# CONVERTING CODE CLONES TO ASPECTS USING ALGORITHMIC APPROACH

by

**Angad Singh Gakhar,** B.Tech., Guru Gobind Singh Indraprastha University, 2009

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

**Master of Applied Science**

in

**Electrical and Computer Engineering**

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

The undersigned recommend to

the Faculty of Graduate and Postdoctoral Affairs

acceptance of the thesis

# CONVERTING CODE CLONES TO ASPECTS USING ALGORITHMIC APPROACH

submitted by

**Angad Singh Gakhar,** B.Tech., Guru Gobind Singh Indraprastha University, 2009

in partial fulfillment of the requirements for

the degree of Master of Applied Science in Electrical and Computer Engineering

_____

Chair, Howard Schwartz, Department of Systems and Computer Engineering

_____

Thesis Supervisor, Professor Samuel Ajila, Department of Systems and Computer Engineering

Carleton University

April 2012

## Abstract

The techniques of duplicating code are not only considered bad practices, they are also considered to be threats towards software maintenance. Nevertheless code clones have been found to be a common occurrence. While, there are tools that can detect code clones, the big question is what do we do with the clones? We can remove them manually – which is cumbersome and may not be effective because the clones may be needed for the software to function properly. A better solution is to apply the principle of modularity by using aspect oriented approach. In this work we present an algorithmic approach for converting code clones to aspects, and do aspect composition. We also carried out a performance analysis of the composed code. Our results show that the composed code performs as well as the original code (with clones) and even better.

## Acknowledgements

I would like to thank my supervisor Professor Samuel A. Ajila, for providing me with great support, wise guidance, patience and funding during the course of my studies and research. I would also like to thank the Faculty of Graduate and Postdoctoral affairs, and the Department of Systems and Computer Engineering for their funding during the course of my studies. I would like to thank Cistel Inc., Ottawa, Ontario for their support and the use of their source code to test the prototype and NSERC for financial support through the NSERC Engage grant number EGP 401451-10.

I would also like to thank my mother, my father, and my brother for their unconditional love, support and sacrifice without which this work would not have been completed.

I would also like to thank all my friends for their help and support.

## Table of Contents

## List of Figures

## List of Tables

## List of Appendices

# Chapter 1: Introduction

## 1.1 Introduction

Software engineers for a long time have handled the development of complex software systems by utilizing the principles of Separation of Concerns. While care is taken to separate the software system into smaller modules with minimal amount of overlapping functionalities between them, the complexity of the software systems is now increasing and so are the concerns/functionalities that span over multiple modules of the software system. These overlapping concerns are known as cross-cutting concerns, and have been found to have detrimental effects on software systems. With cross-cutting concerns, the code of a particular functionality is usually spread over multiple modules. This leads to problems in software maintainability, as well as making software comprehension more difficult.

The Aspect Oriented Programming (AOP) methodology explicitly provides the language support to modularize the design decisions that originally would have cross-cutting effects across a functionally decomposed program [Walker 1999]. It provides programmers the ability of expressing the design decision as a single coherent piece of code, instead of spreading it across multiple modules.

Code clones are duplicated code fragments, and are created using either exact replication, or a replication with certain modifications. These clones are harmful to the software systems as they not only cause code bloating, but also

increase the maintenance costs of the software system. They also increase the risk of making inconsistent changes to the code, thereby increasing the risks of faults in the software system. Code clones can be divided into four distinct types, with types 1-3 based on the textual similarity, while type 4 code clones are based on their similarity being functional in nature. Type 1 code clones are basically identical code fragments with a few variations in whitespaces, or comments etc., Type 2 code clones are structurally / syntactically identical code fragments, Type 3 code clones are duplicated modified fragments and are much more complex than Type 1 and Type 2 clones, while Type 4 code clones perform the same functionality but are syntactic variants of each other. Further description of code clones is provided in section 2.1 of this report. The approach we describe in chapter 3 of this report, as well as the implementation described in section 4.4 is able to handle all four code clone types as long as certain limitations / restrictions are followed. These limitations are described in section 5.2 of this report.

## 1.2 Thesis Motivation and Objectives

Code Fragments (CF's) are considered as clones of each other if there exists some kind of similarity between them [Roy 2009]. The process of duplicating and modifying code is known as Code Cloning [Krinke 2007]. These techniques of duplicating code are not only considered bad practices, they are also considered to be threats towards software maintenance. Nevertheless code clones have been found to be a common occurrence. Studies have shown that around 7% to 23% of the source code in a software system contains code clones

[Schulze 2010]. It is evident that there are tools that can detect code clones but the big question is what do we do with the clones after detection? We can remove them manually – which is cumbersome and may not be effective because the clones may be needed for the software to function properly. In addition, there is no way of removing all the clones especially those that are not method type clones. So, what can we do? We can apply the principle of modularity by using aspect oriented approach and thereby improving the maintainability as well as reusability of the code.

Apart from Aspect Oriented programming the process of code refactoring was also looked at. Code refactoring is the technique of restructuring an existing code by altering its internal structure without changing its external behavior. This is done by applying a series of tiny changes or “refactorings” in the source code while making sure not to modify the functional requirements of the code in question. Now while using Code Refactoring does improve the readability of the program code in question it does not work on the core problems being caused by code clones, namely issues like code bloating and increased risks of bugs and inconsistent behavior of the software system due to inconsistent changes made to code clones. Aspect Oriented Programming on the other hand provides us with the ability of handling those problems, as well as provides advantages like modularity in the source code, ease of maintainability of the software system, and an improvement in the performance of the software system.

The objective of this work then is to convert code clones to aspects, compose the aspect with the original code, and carry out a performance analysis

of the composed code. We undertook the following four steps to achieve our goal: Firstly, we use an existing code-clone detection tool to identify code clones in a source code. Secondly, we design algorithms to convert the code clones into aspects and do aspect composition with the original code. Thirdly, we implement a prototype that converts selected methods-type code clones to aspects and performs aspect composition. Fourthly, we carry out a performance analysis in order to make sure that the aspect composed code performs as well as the original code (with clones) and even better.

## 1.3 Contributions of the Thesis

The main contributions of this work are summarized as follows:

1 Design of four algorithms (File Loading, Aspects Import and Package, Aspect Composition, and File Composition) used for converting code clones present in Java source files to Aspects (see chapter 3 of this report).

2 Design and implementation of a prototype (CC2ASPECT) based on the four algorithms (see section 4.4 of this report).

3 A performance analysis (see section 4.5.3 of this report).

## 1.4 Thesis Outline

The rest of this document is structured as follows. Chapter 2 introduces Code Clones, the different types of Code Clones, Aspect Oriented Programming,

as well as topics related to code clone detection. Chapter 3 presents the four algorithms: File Loading, Aspect Import and Package, Aspect Composition, and File Composition used for converting code clones to aspects. Chapter 4 deals with an approach towards designing and implementation of the algorithms. It first introduces the Eclipse environment, the AspectJ language, as well as the CCFinderX code clone detection tool. The chapter then describes the architecture of our implementation, followed by the actual implementation. This chapter also describes the software's used to test the prototype, as well as the testing procedure and the results obtained after the testing. Chapter 5 summarizes the work done in this thesis, and presents the future work.

# Chapter 2: Literature Review

This chapter discusses code clones, types of code clones, aspect oriented programming, join points, aspect oriented models, code clone detection techniques, and steps involved in the code clone detection process.

## 2.1 Code Clones

According to J. Krinke, “code cloning is defined as the process of duplicating and modifying code, or creating replication of code fragments in the source code. The duplicate code in question is known as a code clone, while groups of code clones are known as clone groups. Clone groups generally consist of code clones that are also clones of each other” [Krinke 2007]. Figure 1 below shows a representation of code clones present in source text. The three parallel lines represent different source files in the project, while the box represents the duplicated code fragments. These techniques of duplicating code though considered bad practice, also known as threats towards software maintenance, are a common occurrence. Studies have shown that the occurrence of code clones is between 7 to 23% of the source code [Schulze 2010].

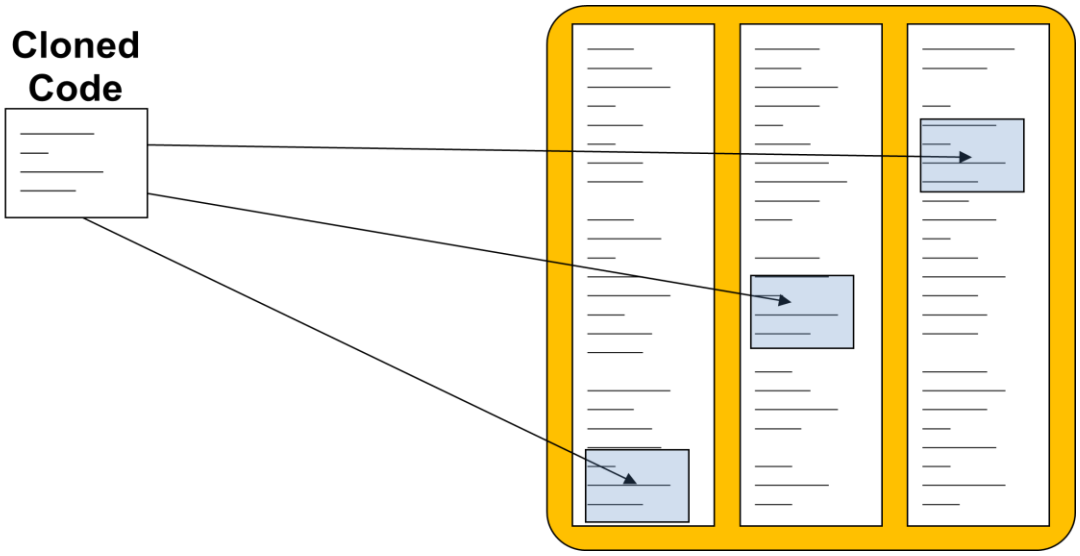


**Figure 1: Code Clone**

The creation of code clones, is considered harmful to the system because, it creates unnecessary duplicates, increases both the code size (code bloating), and the maintenance costs of the system in question. Although these duplicate clones themselves might not directly cause the faults, the inconsistent changes to the created clones often do cause faults in the code, thereby leading to inconsistent and incorrect behavior by the software [Juergens 2009]. Hence if a bug is detected in a code fragment, then all clones of that fragment need to be checked for the presence of the bug [Roy et al. 2009]. With code cloning, there is also a risk of bug propagation, i.e., if a code fragment contains a bug, and that fragment is cloned, then the new location might also contain the bug [Roy 2007].

C. K. Roy in his PhD thesis [Roy 2009] defined several terms that are frequently used while dealing with code clones. A few of them are given below.

### 2.1.1 Code Fragment

A Code Fragment (CF) is basically a sequence of code lines, and can have any granularity. It can be identified by the starting and the ending line numbers of that fragment, along with the file name in which the code fragment is present [Roy 2009].

### 2.1.2 Clone Pair / Group

Code Fragments (CF's) are considered clones of each other if there are similarities between them. This similarity is of two types, and will be explained later in section 2.1.3. When two Code Fragments are clones of each other, they are termed as a Clone Pair (CP). When multiple Code Fragments (CF's) are found to be clones of each other, they are termed as a Clone Group [Roy 2009].

### 2.1.3 Types of Code Clones

Code clones, based on similarity, can broadly be divided into two categories. In this case, some clones are similar to each other due to the textual similarity of their program code, while some other clones are similar to each other based on the similarity in their functionality [Roy et al. 2009].

Clones can further be subdivided into 4 Types, i.e., Type 1 through to Type 4. Clone Types 1 – 3, are clones whose similarity is of a textual type, while Type-4 clones are those that have similarity based on a functional type. [Roy et al. 2009],[Koschke 2007], and [Roy 2007].

**Type 1** clones are identical code fragments, i.e. they are exact copies, except for variations in whitespaces, comments, and textual

layout. Figure 2 below shows the different forms of Type 1 code clone. Figure 2(A) is the original code fragment. Figure 2(B) shows variations in whitespaces, i.e. an extra horizontal-tab space has been added to each line of code. Figure 2(C) shows variations in comments. Certain comments present in the original code fragment have been removed. Figure 2(D) shows the variations in the formatting of the code fragment. Here the opening brace has been moved from line ‘s3’ to line ‘s4’. On removal of the whitespaces and comments in Figure 2(B) and Figure 2(C), we receive code fragments which are similar to that present in Figure 2(A). While Figure 2(D) is not similar to Figure 2(A) on a line-by-line basis due to the repositioning of the “{“, and considering the internal working of the method as a whole, it is similar to the original code fragment.

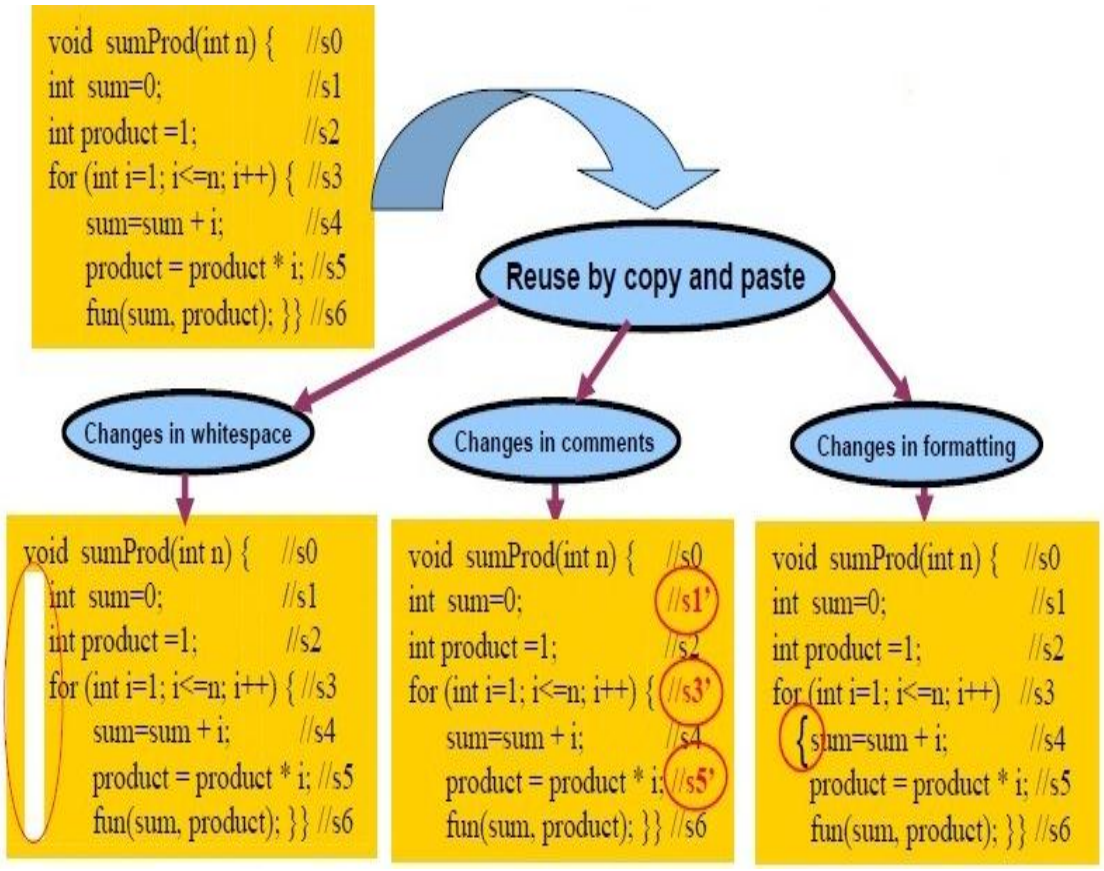


**Figure 2: Type 1 Code Clones (Adapted from [Roy et al. 2009])**

**Type 2** clones are structurally/syntactically identical code fragments. These clones have variations in identifiers, literals and type, in addition to variations in layout, whitespaces and comments. The keywords present in the code statements of these clones are usually the same in both fragments. Figure 3 below shows the different forms of Type 2 code clone. Figure 3(A) shows the original code fragment. Figure 3(B) shows the code fragment with its identifier names changed. In it, the method name “sumProd” was modified to “addTimes”, and the variable names “sum” and “product” were modified to “add” and “times” respectively. Similarly the Figure 3(C) shows the code fragment with variations in data

types and literals. The type “int” is modified to type “double”. Correspondingly, the values for variable “sum” is changed from “0” to “0.0”, and for the variable “product” from “1” to “1.0”. While the identifiers and/or the data types and literals may have been changed, the essential structure of the code fragment is similar to the original one due to placement of keywords, syntax etc.

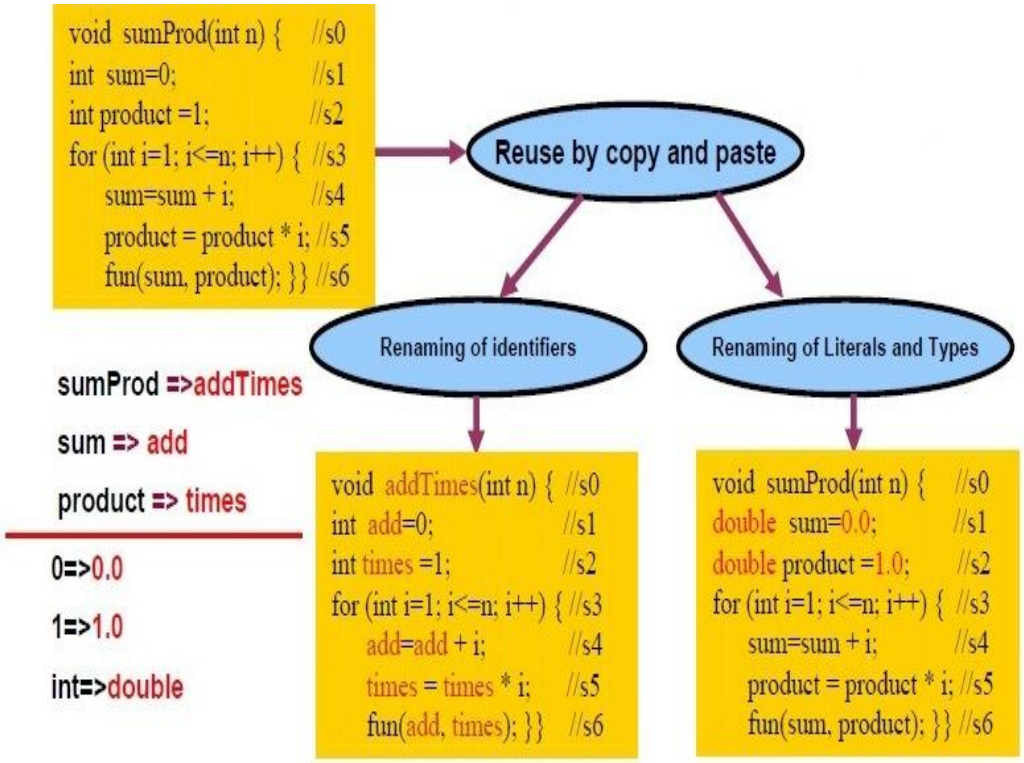


**Figure 3: Type 2 Code Clones (Adapted from [Roy et al. 2009])**

**Type 3** clones are code fragments containing further modifications than those present in Type 1 and Type 2 clones. Here code statements could have been changed, modified or removed, in addition to changes in whitespaces, layout, comments, types, identifiers and literals. Figure 4 below shows the different forms of Type 3 code clone. Figure 4(A) shows

the original code fragment. Figure 4(B) shows the code fragment with one of its lines modified. With this modification, the code (statement s4 has been modified) would only be executed for even values of the variable "i". Figure 4(C) shows the code fragment with the addition of a new line (statement s3b has been added), while the Figure 4(D) shows the code fragment with a line deleted (statement s5 has been deleted).

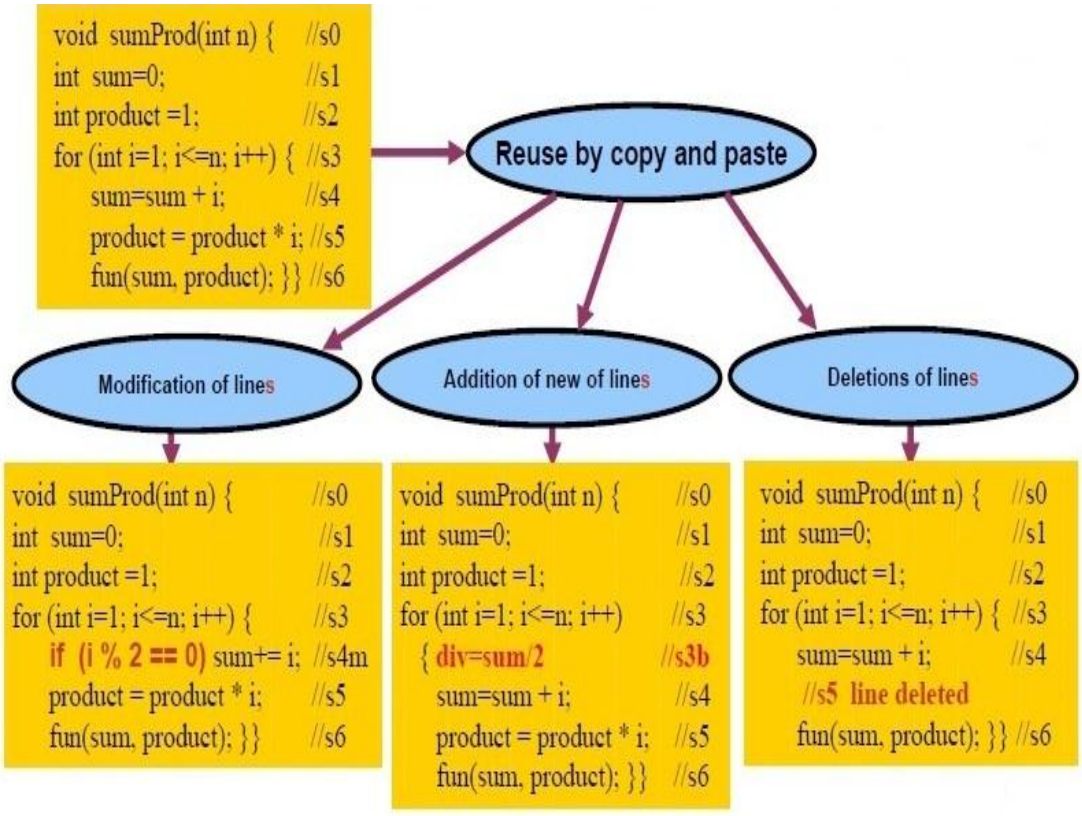


**Figure 4: Type 3 Code Clones (Adapted from [Roy et al. 2009])**

**Type 4** clones are code fragments which though perform the same functionality / computation, are implemented using different syntactic variants. Hence it can be said that there is a semantic similarity in these clones. Figure 5 below shows the different forms of Type 4 code clone.

Figure 5(A) shows the original code fragment. Figure 5(B) shows the code fragment with its statements reordered. The positions of statements ‘s2’ and ‘s1’ are exchanged. Figure 5(C) shows the code fragment with its control statements changed. It now contains a ‘while’ loop instead of the ‘for’ loop in the original code fragment. From a semantic point of view, both code fragments of Figure 5(B) and Figure 5(C) are similar in functionalities to the code fragment at Figure 5(A).

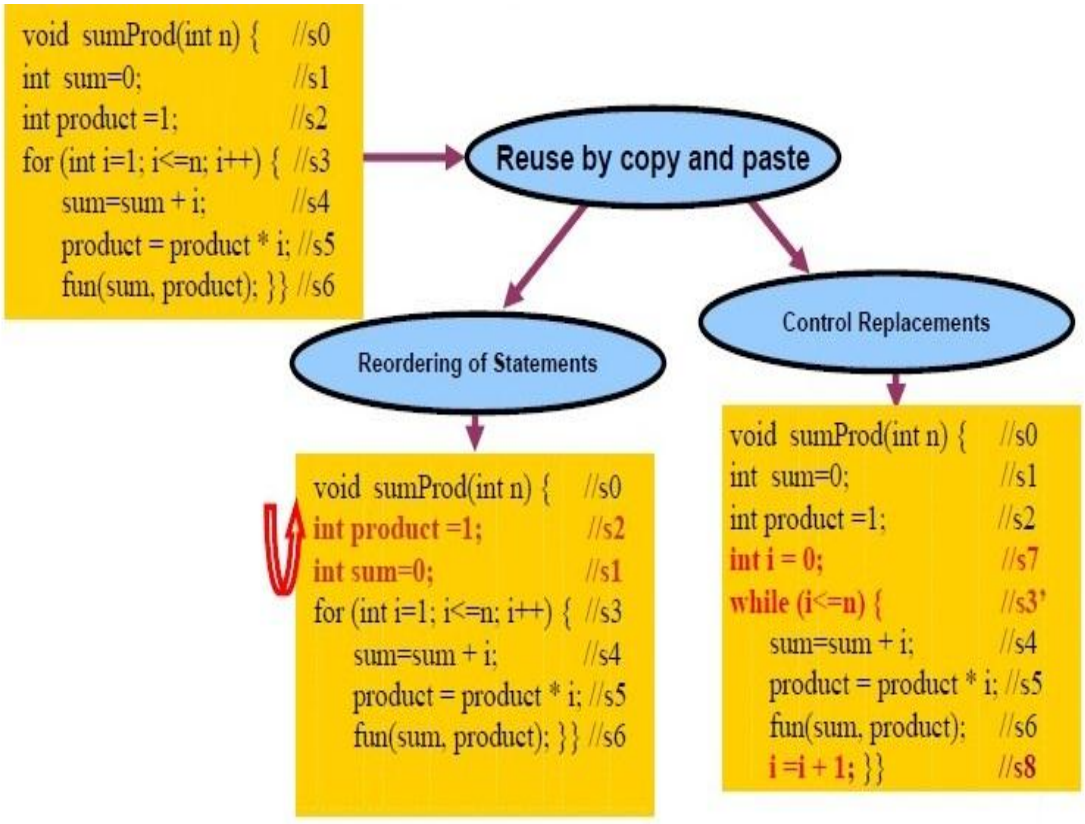


**Figure 5: Type 4 Code Clones (Adapted from [Roy et al. 2009])**

With Type 1 being the easiest, and Type 4 being the hardest, this particular division of clones into Types 1-4, is not only an indicator of an increasing level of complexity and sophistication of the clone, but is also an

indicator of the level of difficulty to identify and detect a particular clone [Roy 2007].

## 2.2 Clone Detection

Mayrand et.al [Mayrand 1996] defines Clone Detection as a “technique that finds functions which are exact copies or mutant copies of another function in the software system.” Roy et.al [Roy 2007] classified the different clone detection techniques into four main categories based on the different levels of analysis of the source code. These techniques are: *textual, lexical, syntactic, and semantic*.

### 2.2.1 Textual Approaches

These techniques, also known as *text-based techniques* use little to no transformation on the source code before the comparison process. Usually they use the source code in its original form. This approach first hashed the code fragments of a fixed number of lines. Then a sliding window technique is used in combination with an incremental hash function to identify sequences having similar hash values [Johnson 1993] [Johnson 1994].

### 2.2.2 Lexical Approaches

These techniques are also known as *token-based approaches*. First, the source code is transformed into a sequence of lexical tokens. The sequence is then scanned for the presence of duplicated sub sequences. Once found, the original source code of the of the corresponding sub sequences is returned.

Efficient token-based detection was pioneered by Brenda Baker in her detection tool called *Dup* [Baker 1992] [Baker 1995]. In it, a lexical analyzer is used to divide source code lines into tokens. These tokens are then divided into parameter and non-parameter tokens. A "hashing *functor*" is used to summarize the non-parameter tokens of a line, while parameter-tokens are encoded using a position index. A suffix tree is used to represent the prefixes of the resulting sequence of symbols. The idea behind this was that if two suffixes have a common prefix, then it can be considered that the prefix occurs more than once and hence can be considered a clone. This could be used to identify Type-1 and Type-2 clones, while Type-3 clones could be found by concatenating Type-1 and the Type-2 clones.

### 2.2.3 Syntactic Approaches

In these approaches, a parser is used to convert the source code into either Abstract Syntax Trees (ASTs), or Parse Trees. Clones are then found using either tree-matching or structural metrics.

#### *2.2.3.1 Tree-matching Approach*:

These approaches find clones by identifying similar sub trees. Baxter et.al pioneered tree-matching clone detection techniques in his tool called CloneDr [Baxter 1998].

#### *2.2.3.2 Metrics-based Approach*:

These approaches gather numerous metrics of code fragments, and then compare the metric vectors and not the Abstract Syntax Trees or code. One of

the techniques uses “fingerprinting functions” [Roy 2007]. Here the metrics calculated for syntactic units like classes, methods, etc., are compared to find the clones. Usually, the code is first parsed to generate an Abstract Syntax Tree (AST) or a Control Flow Graph (CFG), then the metrics are calculated [Roy 2007].

### 2.2.4 Semantic Approaches

In Semantic Approaches, the source code is represented as a Program Dependency Graph (PDG) [Roy 2007]. The edges of the graph represent control and data dependencies, while the nodes represent the statements and expressions. Clones are then found by searching for isomorphic sub graphs.

### 2.2.5 Clone Detection Process

Roy et.al [Roy 2007] provides a summary of the basic steps that a clone detector may have (see Figure 6 below).

#### *2.2.5.1 Preprocessing*

In this stage, the source code is partitioned and the domain of the comparison is determined. Next, all the uninteresting source code is removed. Next, the remaining source code is divided into source units, which are basically sets of disjoint fragments. These units can have any level of granularity, e.g. classes, files, etc. In addition, depending on the comparison technique being used, some source units might need further partitioning into lines or tokens. These partitioning are known as comparison units.

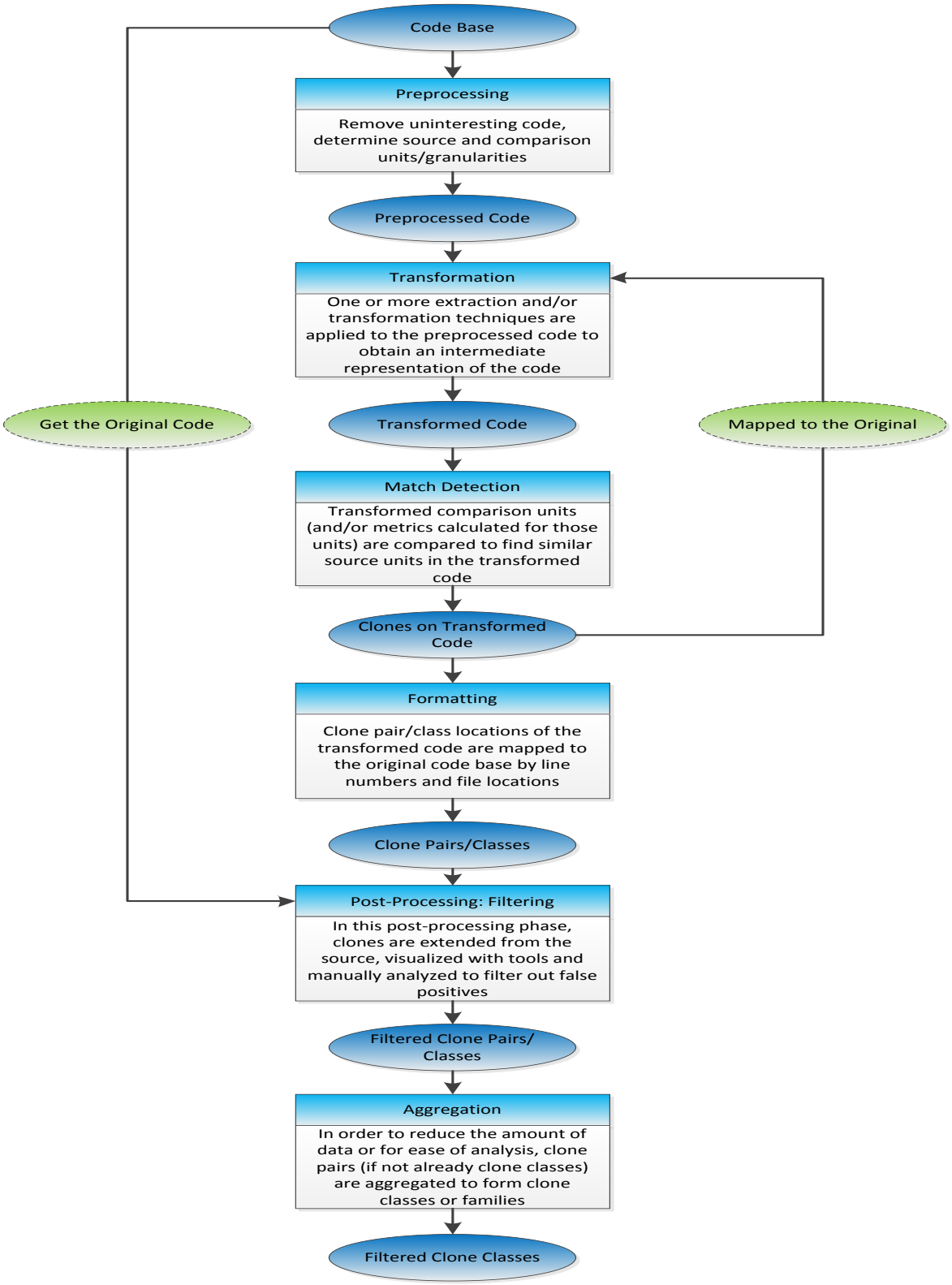


**Figure 6: A Generic Clone Detection Process [Roy 2007]**

#### *2.2.5.2 Transformation*

Once the comparison units have been found, if the comparison technique is not textual, then extraction is done to the comparison units, i.e. their source codes are transformed to an intermediate representation for comparison. Some tools also require additional normalization after the extraction process to detect clones.

Normalization is basically an optional step, used to eliminate any superficial differences between clones. It basically includes the removal of whitespace, removal of comments, identifier normalization, structural transformations of the code, and pretty-printing of the source code.

#### *2.2.5.3 Match Detection*

In this stage, the transformed code from the previous stage is fed to a comparison algorithm. Here the different comparison units are compared with each other. Many times, adjacent comparison units are joined to form larger comparison units. The output of this step is a list of matches in the transformed code, which is then aggregated to form candidate clone pairs.

#### *2.2.5.4 Formatting*

Here, the transformed code clone pair list formed in the previous step is converted to its corresponding original source code list. The clone pair coordinates that were found in the previous step are also mapped to the original source code files.

#### *2.2.5.5 Post-processing: Filtering*

In this stage, clones are filtered and ranked using either manual analysis or automated heuristics. In the case of manual analysis, clones are subjected to manual analysis, where false positive clones are removed. In case of automated heuristics, the heuristics can be based on characteristics like length, frequency, etc.

#### *2.2.5.6 Aggregation*

This stage is used to reduce the amount of data received, or for ease of analysis of the data, the clone pairs are often aggregated to create clone classes.

## 2.3 Aspect Oriented Programming (AOP)

According to Walker et.al, “Aspect Oriented Programming (AOP) is a new programming technique that takes another step towards increasing the kinds of design concerns which can be captured cleanly within the source code” [Walker 1999]. Software engineers have since managed the process of developing complex software systems using the principle of Separation of Concerns (SOC). SOC is the process of separating software into distinct features, with as little overlapping in functionality as possible, i.e., each piece of functionality is implemented in its own distinct module. Aspect Oriented Programming explicitly provides the language support to modularize the design decisions that originally would have been cross-cutting a functionally decomposed program [Walker 1999].

Due to the increase in the complexity of software systems, certain concerns (functionalities) span over multiple modules in the software system architecture. Cross-cutting concerns, therefore, are defined as those concerns which affect multiple system functions and features [Albunni 2008] [Eaddy 2007]. These cross-cutting concerns can have detrimental effects on software systems, like, 1) make software comprehension more difficult, since the programmer would have to keep multiple concerns in mind while inspecting certain sections of code, and 2) make software maintainability more difficult, since the code of a particular functionality would be spread in multiple places [Albunni 2008]. For example, when a concern needs to be modified, a developer usually has to localize the code that implements the functionality. With a cross-cutting concern, this would require the developer to inspect multiple modules, as the code might be scattered across a number of modules. [Eaddy 2008]

Aspect Oriented Programming gives programmers the ability to express a decision in a separate and coherent piece of code, rather than spreading the code for that decision through multiple modules. For example, if required that a particular set of operations did not occur concurrently, a programmer would have to spread the code of those operations through different source files, but, an aspect-oriented approach would allow the synchronization constraint to be specified in a separate piece of code. The aspect code would then later be combined with the primary program code by an Aspect Weaver [Walker 1999].

An aspect-oriented approach makes reasoning about, developing and maintaining certain kinds of code possible, where it was earlier difficult to cleanly

capture design decisions to actual code, easier [Walker 1999] [Kiczales 1997]. Walker et.al performed experiments to test the above mentioned claim, using AspectJ. The first experiment was to measure the ease of debugging between programming in Java and AspectJ. The intention was to investigate whether programmers would find and fix faults in multithreaded programs while working with Aspect Oriented Programming. They paired programmers into groups, and three synchronization errors were introduced to a digital library code. In each pair, one programmer had control of the system with the problem, while the other had access to the reports describing the symptoms and other online documentation. The participants were told to fix the faults in sequential order, though the faults themselves were cascading, and each fault hid the symptoms of the next one. It was found that the AspectJ teams fixed the first faults faster than the Java teams, while the difference was less in the case of the second and third faults. It was also found that AspectJ pairs used fewer switches between the files they were examining, and that both AspectJ, and Java pairs spent equal time to weave and execute their programs, i.e. the additional weaving time was negligible. These results are graphically shown in Figure 7 below. Figure 7A shows the time taken by the groups to correct each fault. Figure 7B shows the number of times each group switched between the different files of the code. Figure 7C shows the number of instances of semantic analysis by each group.

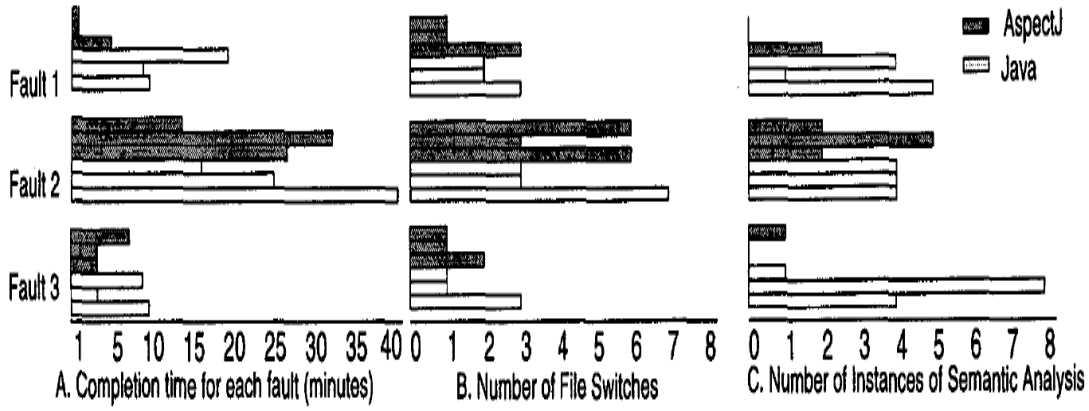


**Figure 7: Debugging Results for the "Ease of Debugging" experiment by Walker et.al [Walker 1999]**

### 2.3.1 Join Point Model

"Underlying any aspect-oriented approach is something called a join-point model. This defines a series of events (Join Points), visible to an aspect during the program execution. Aspects specify which of these events they are interested in through a Pointcut" [EclipseAspectJ]. There are three components to any aspect – Join Point, Pointcut, and the Advice. When a program is executed, certain events occur. These events are what would be considered as Join Points. Pointcuts are rules used to select Join Points. During program execution, pointcuts are used as filters to identify and separate those Join Points that the developer is interested in. The Advice section of the aspect is where the developer specifies what action is to be taken at the Join Point selected by a particular Pointcut. Further description of Join Point's, Pointcut's and Advice is provided in Section 4.2 of this report.

### 2.3.2 Aspect Oriented Programming Model

Wand et.al [Wand 2004], described a conceptual model of aspect-oriented programming which contained dynamic Join Points. In this model, the system contained a base program, along with pieces of Advice. The program would be executed using an interpreter. During the execution when the interpreter reaches certain Join Points, the aspect weaver is invoked. Each piece of Advice contains what is called a Point Cut Designator (PCD). The Point Cut Designator is basically a formula which specifies the set of Join Points to which a piece of Advice is applicable, along with a body containing the actions it intends to perform at that Join Point. It is the job of the aspect weaver to distribute the Join Point under consideration, to its proper Advice, and then execute the body of that Advice using the interpreter [Wand 2004]. Further description of Join Point's, Point Cut Designator's and Advice is provided in Section 4.2 of this report.

## Chapter 3: Code Clone to Aspect

The aim of this research work is to remove code clones from source files, convert these code clones to aspects, and then compose the aspects, back into the source code using an algorithmic approach. The target was achieved using four main algorithms which are presented in this chapter.

These algorithms work under the assumption that the clones being submitted for conversion are java methods. These methods either should be self-contained, or if they are using any variables/methods from a different class, then the object of that class must have been declared inside this method. The final output received after the execution of these algorithms will not work if the method is referring to any variables declared outside of its body.

Figure 8 describes our work in a bird's eye view. The target source code is run through the code clone detection software to obtain the line numbers of the code clones in the source code. Both the target source code and the line numbers of the clones are sent as input to the CC2ASPECT software (see section 4.4). The CC2ASPECT applies the implementation of our algorithms to the input and produces a target source code, whose code clones have been removed by commenting them out, and a file which contains those clones in the form of aspects.

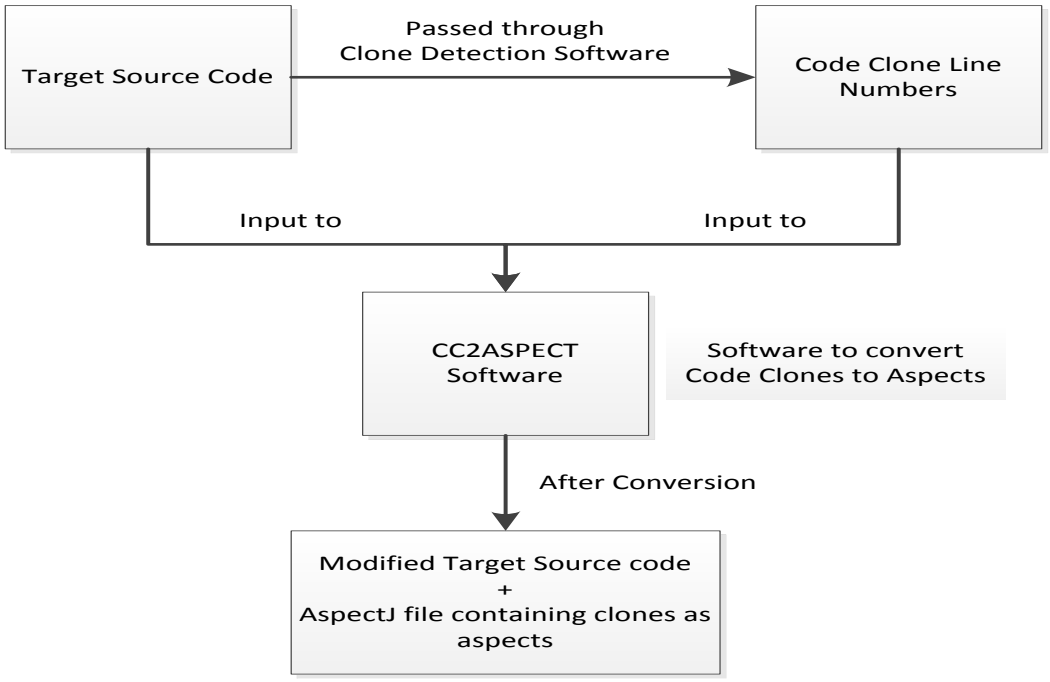


**Figure 8: Bird's Eye view of the research work**

## 3.1 File Loading Algorithm

This section describes the File Loading Algorithm. The file names and the line numbers of the clones have to be entered by the user.

The different variables used in this algorithm are:-

'n' → variable denoting the total number of clones to be removed

'name[ ]' → Array to store the file paths entered by the user.

'full_File[ ][ ]' → A 2-dimensional array to store the code present in the distinct files.

'startNo_Clone$_i$' → Variable to store the starting line number of clone$_i$, i={1,..,n}.

'endNo_Clone$_i$' → Variable to store the ending line number of clone$_i$, i={1,..,n}

'fileLength[ ]' → Array to store the file lengths of distinct files.

'finalName[ ]' → Array to store the file paths of the distinct files.

**Algorithm-1** File Loading Algorithm

```
1)Begin
2)      //Get all filenames
3)      Initialization: name[i]:= file path of clone_i, i={1,..,n}
4)      Initialization: finalName[i]:="abcdTestFile", i={1,..,n}
5)      Initialization: i:=1
6)      while(i<=n)
7)              copy code clone to corresponding text area
8)              i++
9)      end while
10)     Initialization: cloneNumber:=1
11)     while(cloneNumber<=n)
12)             if(filename is distinct) then
14)                     copy entire file to array full_File[cloneNumber][ ]
15)                     copy file path to array finalName[cloneNumber]
16)                     copy file length to array finalLength[cloneNumber]
17)             end if
18)             cloneNumber++
19)     end while
20)end begin
```

The algorithm begins by initializing the array name[ ], with the file paths of the files containing the code clones, in line 3 of the pseudo code. It then pre initializes the array finalName[ ] with the string "abcdTestFile" in line 4. This array is used to store the file paths of all the distinct files which have been entered by

the user, and the assumption taken here is that no file will be named “abcdTestFile”. Later this is used to identify the storage locations of the data of the distinct files, so that the algorithm can know where to make changes.

Each clone has its own text area so that the user is able to visually verify that the correct line numbers had been entered. Lines 6-9 of the pseudo code load the code clone lines to their respective text areas. The algorithm next, in lines 10-19, starts to save the information about the distinct files that had been entered by the user. It first checks if the file in question is distinct or not. This is done by comparing the file paths. If $file_i$ was found to be distinct, then the data in that file is stored in the array full_File[i][ ], then its file path and its length are stored in the arrays finalName[i] and finalLength[i], i={1,..,n}. This was done so that we would be able to identify the storage locations of the data of a particular file, by comparing its file path with those stored in the array finalName[ ].

The complexity of this algorithm is found to be 2n (i.e. lines 6 to 9 is n, and lines 11 to 19 is n). Therefore the algorithm is linear O(n).

## 3.2 Aspect Import and Package Algorithm

We are dealing with Java source files. These files for ease of execution import other files and packages. This section describes the algorithm that adds the required lines of code, which import other files and packages, to the aspect.

The different variables used in this algorithm are:-

'n' → variable denoting the total number of clones to be removed

'finalName[ ]' → Array to store the file paths of the distinct files.

'full_File[ ][ ]' → A 2-dimensional array to store the code present in the distinct files.

'line' → variable denoting the line number under consideration

**Algorithm-2** Aspects Import and Package Algorithm

```
1)Begin
2)      Initialization: i:=1
3)      while(i<=n)
4)              if(finalName[i]!=”abcdTestFile”) then
5)                      Initialization: line:=1
6)                      while(line<=30)
7)                              if((full_File[i][line] contains “package”)||
                                (full_File[i][line] contains “import”)) then
8)                                      copy line to aspectJ text area
9)                              end if
10)                             line++
11)                     end while
13)             end if
14)     i++
15)     end while
16)end begin
```

Firstly, in line 4, the algorithm tries to identify the storage locations of the distinct files. This is done by checking whether the string stored in any particular location of the array finalName[ ], equals the text “abcdTestFile” or not. Since all locations of the array finalName[ ] were pre-initialized with the text “abcdTestFile”, then if the string at any location does not match “abcdTestFile”, it means that that location was modified in the File Loading algorithm, to contain

the file path to a distinct file, and that its corresponding location in the array full_File[ ][ ] contains the data of that distinct file.

Once the file storage location in the array full_File[ ][ ] is found, then the top 30 lines of that file are read. This number of 30 lines was chosen after going through the sample projects available to us and was found to be more than sufficient for the number of lines code importing files and packages in our sample projects source files. If the line under consideration contains keywords like “package” or “import”, then that particular line is copied to the aspect text area. This text area is where the aspect file is being created. This occurs between lines 4-13 of the algorithm.

Due to the presence of a nested while loop in line 6 of the algorithm, the complexity of the Aspects Import and Package Algorithm was found to be $n^2$. Therefore the algorithm is quadratic $O(n^2)$.

## 3.3 Aspect Composition Algorithm

This section contains the algorithm that actually creates the pointcuts, and the advice bodies that are required by the aspect file. It also removes the code clones from the original file.

The different variables used in this algorithm are:-

‘n’ → variable denoting the total number of clones to be removed

'full_File[ ][ ]' → A 2-dimensional array to store the code present in the distinct files.

'startNo_Clone$_i$' → Variable to store the starting line number of clone$_i$, i={1,..,n}.

'endNo_Clone$_i$' → Variable to store the ending line number of clone$_i$, i={1,..,n}

'line' → variable denoting the line number under consideration

**Algorithm-3** Aspect Composition Algorithm

```
1)Begin
2)      Initialization: i:=1
3)      while(i<=n)
4)              if(clone_i is distinct from any previously processed clone) then
5)                      if(startNo_Clone_i has method parameters) then
6)                              compose advice specification with parameter binding
7)                      else
8)                              compose advice specification without parameter
                                binding
9)                      end if
10)                     //create advice body
11)                     Initialization: line:= startNo_Clone_i + 1
12)                     while(line<= endNo_Clone_i)
13)                             copy line to Aspect text area
14)                             line++
15)                     end while
16)             end if
17)             i++
18)     end while
19)     //comment out clones
20)     Initialization: i=1
21)     while(i<=n)
22)             comment out clone_i from array full_File[ ][ ]
23)             i++
24)     end while
```

25)**end begin**

In this algorithm firstly, with the aid of the while loop starting in line 3, the algorithm checks whether the code clone under consideration is distinct from any previously processed clones. This is done to try and prevent duplicate aspects from being created.

If the clone was not distinct, then nothing is done to it, and the processing is moved to the next clone. But if the clone was found to be distinct, then its advice declaration containing an anonymous pointcut, and corresponding advice body is created. Upon finding a distinct clone, a check is done in line 5 of the algorithm, to identify if the clone starting line has any method parameters or not. This is done so that the algorithm can decide whether it needs to create advice with parameter binding or not. In AspectJ, if we create an advice specification for a method, and if that method in the original code has method parameters, then those original method parameters have to be bound to the aspect parameters for use.

If the method header in the clone starting line has method parameters, then line 6 of the algorithm composes the advice specification with proper parameter binding. If the method header does not contain any method parameters, then line 8 of the algorithm composes the advice specification without any parameter binding. Lines 11-15 are then used to create the body of the advice by copying the required lines from the code clone to the aspect text area.

Once all the advice specifications and their corresponding advice bodies are created, lines 20-24 are used to comment out all the code clones. Since the complete original code is stored in the array full_File[ ][ ], all code clone sections are commented out in this array. Later the entire contents of the original file would be overwritten with the now modified contents present in the array full_File[ ][ ]. This commenting out of the original code clone is done so as to reduce the number of lines of code which would now be read by the interpreter.

Due to the presence of a nested while loops in line 12 of the algorithm, the complexity of the Aspect Composition Algorithm was found to be $n^2$. Therefore the algorithm is quadratic $O(n^2)$, (i.e. $n^2+n$).

## 3.4 File Composition Algorithm

This section contains the algorithm that saves the newly created aspect file as well as the modifications done on the original source code. It begins by making the user select the destination file in which to store the aspect. This is done in line 2 of the algorithm. While the user does have the option of selecting any type of file (example a notepad file), and not necessarily only aspect files (*.aj), it would be beneficial to the user to select an AspectJ file because the running environment (eclipse) can go towards execution of the new aspect without it having to be later copied into an AspectJ file for execution.

**Algorithm-4** File Composition Algorithm

```
1)Begin
2)      choose target file using JFileChooser
3)      //copy text from Aspect text area to selected file
4)      open target file
5)      copy text in Aspect text area to target file
6)      close target file
7)      //Save modifications to original files containing the clones
8)      Initialization: i:=1
9)      while(i<=n)
10)             //find location of modified file data
11)             if(finalName[i]!=”abcdTestFile”) then
12)                     open file whose file path is stored in finalName[i]
13)                     overwrite original code at file location of file path stored in
                        finalName[i] with modified code at full_File[i][ ]
14)                     close file whose file path is stored in finalName[i]
15)             end if
16)             i++
17)     end while
18)end begin
```

Once this is done, in lines 4-6 the target file is opened, then the composed aspect file in the aspect text area is copied to it, and finally the file is closed, thereby saving the aspect in the target file.

Next in lines 8-17 of the algorithm, the locations of the distinct filenames are found, and then those files are overwritten by a modified copy stored in the array full_File[ ][ ]. The actual identification is done in line 11 of the algorithm. The array finalNane[ ] was pre-initialized with the text string “acdTestFile” earlier in the File Loading Algorithm. Now after the File Loading Algorithm, any location in the array finalName[ ] not containing this text string will contain the file path of a distinct file. Also, the corresponding location in the array full_File[ ][ ] contains the modified code (code with the code clones commented out), for this particular file

path. Once the file location of the distinct file has been found, lines 12-14 of the pseudo code open the file whose file path was stored in the location of the array finalName[ ], and overwrite the original code of that file with the modified code stored in corresponding locations in the array full_File[ ][ ], and finally close that file, thereby saving the modifications.

The time complexity of the File Composition Algorithm was found to be linear, i.e. O(n).

## 3.5 Conclusion

This chapter describes in detail the four algorithms created to convert code clones into Aspects. The implementation of these algorithms in our CC2ASPECT prototype is described later in section 4.4 of this report. The flowcharts for these algorithms are provided in Appendix A of this report.

# Chapter 4: Design and Implementation

This chapter starts by describing our implementation environments (i.e. Eclipse, AspectJ, and CCFinderX). Then we present the architecture of our prototype based on the algorithms in Chapter 3. We also describe the user interface, verification and validation of the prototype, and some performance analysis.

## 4.1 Eclipse

“Eclipse is an open source software development platform that provides users with the necessary functionality to develop a wide range of applications”. [EclipseAspectJ]

The Eclipse project basically has 3 components – the Eclipse platform, the JDT (Java Development Kit), and the PDE (Plug-in Development Environment). Both the JDT and the PDE are plug-in’s to the eclipse platform itself. Taken together, these three components create the Eclipse SDK (Software Development Kit), which basically is “a complete development environment for eclipse based tools and for the development of eclipse itself.” [Eclipse.org].

“The Eclipse platform itself is a sort of universal tool platform – it is an IDE for anything and nothing in particular” [Eclipse.org]. The platform can handle all types of files, for example, Java files, C files, etc. on its own, the platform does not have the necessary knowledge of how to work with the different file types. It

is the different eclipse plug-in's that inform the platform about what can be done regarding a particular file type.

## 4.2 AspectJ

AspectJ is basically a language that extends the principles of Aspect Oriented Programming (AOP) to Java [AspectJ]. Kiczales et. al. [Kiczales 2001], describes AspectJ as a general purpose language which was designed to be a compatible extension to Java, so as to aid the current Java practitioners. Kiczales stated that this compatibility was of 4 types – Upward compatibility, Platform compatibility, Tool compatibility, and Programmer Compatibility.

AspectJ supports two different kinds of crosscutting concerns – Dynamic Crosscutting, and Static Crosscutting. Dynamic Crosscutting defines additional implementation/behavior to run at certain well defined points of the program code execution. Static Crosscutting modifies the static structure of the program, i.e. add new methods, modify class hierarchy, implement new interfaces, etc. [Rodriguez 2004] [Kiczales 2001]. Using our conversion algorithms (described in chapter 3), we modify the target software such that it implements Dynamic Crosscutting concerns at runtime.

**4.2.1 Join points** – Adrian Colyer in his book [EclipseAspectJ] defines Join Points as events in the control flow of a program. Kiczales defines join points as well defined points of execution in the program [Kiczales 2001] [EclipseAspectJ] [Kiczales G 2001] [Lopez-Herrejon 2006] [AspectJGuide].

Table 1 describes the different dynamic Join Points provided by AspectJ [Kiczales 2001].

**Table 1: Some dynamic join points present in AspectJ [Kiczales 2001]**

| *kind of join point* | *points in the program execution at which…* |
|---|---|
| method call<br>constructor call* | a method (or a constructor of a class) is called. Call join points are in the calling object, or in no object if the call is from a static method. |
| method call reception<br>constructor call reception | an object receives a method or constructor call. Reception join points are before method or constructor dispatch, i.e. they happen inside the called object, at a point in the control flow after control has been transferred to the called object, but before any particular method/constructor has been called. |
| method execution*<br>constructor execution* | an individual method or constructor is invoked. |
| field get | a field of an object, class or interface is read. |
| field set | a field of an object or class is set. |
| exception handler execution* | an exception handler is invoked. |
| class initialization* | static initializers for a class, if any, are run. |
| object initialization* | when the dynamic initializers for a class, if any, are run during object creation. |

**4.2.2 Pointcuts** – Kiczales et.al. [Kiczales 2001] describes Pointcuts as a set of Join Points, or as a means of referring to collections of Join Points. AspectJ contains within it a number of primitive Pointcut designators, which are then used to match the required Join Points at runtime. A simplified way to think of a Pointcut would be in the case of a filter, one which filters through only Join Points containing certain required features out of all the Join Points in the code [Kiczales G 2001] [Lopez-Herrejon 2006] [AspectJGuide].

Pointcuts could be both primitive as well as user defined in nature. The Table 2 below shows some of the primitive Pointcut designators contained within AspectJ [Kiczales 2001].

**Table 2: Some primitive Pointcut designators [Kiczales 2001]**

| |
|---|
| `calls(signature)`<br>`receptions(signature)`<br>`executions(signature)`<br>Matches call/reception/execution join points at which the method or constructor called matches *signature*. The syntax of a method signature is:<br>`ResultTypeName RecvrTypeName.meth_id(ParamTypeName, ...)`<br>The syntax of a constructor signature is:<br>`NewObjectTypeName.new(ParamTypeName, ...)` |
| `gets(signature)`<br>`gets(signature)[val]`<br>`sets(signature)`<br>`sets(signature)[oldVal]`<br>`sets(signature)[oldVal][newVal]`<br>Matches field get/set join points at which the field accessed matches the signature. The syntax of a field signature is:<br>`FieldTypeName ObjectTypeName.field_id` |
| `handles(ThrowableTypeName)`<br>Matches exception handler execution join points of the specified type. |
| `instanceof(CurrentlyExecutingObjectTypeName)`<br>`within(ClassName)`<br>`withincode(signature)`<br>Matches join points of any kind at which the currently executing:<br>- object is of type `CurrentlyExecutingObjectTypeName`<br>- code is contained within `ClassName`<br>- code is contained within the member defined by the method or constructor `signature` |
| `cflow(pointcut_designator)`<br>Matches join points of any kind that occur strictly within the dynamic extent of any join point matched by pointcut_designator. |
| `callto(pointcut_designator)`<br>Matches method call join points that in one step lead to any reception or execution join points matched by pointcut_designator. |
| `staticinitializations(TypeName)`<br>`initializations(TypeName)`<br>Matches class or object initializations of the specified type. |

**4.2.3 Advice** – Kiczales et.al. [Kiczales 2001] [Kiczales G 2001] [Lopez-Herrejon 2006] [AspectJGuide] defines Advice as a method-like mechanism, which is used to identify the code to be executed at the Join Point selected by the Pointcut. Advice is primarily of 3 types – *Before* advice, *After* advice, and *Around* Advice. *After* advice further contains two special cases – *After Returning* advice, and *After Throwing* advice.

Before advice as the name suggests runs before the Join Point is executed. After advice similarly runs just after the execution of the Join Point in question. Around advice runs when the Join point is reached. It has the power to decide whether or not to actually execute the Join Point or not.

**4.2.4 Aspect** – Kiczales et.al. [Kiczales 2001] [Kiczales G 2001] defines Aspects as "*modular units of crosscutting implementation*". They are declared using the keyword "aspect", similar to how a class is declared using the keyword "class" in Java. They contain within them Pointcut declarations, Advice declarations, method declarations, variable declarations, etc. [Avgustinov 2005] [AspectJGuide]

## 4.3 CCFinderX

CCFinderX is a code clone detection tool designed by Toshihiro Kamiya. CCFinderX software license revision of 15th October 2006, allows us to freely use the product without modification for the purpose of research, education, evaluation and/or in-house use.

CCFinderX can be used to detect code clones from source files written in Java, C/C++, COBOL, VB and C#. Kamiya et.al had earlier designed another code clone detection tool named CCFinder [Kamiya 2002]. CCFinderX is a re-designed version of the tool CCFinder. This is aimed at improving the performance of the tool, as well as to provide the users with an interactive analysis based on certain metrics [CCFinderX]. A snapshot of the tool is provided below in Figure 9.

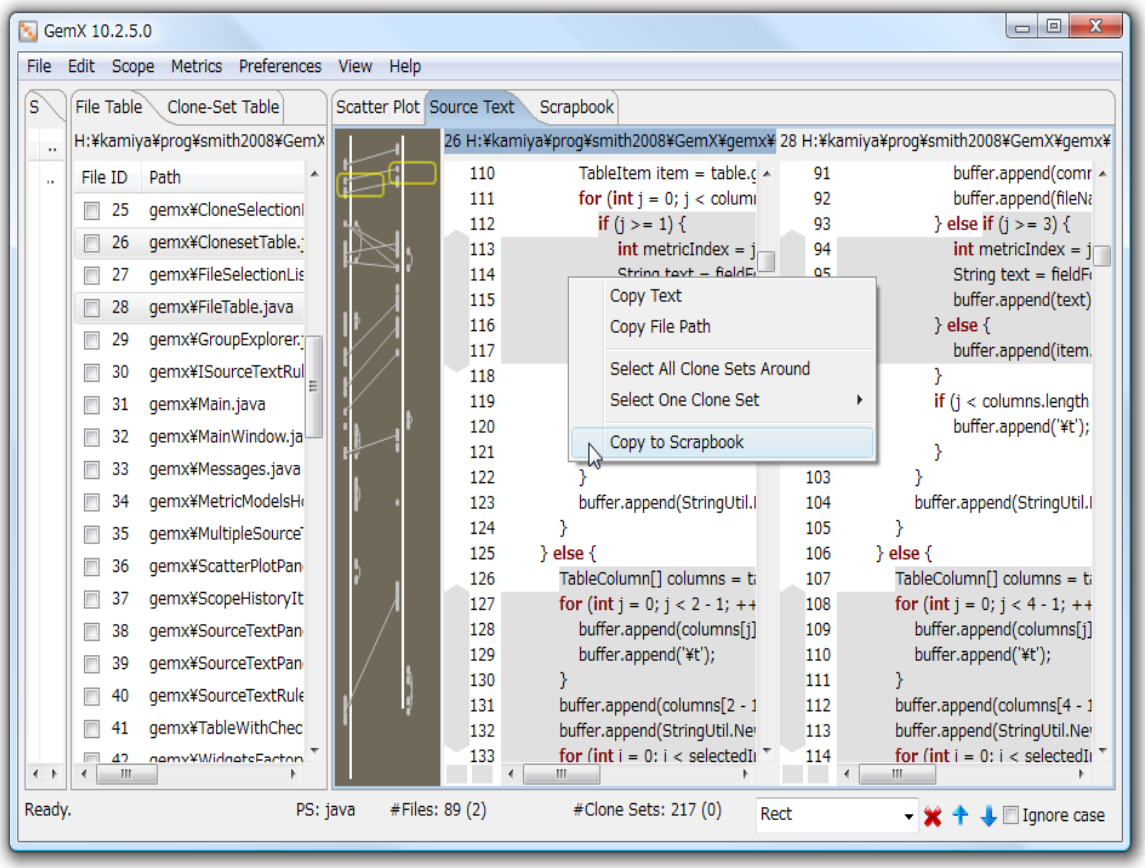


**Figure 9: A snapshot showing CCFinderX [CCFinderX]**

## 4.4 CC2ASPECT Software Implementation

This section describes the CC2ASPECT software prototype developed by us to test the algorithms described in chapter 3 created to convert the code clones to aspects.

### 4.4.1 Architecture of the Design

Figure 10 below shows the architecture we followed in designing our implementation of all the algorithms in chapter 3, which culminated in the CC2ASPECT software prototype.

The first stage of the work requires the identification of the code clones. For this, CCFinderX is used. The original source code of the Java project is used as an input to this CCFinderX process, and the locations of the identified code clones inside the Java project are produced as output. While there are numerous code clone detection tools identified from different research papers, e.g. CP-Miner, CloneDr, Deckard, CPDetector, RTF, Asta, NICAD, CCFinderX, Duplo, Simian, etc, some could not be found, while others were not freeware. Of the detection tools that were found freely available, like Duplo, CCFinderX, and Simian, we found that CCFinderX had the best GUI. With CCFinderX we have the line numbers of the matching code fragments, as well as visually see the exact lines of code, and how they were similar to each other.

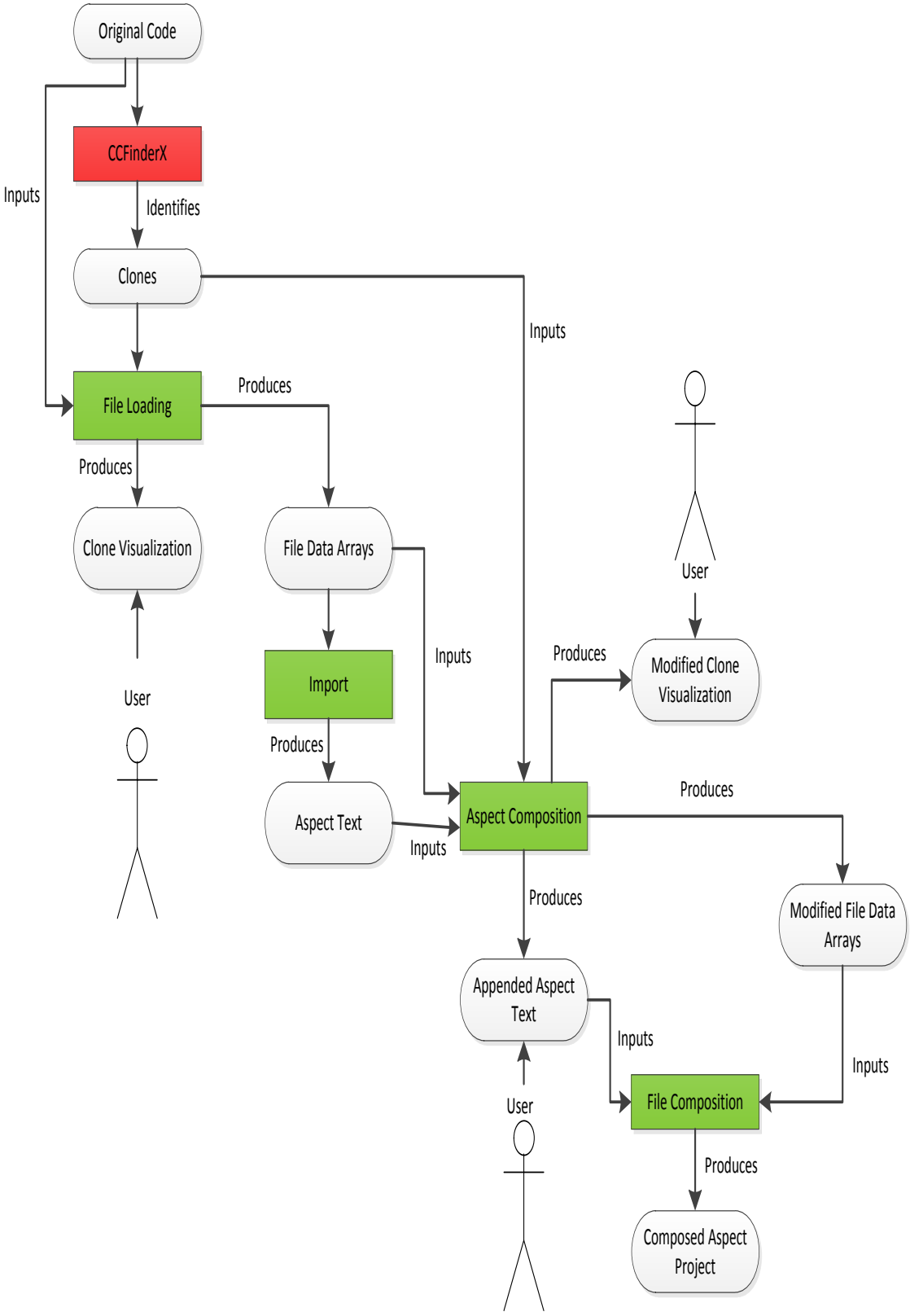


**Figure 10: Architecture of the prototype design**

The identified code clone locations, along with the original source code are used as inputs to the File Loading process. This process produces two outputs. The first output is the visualization of the code clones, so that the user can verify the accuracy of the code clone line numbers entered earlier. Secondly this process also converts the files containing the code clones into distinct two-dimensional data arrays, so that modifications can be done on the data contained within those files, while maintaining the integrity of the original data till the last step.

The File Data Arrays, which were created as outputs by the File Loading process, are used as inputs to the Import process. This process starts the creation of the Aspect text as its output. The process goes through the first 30 lines of each file array submitted to it. Upon finding lines of code where packages or files are being imported for the code to work, the process copies that line to the new Aspect text.

The File Data Arrays earlier produced by the File Loading process, and the identified code clone locations produced by the CCFinderX process are both used as inputs to the Aspect Composition process. It is the job of this process to compose the code clones present in the data arrays, into the required pointcuts and advice, and then comment out the code clone from the data array.

This creates a new modified data array as an output. The newly formed pointcuts and advice are then appended to the Aspect text which had been previously received as the output from the Import process. This process finally

produces visualization of both the commented and modified code clones, as well as the appended Aspect text for the user.

Both the Modified File Data Arrays and the appended Aspect text, which were created as outputs of the Aspect Composition process are taken as inputs to the final File Composition process. This process has two tasks. First it is used to write the Aspect text into an AspectJ file, and then save it. Secondly the process is used to overwrite the original source code files containing the code clones, with their respective modified file data arrays, in which the code clones had been commented out. These files are then saved.

Taken together at the end of the four processes, the code clones in the source code have been commented out and converted to Aspects.

### 4.4.2 Prototype Design

The Graphical User Interface (GUI) shown in Figure 11 below, was created for our prototype, and can take up to 4 clones at a time. For each clone we provide a text field to enter the file path of the file containing the code clone, two text fields to enter the starting and ending line numbers of the code clone segment, and a text area to display the code clone. At the bottom of the GUI, we have three control buttons to initiate the Loading, the Aspects Import and Package, the Aspect Composition, and the File Composition algorithms. To the right of these control buttons, is a text area to store the aspect text as it is being created. This is the text that is copied to the AspectJ file during the File Composition process.

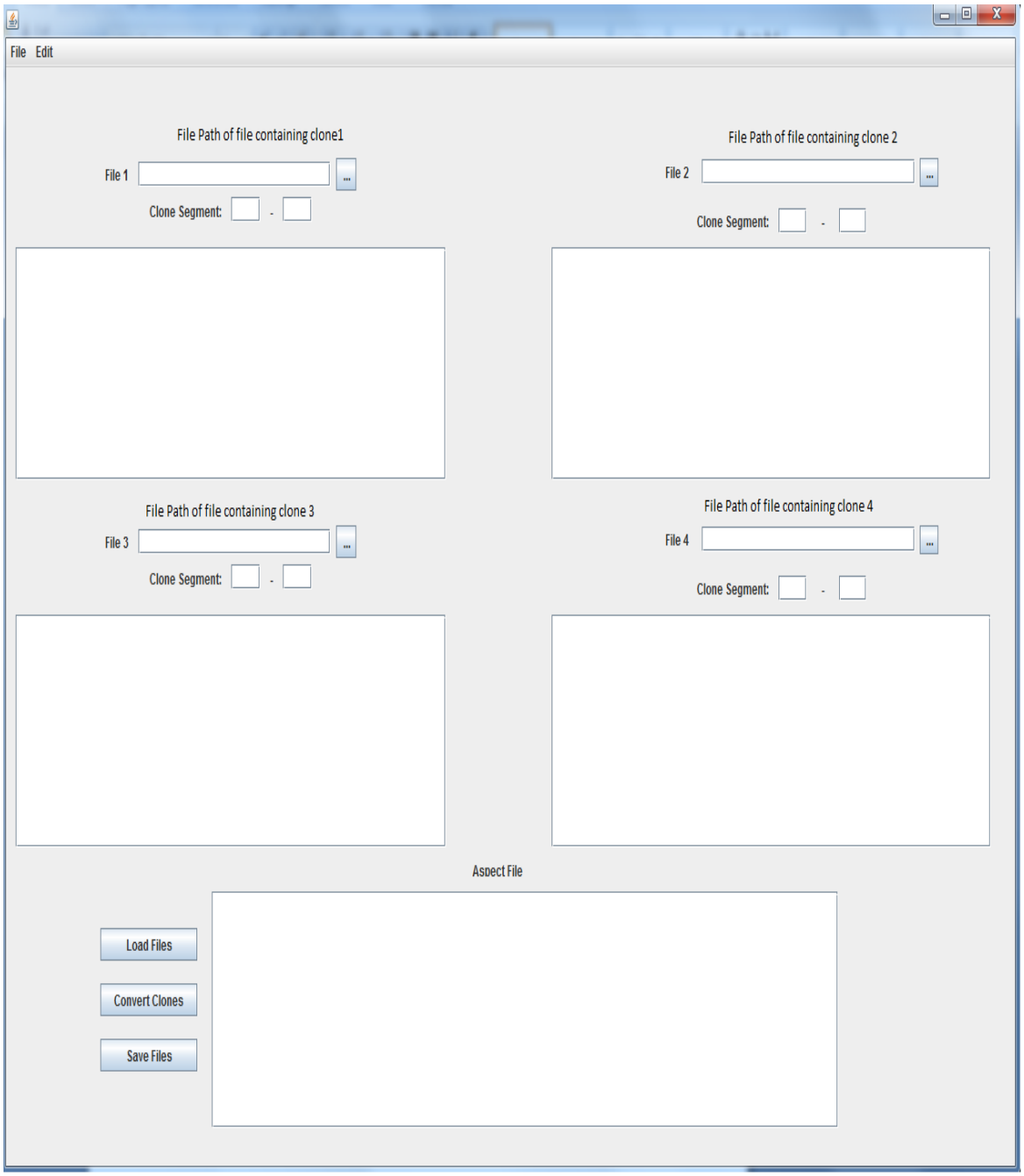


**Figure 11: CC2ASPECT Graphical User Interface**

#### *4.4.2.1 Load Files*

Figure 12 below shows the output received after loading the file. In it, the file paths of the files containing the clones were added to the text fields by the


**Figure 12: CC2ASPECT Load Files**

user, along with their corresponding clone starting and ending line numbers. Clicking the “Load Files” button in the GUI initiates the Java method that implements the File Loading Algorithm. This method first retrieves the four file paths entered by the user in the four text fields, and stores them in a string array name[ ]. A check is done to verify that the user has actually entered all four file paths or not. To accomplish this, all four text fields were pre-initialized with the text “Enter Path/Filename$_i$”. Now if the text present in the array location name[ i ] matches this string, it is taken that the user has not entered any data at that location, and an error message describing the same is shown to the user.

Once it has been verified that the user had entered all the fields, a second method is called to copy the code clone text to their respective text areas. This is done to provide visualization of the code clones to the user. Variables containing the file path, destination text areas, clone starting and ending line numbers are passed on as parameters to this method. This method is called four times, once for each code clone entered.

After loading all code clones to their respective text areas, the method containing the File Loading Algorithm, copies the entire file present at the destination file path for clone1 to the array full_File[1][ ], the file path itself is copied to the array finalName[1] and the total number of lines in the file containing the code clone to the array fileLength[1]. When dealing with second, third, and the fourth code clones, the method compares the file path of the clone under consideration with all previously processed file paths. For example, for code clone number 3, the method compares the file paths stored at array location

name[3] with those stored in locations name[2] and name[1]. Only if the file path at name[3] was found to be distinct, the method stores the complete file in array full_File[3][ ], its file path to the array finalName[3], and the number of lines in the file to array fileLength[3].

The final output of this entire process is the visualization of the code clones to the user, as well as the identification and storage of all distinct files and their corresponding data for [later] ease of access and retrieval.

#### *4.4.2.2 Convert Clones*

Figure 13 below describes the output received after converting the code clones to aspects. Clicking the “Convert Files” button in the GUI initiates the Java methods that contain the Aspects Import and Package algorithm, as well as the Aspect Composition algorithm. The method has to first copy all the lines of code which import files or packages to the Aspect text.

The array finalName[ ], from the File Loading process, now contains the file paths of the distinct filenames. This array was pre-initialized with the text string “abcdTestFile”.

The method then cycles through all four positions of the array finalName[ ]. Upon finding the path of a distinct file, the method opens that file, checks the first 30 lines of code present in that file, for the presence of keywords like “import” or “package”. If any of the required keywords is found, then the line in question is copied to the Aspect text area, and the method moves to the new


**Figure 13: CC2ASPECT Convert Clones**

line. If not, then the method moves on to the next line and tries again. This process continues till the top 30 lines of the file in question have been checked.

The conversion method now starts the process to actually convert the code clones into their corresponding Advice. The method next creates the advice specification and its corresponding advice body for the first code clone. It breaks down the method header using the Java method split(<delimiter>), and attempts to check whether the header has any method parameters or not. This is done due to the fact that method parameters need to be bound to those of the pointcut (Parameter Binding). So depending on whether or not the parameters are found, our conversion method creates the proper corresponding “Around” advice header by appending the chunks of code received from splitting the code clone method header, into the proper sequence required by the Around advice. The body of the advice is created by copying and appending the body of the code clone method.

To handle the second, third and the fourth code clones, the conversion method iteratively matches the clone text of the code clone under consideration with all the code clones processed before it. Only if the code clone text was found to be distinct, would its advice declaration and corresponding advice body be created.

Once all processing for the code has been done, the conversion method calls the method tasked with the commenting out of the code clones from the data array full_File[ ][ ]. The file path of the file which contains the code clone in question, along with the code clone starting and ending line numbers, as well as

the text area where the particular code clone was displayed to the user are passed as arguments to this method.

The process of commenting out the code clone lines continues till the last 5 lines of the clone. After this, it iterates from the last line up. If the method header contained a return type, its body must contain a “return” statement of a similar type. If such a statement is not present, the compiler will issue an exception against it. We have taken the assumption that the “return” statement, if present, would be the last statement of the code clone. Hence we, while reverse iterating these last lines, check if the code statement contains the keyword “return”. If found all lines except it are commented out. Lastly, our method overwrites the old code clone in its text area with the new commented version.

#### *4.4.2.3 Save Files*

Clicking the “Save Files” button, initiates the method tasked with the saving and finalizing of the Aspect text, as well as all the modifications done in previous conversion process. Figure 14 below shows a snapshot of this process. Up to this step, the user has the ability to abort the conversion operation without there being any changes or modifications to the original source files.

The File Composition method, with the aid of a JFileChooser, inputs the user’s selection of the target file where he/she wants the newly created Aspect code to be stored. While there is no restriction on the type of file where the

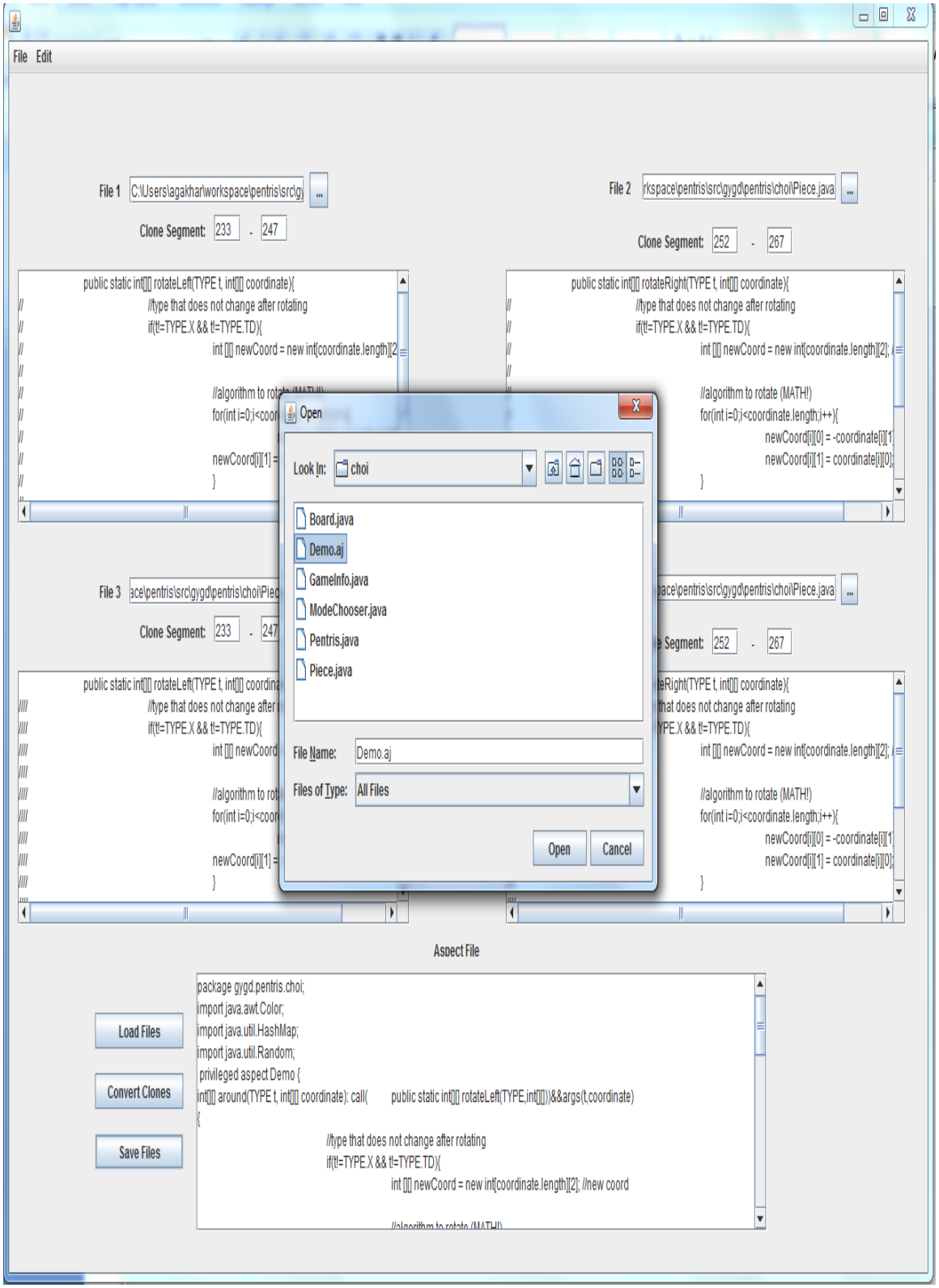


**Figure 14: CC2ASPECT Save Files**

aspect is to be saved, it is advisable to have an AspectJ (.aj) file ready to accept and store the newly created aspect code.

The method overwrites the target file with the aspect text from the Aspect Text area. This gives the users the ability to look through the aspect text and make any changes / modifications they desire, if they were not satisfied with the results previously obtained. For example, the user might not be satisfied with the indentation of the aspect text, and might want to change it according to their preferences. However these changes must be done prior to clicking the “Save Files” button. Once the Aspect text is overwritten, the file is saved and then closed.

The File Composition method then begins the process of overwriting the original code clones, with the modified data contained in the array full_File[ ][ ] where the code clones have been commented out. To do this, the method must first find the locations where the modified file data has been stored. This it does by checking the text strings stored in the array finalName[ ]. At the end of the File Loading process, the locations of this array would either hold the file paths of the distinct files, or contain the text “abcdTestFile”. Hence going through the array locations iteratively, if the method finds that the value at that location does not match the above mentioned text, then the corresponding location in the array full_File[ ][ ] would contain the modified data of that particular file. The method then opens the file pointed to by the file path stored, and overwrites it with the modified content of the array full_File[ ][ ]. It then saves and closes the file.

## 4.5 Experimentation and Analysis

In this section we outline the experiments conducted and present the results obtained. The experiments were conducted to verify that the conversion process did not have any adverse effects on the execution time of the software systems.

### 4.5.1 Experiment Settings

Figure 15, Figure 16 and Figure 17 below present snapshots of the hardware specifications of the machine where we performed our experimentation. These were obtained from the software CPU-Z [CPU-Z]. The machine is composed of a 2.4GHz Intel Core 2 Quad CPU with 3GB DDR2 RAM. The machine hosts a 64bit Windows 7 operating system.

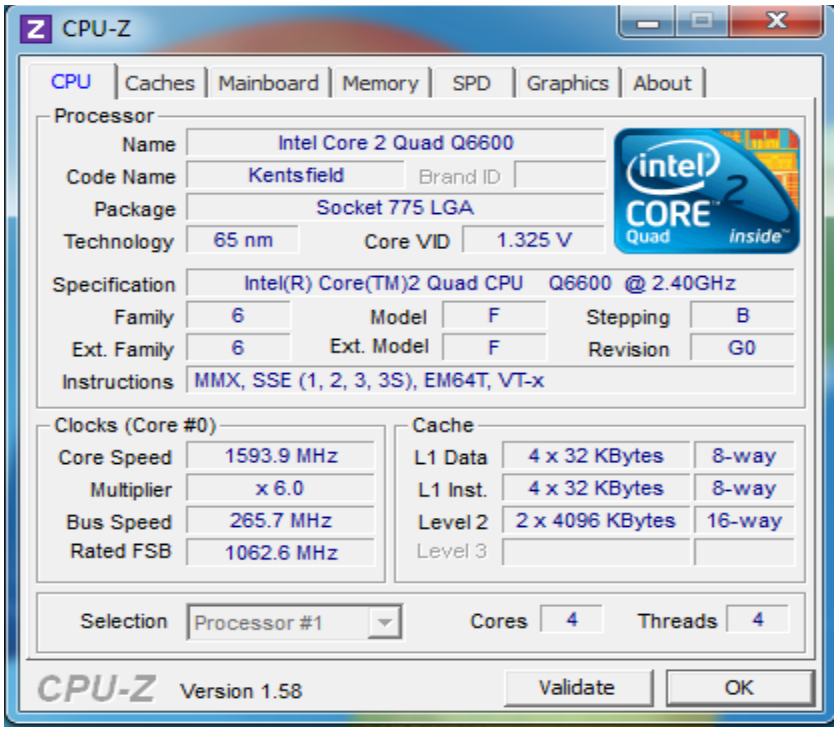


**Figure 15: CPU Specification**

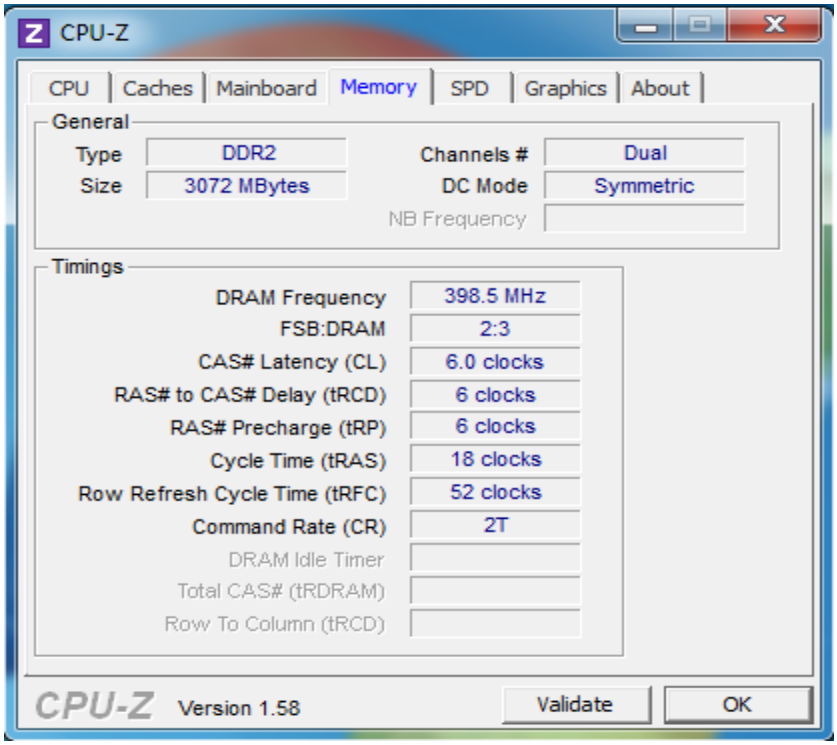


**Figure 16: Memory Specification**

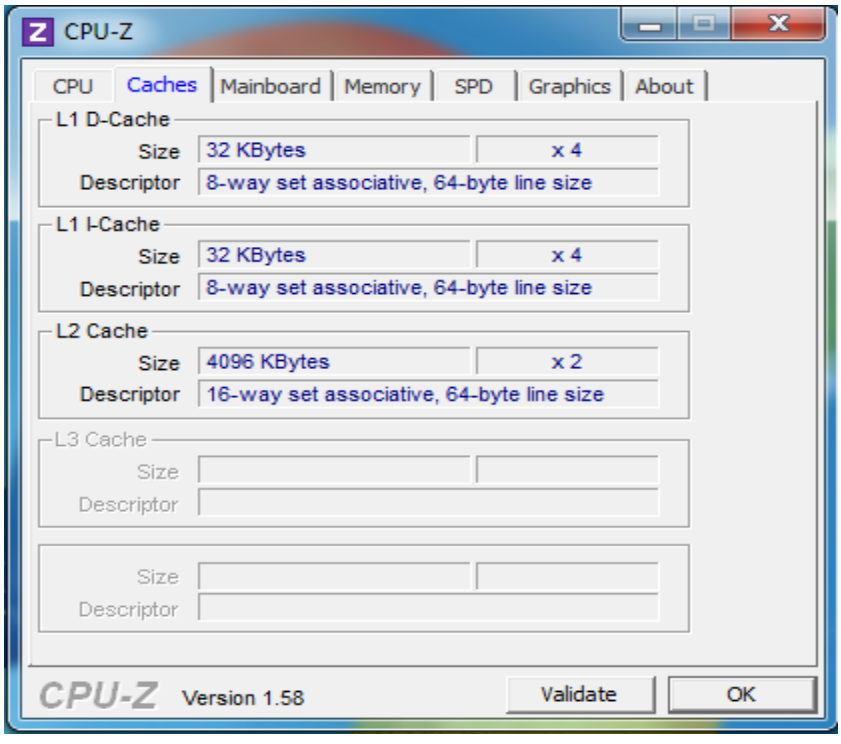


**Figure 17: Cache Specification**

### 4.5.2 Performance Measurements

This section presents different measurements being considered in this experiment.

1. **Average Execution Time**: This is the average execution time of multiple runs of the software system. It is found for each testing round and is in milliseconds.
2. **Performance Impact of Aspect program**: This is a measurement of the performance overhead of the program containing the code clones as Aspect's, compared to the original non-Aspect based program containing code clones. This is calculated as shown below in Figure 18.

$$\frac{\text{Average Execution Time of Original Code - Average Execution Time of Modified Code}}{\text{Average Execution Time of Modified Code}} \times 100\%$$

**Figure 18: Calculation of performance Impact [Liu 2011]**

This value can either be positive or negative. A positive value indicates that the modified code containing code clones as Aspect's runs faster than the original version, and vice versa.

### 4.5.3 Experiment Setup

Experimentation was performed by comparing the output produced by both the original version and the modified versions of the software systems, and tabulating their execution times. At the end of the experimentation round, the Average Execution Time, and the Performance Impact of the Aspect program

were calculated. This was done to check if converting the clones to aspects had any adverse effects on the performance of the software [Liu 2011] [Ajila 2010].

The system time was found using the method currentTimeMillis(). This is a method of type long, and a part of the java class System. This method returns the current time, measured in milliseconds. It does this by returning the difference between the current time and January 1st, 1970 UTC. The method main() is where the compilation and execution of a java program starts. So we store the system time received from the method currentTimeMillis(), both at starting time in variable startTime, and at ending time in variable endTime. The actual execution time is found by finding the difference between the two variables. It is assumed that the time taken for this calculation is miniscule compared to the execution time for the rest of the system code, and that the compiler exits immediately after displaying the execution time. The Figure 19 below describes the code used to find the execution time in the software PENTRIS and Figure 20 shows the output.

```
public static void main(String args[]) {
    long startTime = System.currentTimeMillis();
    new Pentris().startGame();
    long endTime = System.currentTimeMillis();
    System.out.println("Total elapsed time in execution is :"+ (endTime-startTime));

}
```

**Figure 19: Image describing code used to find program execution time**

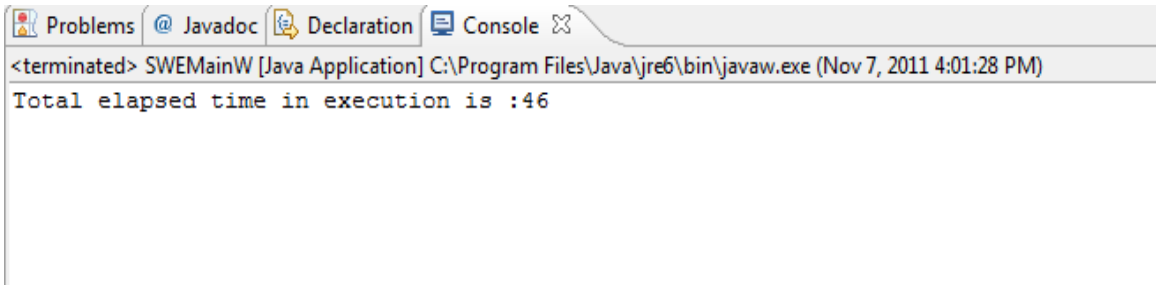


**Figure 20: Image showing the execution time of the program**

Two rounds of experimentation were performed each having 50 executions of both the original and the modified versions of the software systems. While it is not considered that the actual number of executions has a bearing on the Performance Impact of the software system, having two rounds of equal number of executions providing similar results provides a better indication of the results.

Two Software systems (SWEF and PENTRIS) were used in experimentation to find the Performance Impact of the Aspect Composed code. They are described in more detail in section 4.5.4 and section 4.5.5 below.

### 4.5.4 SWEF

The first software system is called the Software Engineering Framework (SWEF), designed and created by Cistel Technology Inc., an Ottawa based company providing technology and management consulting services.

The SWEF software

- Allows the user to open a project with its source code or design documents.

- Project metrics can be extracted from the source code and/or its design documents.
- Project metrics can be derived and analyzed for more useful information about the software system.
- Is suitable for large scale software development and understanding of legacy and/or third-party code.

While the clone detection software identified numerous code clones in the software, an example of a code clone is shown below in Figure 21. This clone belongs to the category of Type 1 code clones. The clones are present in different files of the source code.

```
public static CategoryDataset createDataset(Vector series, Vector category, Vector value) {

    // create the dataset...
    DefaultCategoryDataset dataset = new DefaultCategoryDataset();

    for (int i = 0; i < series.size(); i++) {
        String s = (String) series.elementAt(i);
        for (int j = 0; j < category.size(); j++) {
            String cat = (String) category.elementAt(j);
            int val = new Integer((Integer) ((Vector) value.elementAt(i)).elementAt(j));
            dataset.addValue(val, s, cat);

        }
    }
    return dataset;
}
```

**Figure 21: Code Clone in software SWEF before removal**

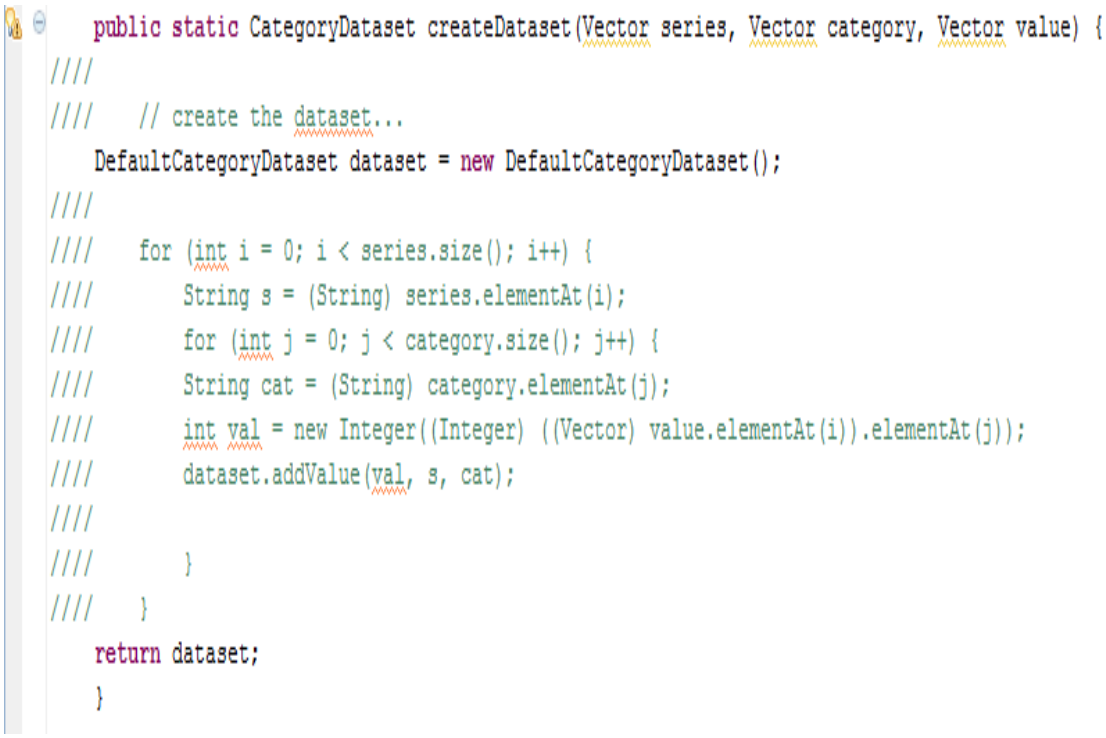


**Figure 22: Code Clone in software SWEF after removal**

After executing CC2ASPECT on SWEF, the code clone is commented out (cf. Figure 22) and the code clone is converted to aspect (cf. Figure 23).

```
privileged aspect Demo {
CategoryDataset around(Vector series, Vector category, Vector value): call(public static CategoryDataset createDataset(Vector,Vector,Vector))&&args(series,category,value)
{

        // create the dataset...
        DefaultCategoryDataset dataset = new DefaultCategoryDataset();

        for (int i = 0; i < series.size(); i++) {
            String s = (String) series.elementAt(i);
            for (int j = 0; j < category.size(); j++) {
            String cat = (String) category.elementAt(j);
            int val = new Integer((Integer) ((Vector) value.elementAt(i)).elementAt(j));
            dataset.addValue(val, s, cat);

            }
        }
        return dataset;
    }
}
```

**Figure 23: Code Clone in software SWEF as an aspect**

### 4.5.5 PENTRIS

The second test software system is PENTRIS. PENTRIS is a gaming software developed by Wonjohn Choi of the Global Youth Game (Software) Developers. It is a variation of the Tetris game, but instead of the normal shapes, he used pentominoes (polyomino composed of 5 blocks). The link to the software website is http://gygd.wordpress.com/ .

Here also the clone detection software identified numerous code clones, and examples of viable and acceptable code clones present are given in

Figure 24 below. The clones belong to the category of Type 3 code clones as there have been modifications to the code statements of these clones.

```
public static int[][] rotateLeft(TYPE t, int[][] coordinate){
    //type that does not change after rotating
    if(t!=TYPE.X && t!=TYPE.TD){
        int [][] newCoord = new int[coordinate.length][2]; //new coord

        //algorithm to rotate (MATH!)
        for(int i=0;i<coordinate.length;i++){
            newCoord[i][0] = coordinate[i][1];
        newCoord[i][1] = -coordinate[i][0];
        }

        return newCoord;
    }
    return coordinate.clone();
}

/**
 * return a copy of rotated coordinate
 */
public static int[][] rotateRight(TYPE t, int[][] coordinate){
    //type that does not change after rotating
    if(t!=TYPE.X && t!=TYPE.TD){
        int [][] newCoord = new int[coordinate.length][2]; //new coord

        //algorithm to rotate (MATH!)
        for(int i=0;i<coordinate.length;i++){
            newCoord[i][0] = -coordinate[i][1];
            newCoord[i][1] = coordinate[i][0];
        }

        return newCoord;
    }
    return coordinate.clone();

}
```

**Figure 24: Code Clones in the software PENTRIS**

The clone locations and their line numbers were found using the CCFinderX clone detection software. These were used as input to the CC2ASPECT conversion software. The output of that software removes the clones from the original files by commenting them out (cf. Figure 25). It also

converts those clones to aspects (cf. Figure 26), and saves them in a user selected file.

```
    public static int[][] rotateLeft(TYPE t, int[][] coordinate){
////        //type that does not change after rotating
////        if(t!=TYPE.X && t!=TYPE.TD){
////            int [][] newCoord = new int[coordinate.length][2]; //new coord
////
////            //algorithm to rotate (MATH!)
////            for(int i=0;i<coordinate.length;i++){
////                newCoord[i][0] = coordinate[i][1];
////            newCoord[i][1] = -coordinate[i][0];
////            }
////
////            return newCoord;
////        }
        return coordinate.clone();
    }

    /**
     * return a copy of rotated coordinate
     */
    public static int[][] rotateRight(TYPE t, int[][] coordinate){
////        //type that does not change after rotating
////        if(t!=TYPE.X && t!=TYPE.TD){
////            int [][] newCoord = new int[coordinate.length][2]; //new coord
////
////            //algorithm to rotate (MATH!)
////            for(int i=0;i<coordinate.length;i++){
////                newCoord[i][0] = -coordinate[i][1];
////                newCoord[i][1] = coordinate[i][0];
////            }
////
////            return newCoord;
////        }
        return coordinate.clone();
////
    }
```

**Figure 25: Code Clones in software PENTRIS after removal**

```
package gygd.pentris.choi;
import java.awt.Color;
import java.util.HashMap;
import java.util.Random;
privileged aspect Demo {
    int[][] around(TYPE t, int[][] coordinate): call(  public static int[][] rotateLeft(TYPE,int[][]))&&args(t,coordinate)
    {
            //type that does not change after rotating
            if(t!=TYPE.X && t!=TYPE.TD){
                int [][] newCoord = new int[coordinate.length][2]; //new coord

                //algorithm to rotate (MATH!)
                for(int i=0;i<coordinate.length;i++){
                    newCoord[i][0] = coordinate[i][1];
                newCoord[i][1] = -coordinate[i][0];
                }

                return newCoord;
            }
            return coordinate.clone();
    }
    int[][] around(TYPE t, int[][] coordinate): call(  public static int[][] rotateRight(TYPE,int[][]))&&args(t,coordinate)
    {
            //type that does not change after rotating
            if(t!=TYPE.X && t!=TYPE.TD){
                int [][] newCoord = new int[coordinate.length][2]; //new coord

                //algorithm to rotate (MATH!)
                for(int i=0;i<coordinate.length;i++){
                    newCoord[i][0] = -coordinate[i][1];
                    newCoord[i][1] = coordinate[i][0];
                }

                return newCoord;
            }
            return coordinate.clone();

    }
}
```

**Figure 26: Code Clones in software PENTRIS as Aspects**

### 4.5.6 Experimentation Results

Two rounds of experimentation were performed. In each round, both the SWEF and the PENTRIS software systems were executed 50 times, first in their original versions with the code clones present, and then with the modified versions where the code clones had been converted to aspects via the CC2ASPECT software prototype. In each execution, the execution time of the software was calculated and tabulated. The tables showing the exact values of

the execution times found in both rounds are provided in Appendix B of this report. At the end of the round of experimentation, we calculated the Average Execution Time of both the original and the modified versions of the software system, and then calculated the Performance Impact of the Aspect Program. These results are tabulated and shown in Table 3 and Table 4 below. Table 3 provides the results for the SWEF software system, while Table 4 shows the results for the PENTRIS software systems.

**Table 3: SWEF Software System Results**

| | Experiment Round 1 | Experiment Round 2 |
|---|---|---|
| Average Execution Time of SWEF with clone (m s) | 43.30 | 43.32 |
| Average Execution Time of SWEF with Aspect (m s) | 38.32 | 38.66 |
| Performance Impact of Aspect program (%) | 12.99 | 12.05 |

In the first round of experimentation, the original version of the SWEF software containing code clones had an Average Execution Time of 43.30 milliseconds, while the modified version with the code clones as aspect had an Average Execution Time of 38.32 milliseconds. The Performance Impact of the Aspect program in this round was found to be 12.99%. In the second round of experimentation, the original version of the SWEF software containing code

clones had an Average Execution Time of 43.32 milliseconds, while the modified version with the code clones as aspect had an Average Execution Time of 38.66 milliseconds. The Performance Impact of the Aspect program in this round was found to be 12.05%. Graphs depicting the Average Execution Times for the SWEF software system in both rounds of experimentation are shown in Figure 27 and Figure 28 below.

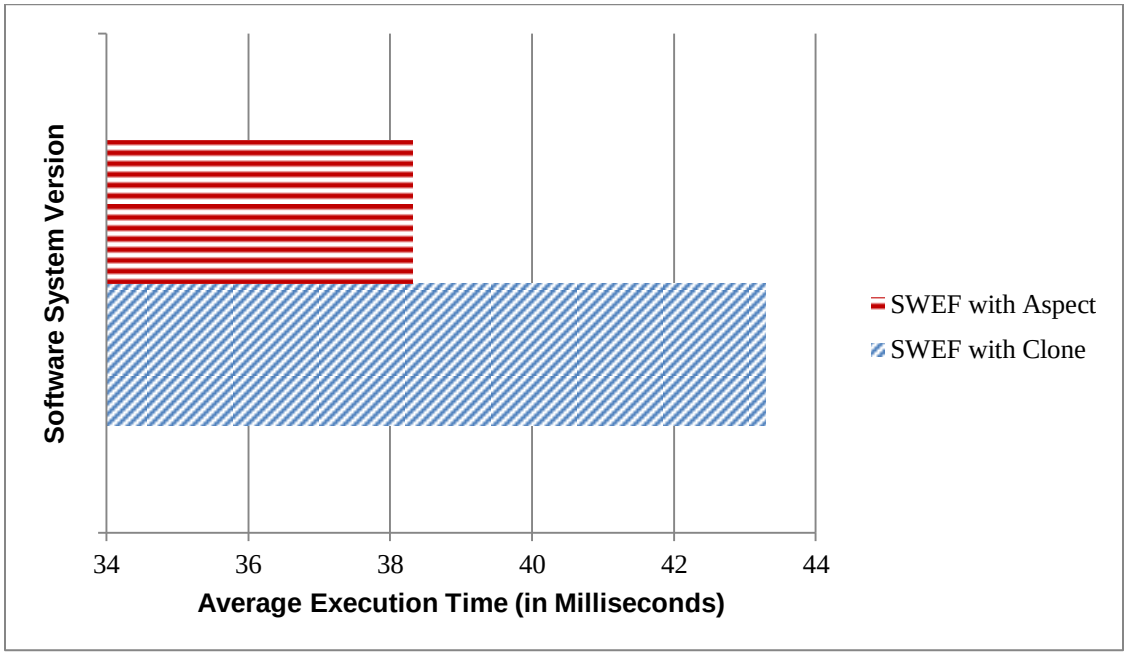


**Figure 27: Graph describing the Average Execution Time (in Milliseconds) of the SWEF Software versions in Experiment Round 1**

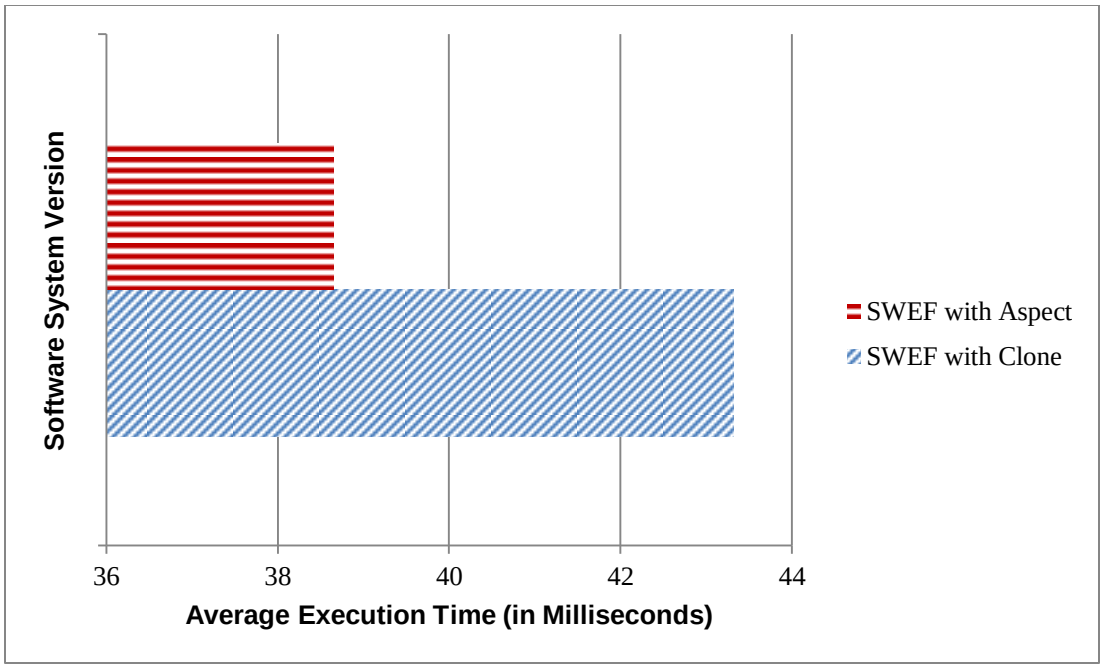


**Figure 28: Graph describing the Average Execution Time (in Milliseconds) of the SWEF Software versions in Experiment Round 2**

**Table 4: PENTRIS Software System Results**

| | Experiment Round 1 | Experiment Round 2 |
|---|---|---|
| Average Execution Time of SWEF with clone (m s) | 769.52 | 740.76 |
| Average Execution Time of SWEF with Aspect (m s) | 687.38 | 655.46 |
| Performance Impact of Aspect program (%) | 11.95 | 13.01 |

As seen from the above Table 4, in the first round of experimentation, the original version of the PENTRIS software containing code clones had an Average

Execution Time of 769.52 milliseconds, while the modified version with the code clones as aspect had an Average Execution Time of 687.38 milliseconds. The Performance Impact of the Aspect program in this round was found to be 11.95%. In the second round of experimentation, the original version of the PENTRIS software containing code clones had an Average Execution Time of 740.76 milliseconds, while the modified version with the code clones as aspect had an Average Execution Time of 655.46 milliseconds. The Performance Impact of the Aspect program in this round was found to be 13.01%. Graphs describing the Average Execution Times for the PENTRIS software system in both rounds of experimentation are shown in Figure 29 and Figure 30 below.

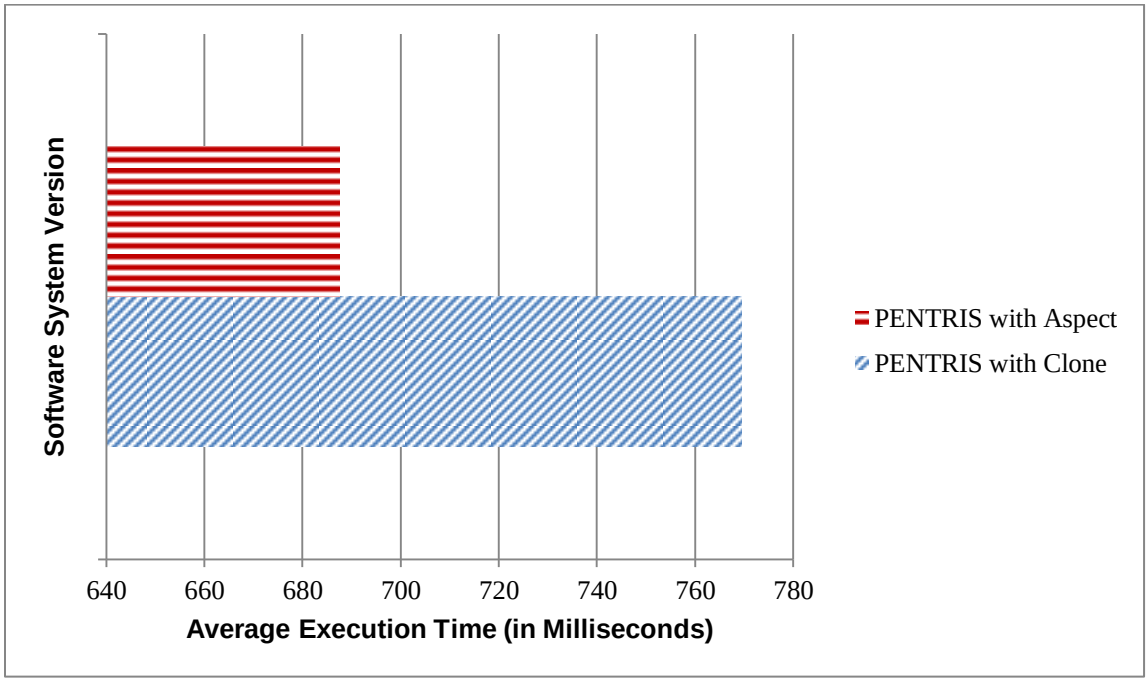


**Figure 29: Graph describing the Average Execution Time (in Milliseconds) of the PENTRIS Software versions in Experiment Round 1**

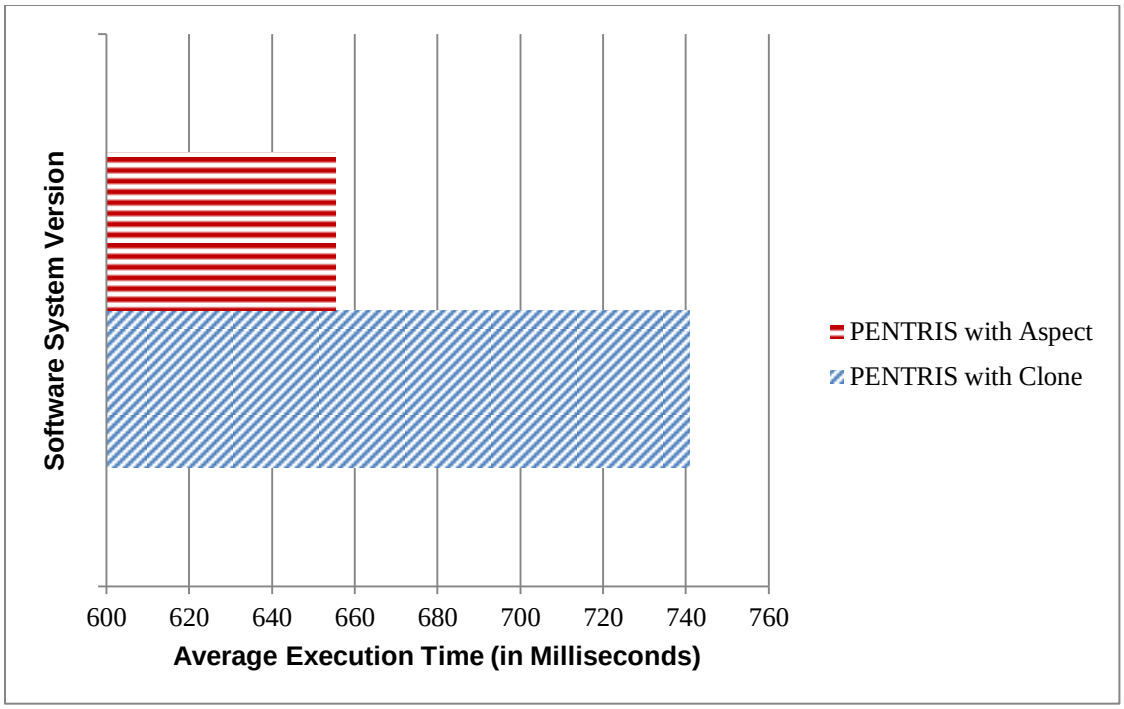


**Figure 30: Graph describing the Average Execution Time (in Milliseconds) of the PENTRIS Software versions in Experiment Round 2**

Judging from the results obtained from both experimentation rounds of the two software systems (Table 3 and Table 4), it is seen that the Average Execution Times of the modified software systems containing Aspect is lower than that of the original version containing code clones. We also see an improvement in the performance of the modified software systems ranging from a Performance Impact of 11.9% to 13.01%. This improvement in performance is contrary to the belief that using of Aspect Oriented Programming would cause a performance overhead due to the additional weaving time required. This behavior shown could be due to the effects of the operating system and its cache/memory management, however it cannot be confirmed without a detailed study into this phenomenon [Liu 2011].

# Chapter 5: Conclusions and Future Work

## 5.1 Conclusion

The main goal of this work is to convert code clones to aspects and compose the aspects. Towards this end, we used an algorithmic approach. The goal was achieved in four stages: The first was to use an existing code clone detection tool to identify code clones in source code. The tool selected for this purpose is CCFinderX (see section 4.3). The second was the designing of algorithms to convert code clones to aspects, and perform aspect composition with the original code. To fulfill this, we created four algorithms, namely the File Loading Algorithm, the Aspect Import and Package Algorithm, the Aspect Composition Algorithm, and the File Composition Algorithm (see chapter 3). The third was to implement a prototype which converts code clones to aspects and performs aspect composition. For this we created the software prototype CC2Aspect. This implementation was discussed in section 4.4. The prototype can only take 4 code clones at any given point of time because the GUI was designed for 4 code clones. There is no upper bound on the number of clones that our approach can remove. If we want to remove more than 4 code clones, only the GUI would need to be modified. The algorithms created in chapter 3 would remain unchanged. If we do not want to modify the GUI, and we have more than 4 code clones, we would first remove the first four code clones, and then re-run the CC2ASPECT software prototype again with the location information of the next batch of clones we want removed. Finally we carried out a performance analysis to make sure that the aspect composed code performed as

well as the original code. For this we conduct two sets of testing for both software's (SWEF and PENTRIS). In both cases it was found that the aspect composed code performed as well as the original code. This was shown in section 4.5.

## 5.2 Limitations and Future Work

We have identified two limitations with our algorithms. The first is that our algorithms require code clones that are self defined methods. By self defined we infer that every variable required for the proper execution of the method under consideration should be either defined and declared within the method itself, or should be passed as an argument to the method in the method header.

The second limitation is that in case of certain variants of clones, especially in Type 2 and Type 3, it is possible that the modifications made to one of the methods will produce a different result due to the underlying functionality in the other method. If such a scenario arises, then there has to be a variation between the two method headers. This is because the pointcuts being created by the algorithms are using the method headers themselves. Without the difference in their headers, the pointcuts would end up being the same, causing unnecessary confusion between which pointcut and advice to follow. This situation could arise because of modifications caused due to either, renaming of identifiers, renaming of literals/data types, modifications of the source code lines, addition and/or deletion of source code lines, reordering the source code statements or replacing the control statements. For example, comparing the code

clone pairs in Figure 3(A) and Figure 3(C) we find that the former deals with integer values, while the latter deals with floating point values. This causes a difference in their final outputs. Similar situations can be seen while comparing the code clones shown in Figure 4(A) and Figure 4(B), or Figure 4(A) and Figure 4(C), or Figure 4(A) and Figure 4(D). In all these situations we find that the final result of the method would change due to the internal modifications.

Part of future work includes undoing the limitations discussed above. Another part is integrating the environments together as a plug-in tool in eclipse, i.e. integrate the code clone detection tool and the code clone removal tool together to make it seamless. This would go a long way to remove user intervention in the code clone removal process.

## References


[Ajila 2010] S.A. Ajila, D. Petriu, P. Motshegwa, “Using Model Transformation Semantics for Aspects Composition”, 2010 IEEE 4th International Conference on Semantic Computing (IEEE-ICSC 2010), 22-24 September 2010, Page(s): 325-332.

[AspectJ] Eclipse AspectJ project Website

http://www.eclipse.org/aspectj/ [Accessed: 24th April, 2012]

[AspectJGuide] AspectJ programming guide

http://eclipse.org/aspectj/doc/released/progguide/language.html [Accessed: 24th April, 2012]

[Albunni 2008] N. Albunni, M. Petridis, “Using UML for Modelling Cross-Cutting Concerns in Aspect Oriented Software Engineering”, 3rd International Conference on Information and Communication Technologies: From Theory to Applications, 2008, ICTTA 2008, 7-11 April 2008, Page(s): 1-6.

[Avgustinov 2005] P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O.de. Moore, D. Sereni, G. Sittampalam, J. Tibble, “Optimising AspectJ”, Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI ’05), ACM 2005, 12-15 June 2005, Page(s): 117-128.

[Baker 1992] B.S. Baker, "A program for identifying duplicated code", Proceedings of Computing Science and Statistics, vol. 24, Interface Foundation of North America, 1992, Page(s): 49-57.

[Baker 1995] B.S. Baker, "On finding duplication and near-duplication in large software systems", Proceedings of the 2nd Working Conference on Reverse Engineering, 14-16 Jul 1995, Page(s): 86-95.

[Baxter 1998] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, "Clone detection using abstract syntax trees", Proceedings of the 14th International Conference on Software Maintenance, ICSM 1998, 1998, Page(s): 368-377.

[CCFinderX] CCFinderX Official Site

http://www.ccfinder.net/ccfinderx.html [Accessed: 24th April, 2012]

[CPU-Z] CPUID Technical Resources

http://www.cpuid.com/softwares/cpu-z.html [Accessed: 27th April, 2010]

[Eaddy 2007] M. Eaddy, A. Aho, G.C. Murphy, “Identifying, Assigning, and Quantifying Crosscutting Concerns”, First International Workshop on Assessment of Contemporary Modularization Techniques 2007, ICSE Workshops ACoM ’07, 20-26 May 2007.

[Eaddy 2008] M. Eaddy, T. Zimmermann, K.D. Sherwood, V. Garg, G.C.Murphy, N. Nagappa, A.V. Aho, “Do Crosscutting Concerns Cause Defects?”, IEEE Transactions on Software Engineering, Vol. 34, Issue 4, 2008, Pages: 497-515.

[Eclipse.org] Eclipse website

http://eclipse.org/ [Accessed: 24th April, 2012]

[EclipseAspectJ] Adrian Colyer, Andy Clement, George Harley, Matthew Webster, *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development tools*, Addison-Wesley, ISBN 0-32-124587-3.

[Johnson 1993] J.H. Johnson, “Identifying redundancy in source code using fingerprints”, Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1993, 1993, Page(s): 171-183.

[Johnson 1994] J.H. Johnson, “Visualizing textual redundancy in legacy source”, Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative research, CASCON 1994

[Juergens 2009] E. Juergens, F. Deissenbocck, B. Hummel, S. Wagner, “Do Code Clones Matter?”, IEEE 31st International Conference on Software Engineering, 2009, (ICSE’09), May 16-24, 2009, Vancouver, Page(s): 485-495.

[Kamiya 2002] T. Kamiya, S. Kusumoto, K. Inoue, “CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code”, IEEE Transactions on Software Engineering, Volume 28, Issue 7, July 2002, Page(s):654-670.

[Kiczales 1997] G. Kiczales, J. Lamping, A. Mendhakar, C. Maeda, C. Lopes, J.-M. Irwin, “Aspect-Oriented Programming”, European conference on Object-Oriented Programming (ECOOP), Finland, June 1997, Lecture Notes in Computer Science (LNCS), 1997, Volume 1241/1997, Page(s): 220-242.

[Kiczales 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, “An Overview of AspectJ”, ECOOP 2001 – Object-Oriented Programming, Lecture Notes in Computer Science (LNCS), Volume 2072/2001, 2001, Page(s): 327-353.

[Kiczales G 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, “Getting started with ASPECTJ”, Communications of the ACM, Volume 44, Issue 10, October 2001, Page(s):59-65.

[Koschke 2007] R. Koschke, “Survey of Research on Software Clones”, In Proceedings, 06301-Duplication, Redundancy, and Similarity in Software, 19th April 2007, URL: http://drops.dagstuhl.de/opus/volltexte/2007/962/ [Accessed: 24th April, 2012]

[Krinke 2007] J. Krinke, “A Study of Consistent and Inconsistent Changes to Code Clones”, 14th Working Conference on Reverse Engineering 2007 (WCRE 2007), 28-31 October 2007, Page(s): 170-178, Vancouver, BC.

[Liu 2011] W.L. Liu, C.H. Lung, S. Ajila, “Impact of Aspect-Oriented Programming on Software Performance: A Case Study of Leader/Followers and Half-Sync/Half-Async Architectures”, 2011 35th IEEE Annual Computer Software and Applications Conference (COMPSAC), 18-22 July 2011, Page(s): 662-667.

[Lopez-Herrejon 2006] R. Lopez-Herrejon, D. Batory, C. Lengauer, “A Disciplined Approach to Aspect Composition”, Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM ’06), 9-10 January 2006, Page(s): 68-77.

[Mayrand 1996] J. Mayrand, C. Leblanc, E.M. Merlo, “Experiment on the Automated Detection of Function Clones in a Software System using Metrics”, Proceedings, International Conference on Software Maintenance 1996, 4-8 Nov 1996, Page(s): 244-253.

[Rodriguez 2004] L. Rodriguez, E. Tanter, J. Noye, "Supporting dynamic crosscutting with partial behavioral reflection: a case study", 24th International Conference of the Chilean Computer Science Society, 2004, SCCC 2004, 11-12 Nov. 2004, Page(s): 48-58.

[Roy 2007] C.K. Roy, J.R. Cordy, “A Survey on Software Clone Detection Research”, Technical Report No. 2007-541, School of Computing, Queens University, Canada, September 26, 2007.

[Roy 2009] C. K. Roy, “Detection and Analysis of Near-Miss Software Clones”, Ph.D. Thesis, Queen’s School of Computing, Queens University, 2009-08-31, 14:05:30.233. http://hdl.handle.net/1974/5104 [Accessed: 24th April, 2012].

[Roy et al. 2009] C.K. Roy, J.R. Cordy, R. Koschke, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach”, Science of Computer Programming, Vol. 74, Issue 7, 1 May 2009, Page(s): 470-495.

[Schulze 2010] S. Schulze, S. Apel, C. Kästner, “Code Clones in Feature-Oriented Software Product Lines”, Proceedings of the ninth international conference on Generative Programming and Component Engineering GPCE’10, Oct. 10-13, 2010, Page(s): 103-112, Eindhoven, The Netherlands.

[Walker 1999] R.J. Walker, E.L.A. Baniassad, G.C. Murphy, “An Initial Assessment of Aspect-Oriented Programming”, ICSE’99

Proceedings of the 21st international conference on Software Engineering, May 1999, Page(s): 120-130.

[Wand 2004] M. Wand, G. Kiczales, C. Dutchyn, “A Semantic for Advice and Dynamic Join Points in Aspect-Oriented Programming”, ACM Transactions on Programming Languages and Systems, Vol. 26, No. 5, September 2004, Pages 890-910.

## Appendix A

This appendex contains the flowcharts for the different algorithms which were described in chapter 3.

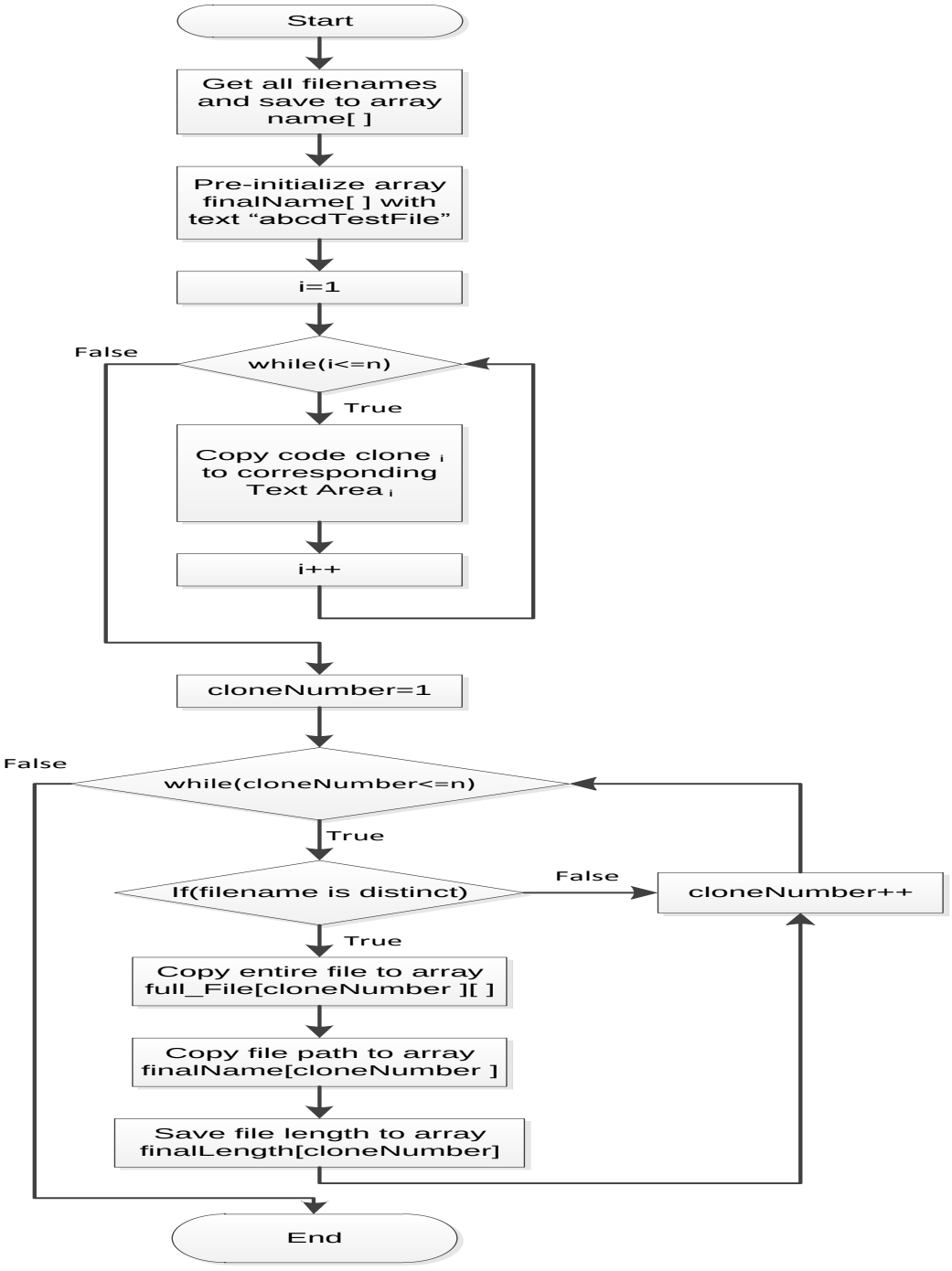


**Figure 31: Flowchart describing the File Loading Algorithm**

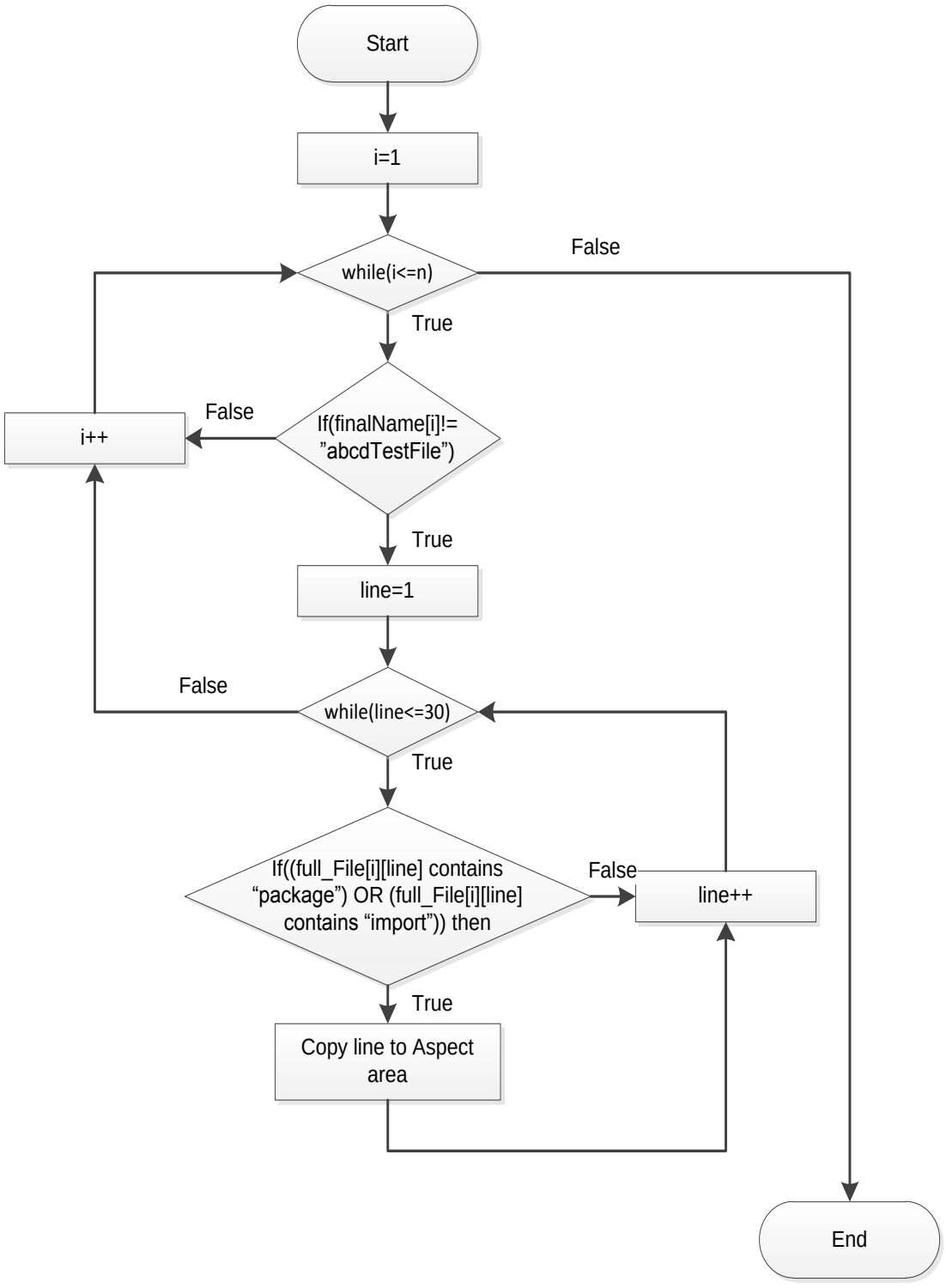


**Figure 32: Flowchart Describing the Aspects Import and Package Algorithm**

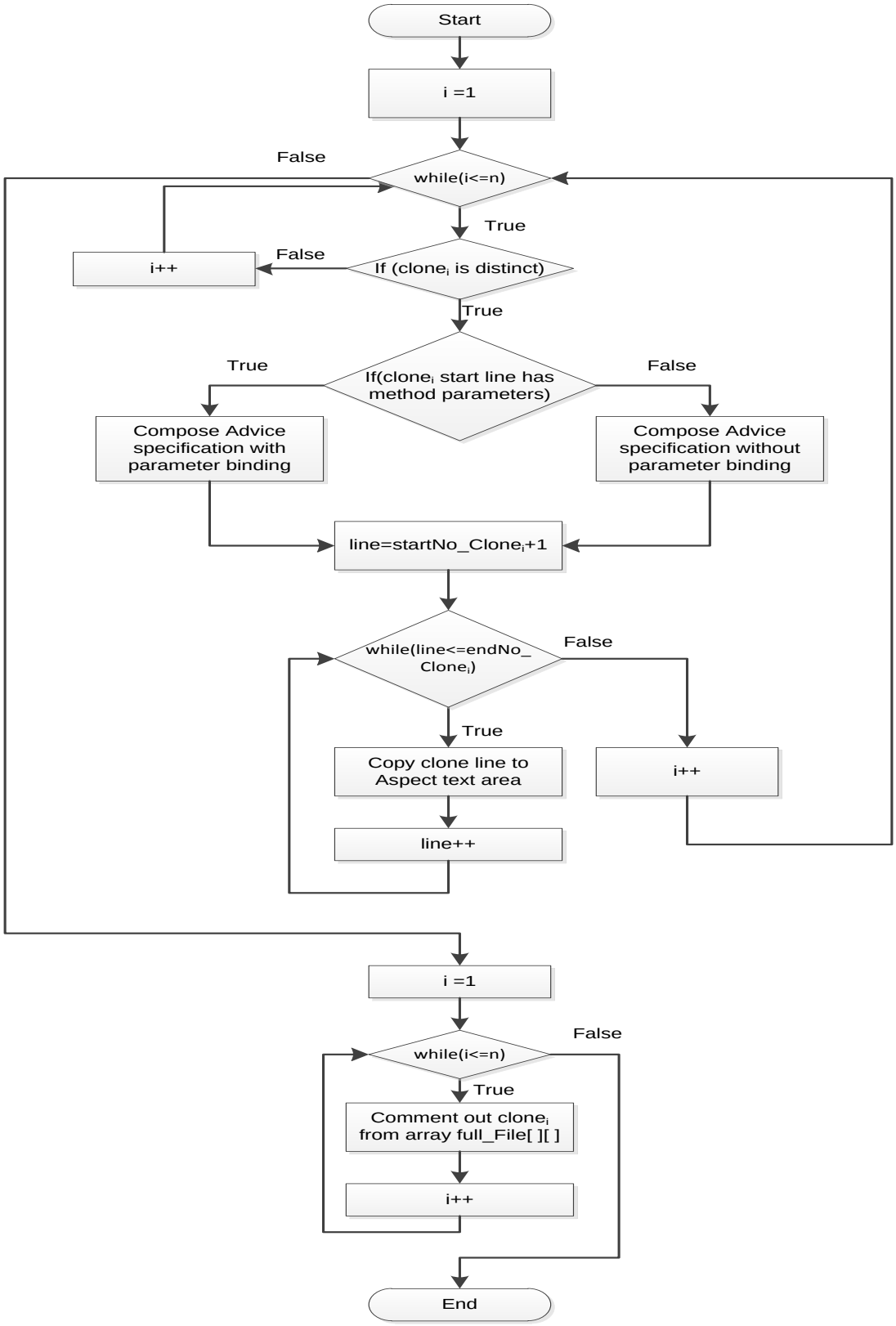


**Figure 33: Flowchart describing the Aspect Composition Algorithm**

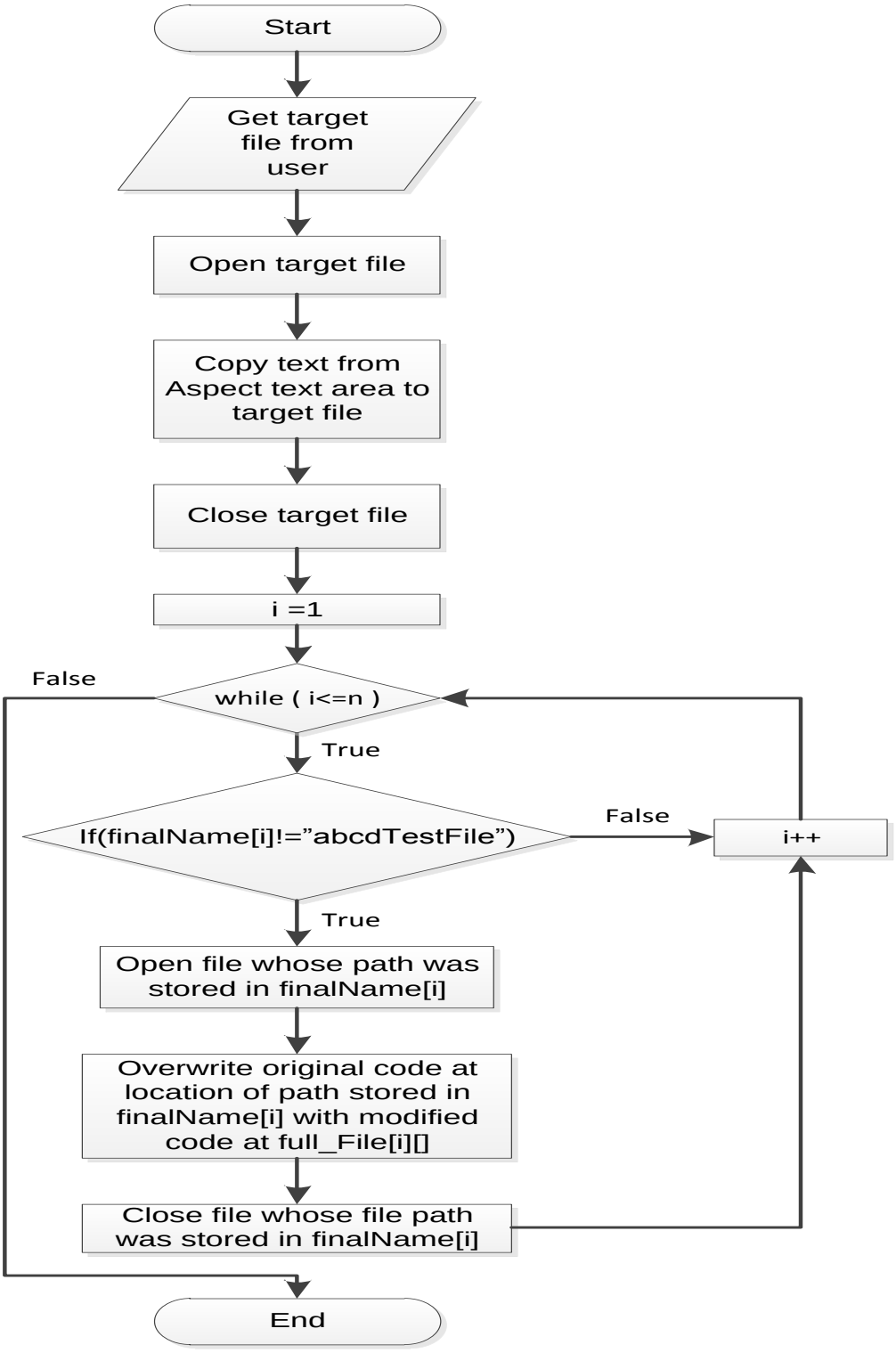


**Figure 34: Flowchart describing the File Composition Algorithm**

## Appendix B

This appendix contains the testing results of both rounds of experiments. Table 5 below provides the results for the SWEF software system. All execution time values are in milliseconds.

**Table 5: Execution times of SWEF software system in both experiment rounds**

| Serial Number | SWEF with Clone | | SWEF with Aspect | |
|---|---|---|---|---|
| | Round 1 | Round 2 | Round 1 | Round 2 |
| 1 | 47 | 47 | 31 | 47 |
| 2 | 47 | 31 | 46 | 47 |
| 3 | 46 | 62 | 40 | 31 |
| 4 | 31 | 31 | 47 | 32 |
| 5 | 47 | 47 | 46 | 32 |
| 6 | 31 | 47 | 31 | 47 |
| 7 | 31 | 46 | 47 | 47 |
| 8 | 31 | 46 | 47 | 31 |
| 9 | 47 | 31 | 31 | 32 |
| 10 | 62 | 31 | 47 | 31 |
| 11 | 47 | 31 | 31 | 47 |
| 12 | 31 | 47 | 31 | 47 |
| 13 | 47 | 62 | 47 | 31 |

| | | | | |
|---|---|---|---|---|
| 14 | 31 | 47 | 47 | 46 |
| 15 | 47 | 47 | 32 | 31 |
| 16 | 46 | 47 | 47 | 47 |
| 17 | 47 | 47 | 32 | 47 |
| 18 | 47 | 31 | 31 | 31 |
| 19 | 32 | 47 | 47 | 47 |
| 20 | 47 | 31 | 46 | 31 |
| 21 | 47 | 47 | 31 | 31 |
| 22 | 31 | 47 | 32 | 31 |
| 23 | 47 | 47 | 31 | 31 |
| 24 | 47 | 31 | 32 | 47 |
| 25 | 62 | 46 | 31 | 46 |
| 26 | 47 | 47 | 46 | 31 |
| 27 | 31 | 31 | 47 | 47 |
| 28 | 46 | 47 | 47 | 46 |
| 29 | 46 | 32 | 31 | 47 |
| 30 | 47 | 46 | 32 | 31 |
| 31 | 47 | 47 | 31 | 32 |
| 32 | 46 | 32 | 47 | 47 |
| 33 | 47 | 31 | 47 | 46 |

| | | | | |
|---|---|---|---|---|
| 34 | 46 | 47 | 47 | 31 |
| 35 | 47 | 46 | 31 | 47 |
| 36 | 46 | 47 | 47 | 31 |
| 37 | 47 | 46 | 47 | 32 |
| 38 | 47 | 47 | 31 | 46 |
| 39 | 31 | 31 | 31 | 47 |
| 40 | 47 | 47 | 32 | 31 |
| 41 | 31 | 47 | 32 | 31 |
| 42 | 46 | 47 | 31 | 31 |
| 43 | 47 | 62 | 32 | 46 |
| 44 | 31 | 47 | 46 | 31 |
| 45 | 47 | 46 | 47 | 31 |
| 46 | 46 | 47 | 47 | 31 |
| 47 | 47 | 31 | 32 | 47 |
| 48 | 47 | 47 | 31 | 47 |
| 49 | 47 | 47 | 32 | 46 |
| 50 | 32 | 47 | 31 | 32 |

Table 6 below provides the results for the SWEF software system. All execution time values are in milliseconds.

**Table 6: Execution times of PENTRIS software system in both experiment rounds**

| Serial Number | PENTRIS with Clone | | PENTRIS with Aspect | |
|---|---|---|---|---|
| | Round 1 | Round 2 | Round 1 | Round 2 |
| 1 | 983 | 686 | 765 | 670 |
| 2 | 749 | 671 | 577 | 592 |
| 3 | 734 | 874 | 733 | 515 |
| 4 | 905 | 780 | 624 | 655 |
| 5 | 827 | 874 | 639 | 499 |
| 6 | 796 | 921 | 624 | 733 |
| 7 | 733 | 437 | 919 | 593 |
| 8 | 749 | 873 | 655 | 765 |
| 9 | 718 | 733 | 640 | 733 |
| 10 | 452 | 717 | 702 | 546 |
| 11 | 499 | 749 | 827 | 624 |
| 12 | 546 | 748 | 562 | 530 |
| 13 | 796 | 748 | 577 | 530 |
| 14 | 795 | 718 | 734 | 702 |
| 15 | 998 | 889 | 639 | 765 |
| 16 | 718 | 686 | 670 | 686 |

| | | | | |
|---|---|---|---|---|
| 17 | 749 | 452 | 655 | 780 |
| 18 | 796 | 780 | 780 | 796 |
| 19 | 780 | 920 | 812 | 717 |
| 20 | 889 | 499 | 780 | 780 |
| 21 | 889 | 514 | 639 | 639 |
| 22 | 811 | 671 | 686 | 687 |
| 23 | 682 | 749 | 671 | 671 |
| 24 | 702 | 935 | 593 | 608 |
| 25 | 795 | 733 | 607 | 640 |
| 26 | 889 | 702 | 655 | 655 |
| 27 | 962 | 702 | 608 | 656 |
| 28 | 961 | 702 | 702 | 671 |
| 29 | 733 | 624 | 702 | 515 |
| 30 | 765 | 701 | 609 | 499 |
| 31 | 780 | 670 | 858 | 812 |
| 32 | 858 | 639 | 640 | 562 |
| 33 | 700 | 686 | 827 | 655 |
| 34 | 842 | 608 | 656 | 640 |
| 35 | 749 | 779 | 842 | 702 |
| 36 | 795 | 686 | 640 | 639 |

| | | | | |
|---|---|---|---|---|
| 37 | 671 | 874 | 562 | 811 |
| 38 | 655 | 717 | 718 | 624 |
| 39 | 499 | 873 | 718 | 671 |
| 40 | 764 | 874 | 577 | 468 |
| 41 | 718 | 796 | 624 | 686 |
| 42 | 780 | 920 | 687 | 919 |
| 43 | 796 | 795 | 765 | 624 |
| 44 | 750 | 857 | 608 | 656 |
| 45 | 717 | 904 | 640 | 655 |
| 46 | 740 | 827 | 812 | 764 |
| 47 | 796 | 858 | 718 | 655 |
| 48 | 780 | 593 | 780 | 671 |
| 49 | 811 | 686 | 671 | 546 |
| 50 | 874 | 608 | 640 | 561 |