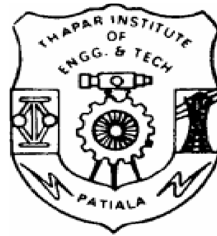
THREE STAGE TRANSFORMATION FOR SOFTWARE CLONE DETECTION

A thesis
Submitted in partial fulfillment of the requirements for the award of degree
of

Master of Engineering
in
Software Engineering



Under the Supervision of
Sh. Rajesh K. Bhatia
Assistant Professor
CSED, TIET, Patiala.

Submitted By
Kunal Pandove
(Roll No 8033109)

Computer Science & Engineering Department
Thapar Institute of Engineering & Technology
(Deemed University), Patiala-147004

May 2005

Declaration

I hereby certify that the work which is being presented in the thesis entitled, "**THREE STAGE TRANSFORMATION FOR SOFTWARE CLONE DETECTION**", in partial fulfillment of the requirements for the award of degree of Master of Engineering submitted to Computer Science and Engineering Department of Thapar Institute of Engineering and Technology (Deemed University), Patiala, is an authentic record of my own work carried out in the supervision of Mr. Rajesh K. Bhatia. The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other university.

Kunal Pandove

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Mr. Rajesh K. Bhatia
Assistant Professor
Computer Science & Engineering Department
Thapar Institute of Engg. & Tech. ,Patiala –147004.

Countersigned By:

Mr. R.S.Salaria
Head of Department
Computer Science & Engineering Department
Thapar Institute of Engineering & Technology
Patiala –147004.

Dr. D.S.Bawa
Dean (Academic Affairs)
Thapar Institute of Engg. & Tech.
Patiala – 147004.

The M.E. (Thesis) Viva-Voce examination of Kunal Pandove, Roll No. 8033109, M.E. (Software Engineering), Thapar Institute of Engineering and Technology Patiala has been held on ………………….

Supervisor **External Examiner**

Acknowledgements

First of all I would like to pay most sincere thanks to all the staff of Computer Science & Engineering Department, Thapar Institute of Engineering & Technology (Deemed University) Patiala.

Sincere thanks goes to Mr. Rajesh K. Bhatia, Assistant Professor and my guide. For providing his uncanny guidance and support throughout the preparation of this thesis.

I express appreciation to staff members and my batch mates for making my stay at Thapar Institute of Engineering & Technology (Deemed University) Patiala an unforgettable experience

Last but not the least I would like to express my gratitude and heartfelt thanks to all the people and acquaintances that I got to interact with and see at Thapar Institute of Engineering & Technology (Deemed University) Patiala.

Kunal Pandove
Roll No. 8033109

Contents

Abstract...1

Organization Of The Thesis...2

Chapter 1: Introduction…………………………………………………...3 –17

1.1 Problems with clones in code..4

1.2 Types of clones..4

1.2.1 Duplicated blocks within same function..............................4

1.2.2 Similar functions, same file...5

1.2.3 Functions cloned between files within same directory.......6

1.2.4 Functions cloned across directories..................................7

1.2.5 Cloned files, possibly with some changes.........................7

1.2.6 Blocks across files...8

1.2.7 Initialization and finalization clones...................................8

1.3 Clone detection techniques……………………………………….....9

1.3.1 String based...10

1.3.1.1 Simple line matching...10

1.3.1.2 Parameterized line matching.............................10

1.3.2 Token based..11

1.3.3 Parse tree based..12

1.4 Issues in applicability of the different techniques...........................13

1.4.1. How much configuration is needed to apply on another language?..13

1.4.2 What kind of matches are found? 14

1.4.3 How accurate are the results? 14

1.4.4 How does it perform? 16

1.4.5 Conclusion regarding applicability of the detection techniques 16

1.5 Summary..17

Chapter 2: Approaches To Clone Detection……………………….18 – 28
2.1 A language independent approach to detecting clones................18
2.1.1 Algorithm for language independent approach................18
2.1.1.1 Source code transformation..............................19
2.1.1.2 Comparison algorithm......................................19
2.1.2 Visualization..20
2.1.3 Pattern matching to extract copied sequences................20
2.1.4 Textual reports...21
2.1.5 Visualization vs. Automated detection............................21
2.2 Substring matching for clone detection...21
2.2.1 Algorithm for substring matching for clone detection.......21
2.2.1.1 Text-to-text source transformation 22
2.2.1.2 Generation of candidate substrings....................23
2.2.1.3 Identification of raw substring matches.............23
2.2.1.4 Information-preserving simplification of match database...23
2.2.1.5 Data reduction..24
2.2.1.6. Presentation of file clusters and multi-file matches..24
2.3 Clone detection using abstract syntax trees.................................24
2.3.1 Finding sub-tree clones..25
2.3.2 Basic clone detection algorithm.......................................27
2.4 Summary..28

Chapter 3: Proposed Model: Three Stage Transformation For Clone Detection...29 – 38
3.1 Approach: Three stage transformation for clone detection............29
3.1.1 Pretty print formatting...30
3.1.2 Dictionary replacement...30
3.1.3 Tokenization of variables...30
3.2 Modeling of the system 31
3.3 Implementation...35
3.3.1 Class structure...35
3.4 User Interface..36

3.5 Output..38

Chapter 4: Conclusion And Future Work…………………………..39 - 47

4.1 Case Study: Application of the three stage clone detection tool on a set of files.............................40

4.1.1 Input files...40

4.1.2 Output files...41

4.1.2.1 MS-Excel format file...42

4.1.2.2 Rtf format file...42

4.1.3 Analysis of results...45

4.1.3.1 Pretty print formatting..45

4.1.3.2 Dictionary replacement......................................46

4.1.3.3 Tokenization of variables...................................46

4.2 Conclusions..46

4.3 Future work..47

Bibliography……………………………………………………………...48 – 50

List Of Publications……..……………………………………………………...51

List Of Figures

1.1 Detection steps for the metric fingerprint technique................................12

2.1 Overview of the language independent approach...................................19

3.1 Component diagram...31

3.2 Use case diagram...32

3.3 Activity diagram..33

3.4 Data flow diagram...34

3.5 GUI for clone detection tool..37

3.6 Open File Dialog...37

3.7 Error in case filename field(s) are left empty...38

3.8 Result of the clone detection process..38

Abstract

Software engineers often build new procedures by cloning, copying an existing one with similar requirements, and slightly modifying it. While this may be easier than extracting the common part, and sharing it in a library, it increases the system size and often leads to higher maintenance costs. The occurrence of clones is highly dependent on the system architecture and development model, and has been studied in the past for a few large procedural systems.

In the last decade, several researchers have investigated techniques to detect duplicated code in programs exceeding hundreds of thousands lines of code. All of these techniques have known merits and deficiencies, but as of today, little is known on where to fit these techniques into the software maintenance process.

Current work proposes a three stage transformation for software clone detection. This technique makes use of substring matching and derives partially from token based technique for clone detection. Ambiguity problems in string matching algorithm are reduced by the dictionary replacement for similar keywords.

Organization Of The Thesis

The thesis is organized in four parts, covering the introduction to cloning in software, clone detection techniques and a model proposed for clone detection.

Chapter 1 provides the basic introduction to clones, the reasons for occurrence of clones, the types of clones, clone detection techniques and applicability of clone detection techniques.

Chapter 2 gives the algorithmic details of three clone detection approaches. The first approach is a language independent approach. The second is based on substring matching. The third technique is based on use ob Abstract Syntax Trees.

In Chapter 3 a three stage transformation model for software clone detection has been proposed.

The work done has been concluded along with future directions in Chapter 4.

Chapter 1
Introduction

A “clone” in software is a segment of code that has been created through duplication of another piece of code. [1] [2] e.g. Copy and paste. Clones may start to appear for any one of the following reasons:

- Development time: A software engineer clones a procedure when needing similar functionality instead of extracting the common reusable part. It looks quicker to achieve, may be faster for the initial implementation, but often leads to more expensive code to maintain.
- Communication: A software engineer borrows code from a colleague but cannot extract the common reusable part. Either he is not sufficiently knowledgeable about the cloned procedure, or he cannot convince the other software engineers involved to include this reusable procedure in the library and modify their code to use it.
- Structural: A software engineer borrows code from another subsystem but cannot avoid cloning because the other subsystem may not be modified; the other subsystem may belong to a different department or may not be modifiable (stored in non-volatile memory in embedded systems, or frozen after a lengthy testing/qualification procedure).
- Coincidence: It may happen that two software engineers came up with similar procedures independently, thus leading to look-alikes more than clones. It would be beneficial to replace them with a reusable procedure. These are typically much more difficult to detect as they may achieve the same functionality with somewhat different apparent structures.
- Efficiency: Considerations of efficiency may make the cost of a procedure call or method invocation, too high a price.

1.1 Problems with clones in code:

Unjustified duplicated code gives rise to severe problems:

- If one repairs a bug in a system with duplicated code, all possible duplications of that bug must be checked.
- Code duplication increases the size of the code, extending compile time and expanding the size of the executable.
- Code duplication often indicates design problems like missing inheritance or missing procedural abstraction. In turn, such a lack of abstraction hampers the addition of functionality.
- Errors in the systematic renaming can lead to unintended aliasing, resulting in latent bugs that show up much later.
- The effect of all of these is a form of “software aging” or “hardening of the arteries” that result when even small design changes become very difficult to make.

1.2 Types of clones

In general, clones may be described using the following typology:

- Type I: An exactly identical source code clone, i.e. no changes at all.
- Type II: An exactly identical source code clone, but with indentation, comments, or identifier (name) changes.
- Type III: A functionally identical clone, but with small changes made to the code to tailor it to some new function.
- Type IV: A functionally identical clone, developed possibly with the originator unaware that there is a function already available that accomplishes essentially the same function.

The following subsections present another way to classify the types of clones found in software [11] [15]:

1.2.1 Duplicated blocks within same function

Characterized as repeated blocks of code within the same function, these blocks are of non-trivial and each copy expresses the same semantic idea, generally with very few variables changed. The major problem that this can

cause is increased code size; in particular it can cause functions to grow long and unreadable. In addition, this type of cloning may lead to unintended diverging evolution of the code blocks if a developer changes one block, and not another. A bad initialization or 'value changed' type of error can very easily happen in this type of code, because it is likely variables are shared, intentionally or unintentionally, by the individual blocks. Situations where this typically occurred is in control structures such as switch and if/else statements. The cause of this may be that some developers do not anticipate a condition that may require a similar block, so they do not think to make the block a function from the start. Also, making the function that encapsulates the functionality of this clone block may appear to be too much work, because of the number of local variables involved in the code block. Another reason may be time: it is very fast to just copy and paste the block just a few lines down, and the developer "knows" the code works, so it is a quick and dirty solution. Performance may also be an issue, if many local variables are required to be passed, stack creation and destruction may be time consuming. A solution to this problem, as with many code clones, would be to create a new function or macro to represent the block, and call the function where these clones occur. Parameters to the function would be the few changed variables that occur in the code block. One would expect this change to be simple and straightforward to implement.

1.2.2 Similar functions, same file

This type of clone occurs when a programmer has two functions performing very similar tasks, with minor variations. These types of clones are often characterized by changing only a few function calls, variable initializations, constants, or other minor things. Any functions which both match 60% of their code, are considered to be cloned functions. Consequences of this type of clone are increased code size. Also fixing bugs may be harder because same error may be spread across several functions, as well as the functions may evolve on separate paths as various maintainers update them. Developers are likely to do this when the effort required to parameterize the code block and create a more general function appears to be too great when compared to simply copying the code. Also cloning the function may actually make the

program conceptually simpler, because the function names can be specific and meaningful. This type of cloning is not considered extremely harmful because clones are not physically far apart, but it is recommended that such cloning activity should be documented as it may not be apparent to future maintainers which functions are clones of each other. Solutions for this can be very simple, or quite complex. Possible solutions would be to introduce function pointers to the parameter list, adding more parameters for initialization, etc.

1.2.3 Functions cloned between files within the same directory

This type of clone occurs when the same functionality is required among multiple files. The majority of code duplication that occurs within a directory ,excluding duplication with the same file, is related to duplicated functions, more than 80% of clone pairs that occur within the same directory (but not in the same file) are related to the duplication of functions. It often occurs with no changes at all to the cloned segment of code, or minor changes such as the function name and some variable or function calls. At times, several constants may be changed, global variables accessed and in these cases a solution is harder to find. Consequences of this type of clone are code size increase, and increased difficulty in error finding and correcting. The copied code segments are no longer localized in the same file and easily identified, but may be scattered across as many as four or five files. At times, this type of code duplication may contribute to source code that is easier to read. Functions will be easier to understand because they will not include extra logic and flows of control which would be required to restructure a function to encompass the more general functionality required of it to eliminate duplicates. This case is less frequent however, and quite often the use of function pointers or some minor conditional operations would create a function which may perform the desired task. A simple solution to this is to create a common file to use as a library, and migrate the function definitions and prototypes of the cloned functions to this file. This will work best in the case of exact copies, or clones with minor changes.

1.2.4 Functions cloned across directories

This type of clone may occur when the same functionality is common among several different components in the software. As with functions cloned within a subsystem, it may entail no changes at all to the cloned segment, or minor changes such as the function name and some variable or function calls. Consequences of this type of clone are code size increase, and may increase labor for error fixing. Also, it may be the case that one developer created one component, and is unaware of the clones existing in the rest of the system. In this case, when an error is found, repairs may not even have a chance to be propagated to the rest of the clones. This type of cloning may occur when a new subsystem is being created, and the design and implementation is based on previous work of another subsystem. Creating a set of library function may be the easiest solution, but if the function is cloned only between several files, the effort put into creating a new library, and maintaining it, to be shared by all components may be more work than it is worth.

1.2.5 Cloned files, possibly with some changes

This type of clone occurs when a new problem arises with requirements that are very similar to those of an existing software system, and the source code is readily available. For example, when new file system is introduced to the system, it may be possible to copy another's file, and make only minor changes. Consequences of this type of cloning can be much more severe than function cloning, because the clone has now introduced a large number of lines of code that are common between the two files, and must be changed together, especially when bug fixing. Because it is likely that there will be some alterations to some of the code, it may not be clear where or how to change the cloned file when reflecting changes that have been made to the original code. Also, this is one of the worst-case scenarios for code size increase. In addition, it is possible that side effects (such as inefficient device usage and settings) can occur if the developer does not fully understand the code that he/she has copied. This may lead to inefficiencies in the code and instability. This type of cloning will occur when speed of development may be a factor, or a developer may not completely understand the problem at hand. In practice this is seen when drivers are made for related hardware. Solutions

to this problem may not be as simple as other cloning types. Because the two files are used on different products or include different features, they may need evolve separately from this point on. As well, changes that have been made to the duplicated code may make it difficult to refactor both subsystems completely just to remove to code duplicates. That said, a workable solution may be to try to take the common invariant code and place it into a common library file which both subsystems could use. This solution may lead to a slightly more complex architecture.

1.2.6 Blocks across files

Often, in the case of cloning blocks across directories, we see that the cloned block is in fact the remains of what appears to be a cloned function. The function is often changed to suit the developers own personal style and also to meet the specific needs of his/her own project. Based on our observations, we would argue that most clones that occur across files start out as whole function clones and then are manipulated to fit the current project goals until what remains are scattered blocks of code which can still be captured as code duplicates. The main problem with this kind of clone is when the developer wants to modify or change these blocks of code or when they find bugs, it will be very difficult to fix and change these blocks everywhere else, and it is possible that the developer may be completely unaware of the other clones. If any logic on which this block depends changes, then all the blocks may be harmed, and it may be difficult to find all the blocks affected. The solution for this problem is relative to the size and number of clones that occurs across files. In certain contexts it might be proper to leave the clones as it is, such as in the case of if or case statement, sometimes making function calls may break the understanding of the logic of the code. In other cases a common library should be made.

1.2.7 Initialization and finalization clones

This type of clone occurs within the same file or across file systems when initializing data parameters or cleaning up at end of function. This usually occurs when using the same data types or when performing the same tasks such as memory allocation and de-allocation or variable initialization.

Finalization clones often encompass exit conditions and logging. Problems with this type of clone are much less severe than other clone types, and in many cases are unavoidable. Certainly increased code size may be an issue, but other problems related to code duplication do not seem as large of a concern. Solutions to this sort of problem may be the use of macros or functions, but this seems too complex for something that is of such little issue.

1.3 Clone detection techniques

Code cloning or the act of copying code fragments and making minor, non–functional alterations, is a well known problem for evolving software systems leading to duplicated code fragments or code clones. Of course, the normal functioning of the system is not affected, but without countermeasures by the maintenance team, further development may become prohibitively expensive. As far as removal of duplicated code is concerned, the state of the art proposes refactoring which is a technique to gradually improve the structure of programs while preserving their external behavior.

Extract Method which extracts portions of duplicated code in a separate method, is an example of a typical refactoring to remove duplicated code. However, quite often one must use a series of refactorings to actually remove duplicated code. Concerning the detection of duplicated code, numerous techniques have been successfully applied on industrial systems. These techniques can be roughly classified into three categories [25] :

(i) string-based, i.e. the program is divided into a number of strings (typically lines) and these strings are compared against each other to find sequences of duplicated strings [6]

(ii) token-based, i.e. a lexer tool divides the program into a stream of tokens and then searches for series of similar tokens [18] [19]

(iii) parse–tree based, i.e., after building a complete parse-tree one performs pattern matching on the tree to search for similar sub–trees. [3] [8]

Despite all this progress, little is known about the most optimal application of a given clone detection technique during the maintenance process. For instance, which technique should one use in a problem assessment phase,

when one suspects duplicated code but isn't sure how much and in which files? Or which technique works best in combination with a refactoring tool, which has to know the exact boundaries of the code segment to be refactored, including possible renaming of variables and parameters?

The detection of code clones is a two phase process which consists of a transformation and a comparison phase. In the first phase, the source text is transformed into an internal format which allows the use of a more efficient comparison algorithm. During the succeeding comparison phase the actual matches are detected. Due to its central role, it is reasonable to classify detection techniques according to their internal format.

1.3.1 String based

String based techniques use basic string transformation and comparison algorithms which makes them independent of programming languages [6] [25]. Techniques in this category differ in underlying string comparison algorithm. Comparing calculated signatures per line is one possibility to identify for matching substrings.

Line matching, which comes in two variants, is an alternative which is selected as representative for this category because it uses general string manipulations.

1.3.1.1 Simple line matching: is the first variant of line matching in which both detection phases are straightforward. Only minor transformations using string manipulation operations, which can operate using no or very limited knowledge about possible language constructs, are applied. Typical transformations are the removal of empty lines and white spaces. During comparison all lines are compared with each other using a string matching algorithm. This results in a large search space which is usually reduced using hashing buckets. Before comparing all the lines, they are hashed into one of n possible buckets. Afterwards all pairs in the same bucket are compared.

1.3.1.2 Parameterized line matching: is another variant of line matching which detects both identical as well as similar code fragments. The idea is

that since identifier–names and literals are likely to change when cloning a code fragment, they can be considered as changeable parameters. Therefore, similar fragments which differ only in the naming of these parameters, are allowed. To enable such parameterization, the set of transformations is extended with an additional transformation that replaces all identifiers and literals with one, common identifier symbol like ”$P”. Due to this additional substitution, the comparison becomes independent of the parameters. Therefore no additional changes are necessary to the comparison algorithm itself.

1.3.2 Token based

Token based techniques use a more sophisticated transformation algorithm by constructing a token stream from the source code, hence require a lexer. [18] [19] The presence of such tokens makes it possible to use improved comparison algorithms.

Parameterized Matching with Suffix Trees consists of three consecutive steps manipulating a suffix tree as internal representation. In the first step, a lexical analyzer passes over the source text transforming identifiers and literals in parameter symbols, while the typographical structure of each line is encoded in a non-parameter symbol. One symbol always refers to the same identifier, literal or structure. The result of this first step is a parameterized string or p-string. Once the p-string is constructed, a criterion to decide whether two sequences in this p-string are a parameterized match or not is necessary. Two strings are a parameterized match if one can be transformed into the other by applying a one-to-one mapping renaming the parameter symbols. An additional encoding prev(S) of the parameter symbols helps us verifying this criterion. In this encoding, each first occurrence of a parameter symbol is replaced by a 0. All later occurrences are replaced by the distance since the previous occurrence of the same symbol. Thus, when two sequences have the same encoding, they are the same except for a systematic renaming of the parameter symbols.

After the lexical analysis, a data structure called a parameterized suffix tree (p-suffix tree) is built for the pstring. A p-suffix tree is a generalisation of the suffix tree data structure which contains the prev()-encoding of every suffix of

a P-string. Concatenating the labels of the arcs on the path from the root to the leaf yields the prev( )-encoding of one suffix. The use of a suffix tree allows a more efficient detection of maximal, parameterized matches. All that is left for the last step, is to find maximal paths in the p-suffix tree that are longer than a predefined character length.

1.3.3 Parse tree based

Parse tree based techniques use a heavyweight transformation algorithm, i.e. the construction of a parse tree [3]. Because of the richness of this structure, it is possible to try various comparison algorithms as well.

Metric Fingerprints builds on the idea that you can characterize a code fragment using a set of numbers [5] [8]. These numbers are measurements which identify the functional structure of the fragment and sometimes the layout. The metric fingerprint technique can be divided in five steps, each with a well-defined task. Figure 1.1 shows the basic steps in the detection process.

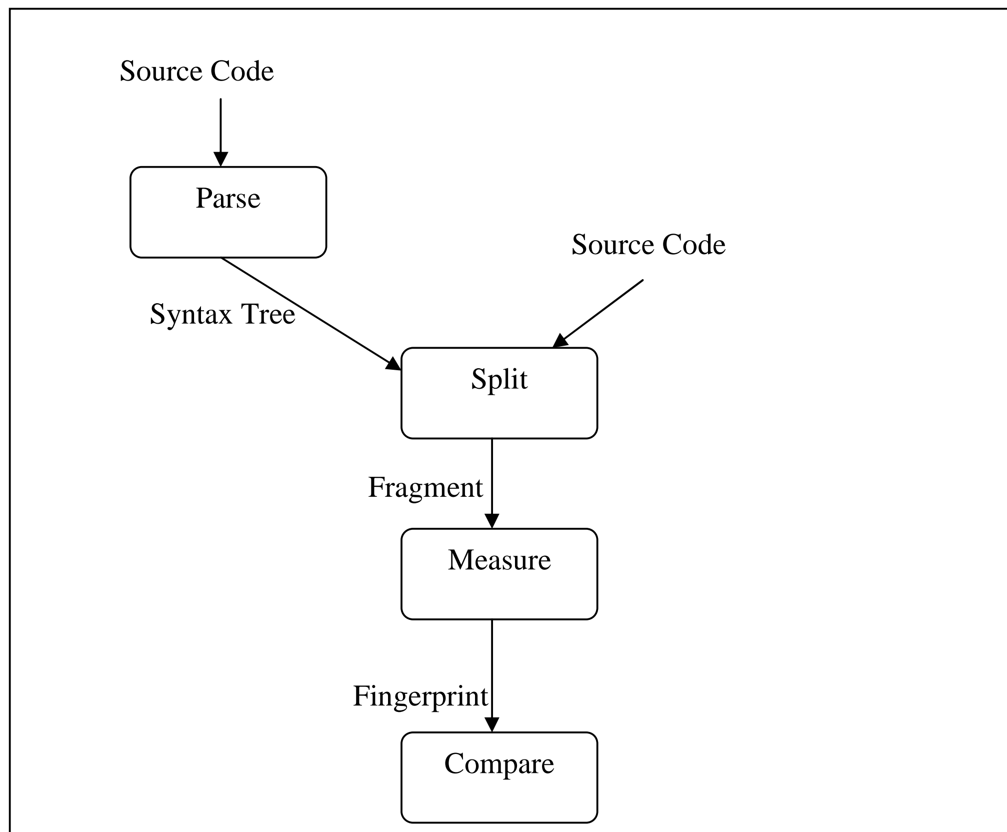


Figure 1.1 - Detection steps for the metric fingerprint technique

Before we can characterize the functional structure of a code fragment with numbers, it's wise to transform the source code into a representation that allows us to calculate such measurements efficiently. This transformation job is done using a parser which builds the syntax tree of the source code. After parsing we end up with one large syntax tree. This tree is then split into interesting fragments. The choice of the type of fragments used is difficult because it affects the detection results. Most of the time, however, method and scope blocks are used as fragments since they are easily extracted from a syntax tree. Afterwards the fragments are characterized through a set of measurements by measuring the values for a set of metrics, chosen in advance. This set of metrics can differ between various implementations, but most of the time it specifies functional properties. However there are implementations in which layout metrics are used as well. Cyclomatic complexity, function points, expression complexity (functional) and lines of code (layout) are examples of possible measures [4]. Finally, these sets of numbers are compared to each other. Depending on the implementation, algorithms with different levels of sophistication or power may be used. One possible approach calculates the Euclidean distance between each pair of fingerprints, considering fragments within zero distance as clones.

1.4 Issues in applicability of the different techniques

To identify which clone detection techniques are more appropriate for specific tasks of the maintenance process, the following issues need to be tackled [2] [3] [6] [18] [25].

1.4.1. How much configuration is needed to apply on another language?

Before using a technique, you like to know how much configuration has to be done to adapt it to your particular programming context. Especially because it may limit the applicability of the technique, certainly in COBOL and C++ environments, where lots of dialects exist. Simple line matching, as it only utilizes basic string manipulations, is a truly language independent technique which is very easy to configure. As a language independent technique, no

modification is required to be applicable on different languages. All the remaining techniques on the other hand, do require configuration.
For parameterized matching the portability to another language is fair. Changing the lexer, which lies at the basis of both techniques, suffices to port it. Because more changes in the lexer are necessary for the parameterized line matching technique, its portability is slightly lower than that of the suffix tree technique. Both parameterized techniques are fairly portable.

The metric fingerprint technique demands much configuration effort as it is syntax dependent due to the use of a parser. One is confronted with this syntax dependence because it fails even due to a syntax error in the analyzed code. The use of a parser limits the technique to syntactically correct sources of one language and makes changing to other languages difficult.

1.4.2 What kind of matches are found?

Depending on the maintenance task at hand, one may be looking for specific kinds of duplication. For instance, during a problem assessment phase, maintainers want to obtain an overall report of the amount of duplication existing in all program files. On the other hand, during a restructuring phase, maintainers are interested in a duplication tool that detects only the programming constructs that one can restructure using a particular tool. Therefore a refactoring tool, moving methods in the class hierarchy, is interested only in duplicated method bodies.

1.4.3 How accurate are the results?

For the clone detection problem, detection accuracy is difficult to define, but in the context of duplicated code detection it is characterized by three quality measures:

- number of false positives (to be minimized): that is, the number of matches the technique incorrectly identified as a piece of duplicated code.

- number of useless matches (to be minimized): that is, the number of matches which are not worth to be removed by means of refactoring. Typically depending on the length of a match.
- number of recognizable matches (to be maximized): that is, the number of matches that are easily recognized as interesting. For instance, in a program restructuring phase these are the matches that are easily removed by the refactoring tool at hand.

Number of false matches— No false matches are reported by both simple line matching and parameterized matching using suffix trees. Simple line matching reports only equal lines which makes it impossible to have false positives, while parameterized matching using suffix trees benefits from its P-string encoding that enforces a strict one–to–one parameterization. Only positive matches (parameterized or exact) are found by them.

Parameterized line matching allows a non systematic renaming of the parameters which leads to few false matches. Such systematic renaming is necessary to ensure that two fragments share the same basis functionality which characterizes duplication.

Even more false matches are reported by the metric fingerprint technique.
For the metric fingerprint technique more useless matches are reported. Most of them are only one to four lines long and are caused because two method calls with the same number of arguments always match.

Simple line matching also reports many useless matches. As an example think of the “return;” statement you tend to write in your program. It is hard to estimate the exact number of useless matches in general but usually it is larger than the amount for metric fingerprints. Number of recognizable matches— For the metric fingerprints technique the number is high. Each match that is returned is a functional block like e.g. scope blocks and method definitions.

Both parameterized matching techniques return a lower number of recognizable matches. It is difficult to decide which matches are important by just looking at the output because each match represents a chunk of duplicated lines or symbols, which lacks context.

The number of recognizable matches for simple line matching is even lower. All exactly matching lines are reported. Visualization can be used to detect the interesting duplicates. However the lack of parameterization makes it more difficult than the parameterized techniques to detect altered duplicates.

1.4.4 How does it perform?

When using a detection technique one wishes to balance the amount of usable information that one can derive, with the time and memory one invested. Therefore you need to establish the performance of each technique and identify performance bottlenecks. This question addresses how the execution time of each technique relates to its input.

Because the actual performance of a technique depends on many factors like implementation and testing platform, calculating the theoretical time complexities is feasible. For both line matching techniques this results in a time complexity of $O(n^2)$ because each line is compared with each other line resulting in an exponential complexity. Parameterized matching using suffix trees on the other hand, has a complexity of $O(|\Pi|*n)$ (with $|\Pi|$ is the number of parameter symbols) as was formally proven by Baker[2]. As a last technique we studied the time complexity of the metric fingerprint technique which shows a time complexity of $O(m^2)$ when a simple comparison is used to compare the m fragments.

1.4.5 Conclusion regarding applicability of the detection techniques

• Simple line matching (representative for the string-based techniques) gives a crude overview of the duplicated code that is quite easy to obtain, hence is most appropriate during problem detection and problem assessment.

• Parameterized matching (representative for the token-based approaches) provides a precise picture of a given piece of duplicated code and is robust against rename operations. Therefore it works best in combination with fine-grained refactoring tools that work on the level of statements.

• Metric fingerprints (representative for the parse-tree based techniques) are very good at revealing duplicated subroutines, irrespective of small differences, hence work best in combination with refactoring tools that work on the method level (i.e. Remove Method and Pull up method);

1.5 Summary

Software clones, its types and problems with clones are discussed in this chapter. Further, various clone detection techniques and their applicability are explored. Algorithmic details of clone detection techniques are considered in the next chapter.

Chapter 2

Approaches To Clone Detection

The various approaches to clone detection discussed are

- A language independent approach to detecting clones
- Substring matching for clone detection
- Clone Detection using Abstract Syntax Trees (ASTs)

Depending on user situation any one or hybrid technique can be applied for clone detection. Language independent approach can be used at even design level. Substring matching is easy to understand and implement but may suffer from problems in similar keywords. Abstract syntax tree parsing and code structure for clone detection but is difficult to implement.

2.1 A language independent approach to detecting clones

Language dependency is a big obstacle when it comes to the practical applicability of duplication detection. Thus there arises a need to employ a technique that is as simple as possible and prove that it is effective in finding duplication. [16]

2.1.1 Algorithm for language independent approach

Clone detection is always a two-step process. First, source code is transformed into an internal format. Second, a more or less sophisticated comparison algorithm is then performed on the internal data. In this case, the code is only slightly transformed using string manipulation operations. To compare the transformed lines, basic string matching operations in used.

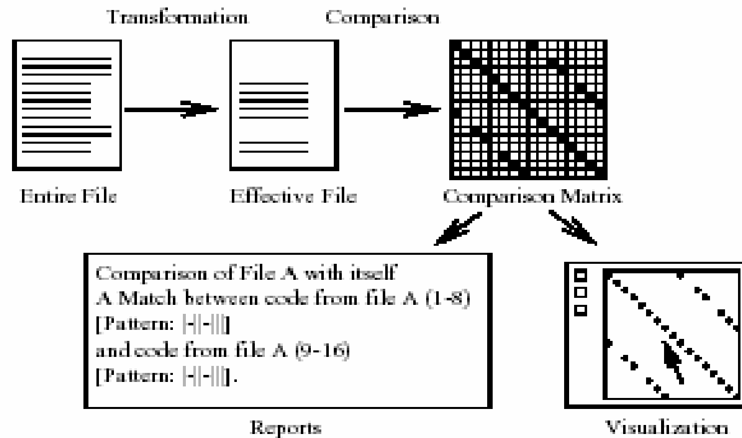


Figure 2.1 - Overview of the language independent approach.

2.1.1.1 Source code transformation

Two decisions must be made: the nature of the transformation and the size of the source code fragment that will be the entity of the incidental comparison. One line of source code is chosen as code fragment entity on which the algorithm is based. The choice is on the one hand motivated by the consideration that the important copy and paste performed by programmers include one or more lines and on the other hand that preprocessing can be kept simple. To stay language independent, the approach refrains from code transformation to more abstract formats like AST's [3] which have to employ parsing, or parameterized strings which need at least a lexer [18] [19]. The transformation this approach applies to a code fragment is minimal and stays in the realm of string manipulation: Comments and all white space are removed until a condensed form is obtained. The transformation reduces the entire file to an ordered collection of effective lines that will be compared against itself and line collections from other files.

2.1.1.2 Comparison algorithm

Every entity (transformed source line) needs to be compared with every other entity. The comparison of two entities is done by string matching. The result is a boolean true for an exact match and a false otherwise. This value is stored in a matrix (see Figure 2.1), taking the coordinates that the two compared entities have in their respective ordered collections as the matrix coordinates

for the comparison result. It is interesting to note that after the comparison process, the matrix only contains matches of individual lines and not yet of whole sequences.

2.1.2 Visualization

The matrix created by the comparison can be visualized using scatter-plots, which were first used by geneticists looking for similar strings of DNA. Such “dot drawings” allow immediate recognition of typical situations. Some interesting configurations formed by the dots in the matrices are the following:

- Diagonals of dots indicate copied sequences of source code.
- Sequences that have holes in them indicate that a portion of a copied sequence has been changed.
- Rectangular configurations indicate periodic occurrences of the same code

Note that due to the line-based comparison, repeated matches of either structural code elements that occur alone on one line (e.g. the break;) or minimal “idioms” (e.g. the frequent C-line int i;) spread spurious dots all over the matrix. To get rid of this noise rather than duplication, we have two possibilities:

- Remove such lines up front by running a filter over the input before the comparison process.
- Use a pattern matcher to sweep over the matrix and remove single dots.

2.1.3 Pattern matching to extract copied sequences

The algorithm as stated above does not catch duplicated code that was changed inside one line of code. In a sequence of copied code that is compared with the original sequence, a changed line shows up as a hole in the diagonal match pattern. To cope for this weakness when extracting whole copied sequences, a pattern matcher is run over the matrix which captures diagonal lines and allows holes up to a certain size in the middle of the line. The sequence extraction is run automatically and produces textual reports containing the detailed locations of the duplication.

2.1.4 Textual reports

The report presents matched sequences that were found in two comparisons. First, the participants of the comparison are identified. Then, a summary of all the sequences found is presented. Finally, all matched sequences are listed in detail Reports are important since they provide the exact location of the duplicated code. This information is handy for the maintainer that has to work with the code. Reports are also the basis for computing more abstract numbers like duplication percentages.

2.1.5 Visualization vs. Automated detection

The advantage of the visualization over the automated pattern matching is twofold:

- First, images of duplication can be striking and allow to grasp situations immediately.
- Second, visualization allows an exploratory approach to the investigation of the duplication situation in a system. Exploratory in the sense that unknown configurations attract the eye of the user and lead to unexpected findings, whereas pattern matching only catches preprogrammed, known configurations.

2.2 Substring matching for clone detection

The substring approach is to consider the source code as text and analyze it the way documents are analyzed [6]. This is closer to the way software developers and maintainers see code and provide, in a language independent manner. A powerful set of tools can access information inaccessible to the compiler based approaches. Identifying software clones and understanding how software changes between releases are two important issues for maintainers where text based approach is likely to be useful.

2.2.1 Algorithm for substring matching for clone detection

The approach can be summarized as:

1. For each file being considered, apply a text-to-text transformation to discard characters not to be considered for matching. For exact matching, this is an identity transformation (output equals input). Various types of approximate matching can be accommodated by discarding different parts of the input.
2. Generate a set of substrings that together cover the source (i.e. every character of text appears in at least one substring).
3. Identify matching substrings (i.e. those with same sequence of characters).
4. Transform the database of raw matches into a form that more concisely expresses the same information.
5. Perform task-specific data reduction.
6. Summarize high-level matches.

Steps (2) and (3) are information collection phases, (4) is an information-preserving transformation, (5) is an aggregation and simplification phase, and (6) presents results in useful form. The algorithm steps are described as:

2.2.1.1 Text-to-text source transformation

To ensure that the approach scales up, it is based on the idea of exact matching. This restriction means that sorting can be used to bring together simultaneously all identical substrings. This approach will not work for approximate matching and they require more complex (and more expensive) approaches.

Exact matching can be generalized in a weak fashion to handle one form of approximate matching that can be referred to as "exact matching on partial information". A text-to-text transformation is applied to the source before substrings are generated. The following candidate transformations seem plausible:

1. Remove all white space characters (blank, tab, carriage return, line feed). The resulting matches are not sensitive to different layout on the page affected using only white space.
2. Remove all white space except for line separators. Results are similar to (1), but the line boundaries are preserved.

3. Replace each (maximal) sequence of white space characters by a single blank. There are variants of this approach based on (1) and (2). Matches will then be found if white space is used in the same places.
4. Remove comments.
5. Replace each identifier by an identifier marker.
6. Various combinations of the above.

2.2.1.2 Generation of candidate substrings

The second phase of the process involves producing a collection of substrings that will be checked for matches. This must be done carefully, since producing too many substrings causes much more processing to be required and producing too few will result in matches being missed completely. It is important that the rate at which substrings are generated can be controlled by the user of the system. Average length measured in number of characters or number of lines is also a necessary parameter. There is some subtlety in how possible substrings are sampled when it is not feasible to produce all substrings of the requested length. Thus, the sampling strategy can depend only on the string itself and a bounded number of characters before and after the string.

2.2.1.3 Identification of raw substring matches

This is a straightforward sorting of a file containing the content of the substring and an indication of its origin. As a result, a fingerprinting scheme based on the Karp and Rabin string matching approach is used. With this approach a 16-byte fingerprint stands in for the content of the substring. Fingerprints agree if the underlying substrings agree, and disagree with high probability if the underlying substrings disagree. The probability of the false matches is kept very low through careful design and a long fingerprint.

2.2.1.4 Information-preserving simplification of match database

The raw substrings used for matching purposes have more or less the same length and overlap. Matches larger than the substring length will be represented by many raw matches. To simplify the database of matches and facilitate later processing, it is important to replace the raw substring and

match information by a new minimal set of non-overlapping substrings and matches that preserves the information obtained in the matching phase. This set of substrings has the minimum number of substrings and each substring is of maximum length. Reducing the number of substrings or lengthening any of the substrings should not result either in overlapping substrings or in a loss of match information.

2.2.1.5 Data reduction

Another way of looking at a set of matches on non-overlapping substrings is an association between substring content (sequence of characters) and a set of places such that the set of places collectively constitutes the total source. Since each place is an offset in a file, this information can be naturally reduced to an association between substring content and a set of files in which it occurs. For each subset of files one can then total the lengths of substring contents that occur in exactly that set of files.

2.2.1.6. Presentation of file clusters and multi-file matches

If one considers whose nodes are files and where arcs have been added whenever a match involves a set of files containing a given pair, then one can define clusters of files corresponding to the connected components in the graph. Intuitively, it might seem that these "clusters" might be large and uninformative but it is not true. Each cluster corresponds to a sharing of text that is comprehensible to the maintainer without almost any noise.

2.3 Clone detection using abstract syntax trees (ASTs)

The basic problem in clone detection is the discovery of code fragments that compute the "same" result. To do this, it is needed to first fragment the program in parts willing to be compared, and then determine if fragment pairs are equivalent. Since determining that even a single fragment halts is impossible, it can not be determined that two arbitrary program fragments halt under the same circumstance, and thus it is impossible in theory to decide that they compute identical results. Since false negatives are acceptable then deep semantic analysis conservatively bounded by time limits could be used

for equivalence detection. However, considerable infrastructure may be required—in the form of semantic definitions, theorem provers, etc. In practice, analyzers are willing to give up detecting complete semantic equivalence because many clones come about due to copy-and-paste editing processes. Simpler definitions of code equivalence may suffice if too many false positives are not produced. This suggests clone detection by more syntactic methods. One can go as far as comparing source lines. Source line equality assumes that the cloning process introduced no changes in identifiers, comments, spacing, or other non-semantic changes, and thus limits clone detection to exact matches.

Consequently, it fails to detect near-miss clones. Closer to full semantics but still a practical possibility would be to compare program representations in which control and data flows are explicit. Semantic Designs is building transformational tools to help modify large software systems.

Such tools typically parse source programs into ASTs as a first step before transformation [3]. Due to the early product state of tools, investigation is made by comparing syntax trees. This has the attraction of directly avoiding confusing but uninteresting changes at the lexical level. As a first step in the clone detection process, the source code is parsed and an AST is produced for it. After that, the main algorithm is to find clones.

- The purpose of this algorithm, the Basic algorithm, is to detect sub-tree clones.

2.3.1 Finding Sub-tree Clones

In principle, finding sub-tree clones is easy: compare every subtree to every other sub-tree for equality. In practice, several problems arise:

- near-miss clone detection,
- sub-clones,
- scale.

Near misses is handled by comparing tress for similarity rather than exact equality. The sub-clone problem is that we wish to recognize maximally large clones, so clone subtrees of detected clones need to be eliminated as reportable clones. The scale problem is harder. For an AST of N nodes, this

comparison process is O(N^3), and, empirically, a large software system of M lines of code has N=10*M AST nodes. If comparing sequences of trees is considered, the process is O(N^4)!). Thus, the amount of computation becomes prohibitively large. In order to tackle this problem it is possible to partition the sets of comparisons by categorizing sub-trees with hash values. This allows the straightforward detection of exact sub-tree clones. If we hash sub-trees to B buckets, then only those trees in the same bucket need be compared, cutting the number of comparisons by a factor of B. B is usually chosen of approximately the same order as N; in practice, B=10% N means little additional space at great savings in terms of computation. It has been observed that the cost of comparing individual trees averages close to a constant, rather than O(N), and so hashing allows this computation to occur in practice in time O(N). This approach works well when we are finding exact clones. When locating near-miss clones, hashing on complete subtrees fails precisely because a good hashing function includes all elements of the tree, and thus sorts trees with minor differences into different buckets. This problem is solved by choosing an artificially bad hash function. This function must characterized in such a way that the main properties one wants to find on near-miss clones are preserved. Near miss clones are usually created by copy and paste procedures followed by small modifications. These modifications usually generate small changes to the shape of the tree associated with the copied piece of code. Therefore, this kind of near-miss clone often has only some different small sub-trees. Based on this observation, a hash function that ignores small sub-trees is a good choice. For example , use a hash function that ignores only the identifier names ,i.e. leaves in the tree. Thus hashing function puts trees which are similar modulo identifiers into the same hash bins for comparison.

Rather than comparing trees for exact equality, compare instead for similarity, using a few parameters. The similarity threshold parameter allows the user to specify how similar two sub-trees should be. The similarity between two sub-trees is computed by the following formula:

Similarity = 2 x S / (2 x S + L + R)

where:

S = number of shared nodes

L = number of different nodes in sub-tree 1

R = number of different nodes in sub-tree 2

The mass threshold parameter specifies the minimum subtree mass (number of nodes) value to be considered, so that small pieces of code (e.g., expressions) are ignored. These methods are combined to detect sub-tree clones, giving the Basic clone detection algorithm.

2.3.2 Basic clone detection algorithm

```
1. Clones=∅
2. For each subtree i:
        If mass(i)>=MassThreshold
        Then hash i to bucket
3. For each subtree i and j in the same bucket
        If CompareTree(i,j) > SimilarityThreshold
                Then { For each subtree s of i
                                If IsMember(Clones,s)
                                Then RemoveClonePair(Clones,s)
                        For each subtree s of j
                                If IsMember(Clones,s)
                                Then RemoveClonePair(Clones,s)
                AddClonePair(Clones,i,j)
```

The Basic algorithm is straightforward. In Step 2, the hash codes for each sub-tree are computed to place them in the respective hash bucket. This step ignores small subtrees, thus implementing the mass threshold in a way that further reduces the number of comparisons required considerably, as the vast majority of trees are small. After that, every pair of sub-trees located in the same hash bucket is compared, if the similarity between them is above the specified threshold, the pair is added to the clone list, and all respective sub-clones are removed.

2.4 Summary

Algorithmic details of three major clone detection techniques namely :

- language independent approach,
- substring matching and
- using abstract syntax trees

are discussed. In this chapter the details regarding steps involved in these techniques are also discussed.

The proposed model is described in the next chapter.

Chapter 3
Proposed Model: Three Stage Transformation For Clone Detection

---

In the proposed model the string matching algorithm for detection and reporting of clones is used. The string based approach was chosen because: [6] [16]

1. String based approach scores much high over token based or parse-tree approaches because of language independent nature. String based detection tool can be used over a wide variety of code samples without having to make significant modifications in the project.
2. String based clone detection algorithms are relatively less complex and are hence much smoother to understand, implement and maintain.

The user is supposed to supply the filenames from the specified User Interface to the system. After obtaining the valid filenames, the system proceeds to create File objects from the filenames. Now once the file objects have been created the code files need to be read into hash-tables so that identity of each line is distinct. The files are read as a key-value pair of line-number and the code line. Once the files have been housed in hash-tables, the lines of code are further formatted in three stages as described in sections 3.1.1 – 3.1.3.

3.1 Approach: Three stage transformation for clone detection

Before the actual comparisons for equality are made, the lines of the files are made to go through a three stage transformation. The transformations are:

- Pretty Print Formatting
- Dictionary replacement
- Tokenization of variables

3.1.1 Pretty print formatting

Firstly we need to remove any tab spaces that occur in the source files. Secondly we need to remove any white spaces that occur in lines of code. Programmers usually use tab spaces and white spaces in their code so that the code becomes more readable and presentable. But this fact may cause havoc in a string matching algorithm, since just inserting a single white space in a copied file may make it unrecognizable. Pretty print formatting helps to overcome such formatting differences, and the following lines are detected as clones:

File1: cin>>param;

File2: cin>>   param;

3.1.2 Dictionary replacement

In a typical scenario, there might be functions with names "add" and "total" which have similar functionality. The user might suspect identical functionality and therefore want them to be identical. To overcome such linguistic difference of nomenclature the system provides a dictionary replacement tool. The user (analyzer) may provide entries in a database for such words. For example for add and total, the user might use the word sum in dictionary. Dictionary is a simple database provided in MS-Access format (mdb). The user can supply replacement values in the database. Due to dictionary replacement the following lines are detected as clones:

File1: inline double add (double x, double y)

File2: inline double total (double x, double y)

3.1.3 Tokenization of variables

In a typical industrial application it might happen that a programmer copies some code from an existing module and use it in his module, and in doing so rename variables to make them more meaningful in his context of use. To tackle this situation, we should make the comparisons to be independent of nomenclature of variables. As a solution we tokenize all the variables defines, i.e. replace all the variables with $P. Thus we generate a p-string in the memory. After attaining this form the code samples become independent of

variables' nomenclature and our effort concentrates on program structure. Due to tokenization of variables the following lines are detected as clones:

```
File1: inline double difference (double funcparam1,
double funcparam2)
File2: inline double difference (double functionparam1,
double functionparam2)
```

3.2 Modeling of the system

The model of the system is shown in four views, namely:

1. Component Diagram
2. Use Case Diagram
3. Activity Diagram
4. Data Flow Diagram

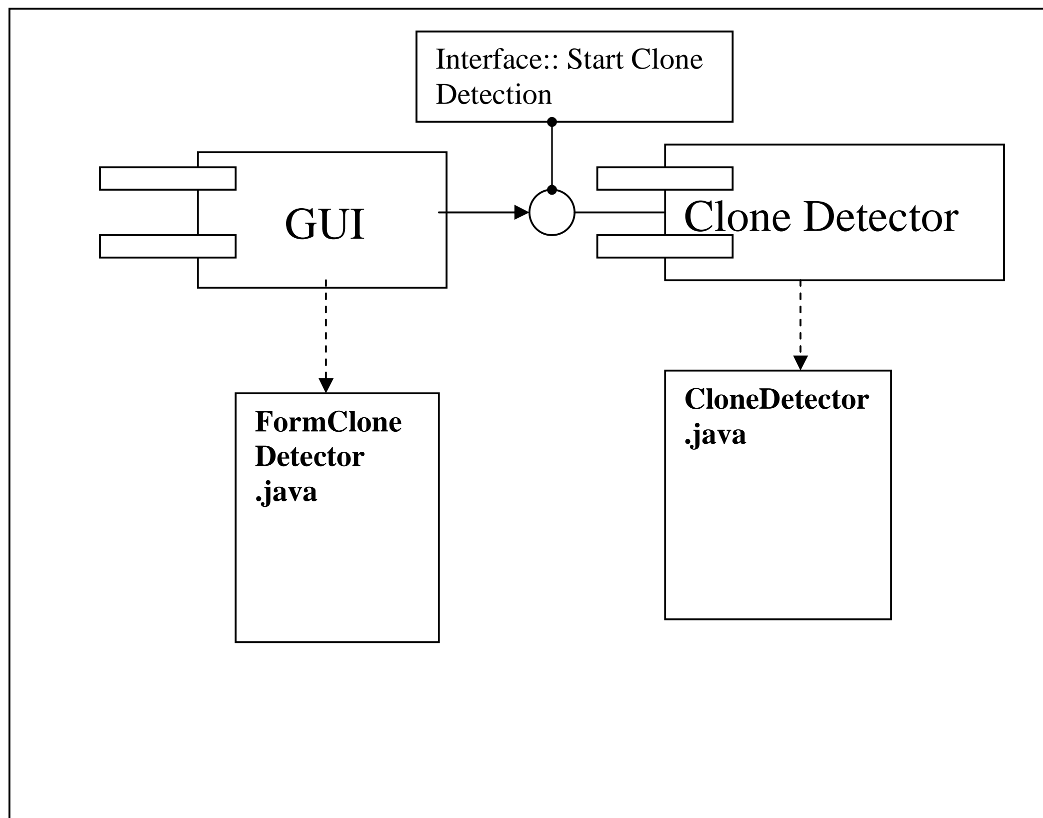


Figure 3.1 - Component Diagram

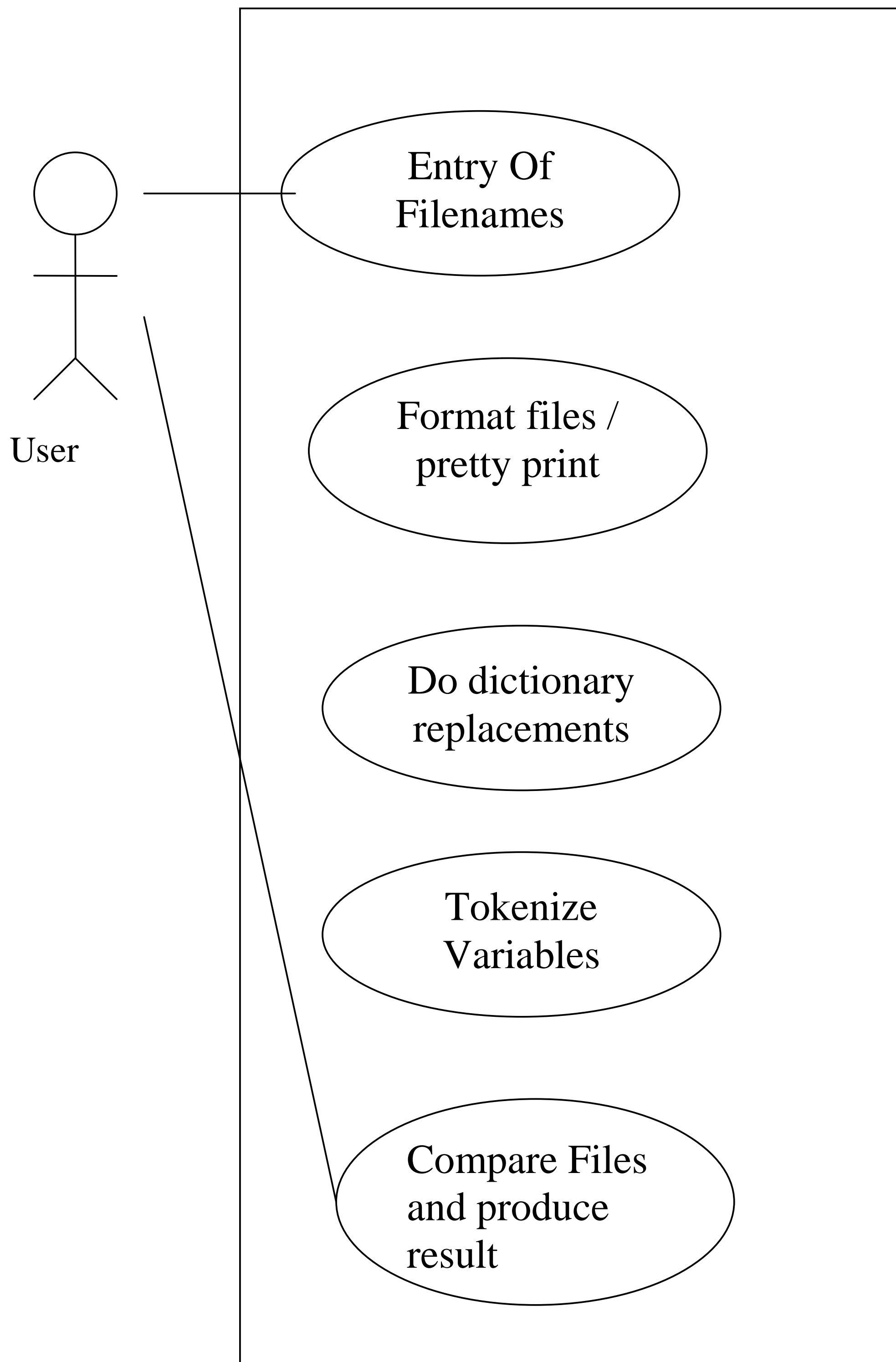


Figure3.2 - Use Case Diagram

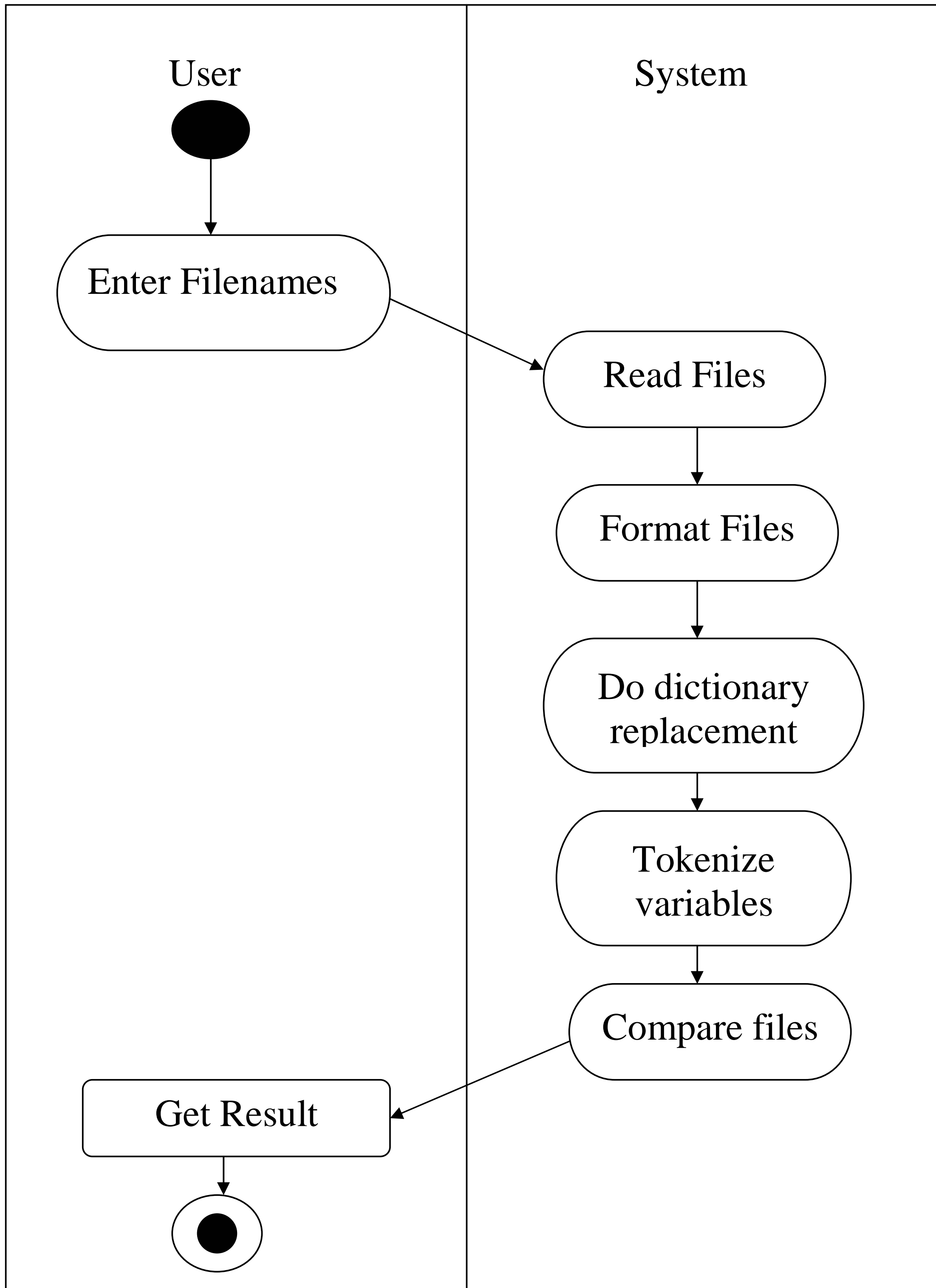


Figure 3.3 - Activity Diagram

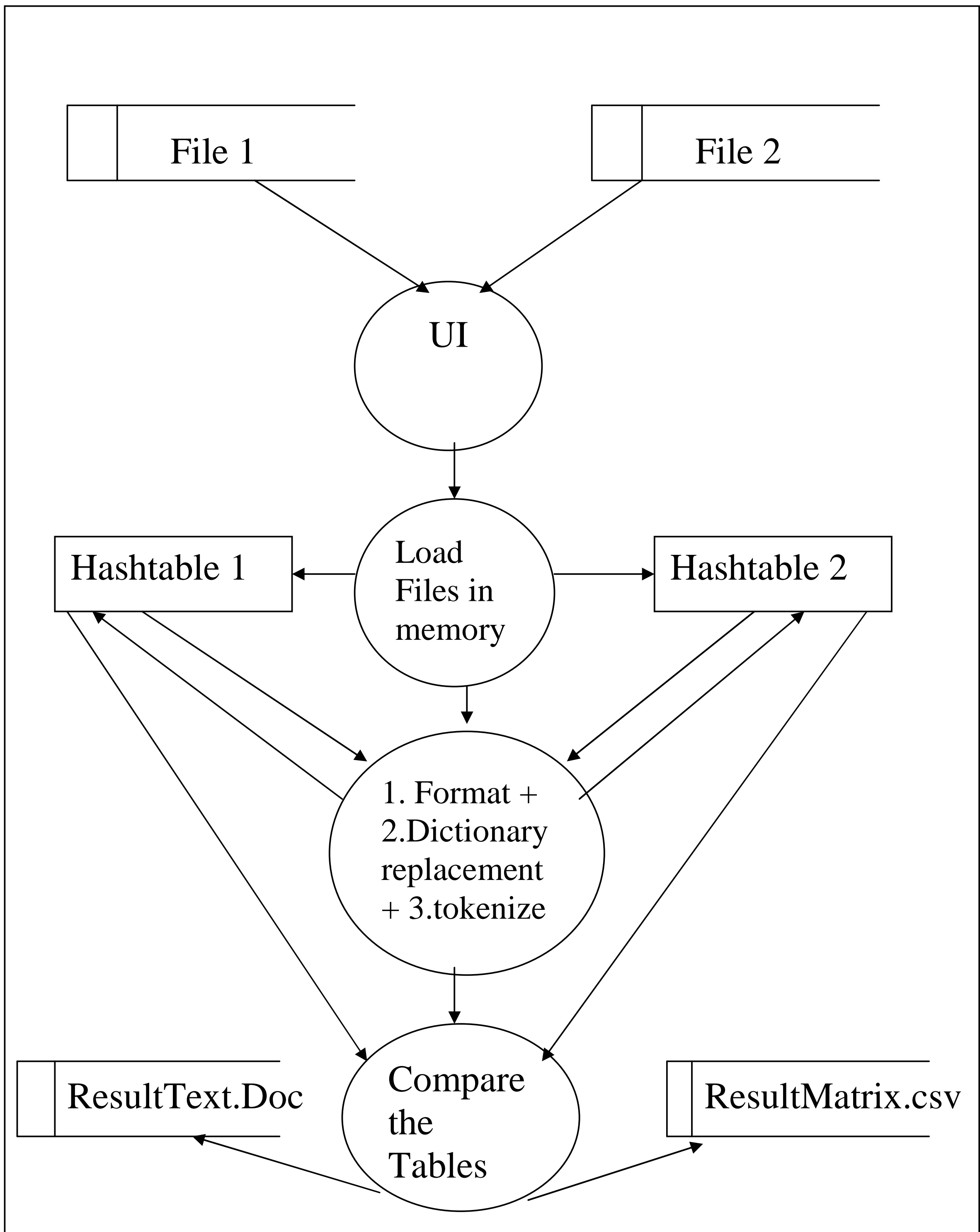


Figure 3.4 - Data Flow Diagram

3.3 Implementation

For implementation purpose, java programming language has been used. The User-Interface is designed using the MS wfc classes. The project is packaged into a Win32 executable. The class structure has been implemented so as to isolate the User-Interface and the functionality. The UI is implemented in a class named **FormCloneDetectorUI**. The functionality is implemented in class named **CloneDetector.** Separating the implementations of the UI and the implementation, creates an easily understandable and manageable code. The API description of the two classes is given as:

3.3.1 Class structure

public class FormCloneDetectorUI extends Form

//Constructor

public FormCloneDetectorUI()

//initializes the UI

initForm();

//cleanup called on delete

public void dispose()

//Click of "…" button to get first file's name

private void buttonFileName1_click(Object source, Event e)

//Click of "…" button to get second file's name

private void buttonFileName2_click(Object source, Event e)

//click of "Start Clone Detection" Button

private void buttonStartCloneDetection_click(Object source, Event e)

//disable buttons once the detection process has started

disableButtons()

//enablebuttons once the detection process has ended
enableButtons();

public class CloneDetector

//private constructor
private CloneDetector()

//Constructor
public CloneDetector(String fileName1, String fileName2)

//Create a hash-bucket for the File
private void createHashBucket(HashTable hash, File file)

//Remove tab spaces from the hash bucket entries
private void removeTabsFromHashBuckets(HashTable hash)

//Remove white spaces from the hash bucket entries
private void removeSpaceFromHashBuckets(HashTable hash)

//Do dictionary replacements
private void doDictionaryReplacement(HashTable hash)

//Tokenize variables
private void tokenizeVariables(HashTable hash, List list)

//Start the Clone Detection Process
public String startCloneDetection()

3.4 User Interface

The user interface is intuitive and guides the user by mere looking at the screen. The User Interface screen looks like:

Figure 3.5 - GUI for the clone detection tool.

The user is required to enter two filenames for the detector to start working on detecting code cloning in the two files. The user can-not enter directly into the text-boxes for entering filenames, but has to launch an open-file dialog box from the 3-dot button and select a file from the dialog. For Open file dialog see Figure B.

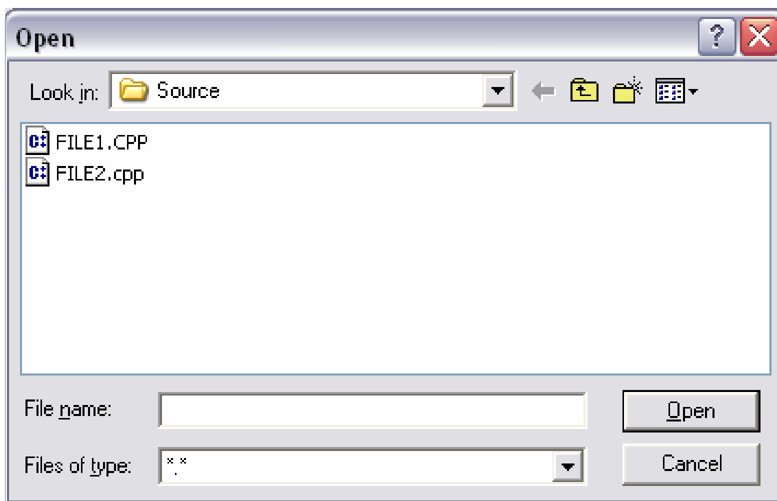


Figure 3.6 - Open File Dialog.

After selecting files from the Open-file dialog, the user hits the 'Start Clone Detection' bar to start the actual clone detection process on the two files specified. In case the 'Start Clone Detection' bar is hit without specifying either of two files the system shows an error like the one shown in Figure C.

Figure 3.7 - Error in case filename field(s) are left empty.

After supplying the filenames and hitting the 'Start Clone Detection' button the user has to wait for confirmation by the system. The system supplies the user with a MS-Excel format file which contains the result matrix for the clone detection process and a rtf format file that contains a clone summary See Figure D.

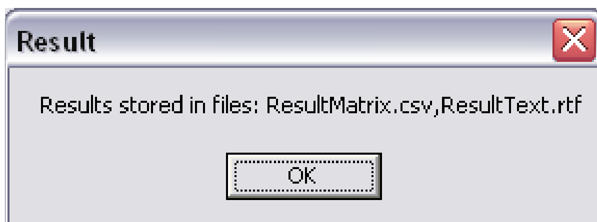


Figure 3.8 - Result of the clone detection process.

3.5 Output

On the start of the code duplication detection process, the system firstly creates an MS-Excel format file which acts as the result matrix. Each line in the first hash-table is compared to each other line in the second hash-table. Thus if there are n lines in the first code file and m lines in the second, the algorithm takes on order (n x m) or $O(n^2)$. For every duplicate line the system puts a "1" entry in the result matrix, otherwise a "0" entry is made in the result matrix. The result matrix can be viewed in MS-Excel application.

Moreover the system also creates a file in rtf format which contains detailed information of the duplicate lines in code. The system reports every duplicate pair with their line number. In the end of the file, the system reports statistics like number of lines in each files, total number of comparisons made by the system and the number of lines found identical.

Chapter 4

Conclusions And Future Scope

Duplicated code is a phenomenon that occurs frequently in large systems. The reasons why programmers duplicate code are manifold and include the following reasons:

- Making a copy of a code fragment is simpler and faster than writing the code from scratch. In addition, the fragment may already be tested so the introduction of a bug seems less likely.
- Evaluating the performance of a programmer by the amount of code he or she produces gives a natural incentive for copying code.
- Efficiency considerations may make the cost of a procedure call or method invocation seem too high a price.

The various clone detection techniques and the various places where they are preferable:

• *Simple line matching,* representative for the string-based techniques, gives a crude overview of the duplicated code that is quite easy to obtain, hence is most appropriate during problem detection and problem assessment.

• *Parameterized matching,* representative for the token-based approaches, provides a precise picture of a given piece of duplicated code and is robust against rename operations. Therefore it works best in combination with fine-grained refactoring tools that work on the level of statements.

• *Metric fingerprints,* representative for the parse-tree based techniques, are very good at revealing duplicated subroutines, irrespective of small differences, hence work best in combination with refactoring tools that work on the method level i.e. *Remove Method* and *Pull up method*.

The three stage transformation hybrid string matching algorithm for detection and reporting of clones is proposed. This hybrid based approach was chosen because:

- During microscopic investigation the user might find out methods with distinct names but performing the similar functions. The analyzer may use the dictionary feature so that the detection process considers them as having the same name.
- Any conflicts in nomenclature of variables has been taken care of in the tokenization of variables which allows detection of code segments where variables may have been renamed.

4.1 Case Study: Application of the three stage clone detection tool on a set of files

Two files written in C programming language are considered here for testing of proposed algorithm. Both the files perform trivial mathematic functions but are a good means of exploring and testing the functionality of the three stage clone detection tool.

4.1.1 Input Files

The files of code are as follows:

File 1:

```
#include<iostream.h>
#include<conio.h>
inline double add (double x, double y)
{
return (x+y);
}
inline double difference (double funcparam1, double funcparam2)
{
return funcparam1 - funcparam2;
}
void main()
{
double param1,              param2;
clrscr();
cout<<"Enter 2 nos. :";
cin>>param1>>param2;
cout<<"\nThe sum is :";
cout<<add(param1,param2);
cout<<"\nThe Dirrerence is: ";
cout<<difference(param1,param2);
getch();
}
```

File 2:

```
#include<iostream.h>
#include<conio.h>
inline double total (double x, double y)
{
return (x+y);
}
inline double difference (double functionparam1, double functionparam2)
{
return functionparam1 - functionparam2 ;
}
void main()
{
double param1, param2;
double testVariable;
clrscr();
cout<<"Enter 2 nos. :";
cin>>param1>>param2;
cout<<"\nThe sum is :";
cout<<total(param1,param2);
cout<<"\nThe Dirrerence is: ";
cout<<difference(param1,param2);
getch();
}
```

These paths and names of these files are supplied to the tool. The tool proceeds by reading there files into memory and applying the three step transformation to the lines of the files. The three stages of transformation are

- Pretty Print Formatting
- Dictionary replacement
- Tokenization of variable,

After the three stage transformation of the lines of both the files the tool proceeds to compare the lines in the file. The tool then produces an MS-Excel format file which contains the result matrix of the comparison and a RTF format file which contains the detailed report of the comparisons.

4.1.2 Output Files

The system creates two files to indicate the results. One file is in MS-Excel Csv format, and the other in Rtf format. The contents of these files are as follows.

4.1.2.1 MS-Excel format file:

The rows in the matrix represent the lines of file1 and the columns represent the lines of file2. A 0 entry means mismatch whereas a 1 entry in the matrix means a match of two lines.

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
```

4.1.2.2 RTF format file

The rtf format file gives a detailed description of every 1 entry in the matrix. It logs the identical lines into the rtf document. In the end of the file it logs important statistics like the number of lines in each file, the number of comparisons and the number of identical lines. The rtf files for the case study look like:

```
Identical Lines::::

File1 Line1 and File2 Line1
File1: #include<iostream.h>
File2: #include<iostream.h>

File1 Line2 and File2 Line2
File1: #include<conio.h>
File2: #include<conio.h>

File1 Line3 and File2 Line3
```

```
File1: inline double add (double x, double y)
File2: inline double total (double x, double y)

File1 Line4 and File2 Line4
File1: {
File2: {

File1 Line4 and File2 Line8
File1: {
File2: {

File1 Line4 and File2 Line12
File1: {
File2: {

File1 Line5 and File2 Line5
File1: return (x+y);
File2: return (x+y);

File1 Line6 and File2 Line6
File1: }
File2: }

File1 Line6 and File2 Line10
File1: }
File2: }

File1 Line6 and File2 Line23
File1: }
File2: }

File1 Line7 and File2 Line7
File1: inline double difference (double funcparam1,
double funcparam2)
File2: inline double difference (double functionparam1,
double functionparam2)

File1 Line8 and File2 Line4
File1: {
File2: {

File1 Line8 and File2 Line8
File1: {
File2: {

File1 Line8 and File2 Line12
File1: {
File2: {

File1 Line9 and File2 Line9
File1: return funcparam1 - funcparam2;
```

```
File2: return functionparam1 - functionparam2 ;

File1 Line10 and File2 Line6
File1: }
File2: }

File1 Line10 and File2 Line10
File1: }
File2: }

File1 Line10 and File2 Line23
File1: }
File2: }

File1 Line11 and File2 Line11
File1: void main()
File2: void main()

File1 Line12 and File2 Line4
File1: {
File2: {

File1 Line12 and File2 Line8
File1: {
File2: {

File1 Line12 and File2 Line12
File1: {
File2: {

File1 Line13 and File2 Line13
File1: double param1,         param2;
File2: double param1, param2;

File1 Line14 and File2 Line15
File1: clrscr();
File2: clrscr();

File1 Line15 and File2 Line16
File1: cout<<"Enter 2 nos. :";
File2: cout<<"Enter 2 nos. :";

File1 Line16 and File2 Line17
File1: cin>>param1>>param2;
File2: cin>>param1>>param2;

File1 Line17 and File2 Line18
File1: cout<<"\nThe sum is :";
File2: cout<<"\nThe sum is :";

File1 Line18 and File2 Line19
```

```
File1: cout<<add(param1,param2);
File2: cout<<total(param1,param2);

File1 Line19 and File2 Line20
File1: cout<<"\nThe Dirrerence is: ";
File2: cout<<"\nThe Dirrerence is: ";

File1 Line20 and File2 Line21
File1: cout<<difference(param1,param2);
File2: cout<<difference(param1,param2);

File1 Line21 and File2 Line22
File1: getch();
File2: getch();

File1 Line22 and File2 Line6
File1: }
File2: }

File1 Line22 and File2 Line10
File1: }
File2: }

File1 Line22 and File2 Line23
File1: }
File2: }

Result Summary
Lines in File1 = 22
Lines in File2 = 23
Total Comparisions = 506
Identical Lines = 34
```

4.1.3 Analysis of results

The tool compares each and every line pair for the files of code. i.e for 22 lines in file1 and 23 lines in file2 it makes (22 * 23) i.e. 506 comparisons. It would be interesting to note how the three stages of transformation help in better detection of clones. The following pieces of code would not have been reported if the three stages were not applied to the files. Such detected lines along with the responsible stage of transformation are:

4.1.3.1 Pretty print formatting

```
File1 Line13 and File2 Line13
File1: double param1,        param2;
File2: double param1, param2;
```

Clearly the string matching algorithm would treat the lines as distinct due to mismatch of tab and blank spaces, but they are rendered identical due to pretty print formation.

4.1.3.2 Dictionary replacement

```
File1 Line3 and File2 Line3
File1: inline double add (double x, double y)
File2: inline double total (double x, double y)

File1 Line18 and File2 Line19
File1: cout<<add(param1,param2);
File2: cout<<total(param1,param2);
```

The dictionary empowers the detector so that the analyzer can supply synonyms and they are considered as duplicates. Clearly in the example the functions are similar and just the name has been changed.

4.1.3.3 Tokenization of variables

```
File1 Line7 and File2 Line7
File1: inline double difference (double funcparam1,
double funcparam2)
File2: inline double difference (double functionparam1,
double functionparam2)

File1 Line9 and File2 Line9
File1: return funcparam1 - funcparam2;
File2: return functionparam1 - functionparam2;
```

Here it can be observed that the samples of code are similar with the exception that the names of variables have been changed. Thus the tokenization of variables makes the approach identify code fragments where the variables may have been renamed to suit the needs of the method where the code has been copied.

4.2 Conclusions

- The ambiguity of exact string match has been plugged using the three stage transformations which widen the match result space.

- Inherent simplicity of the string matching is kept intact. Partial tokenization is applied on variable names which empowers the approach with independence of variable names.
- The tool misses to identify clones if the variable types are changed in copied code. For example variables of a particular class may be replaced by some derived class. Now the execution behavior for the code segments is identical but the tool misses to identify the clones.

4.3 Future scope

- Currently the proposed and implemented model simply reports the clones or identical lines in source. It does not support features for refactoring the code or removal of clones from code. More work is seen in this area of automating the correcting of problem of cloning which is detected.
- The reporting mechanism reports the results in textual format. *Scatter plot* visualizations are a helpful means in analyzing duplication. The implementation needs to be extended to incorporate visual output which is much simpler to investigate than the text files. Moreover scatter-plots provide a means of intuitive judgment to the analyzer.
- The implementation uses a dictionary replacement to replace certain words (phrases) so that any renaming by the programmers in clones may be detected. The implementation needs to be extended so that the user can modify the dictionary database and supply values to it. The implementation should provide an interface to maintain the dictionary database. Moreover, currently the dictionary database is implemented in MS-Access file database, which can be modified in Access application. The implementation should support the editing/ modification of the database from within the detection tool.
- The implementation tokenizes any variable names defined and used in the input files. Current implementation works for C programming language. The tokenization procedure needs to be extended to incorporate code written in a variety of programming languages like Java, VB, C++ etc.

Bibliography

[1] B. S. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 1992.

[2] B. Baker. On finding duplication and near-duplication in large software systems. In *Working Conference on Reverse Engineering 1995*, 1995.

[3] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of ICSM*. IEEE, 1998.

[4] M. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

[5] J. H. Johnson. Identifying Redundancy in Source Code using Fingerprints. In *Proceedings of CASCON93*, 1993.

[6] J. H. Johnson. Substring Matching for Clone Detection and Change Tracking. In *Proceedings of the International Conference on Software Maintence (ICSM)*, 1994.

[7] K. Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In I. Baxter, A. Quilici, and C. Ver-hoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, IEEE Computer Society, 1997.

[8] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software System Using Metrics*, 1996.

[9] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, June1994.

[10] H. Reubenstein, R. Piazza, and S. Roberts. Separating Parsing and Analysis in Reverse Engineering Tools. In *First Working Conference on ReverseEngineering*, 1993.

[11] G. Antoniol, U. Villano, E. Merlo, , and M. Di Penta. Analyzing cloning evolution in the linux kernel. In *Information and Software Technology* 2002.

[12] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the Sixth International Software Metrics Symposium* 1999.

[13] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Partial redesign of java software systems based on clone analysis. In *The Proceedings of the 6th. Working Conference on Reverse Engineering*, 1999.

[14] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Advanced clone analysis to support object-oriented system refactoring. In *Proceedings of the 7th. Working Conference on Reverse Engineering*, 2000.

[15] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Identifying clones in the linux kernel. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, IEEE Computer Society Press,2001.

[16] St´ephane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, IEEE, 1999.

[17] MichaelW. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the 2000 International Conference on Software Maintenance*, 2000.

[18] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. A token-based code clone detection tool - ccfinder and its empirical evaluation. Technical report, 2000.

[19] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. In *Transactions on Software Engineering 8(7)*,. IEEE Computer Society Press, 2002.

[20] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2001.

[21] K. Kontogiannis, R. De Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection, 1996.

[22] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. EigthWorking Conference on Reverse Engineering*, 2001.

[23] Qiang Tu and Michael Godfrey. An integrated approach for studying software architectural evolution. In *Proceedings of 2002 International Workshop on Program Comprehension (IWPC-02)*, 2002.

[24] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the Eighth IEEE Symposium on Software Metrics*. IEEE Computer Society Press, 2002.

[25] Filip Van Rysselberghe, Serge Demeyer Evaluating Clone Detection Techniques.

List Of Publications

- Kunal Pandove and Rajesh K. Bhatia "Three Stage Transformation for Software Clone Detection" communicated to IEEE/ACM International conference on Automated Software Engineering, Long Beach, California, USA, Nov. 7 –11 2005.