *String* appears on line 10. Our strategy collects the statements on lines 2 and 10. After all of the declaration statements have been collected (in this case, the two statements), the strategy then inspects each individual statement to see if it has dependencies. In our case, there are no dependencies for lines 2 or 10, so the statements are moved to the beginning of the method and a comment is generated. Figure 3-4 shows how the code looks after this process.

This step organizes the code by locating the declarations in a block of code to its beginning. This improves the readability of the source code since many of the declarations are now located in a common region.

```
    private void initialize( ) {
1       setLayout(boxLayout);
2       JPanel jPanel;
3       jPanel = new JPanel( );
4       jPanel.setLayout(borderLayout);
5       jPanel.setBackground(Color.white);
6       JPanel.setAlignmentX((float) 0.0);
7       setBackground(Color.white);
8       add(header);
9       jPanel.add(text, BorderLayout.CENTER);
10      String title;
11      title = "My Window";
12      add(title);
13      add(jPanel);
    }
```

Figure 3-3: Example source code.

3.1.2 Algorithm

This strategy constructs a set of all of the declaration statements found in the block being processed. Each statement is then separately examined to find the dependencies associated with it. If a declaration statement can be moved to the beginning of the block without violating any of the data or control dependencies which exist in the code, it is recognized as a valid refactoring. Each time a declaration statement can be moved to the beginning of a block, it is placed immediately after the previous declaration statement that was moved to the beginning of the block. If it is the first statement to be moved, it is placed at the beginning of the block. The general algorithm to perform this process is found in Figure 3-5.

```
    private void initialize( ) {

        // DECLARATIONS
1       JPanel jPanel;
2       String title;

3       setLayout(boxLayout);
4       jPanel = new JPanel( );
5       jPanel.setLayout(borderLayout);
6       jPanel.setBackground(Color.white);
7       JPanel.setAlignmentX((float) 0.0);
8       jPanel.add(text, BorderLayout.CENTER);`
9       setBackground(Color.white);
10      add(header);
11      title = "My Window";
12      add(title);
13      add(jPanel);
    }
```

Figure 3-4: Source code after processing declarations.

```
declarations = all declaration statements from the block of code we are processing
for each statement S in declarations
        if S can be moved to the beginning of the block (following previous
        statements moved into the declaration region) without violating dependencies
                move S after the most recently moved declaration statement
```

Figure 3-5: Declaration region algorithm.

3.1.3 Declaration Region Comment Generation

The process of generating a comment to preclude a set of declaration statements being grouped together is as follows. The process adds a comment which contains "// DECLARATIONS:" immediately before the newly created region of declarations. A trailing space is added after the region to further improve the organization and appearance of the code.

3.2 Initialization Region

Many operations can be performed to initialize objects from their classes. The goal of this strategy is to group statements of code that perform initialization tasks together based on class. We create functional regions of code that initialize objects from the same class. This helps developers to more quickly locate functionality related to initializing objects of a given class. For instance, if developers were looking to see where all of the *JLabels* are initialized to change the text colour, this strategy would have reorganized all of the statements which initialize *JLabels* into a central and single location, thus making it easier for developers to find the code they are searching for.

Initialization statements include statements which use the *new* operator and statements which invoke *setter* methods (we define setter methods as methods that strictly assign to class fields). To detect setter methods, we perform a textual analysis, and look for methods whose names begin with the string "set". Moreover, we examine the semantics of other methods. If a method strictly performs assignment of class fields, it can be considered a *setter* method.

3.2.1 Example

To illustrate this approach, consider the following example. In Figure 3-4, we can see how lines 4 – 7 are statements which initialize an object from the class *JPanel*. These statements either use the *new* operator or invoke *setter* methods, and perform the initialization tasks. All four of the statements share a common purpose which is to initialize an object of the *JPanel* class. Since they deliver a single functional feature (initializing *JPanel* objects) they are grouped together into a region. This group of statements is moved from their original locations, into a new region of code and a comment is generated which describes the feature provided by this new region of code. The results of applying this strategy yield the following code found in Figure 3-6. Also notice, that we have grouped lines 3 and 9 from Figure 3-4, as they initialize an object from the class *InfoPanel*, and have generated a corresponding comment.

This second strategy has improved the organization and readability of the code. The code has now been organized into three functional regions, which each offer a single functional feature. The regions created in this step handle the initialization of an object from the *JLabel* class and an object from the *InfoPanel* class.

3.2.2 Algorithm

This strategy collects all of the initialization statements in a block of code. They are then sorted into groups, based on which class they are performing an initialization task on. Each of these groups (statements which operate on the same class type) is then considered a functional-based refactoring opportunity. Each refactoring then has its dependencies verified. If the refactoring is determined to be valid (i.e.: it does not violate

dependencies), it is performed. Figure 3-7 summarizes the algorithm for this detection strategy.

```
    private void initialize( ) {

        // DECLARATIONS
1       JPanel jPanel;
2       String title;

        // INITIALIZE INFOPANEL
3       setLayout(boxLayout);
4       setBackground(Color.white);

        // INITIALIZE JPANEL
5       jPanel = new JPanel( );
6       jPanel.setLayout(borderLayout);
7       jPanel.setBackground(Color.white);
8       JPanel.setAlignmentX((float) 0.0);

9       jPanel.add(text, BorderLayout.CENTER);
10      add(header);
11      title = "My Window";
12      add(title);
13      add(jPanel);
    }
```

Figure 3-6: Source code after processing initializations.

```
initializations = all initializations statements from the block of code we are processing
groupings = initializations divided into subgroups, based on class
for each group G in groupings
        if all of the statements in G can be moved into a continuous region without violating
        dependencies
                create the initialization region
```

Figure 3-7: Initialization region algorithm.

3.2.3 Initialization Region Comment Generation

The process of generating a comment for Initialization Regions involves using the AST for one of the statements in the group. We use the class name in generating a comment for an Initialization Region. Since all of the statements in an Initialization Region are initializing objects from the same class, we only need to retrieve the class type once. Initialization statements are of two types for our research:

1. Class instantiations: object = new Class(…);
2. Setter methods: object.setterMethodInvocation(…);

The AST for each type of the above statements is different. We will review the AST structure for each statement type. In Figure 3-8 below, the tree on the left represents a statement which instantiates an object by using the *new* keyword. The parent node contains the complete statement of code. Its immediate child then reflects the assignment operation of the statement of code. This assignment node has two children: one represents the actual object being assigned to in the initialization (left), and the other represents the actual initialization (right). This right node parents a node which indicates the class type. The AST on the right represents a statement of code which invokes a *setter* method. The parent node represents the statement of code. Its immediate child represents the actual method being invoked. This node has three children which provide the information related to the method being invoked. The left child represents the object which has the method invoked on it. The center child represents the method which is being invoked. The right child represents the parameter of the method invocation.

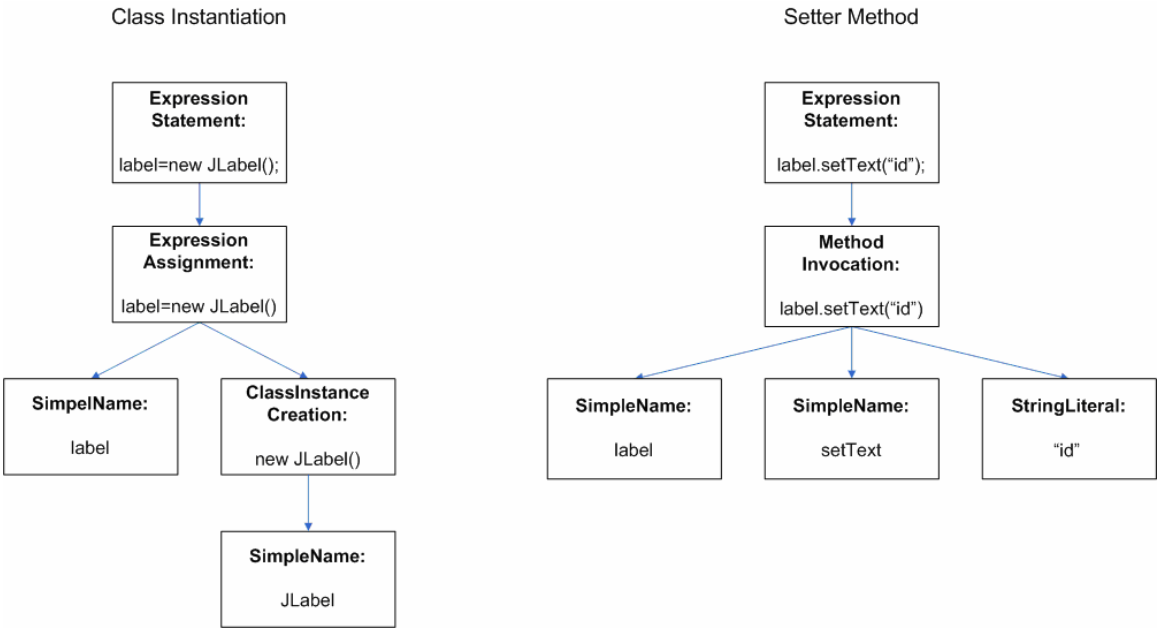

Figure 3-8: AST structure of initialization statements.

For the first case of generating a comment for an Initialization Region, we refer to the AST shown above in Figure 3-8 for the Class Instantiation. We want to find the class type of the objects being initialized in the statements which constitute the refactoring opportunity. In this case, we traverse the AST until we locate the corresponding node which contains the class type, which is the SimpleName node that contains JLabel.

For the second case where the statement is of type Setter Method, refer to the AST for the Setter Method, as shown in Figure 3-8 above. In this case, the actual class type is not stored in this AST. We search the source code for the statement which declares this object. When the statement of code which declares the object is found, we extract the class type.

After the class type is obtained for the refactoring opportunity, the string name of the class is prefixed by "// INITIALIZE". This concatenation of the class name with the strategy being performed yields the new comment for the functional region of code we create from the functional-based refactoring opportunity. For example, code that initializes *JLabel* objects has the comment "// INITIALIZE JLABELS" generated for it.

3.3 Functional Feature Region

Invoking a method provides a specific piece of functionality, or a functional feature. For this strategy, we group statements together which invoke methods with the same name.

3.3.1 Example

To illustrate this approach, consider the following example. In Figure 3-6, we can see that lines 9 – 10, and 12 – 13 all invoke the *add* method. All four of the statements share a common purpose which is to add objects to themselves. Since they deliver a single functional feature they are grouped together into a region. Each statement is then moved from its original location, into a new region of code and a comment is generated which describes the feature provided by this new region of code. The results of applying this strategy yields the code found in Figure 3-9 below.

The source code now consists of four distinct functional regions of code. The newly created fourth region of code delivers a single functional feature, which is to add objects to instantiations of their respective classes.

```
    private void initialize( ) {

        // DECLARATIONS:
1       JPanel jPanel;
2       String title;

        // INITIALIZE INFOPANEL
3       setLayout(boxLayout);
4       setBackground(Color.white);

        // INITIALIZE JPANEL
5       jPanel = new JPanel( );
6       jPanel.setLayout(borderLayout);
7       jPanel.setBackground(Color.white);
8       JPanel.setAlignmentX((float) 0.0);

9       title = “My Window”;

        // ADD
10      jPanel.add(text, BorderLayout.CENTER);
11      add(header);
12      add(title);
13      add(jPanel);
    }
```

Figure 3-9: Source code after functional features.

3.3.2 Algorithm

This strategy collects all of the method invocation statements found in a given block. This set is then further subdivided into groups, by the syntactical name of the method being invoked in the statement. This yields functional-based refactoring opportunities which consist of statements which all invoke the same named method. By grouping these statements together into a region, the region provides a single functional

feature which is invoking similar methods. Figure 3-10 summaries the algorithm used for this detection strategy.

```
methodInvocations = all statements which invoke methods (which are not setter methods) from the block of code we are processing
groupings = methodInvocations divided into groups, based on the textual name of method being invoked
for each group G in groupings
    if all of the statements in G can be moved into a continuous region without violating dependencies
        create the functional feature region
```

Figure 3-10: Functional feature region algorithm.

3.3.3 Functional Feature Region Comment Generation

The process of generating a comment for *Functional Feature Regions* uses the AST also, to obtain the name of the method being invoked. We examine the AST generated for one of the statements in the functional-based refactoring opportunity. We traverse the AST until we locate the node which yields the name of the method being invoked. Figure 3-11 below shows a typical AST for a statement of code which invokes a method. Its structure is identical to the AST shown above for invoking a setter method.

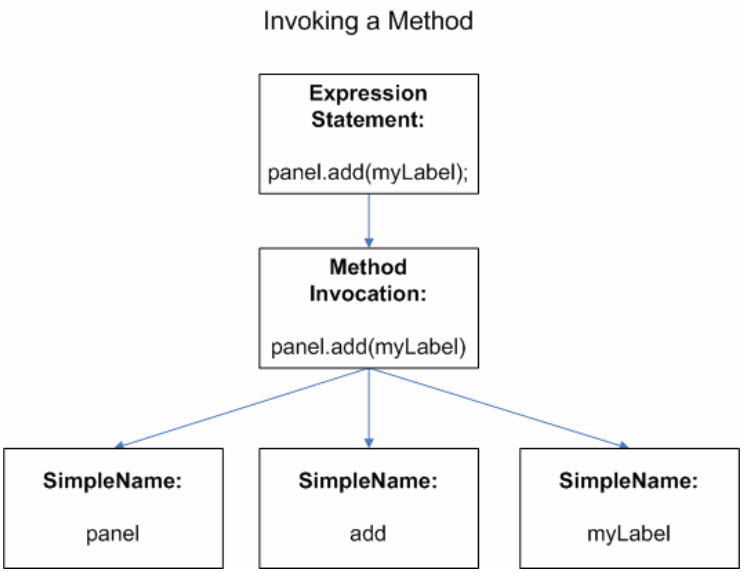

Figure 3-11: AST structure for a method invocation.

The comment generated follows a pattern of "//" + methodName. If we have just created a functional region of code for a set of statements which all invoke the add method, the comment that is generated is "// ADD".

3.3.4 Refactoring Initialization and Functional Feature Regions

The Initialization and Functional Feature Region detection strategies have their refactorings performed in an identical manner. For this reason, we will review this method for both detection strategies in this section.

The dependency verification process explained in section 5.2.1 Bounded Dependency Condition, guides the placement of the new functional region created from the refactoring. The pre and post dependencies given by the bounded dependency condition, tell us the possible locations where the region could be located without

violating dependencies. The functional region being created from the functional-based refactoring must appear after the pre-dependency and before the post-dependency.

After the declaration region is finished being created, a pointer points to the last line inserted into the declaration region. All of the initialization regions are then processed. The goal is to create all of the initialization regions in a consecutive series, which follows the declaration region. After each initialization region is created, the pointer is adjusted to point to the most recently created initialization region. After the initialization regions have all been created, a pointer points to the location where the last initialization region was created. The refactoring process then has the goal to create all of the functional feature regions sequentially, following the last initialization regions. The goal of this refactoring process, is to reorganize, as much as possible, the source into three regions of code: a region of code which declares, followed by a region of code which initializes, which is then followed by a region of code which performs operations. However, due to dependencies which exist in the source code, this goal is not always fully possible.

There are three cases which guide the placement of the Initialization and Functional Feature Regions being created:

Case 1: *The pre-dependency is below or at, AND the post- dependency is above or at, the location pointed to by the pointer:* The goal in placing the functional regions is to place them in succession, one after another. Each time a functional region is placed, the pointer is adjusted to point to the end of the region just placed. Then, the next time a functional region is inserted, it is placed at the location pointed to by the pointer, and this process repeats. However, because of dependencies, the regions may not be

able to be placed in succession. This case deals with a region that can be placed in succession, after the most recently created functional region.

Case 2: *The post-dependency is before the location pointed to by the pointer:* this case covers the situation where the post-dependency for a refactoring opportunity is below the pointer. This means that there is a dependency which would prevent the region being created from being located at or after the pointer. In other words, it cannot be created in succession with the most recently created region. In this case, we would place the region immediately after the pre-dependency, since it cannot be placed in succession with other regions being created.

Case 3: *The pre-dependency is after the location pointed to by the pointer:* In this case, the functional region being created cannot be located in succession with the other successive regions because there is some dependency which necessitates that it be placed at some location later in the code, than where the pointer points to. For this case, the region is placed immediately after its pre-dependency.

3.4 Leading Duplicate Code Fragment in Conditional Region

A Duplicate Code Fragment is a statement, or statements, of code, which appear(s) in all branches of a conditional structure. An example of this would be lines 2, 8, 14 from Figure 3-12 below. All three of the lines are duplicates, and the duplicate line appears in every branch in the conditional structure. Duplicate conditional fragments have many problems associated with them. Duplicate code is difficult to maintain as when one of the duplicates changes, finding and changing the other copies can be a cumbersome task and can quite often introduce bugs. The duplicate fragments also add unnecessary complexity

and size to the code. By eliminating these duplicate fragments, we eliminate duplicate code. We also locate the previously duplicated functionality to a single location, which is immediately before the parent conditional structure. We also reduce the effective size of each branch in the conditional, which in turn reduces the amount of logic and complexity in each branch.

3.4.1 Example

For this strategy, we focus on fragments that lead each branch in a conditional structure. Consider the code found below in Figure 3-12. There are three branches in the conditional structure. The first statement from all three of the branches match (lines 2, 8 and 14). This means that the leading section of each branch is a match up to the first lines in each branch. Now notice how the second statements from each branch match each other (lines 3, 9 and 15). Now, the leading section of each branch matches up to the first and second lines of each branch. However, the third lines in each branch (lines 4, 10 and 16) do not all match. The leading duplicate code fragment would not include the third lines from each branch. Thus, the leading duplicate code fragment is the first and second lines from each branch, as they all match, and are all leading continuous fragments.

Performing this refactoring process on the code found below in Figure 3-12 yields the following code found in Figure 3-13. It should be noted that this strategy also recursively searches through nested conditionals and properly manages the search process and scoping, so that any duplicate fragments are properly matched to their parent conditional structure.

```
1       if( firstName.equals("First") {
2               processFirstName( firstName );
3               processLastName( lastName );
4               age = 1;
5               processBaby();
6       }

7       else if( firstName.equals("Second") {
8               processFirstName( firstName );
9               processLastName( lastName );
10              age = 24;
11              salary = 70 000;
12      }

13 else {
14              processFirstName( firstName );
15              processLastName( lastName );
16              age = -1
17      }
```

Figure 3-12: Leading duplicate code fragments.

3.4.2 Algorithm

For each conditional structure, this strategy examines to see if it has an *else* branch. If there is no *else* branch present in the conditional structure, it is not processed any further. This is because the strategy can't guarantee that the behaviour would be preserved in the case when none of the conditions for the branches are true. For example, in the code found in Figure 3-14 below, the statements at lines 2 and 5 are duplicates. However, we cannot extract the statements from both branches and move a duplicate immediately before the If Statement, because there is no *else* branch. Because there is no *else* branch, there is no guarantee that the duplicate statement executes. If *x* is not equal to

1 or *2*, then the duplicate statement never executes. However, if the last branch is an *else* statement, this guarantees that the duplicate statement executes, regardless of which condition is true.

```
    // DUPLICATE LEADING CONDITIONAL FRAGMENT
1   processFirstName( firstName );
2   processLastName( lastName );

3   if( firstName.equals("First") {
4           age = 1;
5           processBaby();
6   }

7   else if( firstName.equals("Second") {
8           age = 24;
9           salary = 70 000;
10  }

11  else {
12          age = -1
13  }
```

Figure 3-13: Refactored leading duplicate code fragment in conditional region.

The algorithm looks at each branch in the conditional structure. If the first statement in each branch matches, then these statements are marked as being leading duplicate code fragments. The second lines from each branch are then compared. If they all match, then the leading duplicate fragment then expands to include these statements also. This process repeats until the i^{th} statement from each branch does not match. If there is a duplicate leading fragment, then the statements that comprise this fragment are removed from each

of the branches in the conditional. A single copy of these statements is then placed immediately before the If Structure and an appropriate comment is generated for the newly created functional region. The following graph in Figure 3-15 summarizes the process of detecting sequential duplicate fragments which are present in all branches of an If Statement:

```
1    if( x == 1) {
2            processTransaction();
3    }

4    else if( x == 2) {
5            processTransaction();
6    }
```

Figure 3-14: Necessity of else branch.

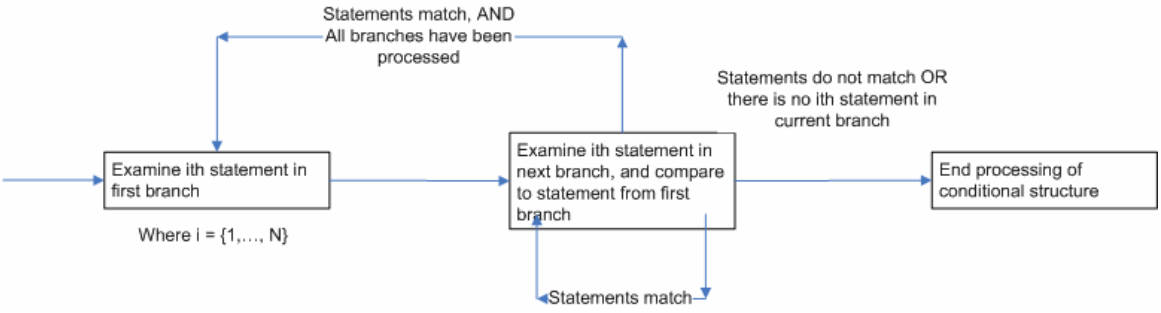

Figure 3-15: Leading duplicate code fragment detection process.

3.5 Trailing Duplicate Code Fragment in Conditional Region

The trailing duplicate code fragment in conditional region strategy is similar to the leading duplicate code fragment in conditional region strategy found above in section 3.4. The only difference is that instead of looking for duplicate consecutive statements found in all branches of the conditional structure from the beginning of each branch, the process starts at the end of the branch and scans to the beginning of the branch. Figure 3-16 below shows an example of trailing duplicate fragments.

```
1    if( x == 1) {
2            verifyTime();
3            processTransaction();
4    }

5    else ( x == 2) {
6            verifyDate();
7            processTransaction();
8    }
```

Figure 3-16: Trailing duplicated fragment in conditional.

Lines 3 and 7 comprise the trailing duplicate fragment in this case. For this example, these two lines would be moved to appear immediately following the conditional structure, as shown below in Figure 3-17.

```
1       if( x == 1) {
2               verifyTime();
3       }
4       else ( x == 2) {
5               verifyDate();
6       }
7               processTransaction();
```

Figure 3-17: Refactored trailing duplicated fragment in conditional.

3.6 Loop Constant Region

Often times, variables are declared within loops. This can be a problem if the variable is set to a constant value that never changes during any iteration of the loop. This unnecessarily recreates the same variable each iteration through the loop, and slows down program execution. By removing these loop constant variables, we also effectively reduce the size of the body of the loop. We also remove unnecessary complexity and improve performance.

3.6.1 Example

An example of a loop constant would be the variable *message* being declared on line 2 in Figure 3-18 below. The value of this variable never changes during any iteration of the loop, and it is needlessly recreated each iteration through the loop.

To refactor this example, line 2 would be moved to immediately before the beginning of the *for loop*, as shown below in Figure 3-19.

```
1       for (int index = 0; index < 5; index++)
        {
2               String message = “approved”;
3               bool approved = getCustomerStatus();
4               if (approved)
                {
5                       sendMessage(message);
                }
        }
```

Figure 3-18: Loop constant.

```
        String message = “approved”;

        for (int index = 0; index < 5; index++)
        {
                bool approved = getCustomerStatus();
                if (approved)
                {
                        sendMessage(message);
                }
        }
```

Figure 3-19: Refactored loop constant.

```
declarations = all statements which declare variables within a loop
for each statement S in declarations
        if S declares a constant value that never changes through the loop’s execution
                move the statement S immediately before the loop (following any other loop
                constant statements which have already been moved)
```

Figure 3-20: Loop constant algorithm.

3.6.2 Algorithm

This detection process looks for all variables that are declared within a loop structure. Each of these declarations is examined to determine if the variable's value is set only once, and if it is set to a constant. If this is the case, the variable is labelled as a Loop Constant, and is moved to immediately before the loop. The algorithm that summarizes this detection strategy is above, in Figure 3-20.

3.7 Execution Order

When performing the refactoring process, the detection strategies are executed in a specified order. The order in which the detection strategies are run is as follows:

1. Loop Constant Region
2. Leading Duplicate Code Fragment in Conditional Region
3. Trailing Duplicate Code Fragment in Conditional Region
4. Declaration Region
5. Initialization Region
6. Same Method Invocation Region

The justification for this ordering is to allow for statements that might be pulled out of conditional structures or loops, to be included in refactorings produced by the last three strategies.

Fully detailed algorithms for each of the detection strategies can be found in Appendix B.

Chapter 4

Exceptions to Detection Strategies

In source code, there exist statements of code that cannot be moved for different reasons. Such statements perform various tasks, including:

- *Synchronization*: code which affects timing, threading and the order in which tasks are processed.
- *Utility*: code for networking, database interaction, etc… This code may need to run tasks in a specific order.
- *Instrumentation*: tracking states in code by logging and writing to a console. When writing to application log files, the statements being logged should preserve their order and location, and not ever be grouped together.
- *Super method invocation*: invoking the class' parent constructor. The super method must usually be the first line invoked in a method.

To address the issue of excluding the above categories of statements of code from our refactoring process, we have developed a series of strategies for detecting these exceptions.

4.1 Synchronization

Threads provide a way for a running program to split itself into two or more simultaneously running tasks. Threads can allow a programmer to more effectively take advantage of available resource and accomplish multiple tasks at the same time. With threading though, care must be taken in spawning, managing and destroying threads. Failure to manage threads correctly could result in crippling a software system.

Consider the following example code in Figure 4-1 below:

```
sleep(500);
if(!stop)
   button.setIcon(icon2);
else
   return;
sleep(500);
if(!stop)
   button.setIcon(icon1);
else
   return;
```

Figure 4-1: Example code to show an exception.

Our *Functional Feature Region* detection strategy traces the code, and makes a grouping of the two lines marked in bold, *sleep(500)*. This group is now recognized as a functional-based refactoring opportunity, and could have the statements grouped together

into their own functional region of code. The dependency verification process inspects both statements and finds that there are no dependencies, and then indicates that they could be placed at any location within the segment of code.

However, upon manual inspection, it can be understood that grouping these two statements of code into a functional region modifies the original behaviour of the code. Instead of setting the icon for the button and pausing, then setting the icon for the button to another icon, there is now just a combined pause (the sleep statement is run twice, consecutively). Even though this refactoring opportunity is marked as being valid, it is not.

To avoid introducing errors to synchronization logic in the code, we do not allow statements which use *Thread* objects to be included in any refactorings.

4.2 Utility

Utility code, such as database access or networking code, serves a general purpose which is not specific to the functional requirements of the software. Oftentimes, utility code must perform tasks in a specific sequence. In the case of file I/O operations, a file needs to be opened before performing read or write operations on the file. Further, a file must also be closed after all processing has been completed.

In our research, we have implemented functionality to detect categories of utility code including:

- File I/O
- Networking

- Database

Utility code is often executed in a specified sequential order. In addition, many operations which use various functionalities from these categories are often surrounded with an individualized exception handling mechanism. Each of these exception handling mechanisms exist in their own block and scope, and cannot be moved. An example would be the code below in Figure 4-2 which writes out to a file. The actual statements that perform the write to file, exist in their own block, delimited by the *try block*, and thus could not move outside of their current scope.

```
FileWriter out = new FileWriter(portFile);

try {
   out.write("b\n");
   out.write(String.valueOf(port));
   out.write("\n");
   out.write(String.valueOf(authKey));
   out.write("\n");
}
finally {
   out.close();
}
```

Figure 4-2: Utility code which must preserve scope.

Quite often, code which performs a specific feature will also be located within a separate and distinct method. For example, code which writes financial transaction records to a database might operate as follows:

- The database connection may be instantiated in a constructor.

- The logic to build a specific query to search for an individual record might exist in a method that only performs this task.
- The logic to run the query and hold the result could exist in its own method.
- The logic to write a new record to the database or modify an existing record could exist in its own separate method also.

An example of this would be the code below in Figure 4-3 . The code which performs the write to the file is located within its own method. Having a method perform a single task is a common practice in the object-oriented paradigm.

```
public void writeToFile(FileWriter out, String message) {
    try {
        out.write("Message received: " + message);
    }
    finally {
        out.close();
    }
}
```

Figure 4-3: Utility code which must preserve scope.

Statements of code which are utility code, are not included in any of the refactorings.

4.3 Instrumentation

Instrumentation is the task of inserting probes into source code. The probes can be used for many different tasks, such as performance analysis, debugging, and logging.

Logging is a type of instrumentation and can serve many purposes. Logging code can be inserted as trace statements, which write important information about a program's execution to log files. For example, a log entry could be generated after the application successfully initializes or completes a significant task. These log messages help developers to track the behaviour of the application. Logging is commonly used by developers as a valuable debugging aid. An example of logging code is found below in Figure 4-4 . In this example, a log entry is written each time a user successfully logs into the system.

```
private bool authenticateUser(String username, String password) {
    if ( attemptLogin(username, password) {
        Log.write("User: " + username + " has successfully logged in.");
        return true;
    }
    else {
        return false;
    }
}
```

Figure 4-4: Example logging code.

When using logging functionality, the actual location of the logging statement is almost always critical. These logging statements must be located immediately before or after the region they are tracking information for. To move them or group them together would defeat the purpose of having them lead or trail their corresponding regions of code which they are instrumenting. Special consideration must be taken when moving a

statement which has instrumentation code attached to it. Care must be taken to keep the instrumentation statement together with the statement it is instrumenting.

Statements used to perform some form of printing can also be considered instrumentation code. Determining the meaning of statements which invoke some sort of printing method can be difficult without performing a full semantic analysis of the code. Often times, statements of code can invoke printing methods to do various debugging or logging to log files or a system console.

All statements that invoke a method with the string '*log*' appearing in method name, are disregarded and are not included in the refactoring process. Any statements that invoke a method with the string 'print' appearing in the method name, are disregarded also and are not included in the refactoring process as valid candidates to be included in functional regions of code being created.

4.4 Super Method Invocation

In Java, the super keyword invokes a method of the current object's parent. This is a feature which allows programmers to take advantage of code which exists in a class' parent, but to allow for extra functionality to be added to the child class. However, there is a restriction that this method invocation must usually be the first line in the method. Therefore, during the refactoring process when statements are being reordered, caution is taken to ensure that if a super method invocation is present that it remains in its original location. An example of code using the *super* method can be found below in Figure 4-5.

```
public QuickNotepad(View view, String position) {
    super(new BorderLayout());
    this.view = view;
    this.floating  = position.equals(DockableWindowManager.FLOATING);
}
```

Figure 4-5: Example code invoking the super method.

Chapter 5

A Framework for Functionality Based Refactoring

We focus on analyzing the behaviour of a statement of source code, in order to determine possible refactorings. We group statements which deliver similar functionality, into the same location. To this end, we use various properties of the source code to guide the refactoring process in detecting the functionality of a statement. We then verify dependencies and perform the refactorings. We have developed a framework to automatically perform this process.

5.1 Overview of the Refactoring Framework

As shown in Figure 5-1, our framework takes source code as input. The source code is then parsed, and used to generate an *abstract syntax tree* (AST) which represents the original source code. A set of detection strategies then looks for functional features by analyzing the AST. This produces a set of functional-based refactoring opportunities within the source code. Each functional-based refactoring that is detected then has its dependencies analyzed. If the functional-based refactoring can be performed without

violating any of the dependencies in the source code, it is marked as a valid refactoring. After our prototype performs the valid refactorings, it automatically generates comments for the regions of code that are created by the refactorings.

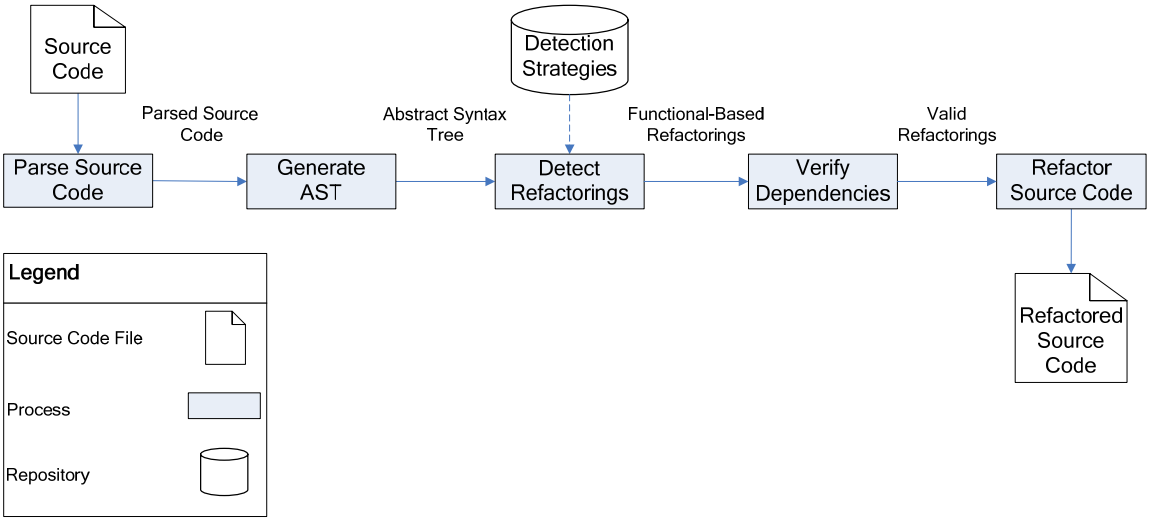

Figure 5-1: Overview of the refactoring framework.

5.2 Dependencies

It is important to correctly identify existing dependencies in source code before performing the refactoring process; otherwise the original behaviour of the software will almost certainly not be preserved. Our research aims to identify and preserve different types of dependencies, including:

- *Data dependency:* a data dependency occurs when an instruction depends on a result from a previous instruction. In the example found in Figure 5-2 below, the second line

depends on the first line, because it prints out the value of the variable *x*, and *x* is defined in the first line.

```
int x = 7;
System.out.println("The value of x is " + x);
```

Figure 5-2: Example data dependency.

- *Control dependency:* a control dependency exists when an instruction, or line of code, depends on the control structure it is located within. Consider the example found in Figure 5-3 below. The variable *index* depends on the for-loop it is located within. Therefore, there exists a control dependency for the variable *index*, as it depends on the loop that contains it.

```
for ( int index = 0; index < 10; index++ )
{
   System.out.println("The value of x has changed to: " + index);
}
```

Figure 5-3: Example control dependency.

We construct dependency graphs to represent the dependencies for each statement in a functional-based refactoring.

5.2.1 Bounded Dependency Condition

Each statement in a refactoring opportunity has a *dependency condition* (see Definition 5-1) associated with it. The entire refactoring opportunity then consists of a set

of *dependency conditions*, which has a computation performed on it, which then yields the *bounded dependency condition*.

Definition 5-1: *We define a Dependency Condition (DC) for some statement of code S_i to consist of a pre-dependency, statement-node(s) (which represents the statement(s) from the refactoring opportunity), and a post-dependency, such that the following two conditions hold:*

1. *The Pre-dependency[1] for a statement S_i, is the statement closest to statement S_i, (which sequentially appears before S_i) where any of the variables used in S_i experience a state-change[2] or are used. If no such pre-dependency exists, the symbol $-\infty$ is used to denote that it does not exist.*
2. *The Post-dependency for a statement S_i, is the statement closest to statement S_i, (which sequentially appears after S_i) where any of the variables used in S_i experience a state-change or are used. If no such post-dependency exists, the symbol $+\infty$ is used to denote that it does not exist.*

[1] Although a statement of code may have many dependencies associated with it, we only consider the dependency which exists closest to our statement appearing before/after our statement, as the pre-dependency/post-dependency.

[2] A state-change for a variable/object, is defined as any values or members of the object changing through assignment or initialization, or any method being invoked on the object.

Therefore, every statement S_i in a refactoring opportunity has a three-node graph associated with it which indicates which statement must appear before S_i (Pre-dependency) and which statement must appear after S_i (Post-dependency), such that the dependencies in statement S_i are respected:

The dependency cndition (**DC**) for some statement (S_i): $DC(S_i)$ is represented as a three-node graph: $(S_{i\text{-}pre}, Si, S_{i\text{-}post})$, where $S_{i\text{-}pre}$ is the pre-dependency and $S_{i\text{-}post}$ is the post-dependency and ***Si*** is the actual statement to which the dependencies belong.

```
1    JLabel nameLabel = new JLabel();
2    nameLabel.setText("Mr. Smith");
3    JLabel ageLabel = new JLabel();
4    ageLabel.setText("50");
```

Figure 5-4: Dependency verification example code.

To illustrate a dependency condition, consider the code found above in Figure 5-4. Assume that we have a refactoring opportunity that consists of the statements on lines 2 and 4 $\{S_2, S_4\}$, where we want to group these two statements together. To create a dependency condition graph for S_2, we notice that the set of variables which is used in this statement is {*nameLabel*}. The pre-dependency for this graph would then be the statement at line 1, since one of the variables from S_2 (*nameLabel*) is used there. There is no post-dependency for this graph since none of the variables from S_2 are used later in the code. To represent the case where there is no post-dependency, we use the symbol $+\infty$. For the case where no pre-dependency exists, we use the symbol $-\infty$. Following a similar

process used to create the graph for S_4, we generate the following graphs found in Figure 5-5.

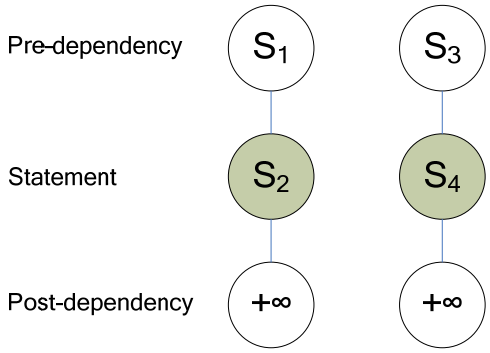

Figure 5-5: Graphs for statements at line 2 and line 4.

Graphs for Dependency Conditions can be chained together. Recall that a graph for a dependency condition has a pre-dependency and post-dependency associated with it. A dependency condition graph for one statement of code can be merged with a dependency condition graph for another statement of code if the following two conditions hold:

1. The statement node represented in the dependency condition graph being merged matches the post-dependency of the graph being merged into.
2. The pre-dependency node in the graph we are merging, matches one of the statement nodes in the graph we are merging it into.

The merging process merges the graphs by adding the post-dependency from the graph we are merging onto the tail of the graph we are merging it into. This post-dependency then becomes the post-dependency for the merged graph.

For example, consider if we have the following two graphs from a refactoring opportunity we wish to merge (Figure 5-6):

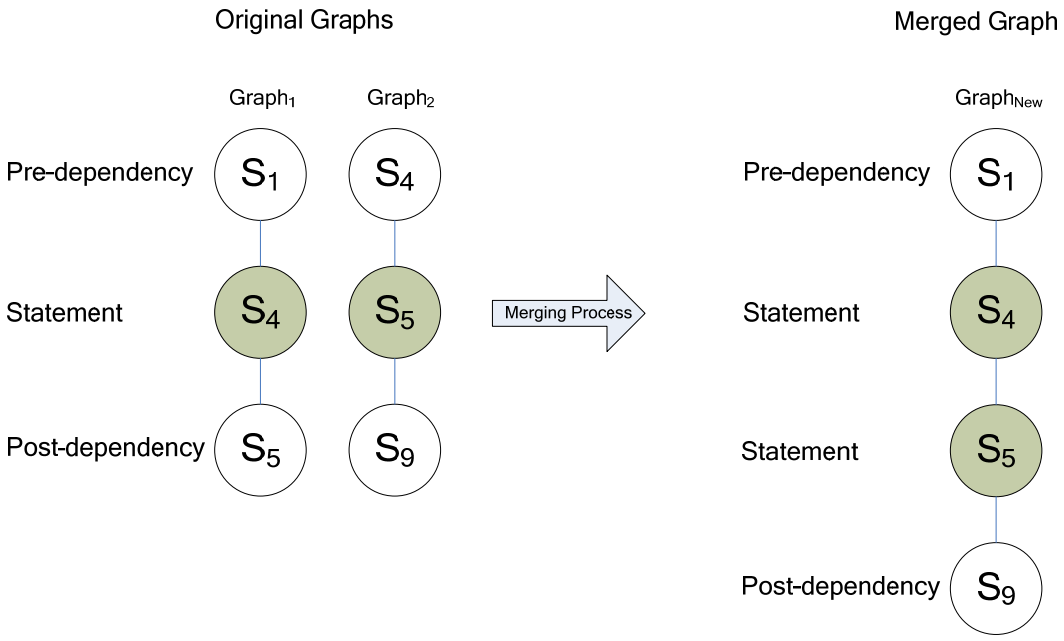

Figure 5-6: Chaining two dependency condition graphs together.

In this case, we wish to merge $Graph_2$ into $Graph_1$. We would first confirm that the statement node in $Graph_2$ ($\mathbf{S_5}$) matches the post-dependency from $Graph_1$ ($\mathbf{S_5}$). Secondly, we confirm that the pre-dependency from $Graph_2$ ($\mathbf{S_4}$) matches one of the statement nodes from $Graph_1$ ($\mathbf{S_4}$). Both conditions hold, and the two graphs are merged, yielding the merged graph.

The merged graph now has S_1 as its pre-dependency and S_9 as its post-dependency. The set of statement nodes from the refactoring opportunity which are represented by the graph is $\{\mathbf{S_4}, \mathbf{S_5}\}$.

Note that as dependency condition graphs are merged and the graph grows, the first node in the graph will be the sole pre-dependency for the graph, and the last node will be

the sole post-dependency. The interior nodes are then designated as statement nodes, which represent actual statements from the refactoring opportunity.

Definition 5-2: *A refactoring opportunity's Bounded Dependency Condition (BDC) is a pair, consisting of:*

1. *Pre-dependency: Each statement in a refactoring opportunity has a dependency condition associated with it. We construct a set of all of the pre-dependencies from each dependency condition associated with each statement in a refactoring opportunity. From this set, we then select the pre-dependency that appears latest in the code. This value is the Bounded Dependency Condition's pre-dependency.*
2. *Post-dependency: We construct a set of all of the post-dependencies from each dependency condition associated with each statement in a refactoring opportunity. From this set, we then select the post-dependency that appears latest in the code. This value is the Bounded Dependency Condition's post-dependency*

These two selected values then represent the Bounded Dependency Condition for the entire functional-based refactoring opportunity. If the properties of this pair are such that the pre-dependency represents a line of code that is before the line of code represented by the post-dependency, then the refactoring opportunity is valid. In the case that the refactoring opportunity is valid, the pair also represents a valid location where the refactoring opportunity could be located in the code. It tells us that the refactoring opportunity can be located after the line of code represented by the pre-dependency and before or at the line represented by the post-dependency.

Consider the dependency condition graphs found above in Figure 5-5. The code associated with these graphs found above in Figure 5-4.

Assume that our refactoring opportunity is an Initialization Region, consisting of the statements at lines 2 and 4 $\{S_2, S_4\}$. At this point, we have created the dependency condition graphs for each statement in the refactoring opportunity (Figure 5-5). To compute the *bounded dependency condition*, we perform the following steps:

1. We construct a set of all of the pre-dependencies from all of the graphs which represent our refactoring opportunity. This set is $\{S_1, S_3\}$. We then select the statement which appears latest in the code, from this set, which is S_3.
2. We construct a set of all of the post-dependencies from all of the graphs which represent our refactoring opportunity. This set is $\{+\infty, +\infty\}$. We then select the statement which appears earliest in the code, from the set, which is $+\infty$.
3. The pair generated as the bounded dependency condition for our refactoring opportunity is $(S_3, +\infty)$. This tells us two important facts:
    a. The refactoring opportunity is valid. The line represented by the pre-dependency is before the line represented by the post-dependency. This means that there is a valid space, range, or location in the code where the statements of code from the refactoring opportunity can be located together in a continuous group, without violating any of the dependencies.
    b. This also tells us the bounds, or locations within the code where the statements could be located. The pair representing the bounded dependency condition tells us that the refactoring opportunity can be

located after the statement at line 3, and before or at the statement at line +∞, which represents the end of the code in the current block. A re-examination of the code from Figure 5-4 above confirms this. Line 3 assigns/declares the variable *ageLabel* which is used in the statement at line 4. Of all of the pre-dependency dependencies in the code, the dependency at line 3 (for the statement at line 4) is the one that appears the latest. It can be seen that in order to place all of the statements from the refactoring opportunity together, they must be located after the dependency at line 3. For dealing with the post-dependency dependencies, the dependency which appears earliest in the code is the dependency found at +∞. This indicates that all of the statements in the refactoring opportunity must be located at or before this dependency, which in this case, is the end of the current block of code.

In the case that the line of code represented by the pre-dependency is at or after the line represented by the post-dependency, the refactoring opportunity is not valid, and cannot be performed.

The algorithm used for this process is shown below in Figure 5-7.

Additional examples, which are more complex and lengthy, can be found at:

- A.1 Bounded Dependency Condition Example – Valid Refactoring Opportunity
- A.2 Bounded Dependency Condition Example – Invalid Refactoring Opportunity.

```
for each statement S, in a refactoring opportunity:
        Construct a three-node graph
        Trace backward from the statement of code to find its closest pre-dependency and
        set this value in the graph
        Trace forward from the statement of code to find its closest post-dependency and
        set this value in the graph
        Add this graph to GraphSet
end for


for index1 = 0 to size of GraphSet
        for index2 = index1+ 1 to size of GraphSet
                if GraphSet[index2] can be merged into GraphSet[index1]
                        merge GraphSet[index2] into GraphSet[index1]
                end if
        end for
end for


PreDependencySet = all of the pre-dependencies from each Graph in GraphSet
PostDependencySet = all of the post-dependencies from each Graph in GraphSet


OverallPredependency = the latest sequential statement in PreDependencySet
OverallPostdependency = the earliest sequential statement in PostDependencySet


if OverallPredependency < OverallPostDependency
        return valid
else
        return invalid
end if
```

Figure 5-7: Dependency verification algorithm.

Chapter 6

Case Studies

This chapter presents the case studies to evaluate the performance of our research techniques being applied to open-source software applications. We aim to validate our approach to refactoring by tracking data related to the accuracy of our functional-based refactoring detections and dependency verification techniques, measured as precision and recall.

6.1 Implementation

We implement our refactoring process as an Eclipse Plug-in. Our prototype tool, **refactorQ**, is written in the Java programming language and also uses the following Eclipse libraries:

- *org.eclipse.jdt.core*: this library provides functionality to work with source code in an *Abstract Syntax Tree* (AST) format. Specifically, we use this library to generate ASTs for Java files, and use various library functions to process the ASTs.
- *org.eclipse.text*: we use this library to manipulate the actual source code contained in the AST and write our modifications to the original Java file.

Our implementation is divided into four main packages. The following list describes the major components of our prototype.

- *detectionStrategies*: this package contains the classes which implement the detection strategies. Each detection strategy is implemented as a class, and a master class acts as a controller, which runs each of the detection strategies.
- *refactoring*: this package contains our dependency verification module and our refactoring module.
- *utilities*: this package contains various data structures and user interfaces.
- *visitors*: this package contains the visitor objects which are used to traverse the AST of a Java file.

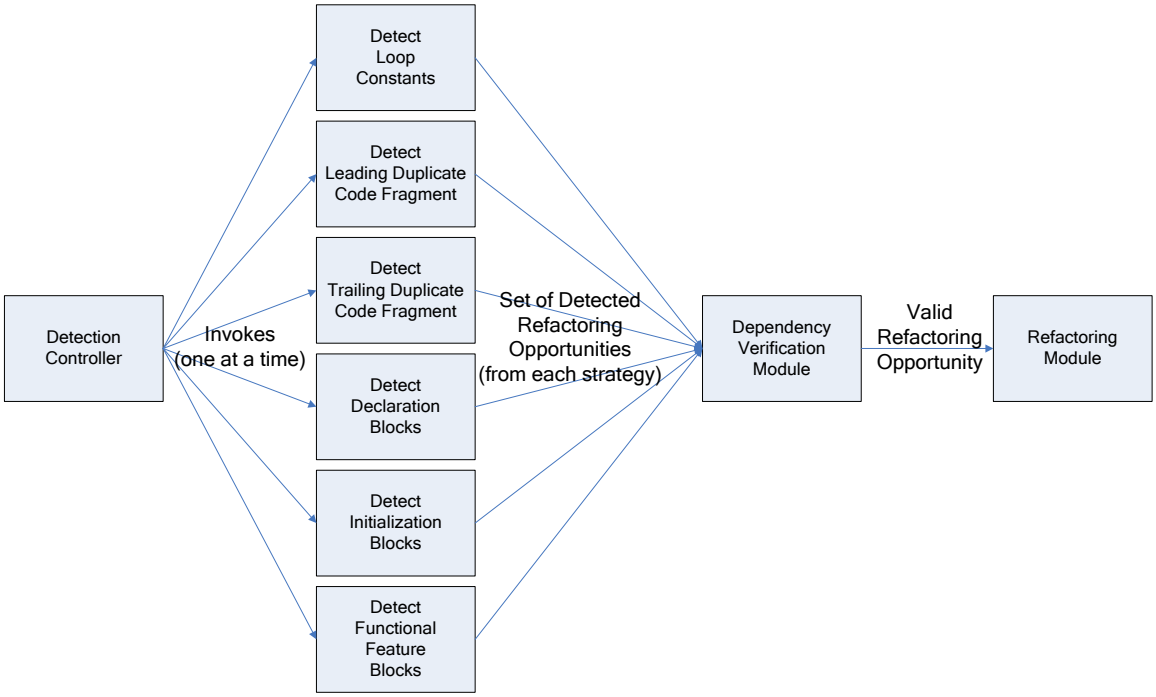

Figure 6-1: Prototype architecture.

Table 1, below, summarizes the properties of our prototype components:

Package	**Lines of Code**	**Num. of Classes**	**Num. of Methods**
detectionStrategies	1476	5	49
refactoring	712	2	35
utilities	417	6	57
visitors	723	21	61
Total	**3328**	**34**	**202**

Table 1: Characteristics of our prototype.

There is a class that acts as a controller and calls each of the detection strategies. Each detection strategy exists as its own class. The dependency verification module and refactoring module each exist as their own class. The interaction sequence among each of these classes is as follows:

1. The controller runs each of the detection strategies, one at a time.
2. Each detection strategy scans the java source file, constructing a set of the refactoring opportunities it detects.
3. After all of the functional-based refactoring opportunities for the detection strategy currently being run have been collected, each functional-based refactoring is processed one at a time, using the following technique:

a. The functional-based refactoring currently being processed is passed to the dependency verification module, and its dependencies are analyzed.

b. If the dependency verification module determines that the refactoring can be performed without violating dependencies, the functional-based refactoring opportunity is passed to the refactoring module and the refactoring is performed.

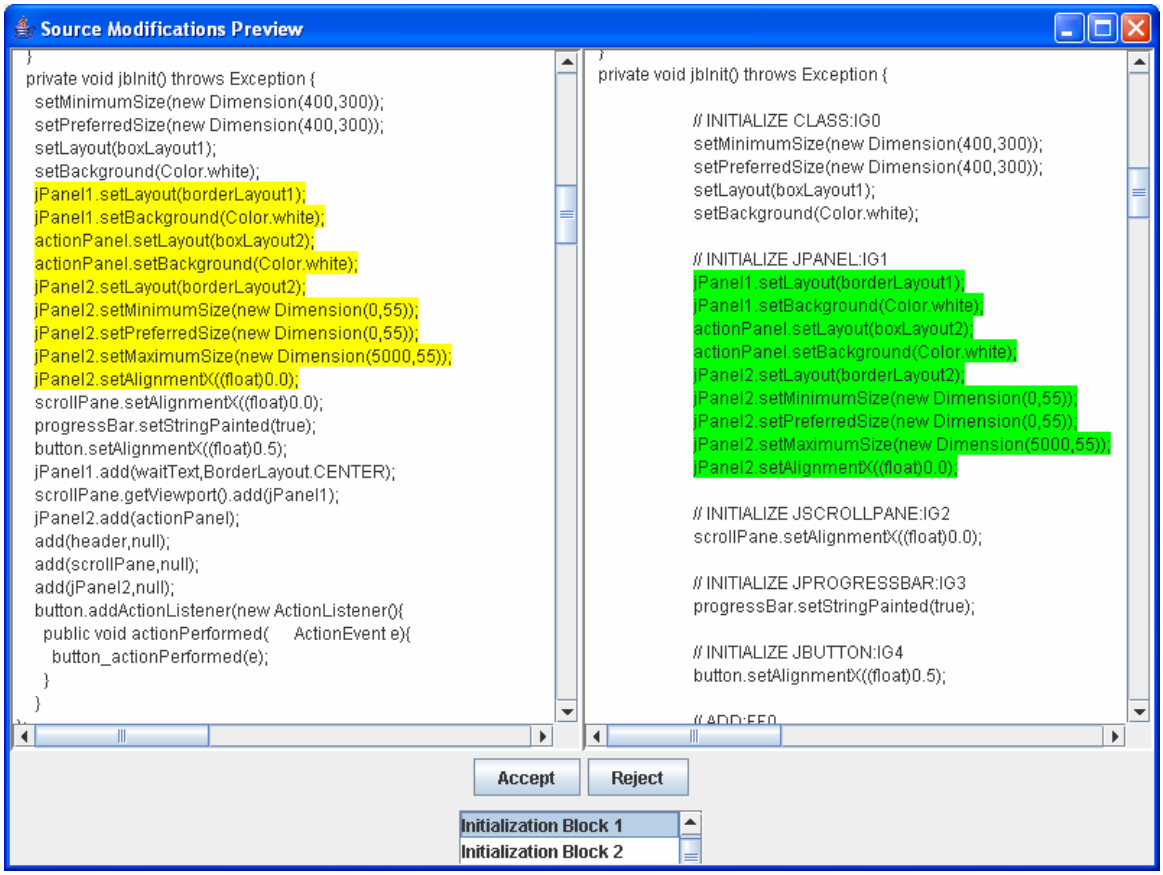

Figure 6-2: Screenshot of prototype.

Figure 6-2 above shows a screen shot of our prototype. On the left side of the figure, source code is shown before the refactoring process. The right side shows the source code after it has been refactored. A list allows the user to select a refactoring they wish to view. The tool then highlights the code associated with the refactoring to show a before and after comparison. The user can then accept or reject the refactored source code.

6.2 Objectives

Our refactorings and methods of managing dependencies constitute our new approach to refactoring source code. The objective of these case studies is to validate that the prototype correctly classifies functional-based refactorings as being valid or invalid.

The core task of this goal is ensuring that the dependencies from each functional-based refactoring opportunity are being correctly verified.

6.3 Subjects

JDictionary is an open-source Java based dictionary application [38]. Users can search dictionaries across many different languages. jEdit is a Java based cross-platform text editor for programmers [39]. Both software systems are open-source software. We perform case studies on both systems. Table 2 below, summarizes their properties.

System	Lines of Code	Num. of Classes	Num. of Methods	Language	Version
JDictionary	8130	80	460	Java	1.8
jEdit	56248	394	5545	Java	4.2

Table 2: Characteristics of open source applications used in case studies.

6.4 Evaluation

Our prototype was run on the source code for both applications. The results generated by the automated tool were recorded. We then manually reviewed each refactoring detected and the dependency verification process for that refactoring. This was done by manually reviewing the source code, manually reviewing each refactoring, and manually performing the dependency verification process. The results generated from the prototype were manually verified. The following list explains how we classify our experimental findings [37][57]:

- *True positive (TP)*: a valid functional-based refactoring opportunity which is correctly marked by our prototype as being valid.
- *True negative (TN)*: an invalid functional-based refactoring opportunity which is correctly marked by our prototype as being invalid.
- *False positive (FP)*: an invalid functional-based refactoring opportunity which is incorrectly marked by our prototype as being valid.
- *False negative (FN)*: a valid functional-based refactoring opportunity which is incorrectly marked by our prototype as being invalid.

The following formulas are used to calculate precision and recall:

Precision is defined as the percentage of items retrieved that are relevant.

- Precision is defined as the percentage of items retrieved that are relevant [57]. We calculate precision using the following formula:

- *Precision* = # true positives / (# true positives + # false positives)

- Recall is defined as the percentage of relevant items retrieved [57]. We calculate recall using the following formula:
  - *Recall* = # true positives / (# true positives + # false negatives)

- Incorrect Positives (IP) is the percentage of false positives to total positives, and is calculated using the following formula (this is a direct percentage calculation, not a formula developed as the result of research):
  - *Incorrect Positives* = # false positives / (#true positives + # false positives)

- Incorrect Negatives (IN) is the percentage of false negatives to total negatives, and is calculated using the following formula (this is a direct percentage calculation, not a formula developed as the result of research):
  - *Incorrect Positives* = # false negatives / (#true negatives + # false negatives

- Accuracy is defined by us as the percentage of correct classifications to total classifications, and is calculated using the following formula:
  - *Accuracy* = (#true positives + #true negatives) / (#true positives + #true negatives + #false positives + #false negatives)

6.5 Results

Table 3 lists the results for JDictionary, and Table 4 lists the results for jEdit. As summarized by Table 3 above, our prototype performs with high precision and recall for the case study on JDictionary. The majority of functional-based refactorings are being correctly classified as valid or invalid, based on whether the refactoring preserves

dependencies or violates dependencies. A precision of 0.99 and recall of 0.99 are achieved.

Detection Strategy	**TP**	**TN**	**FP**	**FN**	**Precision**	**Recall**	**IP**	**IN**
Loop Constant	0	0	0	0	**N/A**	**N/A**	**N/A**	**N/A**
Leading Duplicate Fragment in Conditional Region	0	0	0	0	**N/A**	**N/A**	**N/A**	**N/A**
Trailing Duplicate Fragment in Conditional Region	0	0	0	0	**N/A**	**N/A**	**N/A**	**N/A**
Declaration Region	248	31	1	0	**0.99**	**1.00**	**0.01**	**0.00**
Initialization Region	174	15	3	2	**0.98**	**0.99**	**0.02**	**0.12**
Functional Feature Region	28	11	1	4	**0.97**	**0.88**	**0.03**	**0.27**
Summary	**450**	**57**	**5**	**6**	**0.99**	**0.99**	**0.01**	**0.10**

Table 3: Summary results of JDictionary case study.

For the jEdit case study, our prototype performs with a high precision and recall with values of 0.99 and 0.96 respectively (Table 4).

Overall, our prototype performs with a precision of 0.99 and a recall of 0.96. Table 5 below summarizes the data across both case studies. Table 6 summarizes the accuracy rates for our case studies. Our overall accuracy for both case studies is 0.96.

Detection Strategy	**TP**	**TN**	**FP**	**FN**	**Precision**	**Recall**	**IP**	**IN**
Loop Constant	0	0	4	0	**0.00**	**0.00**	**1.00**	**0.00**
Leading Duplicate Fragment in Conditional Region	0	0	0	0	**N/A**	**N/A**	**N/A**	**N/A**
Trailing Duplicate Fragment in Conditional Region	4	0	0	0	**1.00**	**1.00**	**0.00**	**0.00**
Declaration Region	2421	506	1	48	**0.99**	**0.98**	**0.01**	**0.09**
Initialization Region	785	51	2	44	**0.99**	**0.95**	**0.01**	**0.46**
Functional Feature Region	123	26	4	58	**0.97**	**0.68**	**0.03**	**0.69**
Summary	**3333**	**583**	**11**	**150**	**0.99**	**0.96**	**0.01**	**0.20**

Table 4: Summary results of jEdit case study.

Detection Strategy	**TP**	**TN**	**FP**	**FN**	**Precision**	**Recall**	**IP**	**IN**
Loop Constant	0	0	4	0	**0.00**	**0.00**	**1.00**	**0.00**
Leading Duplicate Fragment in Conditional Region	0	0	0	0	**N/A**	**N/A**	**N/A**	**N/A**
Trailing Duplicate Fragment in Conditional Region	4	0	0	0	**1.00**	**1.00**	**1.00**	**1.00**
Declaration Region	2669	537	3	48	**0.99**	**0.98**	**0.01**	**0.08**
Initialization Region	959	66	5	46	**0.99**	**0.95**	**0.01**	**0.41**
Functional Feature Region	151	37	5	62	**0.97**	**0.71**	**0.03**	**0.63**
Summary	**3783**	**640**	**16**	**156**	**0.99**	**0.96**	**0.01**	**0.20**

Table 5: Summary results of both case studies.

Detection Strategy	**JDictionary Accuracies**	**jEdit Accuracies**	**Summary Accuracies**
Loop Constant	**N/A**	0	0
Leading Duplicate Fragment in Conditional Region	**N/A**	**N/A**	**N/A**
Trailing Duplicate Fragment in Conditional Region	**N/A**	1.00	1.00
Declaration Region	1.00	0.98	0.98
Initialization Region	0.97	0.95	0.95
Functional Feature Region	0.89	0.71	0.74
Summary	**0.98**	**0.96**	**0.96**

Table 6: Accuracy rates for case studies.

The prototype does not perform perfectly. There are instances of false positives and false negatives, which indicate that the prototype is not able to correctly verify if a functional-based refactoring is valid or invalid for all cases. The main reason our prototype does not perform perfectly is that it does not perform a full semantic analysis of the source code.

For example, if a line of code uses a variable that is modified through a method invocation (as opposed to being directly modified in the method) our prototype will not find this. An example of a false positive refactoring opportunity appears below in Figure 6-3.

```
int resultCount=0;
setAbortable(false);
try {
   buffer.readLock();
   for (int i=0; i < selection.length; i++) {
      Selection s=selection[i];
      if (s instanceof Selection.Rect) {
         for (int j=s.getStartLine(); j <= s.getEndLine(); j++) {
            resultCount+=doHyperSearch(buffer,s.getStart(buffer,j),s.getEnd(buffer,j));
         }
      }
      else {
         resultCount+=doHyperSearch(buffer,s.getStart(),s.getEnd());
      }
   }
}
finally {
   buffer.readUnlock();
}
setAbortable(true);
return resultCount;
```

Figure 6-3: Example false positive refactoring opportunity.

In this example, the highlighted two lines of code are identified as a possible Initialization Region refactoring, as they both initialize the same class. The intent of the first line is to prevent the process from being able to be aborted before the try-block has completed execution. After the *try-block* has executed, the process is set so that it is possible for it to be aborted. Both lines of code invoke the same method. However, to group them together would destroy the purpose for which they were written. To group the lines together either before or after the try block would permit the process to be aborted during execution of the try block, which is what the lines of code are intended to prevent.

Our dependency verification process marks this refactoring opportunity as valid because the statements are not directly using any variables. However, if a full semantic analysis were to be performed, the prototype would go into the *setAbortable(…)* method and discover that both statements are modifying the same variable, and would thus recognize the refactoring opportunity as being invalid. This error is the common cause for many of our false positives and false negatives.

As another example of why the false positives and false negatives occur, consider the following example in Figure 6-4 below. In this case, the highlighted line is recognized as a possible Declaration Region refactoring opportunity. When it has its dependencies analyzed, the prototype finds that it uses the variable *mode*. When the prototype scans backward from the highlighted line of code, it finds that the variable *mode* is also used in the first line of code from the example. It treats the first line of code as though the highlighted line of code has a dependency upon it. It then indicates that the highlighted line of code cannot be moved to be the first line in the block of code, because it cannot be moved before a line upon which it has a dependency. The refactoring opportunity is marked as being invalid. This happens because we take such a conservative or safe approach to dependency verification and treat any method being invoked on an object as a state change. In fact, the method being invoked on *mode* in the first line of code, does not alter any of *mode*'s properties. It simply returns a value. If a full semantic analysis were to be performed, the dependency verification process would recognize that there is not actually a dependency for the highlighted line of code on the first line of code, and that the highlighted line of code could indeed be moved to be the first line of code in the block.

Mention must also be made to the fact that since the refactoring data for the case studies was manually reviewed by humans, there is also a slight margin of error that is introduced by this process.

Complete data for our case study can be found in Appendix C.

```
prefix="mode." + mode.getName() + ".";
jEdit.setBooleanProperty(prefix + "customSettings",!useDefaults);
String oldFilenameGlob=(String)mode.getProperty("filenameGlob");
String oldFirstlineGlob=(String)mode.getProperty("firstlineGlob");
if (useDefaults) {
jEdit.resetProperty(prefix + "filenameGlob");
..
```

Figure 6-4: Example false negative refactoring opportunity.

Chapter 7

Conclusions and Future Work

In this section, we summarize the important contributions of our thesis and discuss future work related to our research.

7.1 Thesis Contributions

The main contributions of this thesis are as follows:

- *Development of automated strategies to identify functional regions of source code:* comprehension of source code can be a difficult and time consuming task. By grouping statements of code using our detection strategies, we make it easier for developers to locate code related to a feature and perform bottom-up program comprehension. This is because we reorganize source code into regions where each statement in a region shares common functionality with the other statements in the region, and therefore, are centralizing functional features to specific locations. This makes the reading of source code easier and less-time consuming since, developers have to spend less time locating statements of code they are searching for, because these statements are grouped together based on functionality.

- *Development of an automated approach to dependency management:* our dependency bounding condition approach is an approach to dependency location and dependency verification that helps to determine if statements of code can be reordered and grouped together.

- *Automatic comment generation:* in an effort to help developers understand source code and locate features they are searching for, our prototype tool automatically generates comments for the regions of code it creates, which describe the functionality of statements in that region.

7.2 Future Work

There are many areas which future research should address:

- *Complete semantic analysis:* our approach to refactoring and dependency management relies on a syntactical and semantical approach to recognizing source code functionality and locating dependencies. Future work should include a complete semantic analysis. This could increase our accuracy in automated refactoring process.

- *Expanding exception detection strategies:* Currently, we incorporate various heuristics in detecting source code which should not be refactored. Our approach is by no means complete and future work should address adding more categories of code which should not be refactored. As one example, statements or sections of code can have instrumentation code attached to them. Moving statements of code without moving attached instrumentation code could destroy the purpose for which the code is being instrumented. Future work should, as much as is possible, develop methods for

identifying which statements of code the instrumentation code is attached to, and keep the instrumentation code with the code it is instrumenting when it is moved.

7.3 Conclusion

As the software development industry and the sizes of software continue to grow, the need for methods to refactor source code into a more readable and more comprehensible state will continue to grow. In this research, we developed an automatic approach to reorder functionally related statements into regions, verify dependencies within the source code, and refactor the source code. The concept of being able to improve the readability of source code is of great benefit for all software developers. Also of great benefit is the capability to automatically search through monolithic volumes of source code and automatically refactor them. Future work could incorporate different ideas to expand the range of refactorings we address in our research. As the software industry continues to grow, it is hoped that many of the practices developed during this research can be applied both formally and informally by software engineers.

Bibliography

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint and C. K. I. Williams. Using Machine Learning to Focus Iterative Optimization. In *the Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[2] L. Almagor, K.D. Cooper, A. Grosul, T. J. Harvey, S. W.Reeves, D. Subramanian, L. Torczon and T.Waterman: Finding effective compilation sequences In *LCTES* 2004.

[3] R. Anderson, A. Munshi, and B. Simons. A Scheduling Problem Arising from Loop Parallelization on MIMD Machines. In *the Proceedings of the Third Aegean Workshop on Computing*, Corfu Greece, pages 124-133, 1988.

[4] G. Antoniol, M. Di Penta and M. Merlo. Predicting Refactoring Activities via Time Series. *The First International Workshop on Refactoring (REFACE)*, Victoria, BC, Canada, Nov 13 2003.

[5] G. Antoniol and Y. Gueheneuc. Feature Identification: A Novel Approach and a Case Study. In *the Proceedings of the Twenty-First International Conference on Software Maintenance*, Budapest Hungary, pages 357 – 366, 2005.

[6] R. M. Baecker and A. Marcus. *Human Factors and Typography for More Readable Programs*. Addison-Wesley, Reading, MA, 1990.

[7] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M F. P. O'Boyle, and E. Rohou. Iterative Compilation in a Non-Li*near* Optimisation Space. In *the Proceedings of the Workshop on Profile Directed Feedback-Compilation (PACT'98),* 1998.

[8] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[9] V. Bontchev. Macro and Script Virus Polymorphism. In *the Proceedings of the Twelfth International Virus Bulletin Conference,* 2002.

[10] G. Brat and W. Visser. Combining Static Analysis and Model Checking for Software Analysis. In *the Proceedings of the Sixteenth International Conference on Automated Software Engineering*, page(s):262 – 269, 2001.

[11] R. Brooks. Towards a Theory of the Cognitive Processes in Computer Programming. *International Journal on Man-Machine Studies*, 1977.

[12] R. Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, Volume 18, 1983.

[13] C. J. Brownhill, A. Nicolau, S. Novack, and C. D. Polychronopoulos. The PROMIS Compiler Prototype. In *the Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 116 – 125, 1997.

[14] K. Chen and V. Rajlich. Case Study of Feature Location Using Dependence Graph. In *the Proceedings of the Eighth International Workshop on Program Comprehension*, Limerick Ireland, pages 241 – 249, 2000.

[15] D. Chen and P. Yew. Statement Re-ordering for DOACROSS Loops. In *the Proceedings of the International Conference on Parallel Processing*, North Carolina State University North Carolina, pages 24 – 28, 1994.

[16] C. Chih-Ping and D. L. Carver. Reordering the Statements with Dependence Cycles to Improve the Performance of Parallel Loops**.** In *the Proceedings of the International Conference on Parallel and Distributed Systems*, pages 322 – 328, 1997.

[17] M. Christodrescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *the Proceedings of the Twelth USENIX Security Symposium*, Washington DC, 2003.

[18] Code Smell. http://en.wikipedia.org/wiki/Code_smell.

[19] T. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294 – 306, 1989.

[20] S. Counsell, Y. Hassoun, R. Johnson, K. Mannock and E. Mendes. Trends in Java Code Changes: The Key to Identification of Refactorings? In *the Proceedings of the Second International Conference on Principles and Practice of Programming in Java*, pages 45 – 48, 2004.

[21] R. Cytron. Compile-time Scheduling and Optimization for Asynchronous Machines**.** PhD thesis. University of Illinois at Urbana-Champaign, 1984.

[22] T. Eisenbarth, R. Koschke and D. Simon. Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29(3), pages 210 – 224, 2003.

[23] K. Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis and S. Rai. The Optimal Class Size for Object-Oriented Software. *IEEE Transactions on Software Engineering*, 28(5), May 2002.

[24] D. Eng. Combining Static and Dynamic Data in Code Visualization. In *the Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, Charleston South Carolina, pages 43 – 50, 2002.

[25] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *the Proceedings of the Symposium on Security and Privacy*, Berkeley California, 2004.

[26] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *the Proceedings of the Symposium on Security and Privacy*, 2003.

[27] B. Florian, A. Lozano and S. Takahashi. Automatic Detection of Bad Smells Using Software Metrics. In *the Poster Proceedings of the Twenty-First International Conference on Software Maintenance*, Budapest Hungary, pages 20 – 23, 2005.

[28] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[29] V. French. Establishing Software Metrics Thresholds. In *the Proceedings of the Ninth International Workshop on Software Measurement*, 1999.

[30] A. K. Ghosh and A. Schwartzbard. Learning Program Behavior Profiles for Intrusion Detection. In *the Proceedings of the First USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 51–62, Santa Clara California, 1999.

[31] God Object. http://en.wikipedia.org/wiki/God_object.

[32] A. Goldberg. Programmer as Reader. *IEEE Software*, 4(5), pages 62 – 70, September 1987.

[33] O. Greevy, S. Ducasse and T. Girba. Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis. In *the Proceedings of the Twenty-First International Conference on Software Maintenance*, Budapest Hungary, pages 347 – 356, 2005.

[34] M. Hung and Y. Zou. A Framework for Exacting Workflows from E-Commerce Systems. In *the Proceedings of Software Technology and Engineering Practice*, Budapest Hungary, 2005.

[35] M. Hung and Y. Zou. Recovering Workflows from Multi Tiered E-commerce Systems. In *the Proceedings of the Fifteenth International Conference on Program Comprehension*, Banff Alberta, pages198 – 207, 2007.

[36] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Standard*, 610.12-1990, 1990.

[37] Information Retrieval. http://en.wikipedia.org/wiki/Information_retrieval.

[38] JDictionary. http://jdictionary.sourceforge.net/.

[39] jEdit. http://jedit.org/.

[40] W. Kelly and W. Q. Pugh. A unifying framework for iteration reordering transformations. In *the Proceedings of the First International Conference on Algorithms and Architectures for Parallel Processing*, pages 153 – 162, 1995.

[41] J. Korn, Y. Chen and E. Koutsofios. Chava: Reverse Engineering and Tracking of Java Applets. In *the Proceedings of the Sixth International Working Conference on Reverse Engineering*, Atlanta Georgia, pages 314 – 325, Oct 6 – 8, 1999.

[42] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Park, and K. Gallivan. Finding effective optimization phase sequences. In *ACM LCTES*, 2003.

[43] K. Kumar, A. K, Pappu, S. Kumar and S. Sanyal. A Hybrid Approach for Parallelization of Sequential Code with Function Level and Block Level

Parallelization. *In the Proceedings of the International Symposium on Parallel Computing in Electrical Engineering*, pages161 – 166, 2006.

[44] A. Lakhotia and M. Mohammed. Imposing Order on Program Statements to Assist Anti-virus Scanners. In the Proceedings of the Eleventh Working Conference on Reverse Engineering, Delft The Netherlands, pages161 – 170, 2004.

[45] S. Letovsky. Cognitive Processes in Program Comprehension. In E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers*, Ablex Publishing Corporation, pages 58-79, 1986.

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

[47] L. Y. Liu and R. K. Shyamasundar. Static Analysis of Real-time Distributed Systems. *IEEE Transactions of Software Engineering*, 16(4), pages 373 – 388, 1990.

[48] M. Munro. Product Metrics for Automated Identification of "Bad Smell" Design Problems in Java Source Code. In *the Proceedings of the Eleventh International Software Metrics Symposium*, pages 15 – 23, 2005.

[49] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density**.** In *the Proceedings of the Twenty-seventh International Conference on Software Engineering*, St-Louis Missouri, pages 580 – 586, 2005.

[50] E. Nurvitadhi, W. W. Leung and C. Cook. Do Class Comments Aid Java Program Understanding? *Frontiers in Education*, Volume 1, page T3C-13 - T3C-17, 2003.

[51] P. Oman and C. R. Cook. Typographic Style is More than Cosmetic. *Communications of the ACM*, 33(5), pages 506 – 520, 1990.

[52] G. Parikh and N. Zvegintov. *Tutorial on Software Maintenance*. IEEE Computer Society Press, 1996.

[53] N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, Volume 19, pages 295 – 341, 1987.

[54] N. Pennington. Comprehension Strategies in Programming. In G. M. Olson, S. Sheppard and E. Soloway, eds., *Empirical Studies of Programmers*: Second Workshop, Ablex Publishing Corporation, pages 100 - 112, 1987.

[55] J. Plodzien and K. Subieta. Static Analysis of Queries as a Tool for Static Optimization, In *the Proceedings of the International Symposium on Database Engineering and Applications*, pages 117 – 122, 2001.

[56] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antonial and V. Rajlich. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. In *the Proceedings of the Fourteenth International Conference on Program Comprehension*, Athens Greece, pages 137 – 148, 2006.

[57] Precision and Recall Calculation. http://www.kdnuggets.com/faq/precision-recall.html.

[58] V. Purnell, P. H. Corr and P. Milligan. A Novel Approach to Loop Parallelization**.** In *the Proceedings of the Twenty-third Euromicro Conference on New Frontiers of Information Technology. Short Contributions*, pages 272 – 277, 1997.

[59] L. Rosenberg, R. Stapko and A. Gallo. Object-Oriented Metrics for Reliability. *IEEE International Symposium of Software Metrics*, 1999.

[60] M. Salah and, S. Mancordis. A Hierarchy of Dynamic Software Views: from Object-Interactions to Feature-Interactions. In *the Proceedings of the Twentieth International Conference on Software Maintenance*, Chicago Illinois, pages 72 – 81, 2004.

[61] M. Salah, S. Mancordis, G. Antoniol and M. Di Penta. Towards Employing Use-Cases and Dynamic Analysis to Comprehend Mozilla. In *the Proceedings of the Twenty-First International Conference on Software Maintenance*, Budapest Hungary, pages 639 – 642, 2005.

[62] B. Shneiderman. *Software Psychology, Human Factors in Computer and Information Systems.* Whthrop Publishers, Inc., Chap. 3, pages 39 – 62, 1980.

[63] B. Shneiderman and R. Mayer. Syntactic/semantic Interactions in Programmer Behavior: A Model and Experimental Results. International Journal of Computer and Information Sciences, pages 219 – 238, 8(3), 1979.

[64] F. Simon, F. Steinbruckner and C. Lewerentz. Metrics Based Refactoring. In *the Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, Lisbon Portugal, pages 30 – 38, 2001.

[65] E. Soloway, B. Adelson and K. Ehrlich. Knowledge and Processes in the Comprehension of Computer Programs. In *The Nature of Expertise*, M. Chi, R. Glaser and M. Farr, eds., A. Lawrence Erlbaum Associates, pages 129-152, 1988.

[66] E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, 10(1) pages 595-609, 1984.

[67] M. A. Storey. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In the Proceedings of the Thirteenth International Workshop on Program Comprehension, Pittsburgh Pennsylvania, pages 181 - 191, 2005.

[68] P. Ször and P. Ferrie. Hunting for Metamorphic. In *the Proceedings of the Eleventh International Virus Bulletin Conference*, 2001.

[69] T. Tenny. Program Readability: Procedures versus Comments. *IEEE Transactions on Software Engineering*, 14(9), pages 1271 – 1279, 1988.

[70] A. Von Mayrhauser, A. M. Vans. Industrial Experience with an Integrated Code Comprehension Model. Software Engineering Journal, 10(5), pages 171 – 182, 1995.

[71] N. Wilde, M. Buckellew, H. Page and V. Rajlich. A Case Study of Feature Location in Unstructured Legacy Fortran Code. In *the Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, Lisbon Portugal, pages 68 – 76, 2001.

[72] T. Zhang, X. Zhuang and S. Pande. Compiler Optimizations to Reduce Security Overhead. In *the Proceedings of the International Symposium on Code Generation and Optimization*, pages 345 – 357, 2006.

[73] L. Zhen, S. M. Bridges and R. B. Vaughn. Combining Static Analysis and Dynamic Learning to Build Accurate Intrusion Detection Models, In *the Proceedings of the Third International Workshop on Information Assurance*, pages 164 – 177, 2005.

[74] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl and M. A. Vouk. On the value of static analysis for fault detection in software, IEEE Transactions on Software Engineering, 32(4), pages 240 – 253, 2006.

[75] Y. Zou, M. Hung. An Approach for Extracting Workflows from E-Commerce Applications. In *the Proceedings of the Fourteenth International Conference on Program Comprehension*, Athens Greece, pages127 – 136, 2006.

[76] Y. Zou, T. C. Lau, K. Kontogiannis, T. Tong and R. McKegney. Model-Driven Business Process Recovery. In the Proceedings of the Eleventh Working Conference on Reverse Engineering, Delft The Netherlands, pages 224 – 233, Nov. 8 – 12, 2004.

Appendix A

Additional Refactoring Examples

A.1 Bounded Dependency Condition Example – Valid Refactoring Opportunity

To illustrate our approach to managing dependencies explained above in Chapter 5, consider the following source code found in Figure A-1.

Assume that at this point in the refactoring process, we have grouped the declaration statements and initialization statements into their corresponding regions. Now, we would be ready to process the groups of functional feature regions and turn them into functional regions. Lines {9, 10, 12, 13} would be grouped together as a functional-based refactoring opportunity during the detection process, as they all invoke the *add* method. The processing of their dependencies would be as follows:

We construct the dependency condition graph for each statement in the refactoring opportunity. To illustrate this concept, we will use the functional feature region refactoring opportunity discussed in Chapter 3.

```
    private void initialize( ) {

        // DECLARATIONS:
1       JPanel jPanel;
2       String title;

        // INITIALIZE INFOPANEL:
3       setLayout(boxLayout);
4       setBackground(Color.white);

        // INITIALIZE JPANEL:
5       jPanel = new JPanel( );
6       jPanel.setLayout(borderLayout);
7       jPanel.setBackground(Color.white);
8       jPanel.setAlignmentX((float) 0.0);

9       jPanel.add(text, BorderLayout.CENTER);
10      add(header);
11      title = "My Window";
12      add(title);
13      add(jPanel);
    }
```

Figure A-1: Example dependency source code before refactoring.

To process line 9, we notice that the variable/objects that this statement uses are: *jPanel* and *text*. The pre-dependency for this statement, is the closest statement to line 9, which appears before line 9, which uses any of the variables that appear in this statement (*jPanel*, *text*). We notice that the statement at line 8 uses the object *jPanel*. Therefore, the pre-dependency is S_8. To find the post-dependency, we look for the statement closest to line 9, that appears after line 9, which uses any of the variables that appear in the statement at line 9. We notice that the statement at line 13 uses the object *jPanel*.

Therefore, the post-dependency is S_{13}. The *Dependency Condition Graph* for this statement is:

$$(S_8, \mathbf{S_9}, S_{13})$$

We follow a similar process to obtain the dependency condition graphs for the remainder of the statements which constitute the refactoring opportunity. This yields the following dependency condition graphs for the entire refactoring opportunity:

$$(S_8, \mathbf{S_9}, S_{13})$$

$$(S_4, \mathbf{S_{10}}, S_{12})$$

$$(S_{11}, \mathbf{S_{12}}, S_{13})$$

$$(S_{12}, \mathbf{S_{13}}, +\infty)$$

The next step is to see if we can merge any of the graphs. In this case, we can merge the third and fourth graphs. This brings our set of dependency condition graphs to be the following:

$$(S_8, \mathbf{S_9}, S_{13})$$

$$(S_4, \mathbf{S_{10}}, S_{12})$$

$$(S_{11}, \mathbf{S_{12}}, \mathbf{S_{13}}, +\infty)$$

To verify if the refactoring opportunity is valid, we perform the following computation:

1. We construct a set of all of the pre-dependencies from all of the graphs which represent our refactoring opportunity. This set is $\{S_4, S_8, S_{11}\}$. We then select the statement which appears latest in the code, from this set, which is S_{11}.
2. We construct a set of all of the post-dependencies from all of the graphs which represent our refactoring opportunity. This set is $\{S_{12}, S_{13}, +\infty\}$. We then select the statement which appears earliest in the code, from the set, which is S_{12}.
3. The pair generated as the bounded dependency condition for our refactoring opportunity is (S_{11}, S_{12}). This tells us two important facts:
    a. The refactoring opportunity is valid. The line represented by the pre-dependency is before the line represented by the post-dependency. This means that there is a valid space, range, or location in the code where the statements of code from the refactoring opportunity could be located together in a continuous group, without violating any of the dependencies.
    b. This also tells us the bounds, or locations within the code where the refactoring opportunity could be located. The pair representing the bounded dependency condition tells us that the refactoring opportunity could be located after the statement at line 11, and before or at the statement at line 12. A re-examination of the code confirms this. Line 11 re-assigns the variable *title* which is used in the statement at line 12. Of all of the pre-dependency dependencies in the code, the dependency at line 11 (for the statement at line 12) is the one that appears the latest. It

can be seen that in order to place all of the statements from the refactoring opportunity together, they must be located after the dependency at line 11. For dealing with the post-dependency dependencies, the dependency which appears earliest in the code is the dependency found at line 12. This indicates that all of the statements in the refactoring opportunity must be located at or before this dependency.

```
private void initialize( ) {

   // DECLARATIONS:
   1 JPanel jPanel;
   2 String title;

   // INITIALIZE INFOPANEL:
   3 setLayout(boxLayout);
   4 setBackground(Color.white);

   // INITIALIZE JPANEL:
   5 jPanel = new JPanel( );
   6 jPanel.setLayout(borderLayout);
   7 jPanel.setBackground(Color.white);
   8 jPanel.setAlignmentX((float) 0.0);

   9 title = "My Window";

   // ADD:
   10 jPanel.add(text, BorderLayout.CENTER);
   11 add(header);
   12 add(title);
   13 add(jPanel);
}
```

Figure A-2: Example dependency source code after refactoring.

It is important to note that even though our detection strategies group statements together that are from the same block, lexical level, scope, etc… the dependency management process searches through all of the scopes, lexical levels and blocks for dependencies, within the complete bounds of a method. This means that if a variable in a refactoring opportunity (which exists at lexical level 2) is reassigned in a statement within a loop, (where the statement performing the reassignment exists at lexical level 3), the dependency would still be detected, even though the two statements exist within different blocks and lexical levels.

The source code appears as shown in Figure A-2 after performing the refactoring.

A.2 Bounded Dependency Condition Example – Invalid Refactoring Opportunity

To illustrate our approach when dealing with a refactoring opportunity that is invalid, consider the following source code in Figure A-3, which is almost identical to the code from the example above, with the exception of line 9 (highlighted), which now uses the variable *title*.

We follow the process of generating dependency condition graphs for each line in the refactoring opportunity {9, 10, 12, 13}. This yields the following set of dependency graphs:

$$(S_8, \mathbf{S_9}, S_{11})$$

$$(S_4, \mathbf{S_{10}}, S_{12})$$

$(S_{11}, \mathbf{S_{12}}, S_{13})$

$(S_{12}, \mathbf{S_{13}}, +\infty)$

```
private void initialize( ) {

   // DECLARATIONS:
   1 JPanel jPanel;
   2 String title;

   // INITIALIZE INFOPANEL:
   3 setLayout(boxLayout);
   4 setBackground(Color.white);

   // INITIALIZE JPANEL:
   5 jPanel = new JPanel( );
   6 jPanel.setLayout(borderLayout);
   7 jPanel.setBackground(Color.white);
   8 jPanel.setAlignmentX((float) 0.0);

   9 jPanel.add(title, BorderLayout.CENTER);
   10 add(header);
   11 title = "My Window";
   12 add(title);
   13 add(jPanel);
}
```

Figure A-3: Example source code.

Merging the graphs together, where possible, yields the following set of graphs:

$(S_8, \mathbf{S_9}, S_{11})$

$(S_4, \mathbf{S_{10}}, S_{12})$

$$(S_{11}, \mathbf{S_{12}}, \mathbf{S_{13}}, +\infty)$$

To generate the pair which represents the bounded dependency condition, we perform the following steps:

1. From the set of all of the pre-dependencies for this refactoring opportunity {S_4, S_8, S_{11}}, we select the statement which appears latest in the code. In this case, we select S_{11}.
2. From the set of all of the post-dependencies for this refactoring opportunity {S_{11}, S_{12}, $+\infty$}, we select the statement which appears earliest in the code. In this case, we select S_{11}.
3. This generates the pair of (S_{11}, S_{11}) for the bounded dependency condition for the entire refactoring opportunity. The condition of the pre-dependency being before the line of code represented by the post-dependency does not hold. This tells us that the refactoring opportunity is not a valid one, and cannot be performed.

If we re-examine the code for this refactoring opportunity, it becomes apparent why these statements cannot be grouped together. There is no way that S_9 and S_{12} could be moved to be together and preserve the original behaviour of the code. This is because S_9 appears before the statement where *title* has its value changed. S_9 uses the original value of *title*. S_{12} appears after *title* has had its value reassigned, and therefore uses the new value of *title*. To put S_9 and S_{12} together, they would both then have to use the old or new value of *title* and would not therefore be preserving the original behaviour of the code.

Appendix B

Detailed Algorithms for Detection Strategies

B.1 Declaration Region

Input:

SC: source code for a java class

Output:

RC: refactored source code

Algorithm:

Begin

```
-- initialize the vector which will hold the declaration
statements in each block
```

declarationStatements: Vector

```
-- initialize the vector which will hold the groups of
declaration statements from each block
```

DG: Vector

```
-- process the source code one block at a time
```

for each Block $B \in SC$

```
    -- get all of the DeclarationStatement's for the current
block
```

blockDeclarationStatements = ϕ

for each *Statement* $S \in B$

if *S* is of type *DeclarationStatement*

blockDeclarationStatements = blockDeclarationStatements ∪ S

end if

end for

DG = DG ∪ blockDeclarationStatements

end for

for each *blockDeclarationStatements ∈ DG*

`-- start inserting Declaration Statements at the beginning of the block`

declarationBlockInsertionLocation = start of Block

for each *declarationStatement ∈ blockDeclarationStatements*

`-- get the dependency space bounds for the current Declaration statement`

Bounds = DependencyEngine(*declarationStatement)*

if *Bounds.lowerBound <= declarationBlockInsertionLocation*

moveStatement(*declarationStatement*, *declarationBlockInsertionLocation)*

initializationBlockInsertionLocation ++

end if

end for

end for

B.2 Initialization Region

Input:

SC: source code for a java class

Output:

RC: refactored source code

Algorithm:

Begin

`-- initialize the vector which will hold the initialization statements`

initializationStatements: Vector

```
-- initialize the vector which will hold the groups of
initialization statements
-- from each block
```

IG: Vector

```
-- process the source code one block at a time
```

for each Block $B \in SC$

```
    -- get all of the InitializationStatement's for the current
block
```

initializationStatements = ϕ

Vector *blockInitializationGroupings* = ϕ

for each *Statement S* $\in$ *B*

if *S* is of type *InitializationStatement*

initializationStatements = *initializationStatements* $\cup$ *S*

else if *S* invokes a *SetterMethod*

initializationStatements = *initializationStatements* $\cup$ *S*

end if

end for

```
    -- group the initialization statements into subsets based on
class
```

for each *Statement S1* $\in$ initializationStatements

ClassType *classType* = ClassType of *S1*

Vector *groupedInitializationStatements* = ϕ

for each *Statement S2* $\in$ *initializationStatements*

if *S2* is of type *classType*

groupedInitializationStatements = *groupedInitializationStatements* $\cup$ *S2*

initializationStatements = *initializationStatements* – *S2*

end if

end for

```
        -- add the current grouping to the block's initialization
groups
```

blockInitializationGroupings = blockInitializationGroupings ∪ *groupedInitializationStatments*

end for

`-- add the Block's initialization groupings for the Java Class`

IG = IG ∪ *blockInitializationGroupings*

end for

for each *blockInitializationGroupings* ∈ *IG*

`-- start inserting Initialization Groupings at the location after the Declaration Grouping`

initializationBlockInsertionLocation = declarationBlockInsertionLocation

for each *groupedInitializationStatements* ∈ *blockInitializationGroupings*

`-- get the dependency space bounds for the current group of Initialization statements`

Bounds = DependencyEngine(*groupedInitializationStatements)*

`-- check for a non-existing dependency space, which indicates that the statements cannot be grouped together into a continuous functional block of code`

if Bounds.lowerBound = = -1

break

end if

`-- place the functional block at the proper location of the 3 possible cases`

if *Bounds.lowerBound* <= *initializationBlockInsertionLocation* AND

Bounds.upperBound >= *initializationBlockInsertionLocation*

moveStatements(*groupedInitializationStatements* , *initializationBlockInsertionLocation)*

initializationBlockInsertionLocation += numberOfStatements(*groupedInitializationStatements)*

else if *Bounds.upperBound* < *initializationBlockInsertionLocation*

moveStatements(*groupedInitializationStatements* , *Bounds.lowerBound)*

initializationBlockInsertionLocation += numberOfStatements(*groupedInitializationStatements)*

else

moveStatements(*groupedInitializationStatements* , *Bounds.lowerBound)*

end if

end for

end for

B.3 Functional Feature Region

Input:

SC: source code for a java class

Output:

RC: refactored source code

Algorithm:

Begin

```
-- initialize the vector which will hold the invocation
statements
```

invocationStatements: Vector

```
-- initialize the vector which will hold the groups of invocation
statements
-- from each block
```

FFB: Vector

```
-- process the source code one block at a time
```

for each Block $B \in SC$

```
    -- get all of the MethodInvocationStatement's for the current
block
```

invocationStatements = ϕ

Vector *blockInvocationGroupings* = ϕ

for each *Statement* $S \in B$

if *S* is of type *InvocationStatement*

invocationStatements = *invocationStatements* $\cup$ *S*

```
            end if
        end for
        -- group the invocation statements into subsets based on
method invoked
        for each Statement S1 ∈ invocationStatements
            Method method = Method Invoked of S1
            Vector groupedInvocationStatements = ϕ
            for each Statement S2 ∈ invocationStatements
                if S2 is of type method
                    groupedInvocationStatements = groupedInvocationStatements ∪ S2
                    invocationStatements = invocationStatements – S2
                end if
            end for
            -- add the current grouping to the block's invocation
groups
            blockInvocationGroupings = blockInvocationGroupings ∪
groupedInvocationStatements
        end for
        -- add the Block's invocation groupings for the Java Class
        FFB = FFB ∪ blockInvocationGroupings
end for

for each blockInvocationGroupings ∈ FFB
        -- start inserting Invocation Groupings at the location after
the
         -- Initialization Groupings
        invocationBlockInsertionLocation = initializationBlockInsertionLocation
        for each groupedInvocationStatements ∈ blockInvocationGroupings
            -- get the dependency space bounds for the current group
of Invocation statements
            Bounds = DependencyEngine(groupedInvocationStatements)
            -- check for a non-existing dependency space, which
indicates that the statements cannot be grouped together into a
continuous functional region of code
            if Bounds.lowerBound = = -1
```

```
                    break
                end if
                -- place the functional region at the proper location of
the 3 possible cases
                if Bounds.lowerBound <= invocationRegionInsertionLocation AND
                    Bounds.upperBound >= invocationRegionInsertionLocation
                    moveStatements(groupedInvocationStatements,
invocationRegionInsertionLocation)
                    invocationRegionInsertionLocation +=
numberOfStatements(groupedInvocationStatements)
                else if Bounds.upperBound < invocationBlockInsertionLocation
                    moveStatements(groupedInvocationStatements, Bounds.lowerBound)
                    invocationRegionInsertionLocation +=
numberOfStatements(groupedInvocationStatements)
                else
                    moveStatements(groupedInvocationStatements, Bounds.lowerBound)
                end if
            end for
        end for
```

B.4 Leading Duplicate Code Fragment in Conditional Region

Input:

SC: source code for a java class

Output:

refactored source code

Algorithm:

Begin

```
-- initialize the vector which will hold the duplicate statements
```

duplicateStatements: Vector

```
-- initialize the vector which will hold the groups of leading
duplicate
-- conditional statements from each Conditional Block
```

LDCB: Vector

for each IfStatement Structure $\in$ *SC*

duplicateStatements = ϕ

if there is an else branch in the current IfStatement Structure

for each statement 1 to N in each Branch of the IfStructure

if every statement at position N matches

put each statement in *duplicateStatements*

else

break;

end if

end for

end if

LDCB = *LDCB* $\cup$ *duplicateStatements*

end for

for each *duplicateStatements* $\in$ *LDCB*

copy *duplicateStatements* to the location before their parent block

delete the original *duplicateStatements*

end for

B.5 Trailing Duplicate Code Fragment in Conditional Region

Input:

SC: source code for a java class

Output:

refactored source code

Algorithm:

Begin

```
-- initialize the vector which will hold the duplicate statements
```

duplicateStatements: Vector

```
-- initialize the vector which will hold the groups of trailing
duplicate
-- conditional statements from each Conditional Block
```

TDCB: Vector

for each IfStatement Structure $\in$ *SC*

duplicateStatements $= \phi$

if there is an else branch in the current IfStatement Structure

for each statement N to 1 in each Branch of the IfStructure

if every statement at position N matches

put each statement in *duplicateStatements*

else

break;

end if

end for

end if

TDCB = *TDCB* $\cup$ *duplicateStatements*

end for

for each *duplicateStatements* $\in$ *TDCB*

copy *duplicateStatements* to the location before their parent block

delete the original *duplicateStatements*

end for

B.6 Loop Constant

Input:

SC: source code for a java class

Output:

refactored source code

Algorithm:

Begin

```
-- initialize the vector which will hold the loop constants
```

loopConstants: Vector

for each *Statement S* ∈ *Loop Block*

    if *S* is a loop constant

        loopConstants = *loopConstants* ∪ *S*

    end if

end for

for each *loopConsant* ∈ *loopConstants*

    copy *loopConsant* to the location before their parent block

    delete the original *loopConsant*

end for

Appendix C

Statistics from Case Studies

C.1 JDictionary Case Study Statistics

	CLASS	DECLARATION REGION			
		TP	TN	FP	FN
1	KunststoffButtonUI.java	5	0	0	0
2	KunststoffComboBoxUI.java	0	5	0	0
3	KunststoffListUI.java	5	4	0	0
4	KunststoffLookAndFeel.java	0	0	0	0
5	KunststoffMenuBarUI.java	5	0	0	0
6	KunststoffMenuUI.java	1	0	0	0
7	KunststoffRadioButtonUI.java	5	0	0	0
8	KunststoffScrollBarUI.java	7	2	0	0
9	KunststoffScrollButton.java	9	2	0	0
10	KunststoffTabbedPaneUI.java	4	0	0	0
11	KunststoffTextFieldUI.java	3	0	0	0
12	KunststoffTheme.java	0	0	0	0
13	KunststoffToolBarUI.java	5	0	0	0
14	ModifiedDefaultListCellRenderer.java	0	0	0	0
15	DefaultFrameAssembler.java	2	0	0	0
16	FileDownloader.java	5	1	0	0
17	IconBank.java	0	0	0	0
18	ImageBank.java	0	0	0	0
19	JDictionary.java	13	2	0	0
20	JDictionaryFrame.java	0	0	0	0
21	LatestVersionChecker.java	1	1	0	0
22	NewJDictionaryVersionChecker.java	3	1	0	0
23	NewsAgent.java	10	0	0	0
24	NewsChecker.java	4	1	0	0
25	Plugin.java	0	0	0	0

26	Preferences.java	3	1	0	0
27	Prefs.java	6	0	0	0
28	ProxyAuthenticator.java	0	0	0	0
29	NewJDictionaryVersionEvent.java	0	0	0	0
30	NewsEvent.java	0	0	0	0
31	PluginFilesChangeEvent.java	0	0	0	0
32	PluginScanFinishedEvent.java	0	0	0	0
33	PluginSelectionEvent.java	0	0	0	0
34	PluginStructureChangeEvent.java	0	0	0	0
35	AboutPanel.java	0	0	0	0
36	AllPluginsDeactivatedPanel.java	0	0	0	0
37	InfoPanel.java	0	0	0	0
38	JDictionaryTheme.java	0	0	0	0
39	NetworkErrorCenteredPanel.java	5	0	0	0
40	NetworkErrorPanel.java	4	0	0	0
41	NoPluginsPanel.java	0	0	0	0
42	OutputTabbedPane.java	0	0	0	0
44	ProxyButton.java	0	0	0	0
45	SheetedMutableTreeNode.java	0	0	0	0
46	UpgradeAgent.java	18	0	0	0
47	NewJDictionaryVersionListener.java	0	0	0	0
48	NewsListener.java	0	0	0	0
49	PluginFilesChangeListener.java	0	0	0	0
50	PluginScanFinishedListener.java	0	0	0	0
51	PluginSelectionListener.java	0	0	0	0
52	PluginStructureChangeListener.java	0	0	0	0
53	DefaultFooter.java	0	0	0	0
54	DefaultHeader.java	0	0	0	0
55	DefaultLabelledPane.java	0	0	0	0
56	DefaultMenuBar.java	0	0	0	0
57	DefaultOutput.java	15	0	0	0
58	DefaultToolBar.java	0	0	0	0
59	DefaultTreeView.java	10	0	0	0
60	PluginDeployer.java	2	0	0	0
61	PluginDownloader.java	7	0	0	0
62	PluginInfoSheet.java	23	3	0	0
63	PluginLoader.java	7	1	0	0
64	PluginManager.java	53	7	0	0
65	PluginStatusChanger.java	8	0	1	0

	INITIALIZATION REGION				**FF REGION**			
	TP	TN	FP	FN	TP	TN	FP	FN
1	0	1	0	0	0	1	0	0
2	1	1	0	0	0	1	0	0
3	4	1	0	0	0	1	0	0

4	0	0	0	0	1	0	0	0
5	0	1	0	0	0	1	0	0
6	1	0	0	0	0	0	0	0
7	0	1	0	0	0	1	0	0
8	4	1	0	0	0	1	0	0
9	5	1	0	0	0	1	0	0
10	5	0	0	0	0	0	0	0
11	1	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0
13	0	1	0	0	0	1	0	0
14	1	0	0	0	0	0	0	0
15	12	2	1	0	0	0	0	1
16	1	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0
19	6	2	2	0	2	0	0	0
20	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0
23	7	1	0	0	1	0	0	0
24	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0
26	28	0	0	0	3	2	0	0
27	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0	0
36	0	0	0	0	0	0	0	0
37	2	0	0	0	1	0	0	0
38	0	0	0	0	0	0	0	0
39	3	0	0	0	1	0	0	0
40	3	0	0	1	0	0	0	1
41	0	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0	0
44	2	0	0	0	0	0	0	0
45	0	0	0	0	0	0	0	0
46	17	1	0	0	5	1	0	0
47	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0
49	0	0	0	0	0	0	0	0
50	0	0	0	0	0	0	0	0
51	0	0	0	0	0	0	0	0
52	0	0	0	0	0	0	0	0
53	2	0	0	0	0	0	0	0

54	2	0	0	0	1	0	0	0
55	3	0	0	0	2	0	0	0
56	2	0	0	0	1	0	0	1
57	3	0	0	0	0	0	0	0
58	10	0	0	0	2	0	1	0
59	4	0	0	0	0	0	0	0
60	0	0	0	0	0	0	0	0
61	27	1	0	1	4	0	0	1
62	1	0	0	0	0	0	0	0
63	0	0	0	0	0	0	0	0
64	3	0	0	0	0	0	0	0
65	14	0	0	0	4	0	0	0

C.2 jEdit Case Study Statistics

	CLASS	DECLARATION REGION			
		TP	TN	FP	FN
1	ConsoleInstall.java	11	7	0	0
2	ConsoleProgress.java	0	0	0	0
3	InstallThread.java	3	3	0	0
4	InvalidHeaderException.java	0	0	0	0
5	NonInteractiveInstall.java	7	6	0	0
6	OperatingSystem.java	9	5	0	0
7	Progress.java	0	0	0	0
8	SwingInstall.java	39	3	0	0
9	TarBuffer.java	4	0	0	0
10	TarEntry.java	7	6	0	0
11	TarHeader.java	9	0	0	0
12	TarInputStream.java	9	2	0	0
13	TarOutputStream.java	2	0	0	0
14	VariableGridLayout.java	12	7	0	0
15	Abbrevs.java	30	6	0	0
16	AbstractOptionPane.java	12	0	0	0
17	ActionContext.java	2	2	0	0
18	ActionListHandler.java	3	0	0	0
19	ActionSet.java	20	1	0	1
20	Autosave.java	1	0	0	0
21	BeanShellAction.java	2	0	0	0
22	Buffer.java	136	21	0	0
23	Debug.java	0	0	0	0
24	EBComponent.java	0	0	0	0
25	EBMessage.java	2	0	0	0
26	EBPlugin.java	0	0	0	0

27	EditAction.java	0	0	0	0
28	EditBus.java	3	0	0	0
29	EditPane.java	17	1	0	0
30	EditPlugin.java	5	0	0	0
31	GUIUtilities.java	47	11	0	1
32	JARClassLoader.java	13	3	0	0
33	Java14.java	16	3	0	0
34	Macros.java	17	5	0	1
35	Marker.java	0	0	0	0
36	MiscUtilities.java	54	17	0	0
37	Mode.java	5	5	0	0
38	ModeCatalogHandler.java	2	0	0	0
39	OperatingSystem.java	13	5	0	0
40	OptionGroup.java	3	0	0	0
41	OptionPane.java	0	0	0	0
42	PerspectiveManager.java	14	4	0	0
43	PluginJAR.java	64	21	0	0
44	PropertyManager.java	5	0	0	0
45	Registers.java	31	2	0	0
46	ServiceListHandler.java	4	0	0	0
47	ServiceManager.java	9	0	0	1
48	SettingsReloader.java	5	0	0	0
49	TextUtilities.java	24	5	0	0
50	BrowserCommandsMenu.java	20	1	0	0
51	BrowserIORequest.java	7	6	0	0
52	BrowserListener.java	0	0	0	0
53	FileCellRenderer.java	12	0	0	0
54	VFSDirectoryEntryTable.java	28	2	0	0
55	VFSDirectoryEntryTableModel.java	7	1	0	0
56	BufferChangeAdapter.java	0	0	0	0
57	BufferChangeListener.java	0	0	0	0
58	BufferIORequest.java	31	21	0	1
59	ContentManager.java	3	0	0	0
60	DummyFoldHandler.java	0	0	0	0
61	ExplicitFoldHandler.java	2	2	0	0
62	FoldHandler.java	5	0	0	0
63	IndentFoldHandler.java	3	0	0	2
64	LineElement.java	0	0	0	0
65	LineManager.java	7	0	0	0
66	PositionManager.java	2	2	0	0
67	RootElement.java	0	0	0	0
68	UndoManager.java	10	2	0	0
69	AbbrevEditor.java	9	1	0	0
70	AboutDialog.java	8	7	0	0
71	ActionBar.java	12	4	0	0
72	AddAbbrevDialog.java	6	0	0	0
73	AnimatedIcon.java	0	0	0	0
74	BeanShellErrorDialog.java	0	0	0	0
75	BufferOptions.java	16	2	0	0

76	BufferSwitcher.java	1	0	0	0
77	CloseDialog.java	15	0	0	0
78	ColorWellButton.java	5	0	0	0
79	CompleteWord.java	27	12	0	3
80	DefaultFocusComponent.java	0	0	0	0
81	DockableWindowContainer.java	0	0	0	0
82	DockableWindowManager.java	82	11	0	1
83	EditAbbrevDialog.java	5	0	0	0
84	EnhancedButton.java	0	0	0	0
85	EnhancedDialog.java	4	2	0	0
86	ErrorListCellRenderer.java	5	0	0	1
87	ErrorListDialog.java	9	2	0	0
88	FilesChangedDialog.java	20	7	0	0
89	FloatingWindowContainer.java	2	0	0	0
90	FontSelector.java	26	1	0	1
91	GrabKeyDialog.java	14	6	0	0
92	HistoryModel.java	10	2	0	0
93	HistoryTextField.java	13	1	0	0
94	InputHandler.java	9	4	0	0
95	IOProgressMonitor.java	4	0	0	0
96	JCheckBoxList.java	8	1	0	0
97	KeyEventTranslator.java	7	3	0	0
98	KeyEventWorkaround.java	2	0	0	0
99	LogViewer.java	7	2	0	0
100	OptionsDialog.java	24	6	0	4
101	PanelWindowContainer.java	51	8	0	0
102	PasteFromListDialog.java	10	1	0	0
103	RolloverButton.java	1	0	0	0
104	SelectLineRange.java	11	0	0	0
105	SplashScreen.java	5	0	0	0
106	StatusBar.java	33	11	0	1
107	TextAreaDialog.java	6	0	0	0
108	TipOfTheDay.java	6	3	0	0
109	ToolBarManager.java	0	0	0	0
110	VariableGridLayout.java	11	7	0	0
111	ViewRegisters.java	10	3	0	0
112	HelpSearchPanel.java	15	0	0	0
113	HelpTOCPanel.java	19	3	0	0
114	HelpViewer.java	8	0	0	2
115	FavoritesVFS.java	5	2	0	0
116	FileRootsVFS.java	5	0	0	2
117	FileVFS.java	23	8	0	0
118	UrlVFS.java	0	2	0	0
119	VFSManager.java	7	0	0	0
120	DirectoryProvider.java	8	6	0	0
121	DynamicMenuProvider.java	0	0	0	0
122	EnhancedCheckBoxMenuItem.java	5	0	0	0
123	EnhancedMenu.java	1	2	0	0
124	EnhancedMenuItem.java	4	1	0	0

125	FavoritesProvider.java	7	0	0	0
126	MacrosProvider.java	7	0	0	0
127	MarkersProvider.java	14	0	0	1
128	PluginsProvider.java	13	2	0	1
129	RecentDirectoriesProvider.java	12	0	0	0
130	RecentFilesProvider.java	12	1	0	0
131	BufferUpdate.java	0	0	0	0
132	DockableWindowUpdate.java	0	0	0	0
133	DynamicMenuChanged.java	0	0	0	0
134	EditorExiting.java	0	0	0	0
135	EditorExitRequested.java	0	0	0	0
136	EditorStarted.java	0	0	0	0
137	EditPaneUpdate.java	0	0	0	0
138	PluginUpdate.java	0	0	0	0
139	PropertiesChanged.java	0	0	0	0
140	SearchSettingsChanged.java	0	0	0	0
141	VFSUpdate.java	0	0	0	0
142	ViewUpdate.java	0	0	0	0
143	AbbrevsOptionPane.java	36	2	0	1
144	AppearanceOptionPane.java	5	1	0	0
145	BrowserColorsOptionPane.java	16	4	0	0
146	BrowserOptionPane.java	3	0	0	0
147	ContextOptionPane.java	31	2	0	0
148	DockingOptionPane.java	5	3	0	0
149	EditingOptionPane.java	8	0	0	3
150	GeneralOptionPane.java	5	0	0	0
151	GlobalOptions.java	2	0	0	0
152	MouseOptionPane.java	5	1	0	0
153	PluginManagerOptionPane.java	7	1	0	0
154	PluginOptions.java	4	1	0	0
155	ShortcutsOptionPane.java	16	3	0	0
156	ToolBarOptionPane.java	52	2	0	0
157	InstallPanel.java	33	11	0	0
158	ManagePanel.java	31	13	0	1
159	MirrorList.java	8	0	0	0
160	MirrorListHandler.java	2	0	0	0
161	PluginList.java	7	4	0	0
162	PluginListHandler.java	3	0	0	0
163	PluginManager.java	11	2	0	1
164	PluginManagerProgress.java	3	0	0	0
165	Roster.java	20	4	0	1
166	BufferPrintable.java	17	3	0	5
167	BufferPrinter1_3.java	13	0	0	2
168	BufferPrinter1_4.java	20	3	0	1
169	Handler.java	1	0	0	0
170	PluginResURLConnection.java	3	0	0	0
171	AllBufferSet.java	3	0	0	0
172	BoyerMooreSearchMatcher.java	12	2	0	0
173	BufferListSet.java	2	2	0	0

174	CurrentBufferSet.java	0	0	0	0
175	DirectoryListSet.java	3	0	0	0
176	HyperSearchRequest.java	15	3	0	0
177	HyperSearchResult.java	10	0	0	0
178	HyperSearchResults.java	18	7	0	0
179	RESearchMatcher.java	1	3	0	0
180	SearchDialog.java	33	0	0	1
181	SearchFileSet.java	0	0	0	0
182	SearchMatcher.java	0	0	0	0
183	Chunk.java	12	0	0	2
184	DefaultTokenHandler.java	0	0	0	0
185	DisplayTokenHandler.java	3	3	0	0
186	DummyTokenHandler.java	0	0	0	0
187	KeywordMap.java	2	2	0	0
188	ParserRule.java	0	0	0	0
189	SyntaxStyle.java	0	0	0	0
190	SyntaxUtilities.java	3	1	0	0
191	Token.java	0	0	0	0
192	TokenHandler.java	0	0	0	0
193	TokenMarker.java	18	6	0	0
194	XModeHandler.java	4	1	0	0
195	ChunkCache.java	27	1	0	1
196	ExtensionManager.java	14	0	0	0
197	Gutter.java	0	0	0	0
198	JEditTextArea.java	322	56	1	2
199	MouseActions.java	0	0	0	0
200	ScrollListener.java	0	0	0	0
201	Selection.java	32	11	0	0
202	StructureMatcher.java	4	8	0	1
203	TextAreaExtension.java	1	0	0	0
204	TextAreaPainter.java	35	16	0	2
205	CharIndexedSegment.java	0	0	0	0
206	IntegerArray.java	1	0	0	0
207	Log.java	6	1	0	0
208	ReadWriteLock.java	0	0	0	0
209	SegmentBuffer.java	1	0	0	0
210	WorkRequest.java	0	0	0	0
211	WorkThread.java	1	0	0	0
212	WorkThreadPool.java	4	1	0	0
213	WorkThreadProgressListener.java	0	0	0	0

	INITIALIZATION REGION				**FF REGION**			
	TP	TN	FP	FN	TP	TN	FP	FN
1	2	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	1	0	0	0	1	0	0	0

4	0	0	0	0	0	0	0	0
5	1	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	19	0	0	1	3	0	0	1
9	0	0	0	0	0	0	0	0
10	1	1	0	0	0	0	0	0
11	1	0	0	0	0	0	0	0
12	1	0	0	0	0	0	0	1
13	1	0	0	0	1	0	0	0
14	0	0	0	0	0	0	0	0
15	1	0	0	0	0	0	0	0
16	11	1	0	0	2	0	0	1
17	0	0	0	0	0	0	0	0
18	1	0	0	0	0	0	0	0
19	2	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0
21	2	0	0	0	0	0	0	0
22	40	5	0	0	5	0	0	0
23	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0
29	4	2	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0
31	11	1	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
33	4	0	0	0	2	0	0	0
34	2	0	0	0	3	0	0	0
35	0	0	0	0	0	0	0	0
36	0	0	0	0	1	1	0	0
37	0	0	0	0	0	0	0	0
38	1	0	0	0	0	0	0	0
39	0	0	0	0	0	0	0	0
40	2	0	0	0	0	0	0	0
41	0	0	0	0	0	0	0	0
42	2	0	0	0	2	1	0	0
43	13	0	0	0	5	3	0	0
44	0	0	0	0	0	1	0	0
45	7	0	0	0	1	2	0	0
46	2	0	0	0	0	0	0	0
47	2	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0
49	0	0	0	0	1	0	0	0
50	5	0	0	0	0	0	2	3
51	13	1	0	0	0	0	0	0
52	0	0	0	0	0	0	0	0
53	3	1	0	0	0	0	0	0
54	5	0	0	0	0	0	0	0

55	0	0	0	0	1	1	0	0
56	0	0	0	0	0	0	0	0
57	0	0	0	0	0	0	0	0
58	27	1	0	0	3	1	0	0
59	0	0	0	0	1	0	0	0
60	0	0	0	0	0	0	0	0
61	0	0	0	0	0	0	0	0
62	0	0	0	0	0	0	0	0
63	0	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0	0
65	0	0	0	0	1	0	1	0
66	0	0	0	0	0	0	0	0
67	0	0	0	0	0	0	0	0
68	0	0	0	0	0	0	0	0
69	6	2	0	0	0	1	0	0
70	7	0	0	0	1	0	0	0
71	4	3	0	0	4	0	0	0
72	4	0	0	0	2	0	0	1
73	3	0	0	0	0	0	0	0
74	0	0	0	0	0	0	0	0
75	5	3	0	0	0	1	0	3
76	2	0	0	0	0	0	0	0
77	4	0	0	2	0	1	0	1
78	6	1	0	0	1	0	0	1
79	7	1	0	0	1	0	0	0
80	0	0	0	0	0	0	0	0
81	0	0	0	0	0	0	0	0
82	15	1	0	1	5	2	0	0
83	3	0	0	1	2	0	0	1
84	2	0	0	0	0	0	0	0
85	2	0	0	0	1	0	0	0
86	2	1	0	0	0	0	0	0
87	6	0	0	1	1	0	0	1
88	7	0	0	2	2	0	0	2
89	2	0	0	0	1	0	0	0
90	5	1	0	3	2	0	0	2
91	11	0	0	0	3	0	0	1
92	4	0	0	0	1	0	0	0
93	8	0	0	0	1	0	0	1
94	4	0	0	0	0	0	0	0
95	10	0	0	0	1	0	0	1
96	2	1	0	0	0	0	0	0
97	0	0	0	0	0	0	0	0
98	0	0	0	0	0	0	0	0
99	8	0	0	0	1	0	0	1
100	10	0	0	3	2	0	0	1
101	19	3	0	0	4	2	0	1
102	8	0	0	2	1	0	0	1
103	2	0	0	0	0	0	0	0
104	5	2	0	2	1	0	0	2
105	1	2	0	1	0	0	0	0

106	24	5	0	2	3	3	0	0
107	2	0	0	1	0	0	0	1
108	5	1	0	1	0	0	0	2
109	0	0	0	0	3	0	0	0
110	0	0	0	0	0	0	0	0
111	5	2	0	1	0	0	0	2
112	3	1	0	0	0	1	0	0
113	11	0	0	0	2	0	0	0
114	11	0	0	0	2	0	0	1
115	2	0	0	0	1	0	0	0
116	0	0	0	0	1	0	0	0
117	1	0	0	0	2	0	0	0
118	0	0	0	0	0	0	0	0
119	1	0	0	0	1	0	0	0
120	3	1	0	0	0	0	0	0
121	0	0	0	0	0	0	0	0
122	3	0	0	0	0	0	0	0
123	2	0	0	0	0	0	0	0
124	2	0	0	0	0	0	0	0
125	2	0	0	0	0	0	0	0
126	1	0	0	0	0	0	0	0
127	3	0	0	0	0	0	0	0
128	1	0	0	0	0	0	0	0
129	1	0	0	1	0	0	0	0
130	1	0	0	0	0	0	0	0
131	0	0	0	0	0	0	0	0
132	0	0	0	0	0	0	0	0
133	0	0	0	0	0	0	0	0
134	0	0	0	0	0	0	0	0
135	0	0	0	0	0	0	0	0
136	0	0	0	0	0	0	0	0
137	0	0	0	0	0	0	0	0
138	0	0	0	0	0	0	0	0
139	0	0	0	0	0	0	0	0
140	0	0	0	0	0	0	0	0
141	0	0	0	0	0	0	0	0
142	0	0	0	0	0	0	0	0
143	12	0	0	1	0	0	0	3
144	6	0	0	0	2	0	1	0
145	12	0	0	0	1	0	0	2
146	3	0	0	0	1	0	0	0
147	18	0	0	1	0	1	0	3
148	4	0	0	0	0	0	0	0
149	16	0	0	3	1	0	0	1
150	4	0	0	1	0	0	0	1
151	1	0	0	0	2	0	0	0
152	2	0	0	0	1	0	0	0
153	10	0	1	0	0	0	0	1
154	1	0	0	0	0	0	0	0
155	7	0	0	0	2	0	0	0
156	36	0	0	3	0	1	0	3

157	18	0	0	0	1	0	0	0
158	12	0	0	0	1	0	0	0
159	2	0	0	0	0	0	0	0
160	1	0	0	0	0	0	0	0
161	3	0	0	0	0	0	0	0
162	3	0	0	0	0	0	0	0
163	5	0	0	1	4	0	0	0
164	6	0	0	0	1	0	0	0
165	4	0	0	0	1	0	0	0
166	5	2	0	0	0	0	0	0
167	3	0	0	2	0	0	0	0
168	3	0	0	0	0	0	0	0
169	0	0	0	0	0	0	0	0
170	0	0	0	0	0	0	0	0
171	0	0	0	0	0	0	0	0
172	0	0	0	0	0	0	0	0
173	0	0	0	0	0	0	0	0
174	0	0	0	0	0	0	0	0
175	0	0	0	0	0	0	0	0
176	4	0	1	0	0	0	0	0
177	0	0	0	0	0	0	0	0
178	15	0	0	1	2	0	0	1
179	0	0	0	0	0	0	0	0
180	35	4	0	4	4	0	0	10
181	0	0	0	0	0	0	0	0
182	0	0	0	0	0	0	0	0
183	3	0	0	0	0	0	0	0
184	0	0	0	0	0	0	0	0
185	0	0	0	0	0	0	0	0
186	0	0	0	0	0	0	0	0
187	2	0	0	0	0	0	0	0
188	0	0	0	0	0	0	0	0
189	0	0	0	0	0	0	0	0
190	0	0	0	0	0	0	0	0
191	0	0	0	0	0	0	0	0
192	0	0	0	0	0	0	0	0
193	4	0	0	0	0	0	0	0
194	4	0	0	1	0	0	0	0
195	4	0	0	0	0	0	0	0
196	0	0	0	0	0	0	0	0
197	0	0	0	0	0	0	0	0
198	43	1	0	0	9	2	0	0
199	0	0	0	0	0	0	0	0
200	0	0	0	0	0	0	0	0
201	0	0	0	0	1	1	0	0
202	1	0	0	0	2	0	0	0
203	0	0	0	0	0	0	0	0
204	16	0	0	1	1	0	0	0
205	0	0	0	0	0	0	0	0
206	0	0	0	0	0	0	0	0
207	0	0	0	0	0	0	0	0

208	0	0	0	0	0	0	0	0
209	0	0	0	0	0	0	0	0
210	0	0	0	0	0	0	0	0
211	1	0	0	0	0	0	0	0
212	1	0	0	0	0	0	0	0
213	0	0	0	0	0	0	0	0

		TRAILING DUPLICATE FRAGMENT				LOOP CONSTANT			
	Class	TP	TN	FP	FN	TP	TN	FP	FN
14	VariableGridLayout.java	0	0	0	0	0	0	2	0
59	ContentManager.java	1	0	0	0	0	0	0	0
101	PanelWindowContainer.java	1	0	0	0	0	0	0	0
106	StatusBar.java	1	0	0	0	0	0	0	0
110	VariableGridLayout.java	0	0	0	0	0	0	2	0