ENHANCING CODE CLONE MANAGEMENT BY PRIORITIZING CODE CLONES

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Ridhi Garg
(801231023)

Under the supervision of:
Mr. Rajkumar Tekchandani
Assistant Professor, CSED

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2014

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Enhancing Code Clone Management by Prioritizing Code Clones*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Mr.*Rajkumar Tekchandani* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(Ridhi Garg)
801231023

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

(Rajkumar Tekchandani)
Assistant Professor,
Computer Science and Engineering Department,
Thapar University,
Patiala.

Countersigned by

(Dr. Deepak Garg)
Head,
Computer Science and Engineering Department,
Thapar University,
Patiala.

(Dr. S. K. Mohapatra)
Dean (Academic Affairs),
Thapar University,
Patiala.

Acknowledgement

I express my gratitude and appreciation to all those who have helped me throughout the duration of my research work. It would not have been possible to complete this research without guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

First and foremost, I would like to express my deep and sincere gratitude to my supervisor Mr. Raj Kumar Tekchandani, Assistant Professor, Computer Science and Engineering Department, Thapar University, Patiala. It gives me immense pleasure to pay my gratitude for his valuable advice, constructive criticism and great patience at all the times.

I owe my sincere thanks to Dr. Deepak Garg, Associate Professor and Head of Computer Science and Engineering Department, Thapar University, Patiala, for his continuous motivation, cooperation and for providing facilities.

I wish to express my sincere thanks to all the staff members and my friends who always supported and encouraged me. I was very fortunate to have an unconditional support from my family. I want to acknowledge the contributions of my parents, for their constant motivation, inspirations and for supporting me spiritually throughout my life. Last but not the least; I would like to thank God for giving me inner peace and strength.

Ridhi

Ridhi Garg

(801231023)

Abstract

Software development is often accompanied by the activities of reusing exisiting software, with or without modification, along with building application from scratch. Software reuse done through copy-paste activities gives rise to duplicated fragments of code called clones. Code clones tend to reduce maintainability of the system being developed and numerous code clone detection techniques and tools have been developed for the same. Detected code clones require appropriate management. Code clone management involves a deep analysis of the code clones in order to make informed decisions regarding their removal.

Clone management can be enhanced by deciding the order for removal of clones. For deciding the order, it takes into account the maintenance overhead that would be incured if the clones were not fixed.

This thesis introduces an approach that takes the detected code clones as input and gives a prioritized list of the same. The code clones are ranked on the basis of their criticality or severity. The approach works by estimating the maintenance overhead in terms of size of clones, frequency of their repitition and cyclomatic compexity. Calculating the maintenance overhead, in advance, estimates the efforts that would be required in maintaining the clones if they were not fixed. Thus the clones having higher maintenance overhead would get a higher priority as compared to others and would be removed or refactored first.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Two softwares resembling each other in their functionality can be termed as software clones. However this definition for cloning stands somewhat ambiguous. Precisely saying, cloning stands for writing a program code in such a way that it mimics given software with respect to its functionality and appearance.

Software development goes through several phases such as: requirement specifications, design, coding, testing and maintenance. Cloning can take place in any of the phase, out of which code cloning is the most prominent one where a code fragment is used more than once to reuse its functionality.

Code cloning is not always copying and reusing a piece of program instead it could sometimes happen accidentally when the programmer is just unaware of the occurrence of clones. So it becomes essential to realize the fact that existence of clones does not simply states the presence of exactly copied fragments rather their presence indicates the functional similarity too that exists between two cloned copies.

Although the dictionary meaning of cloning implies ‘duplicity’ but the main goal of cloning is to save the effort of repeatedly indulging in coding and implementing a piece of code that already exists.

This chapter gives the basic and a brief description on software reuse, software cloning, reasons for introducing clones, consequences of having code clones, types of code clones, clone management, motivation for clone management and organization of the thesis.

1.1 Software Reuse

When a piece of code written at the first place is used in a program being written later, it is known as reuse. Reuse is generally done to minimize the costs in terms of time and effort. Software reuse has been widely accepted and is quite advantageous over building software from scratch.

Component Based Software engineering (CBSE) is a reuse based approach which helps in defining and integrating stand alone components in systems. These components are more abstract than classes and are independent service providers.

Thus the main purpose of reuse is to reduce cost, time, effort, risk and to increase productivity, quality, performance, maintainability and interoperability [1].

1.2 Code Cloning

Basically, code cloning means to reuse a code component that implements a required functionality. This type of reuse is done by means of copying and pasting the required code snippet, sometimes with minor modifications. Thus the similar fragments are termed as code clones and this activity of duplication is called code cloning. Figure 1.1 shows a simple example code clones.

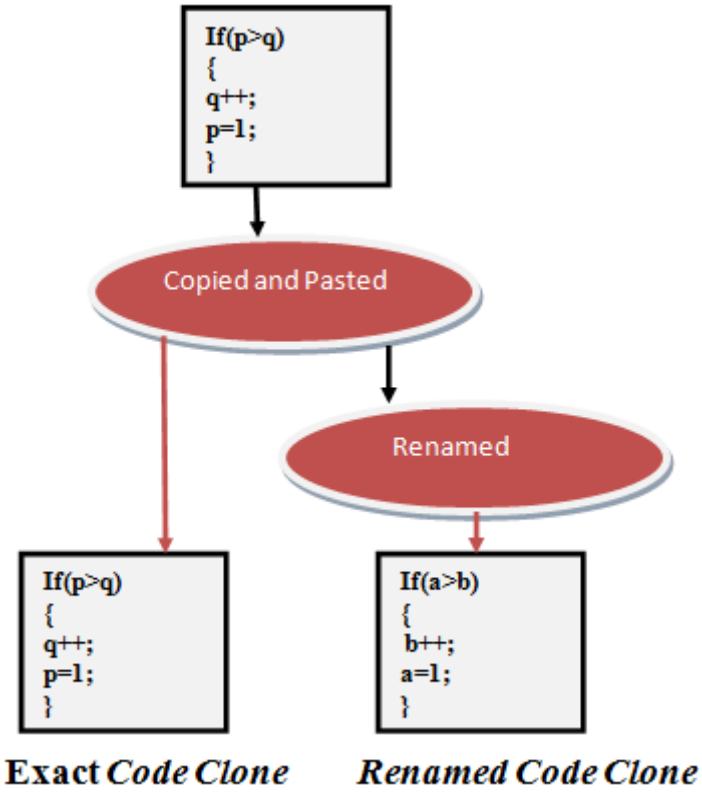


Figure 1.1: Sample code with its clones [2].

According to Ira Baxter, “Clones are segments of code that are similar according to some definition of similarity”.

This adhoc form of duplication has become quite prevalent in the industry. If offers certain benefits to the developer. It saves developer's time and effort in writing the textual code from scratch. It minimizes bug propagation as the code being reused is already tested and is considered error free.

1.3 Reasons for Code Cloning

Code Clones do not originate by themselves. They are either intentionally created or may get introduced accidentally. In both the cases it becomes quite important to deeply understand the root causes for cloning so that the developers might avoid the clones and keep the track of clones which further makes their maintenance easier. Following are some of the reasons of code cloning:

i) Development Strategy

Most of the times, a programmer intentionally introduces code clones with the intent of reusing a logic, a design, a function or the entire program. So, in that case cloning becomes part of developer's strategy in order to save time and effort. Reusing an existing functionality is the most prominent way of code cloning.

ii) Maintenance Benefits

Cloning provides certain maintenance benefits. The foremost benefit is that it reduces the risk factor for the developer in developing new code. More often, a developer is asked to reuse an existing code, for example in financial institutions where the enhancements are done on the existing functionality itself. This is done because of the high risk associated in reengineering a critical fragment which has been already well tested and is assumed to be bug free.

iii) Overcoming the Underlying Limitations

Code Cloning sometimes becomes necessary in order to overcome the underlying limitations. The limitations can be of two types: programmer's limitation and the programming language limitation. The programmer might have to meet certain deadlines and sometimes the programmer lacks the complete knowledge of the given domain. In such cases, the programmer is forced to indulge in copy paste activities.

A programming language might sometimes lack in providing appropriate reuse mechanism or sufficient amount of abstraction. In such cases, the programmer indulges in copy paste and introduces clones in the source code program.

iv) Cloning by Accident

Cloning can take place accidentally too. This happens when two programmers unknowingly develop the same solution for a problem. There can be a case when the same programmer comes up with same solution for similar kind of problems. This kind of repetition is also termed as cloning, although the clones were created unintentionally.

1.4 Consequences of Code Cloning

Although code cloning decreases considerable amount of effort on developer's part and prevents him from developing an application from scratch, yet cloning is not preferred as a good practice. Cloning is considered as a bad smell from maintenance point of view [3] and tends to have a negative impact on the quality of the software being developed. Following are some of the consequences in detail.

i) Increase in maintenance effort and cost

Any kind of bug propagation or unexpected behavior found in any of the fragment would require its correction and updation in all the replicated cloned copies. The numbers of clones are not known previously, so it increases the overhead to first detect the clones and make the necessary corrections in all the detected code clones. Thus, use of cloning leads to maintenance overhead in terms of cost and effort.

ii) Increase in defect probability

The simplest form of cloning consists of copying and pasting code fragments. This adhoc practice of copy paste may cause context conflicts between the copied variables and the destination variables. Also when a code containing a bug is copied and reused at several places, the bug gets replicated and becomes the root cause for bug propagation. Thus cloning results in increased defect probability.

iii) Bad design

Introduction of code clones within a system makes it poorer in design. Such a

system lacks in abstraction mechanisms or a good inheritance structure.

iv) Resource requirements

Code cloning leads to increase in the program size, thereby putting stress on the resources of the system. Increase in program size deteriorates the system's performance by increasing the compilation time and the space requirements.

1.5 Types of Code Clones

Two code fragments can be textually or functionally similar. Also clones can be categorized on the basis of their granularity level. Figure 1.2 shows the different types of clones.

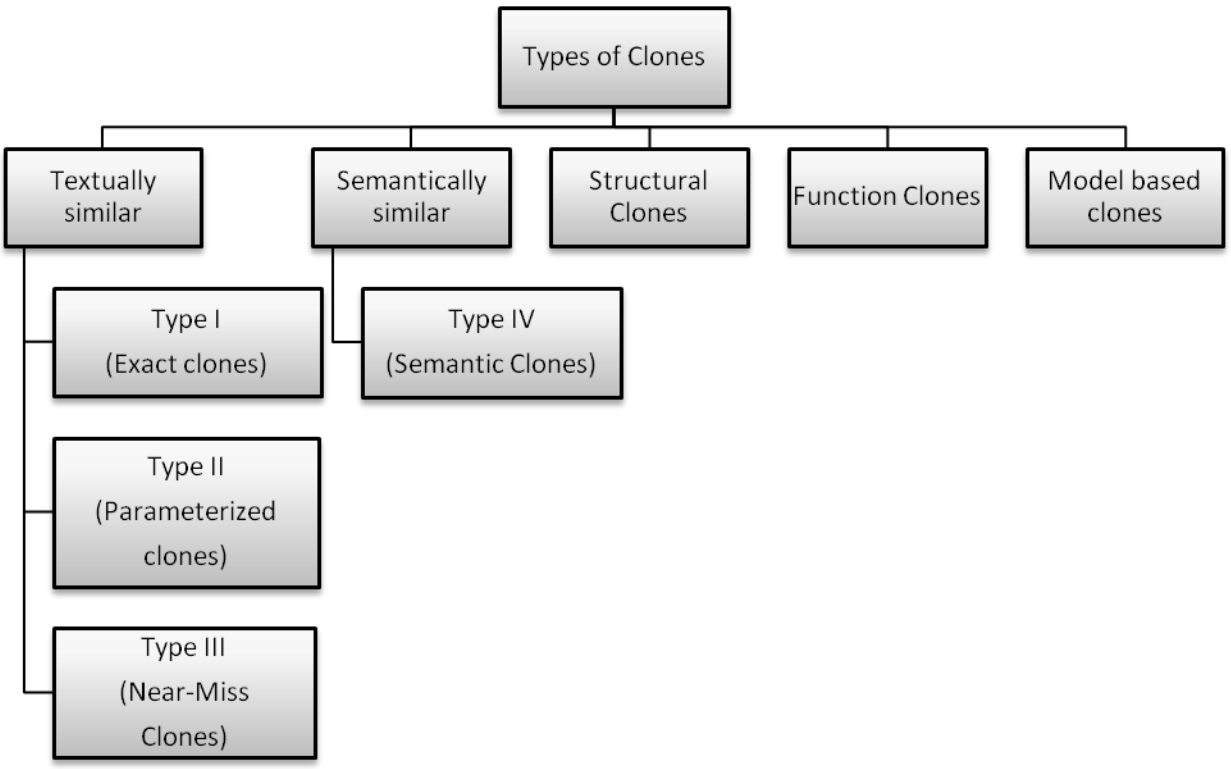


Figure 1.2: Types of Code Clones.

1.5.1 Textually Similar Clones

When a code fragment is reused by copying from one place and pasting it to another with minor modifications, then they are textually similar clones. Following are three types of clones solely based on textual similarity.

1.5.1.1 Type I Clones

These are the exactly identical copies and differ only in whitespaces, comments or layout. Type I clones are popularly termed as Exact Clones. Following example code shows the exact match for a sample code.

```
void  sumProd(int n) {     //s0
int  sum=0;                //s1
int product =1;            //s2
for (int i=1; i<=n; i++) {  //s3
     sum=sum + i;          //s4
     product = product * i; //s5
     fun(sum, product); }} //s6
```

Sample Code

```
void  sumProd(int n) {     //s0
   int  sum=0;             //s1
   int product =1;         //s2
   for (int i=1; i<=n; i++) { //s3
        sum=sum + i;       //s4
        product = product * i; //s5
        fun(sum, product); }} //s6
```

Exact Clone

Figure 1.3: Type I Clones.

1.5.1.2 Type II Clones

In addition to the variations in whitespaces, layouts and comments, two syntactically similar fragments with changes in user-defined identifiers and literals are said to be Type II clones. These are widely known as renamed or parameterized clones.

```
void  sumProd(int n) {     //s0
int  sum=0;                //s1
int product =1;            //s2
for (int i=1; i<=n; i++) {  //s3
     sum=sum + i;          //s4
     product = product * i; //s5
     fun(sum, product); }} //s6
```

Sample Code

```
void  addTimes(int n) {  //s0
int  add=0;              //s1
int times =1;            //s2
for (int i=1; i<=n; i++) { //s3
     add=add + i;        //s4
     times = times * i;  //s5
     fun(add, times); }}  //s6
```

Renaming of Identifiers

Figure 1.4: Type II Clones.

1.5.1.3 Type III Clones

A Type III clone, along with being a Type II clone, is a copied code with further modifications where statements can be added, deleted or changed. Also known as Near-miss clones.

```
void  sumProd(int n) {     //s0
int  sum=0;                //s1
int product =1;            //s2
for (int i=1; i<=n; i++) {  //s3
     sum=sum + i;          //s4
     product = product * i; //s5
     fun(sum, product); }} //s6
```

Sample Code

```
void  sumProd(int n) {     //s0
int  sum=0;                //s1
int product =1;            //s2
for (int i=1; i<=n; i++)   //s3
     if  (i % 2 == 0){     //s3b
     sum=sum + i;          //s4
     product = product * i; //s5
     fun(sum, product); }} //s6
```

Addition of new lines

```
void  sumProd(int n) {     //s0
int  sum=0;                //s1
int product =1;            //s2
for (int i=1; i<=n; i++) {  //s3
     sum=sum + i;          //s4
      //s5  line deleted
     fun(sum, product); }} //s6
```

Deletion of Lines

Figure 1.5: Type III Clones.

1.5.2 Semantically Similar Clones

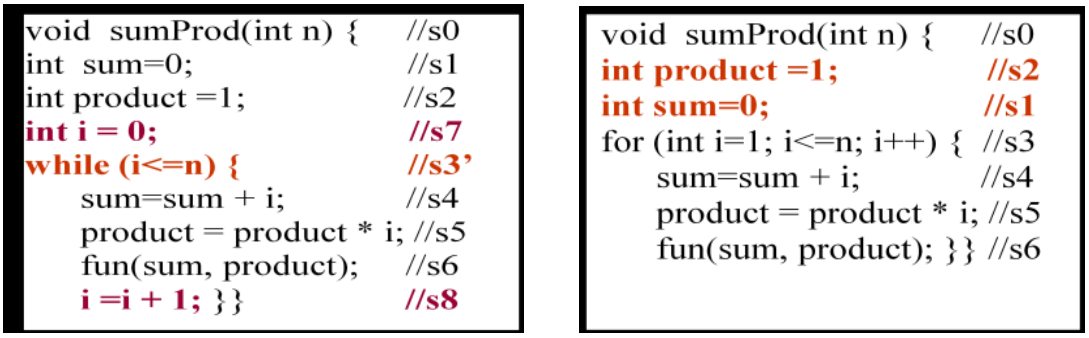
The code fragments similar in their behavior without being textually similar are referred to as semantic clones.

1.5.2.1 Type IV Clones

The two code fragments, claiming to be Type IV clones, may have been developed by two different programmers to implement the same behavior making the code fragments similar in their functionality. They may not be textually identical and are also called semantic clones.

```
void  sumProd(int n) {     //s0
int  sum=0;                //s1
int product =1;            //s2
for (int i=1; i<=n; i++) {  //s3
     sum=sum + i;          //s4
     product = product * i; //s5
     fun(sum, product); }} //s6
```

Sample Code



Control Replacements Reordering of Statements

Figure 1.6: Type IV Clones.

1.5.3 Structural Clones

Structural clones exhibit design level similarities. They can be traced by looking for recurring patterns of simple clones [4]. They emerge at architectural level and can be patterns of cohesive classes or files.

1.5 4 Function Clones

Function clones are the ones having granularity limited to function or a procedure. Such type of clones can be easily extracted into a generalized procedure.

1.5.5 Model based Clones

Model based development is generally used in automotive environment such as embedded systems and uses graph theory to generate code. Nowadays model based development is replacing code as the core artifact of software development. Duplications in these models give rise to model based clones.

1.6 Clone Management

Clone Management can be coined as an 'umbrella activity' covering all aspects of clone detection, clone visualization, analysis, correction and prevention. Existence of redundancy can be in a number of artifacts including requirement specifications, design architectures, test cases, code etc., all of which needs user's attention.

According to Giesecke [5], clone management activities can be related to any of the three categories:

- **Corrective clone management** consisting of clone removal and refactoring activities.
- **Preventive clone management** activities comprising of prevention of creation of new clones.
- **Compensatory management** aiming at minimizing the negative impact of existing clones, in case they can't be removed.

1.7 Motivation for Clone Management

Code clones, basically the similar code snippets, can occur frequently throughout the source code. Any kind of bug propagation or unexpected behavior of any fragment might lead to its updation and thus introduce an updation anomaly in rest of the similar fragments. Therefore need of Clone management arises, which is a much less explored area as compared to clone detection and analysis. This fact can be easily interpreted from Figure 1.7.

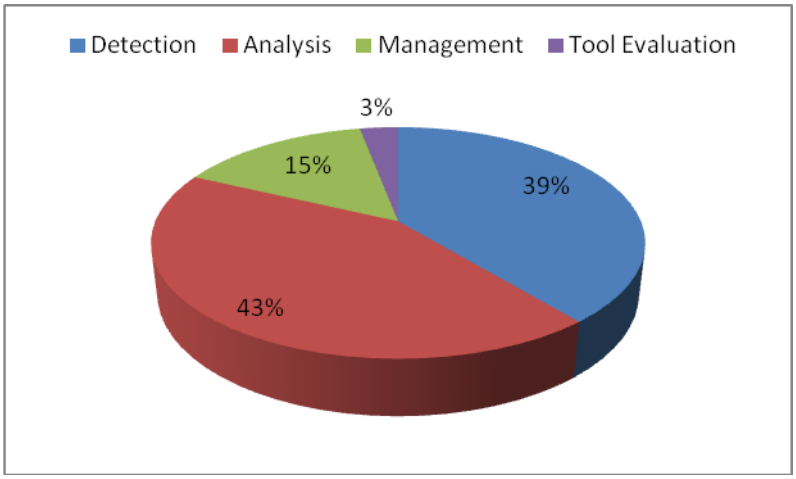


Figure 1.7: Proportion of publications in each category over the period 1994–2013 [6].

The maximum amount of publications took place in the category of code clone analysis and detection that is 43% and 39% of the work has been done in the fields of code clone analysis and detection respectively while the area of code clone management comprises of mere 15% publications. Thus the area of code clone management needs more exploration.

Although code clones are claimed to increase the maintenance overhead and it is advisable to remove the clones but it is not always easy and possible to refactor all the clones. There are certain risks and cost factors associated with refactoring which makes it inappropriate to refactor them, rather some management activities might be in place to help in retaining them.

Code clones play a dual role in software's life, on one hand it helps during development and on the other it increases the overhead while maintenance phase. This gives rise to a practical difficulty of avoiding or refactoring them, due to which researchers have agreed on a fact that code clones should be properly detected and efficiently managed [7] [8].

1.8 Outline of Thesis

The remaining of the report is organized into the following sections:

Chapter 2- Gives a systematic literature review of clone management including the techniques for code clone detection, clone analysis and refactoring of clones.

Chapter 3- Identifies the problem statement and the proposed work.

Chapter 4- Explains the proposed approach and its implementation.

Chapter 5- Discusses the experiments performed along with the results obtained.

Chapter 6– Concludes the work done and provides the future directions.

Thesis concludes with References and Publications.

Chapter 2

Literature Review

2.1 Clone Management

This chapter gives a brief description on the present state of art in the whole clone management process. The clone management process basically acts as an umbrella activity. It covers all the aspects of cloning from clone detection to clone removal [48]. Thus clone management circumscribes a broad range of activities including clone detection, documentation, analysis, visualization and refactoring, where clone documentation and visualization assists towards a better code clone management process.

2.1.1 Clone Terminology

The dictionary meaning of clone implies "duplicity". In simple words, code clone refers to a fragment of code that appears more than once throughout the source code of a program. These fragments can be syntactic or semantic similar structures of code as already described in section 1.5. Roy and Cordy [9] have drawn enough definitional vagueness on code clone. According to their survey, the definitions of code clones given by different authors are vague in one form or another.

Following are some of the clone definitions based on the relationship existing between them.

2.1.1.1 Clone Pair

Clones are considered to have an equivalence relation between them [9]. Any two code fragments detected as clones by a code clone detector constitute a clone pair. Thus it is a kind of relationship that exists between two clone segments.

2.1.1.2 Clone Class

Clone class can be described as a kind of association between code segments that are claimed to be code clones. This relationship exists between more than two clones.

Thus a clone class can be referred to as an integration of all clone pairs.

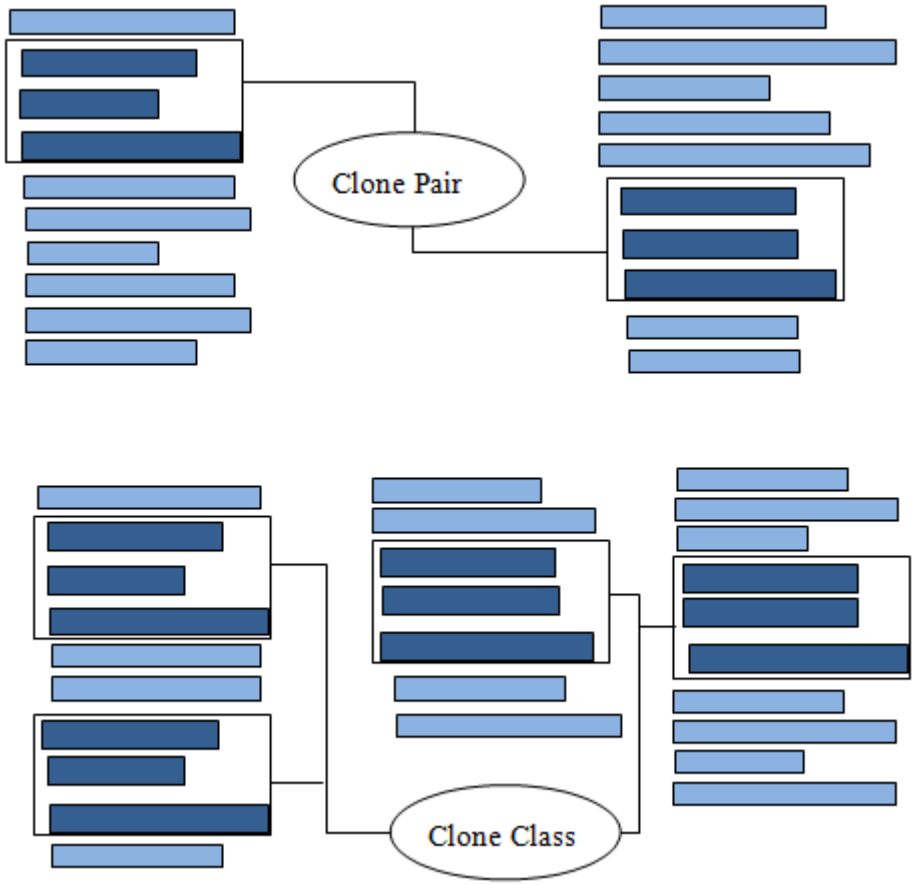


Figure 2.1: Clone pair and Clone Class [10].

2.1.1.3 Clone Class Family

The grouping of all the clone classes, coming under the same domain, constitutes a clone class family. Here domain refers to the source entities from where the clones have been detected. It can be a class, function, package or a file.

The architectural view of a clone pair and a clone class is shown in Figure 2.1.

2.1.2 Clone Management Activities

As a known fact, presence of clones increases the maintenance overhead of a program. So the clones are needed to be managed and for that the first step becomes to detect the clones. The clone information retrieved from the detected clone fragments forms the base for documentation. The documentation records the location of the clones in the source code, their granularity and the relationship existing

between the detected clones. Further the clones are continuously tracked for their evolution, so that any of the change in the clone information might be updated in the documentation. Clone visualization can be an optional activity that aids in clone analysis. Analysis is done to find the potential clone candidates for refactoring. After performing the refactoring operations a verification activity is also done to ensure whether refactoring has caused any change in the behavior of the program. If yes, then the refactoring operation can be rolled back and the clone might be reconsidered for refactoring. Figure 2.2 summarizes the flow of activities in the clone management process.

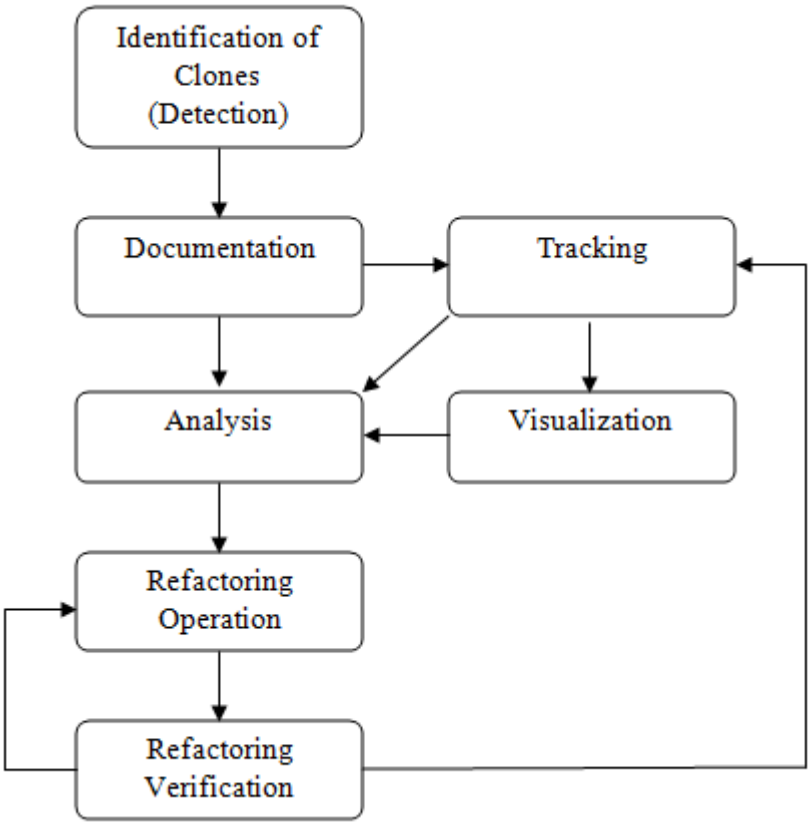


Figure 2.2: Clone Management Flowchart.

2.2 Clone Management Process

This section now describes the literature review on clone management in detail.

2.2.1 Clone Detection

Code clone detection has been widely accepted and is being carried out at the industrial level using a number of tools. These detection tools are based on different

techniques. The detection techniques can be applied over plain text, tokenized sequence of programs, parse trees, program dependency graph (PDG) obtained from the program code, metrics calculations and combinations of any of the above. These can be categorized as follows:

i) Text Based Code Clone Detection Technique

The text based code clone detection technique is the simplest clone detection technique and is claimed to be the fastest one too as it does not require any kind of parsing and does not carries out any syntactic or semantic analysis on the source code. This detection technique is applied on the raw code, may be syntactically incorrect, and goes for similar sequences of code or text and returns them as the clone pairs. This detection approach is claimed to detect Type I clones

Johnson et al. [11] proposed a text based clone detection approach based on fingerprinting. A Fingerprint, over here, refers to a short bit string equivalent for a large number of data objects. The aim of the fingerprint based comparison was to help the maintenance team to understand the large code base in the legacy systems in order to improve maintainability. This was done by taking into account the number of repetitions of a particular code fragment.

Ducasse et al. [12] developed one of the text based clone detection tool. The tool's advancement over the previous ones was that it was a language independent tool and also supported visualization of the compared data in the form of dot plot. The approach used is a two step process and is based on hashing, where firstly code transformation is done. Transformed code is obtained by simply manipulating the strings i.e. by removing white spaces and comments. The second step involves comparison of the transformed code using string matching to find the duplicated substrings. Each match encountered is stored in a matrix. The matrix thus obtained contains the matches of single lines of code and not sequences of code and can be visualized on a dot plot in the form of diagonals.

Roy et al. [13] gave a light-weighted parser-based code clone detection tool which finds near-miss clones with high recall and precision. It is a language specific tool which uses a text comparison technique. The tool named NICAD works in two stages. First it uses pretty-printing and normalization techniques for detecting the potential clones and normalizing them. This is followed by clustering which is done dynamically in order to textually compare the potential code clones.

ii) Token Based Code Clone Detection Technique

This technique is similar to the text based detection approach except that the comparison is applied on the tokenized sequence of code and not directly on the code sequence. The tokenized sequences of code are compared against each other using suffix array or suffix tree algorithm in order to find the duplicated strings. This technique is capable of finding Type I and Type II clones.

Baker [14] proposed a token based tool namely Dup capable of finding the exact matches and the parameterized matches. Parameterized matches, also known as p-matches, are the identical matches with a systematic replacement of variables and identifiers. This tool aims at finding maximal sections of identical code and uses a suffix tree algorithm for the same.

Kamiya et al. [15] developed one of the best known code clone detection tool named as CC-Finder. This tool incorporates the use of a lexical analyzer which removes the comments, white spaces and generates a token sequence out of the source code which is further transformed using certain rules. This transformation regularizes the identifiers by partially removing the context information. A special token replaces the identifiers so that code portions with different variable names could be returned as clone pairs by the matching algorithm. The tool makes use of a visual support environment Gemini [16] which helps visualizing the clones and the related information.

Zhenmin Li et al. [17] gave a tool CP-Miner which uses data mining techniques to find the duplicated code fragments. It detects the clones as well

as the related bugs in large software systems as well as in operating systems. Although its execution time is same as that of CC-Finder but CP-Miner is capable of finding 17-52 % more code clones and is tolerant to insertion and deletion of statements while copy paste.

iii) Tree based Code Clone Detection Technique

In this code clone detection technique, a parse tree or an abstract syntax tree is built for the source code and the tree is searched for similar sub trees by applying pattern matching techniques. Although parsing becomes more expensive in terms of execution time and memory, it is capable of detecting clones consisting of inserted or deleted statements (Type III clones i.e. the near-miss clones).

Baxter et al. [18] instigated an abstract syntax tree based detection approach and developed a tool CloneDr which parses the code into an abstract syntax tree, detects the sub-tree clones using hashing and then performs the sequence detection on the detected sub-trees. This makes the tool capable of detecting near miss clones in addition to the exact clones. The tool also goes for more complicated near miss clones by generalizing other clones' combinations.

Wahler et al. [19] described a new approach which is more flexible and easily configurable. The approach works on the concept of data mining by finding frequent itemsets. The intermediate representation consists of abstract syntax tree in XML. The approach is capable of finding exact and parameterized clones.

Jiang et al. [20] presented an efficient tree-similarity detection algorithm and implemented it as a tool, DECKARD, which works by calculating characteristic vectors from the generated abstract syntax tree and clustering those vectors with respect to the Euclidean distance metric, using locality sensitive hashing. The clones are detected by looking for sub-trees having their vectors in same clusters.

iv) Graph Based Code Clone Detection Technique

As the name suggests, this clone detection technique uses graphs as their intermediate representation. Basically a program dependency graph (PDG) is used which carries out detection of semantic clones because a PDG represents both data and control flow dependencies existing between the statements of a source code. On obtaining the PDG for a source code, they are searched for isomorphic subgraphs and are returned as semantic clones. Although it becomes an expensive approach on the part of time and memory, yet it is the only efficient technique to detect Type IV i.e. semantic clones.

Krinke [21] developed a tool Duplix which detects the duplicated code by searching for similar subgraphs in a program dependence graph. It uses a k-limiting approach to carry out the detection process and gives a good trade-off between precision and recall.

Komondoor et al. [22] presented a PDG based approach in finding the semantic clones. The tool PDG-Dup, based on program slicing is proficient in finding non-contiguous clones, intertwined clones and the ones with reordered statements. Slicing done on a PDG helps in capturing the data dependencies and thus locates the semantic code structures which can be easily extracted.

Higo et al. [23] developed a tool Scorpio, based on two-way slicing, for detecting non-contiguous clones. It was developed to overcome the fact that existing tools and techniques were not that fast and feasible in detecting clones. They pioneered PDG specialization and heuristics for better detection process.

v) Metrics based Code Clone Detection Technique

Other than comparing the source code directly for clone detection, one can find certain metric values and compare them for similarity. Closer metric values for two procedures make them potential clones. Metric calculation can be done at class level, function level or even at statement level.

Mayrand et al. [24] presented an approach to detect clones by calculating metrics at function level. The tool Datrix was used as an assessment framework. The approach used 21 function metrics to find the cloning level in a program. It was developed as an automatic detection technique and used AST as intermediate representation along with estimation of data and control flow dependencies.

Patenaude et al. [25] gave a similar approach of computing metrics and finding clones at function level but extended the approach for handling clones in large java systems.

Lanubile et al. [26] pioneered an approach of clone detection in web applications. This semi-automated approach detects cloned functions present in the html scripted code. Detection is carried out as an automatic process but the follow up verification is done manually.

vi) Hybrid Code Clone Detection Technique

This technique makes use of combination of any of the above stated clone detection technique in order to make a more efficient technique to detect clones.

Kodhai et al. [27] proposed an approach as a combination of metrics based and textual comparison in order to find functional clones in C programs. The approach has been implemented as a tool in java which detects Type I and Type II clones with reduced complexity.

Basit et al. [28] developed a tool Clone Miner in order to find higher level clones. The approach works by first detecting the simple clones and then using frequent itemset mining technique to further detect higher level clones. The data mining and clustering is done on the output of any of the simple clone detection tool.

All the tools discussed above are listed in Table 2.1 which summarizes the given code

clone detection tools along with their corresponding method being used and their citations.

Table 2.1: Summarized List of Code Clone Detection Tool.

Tool/ Ist Author	Method	Citations
Text Based Detection Tools		
Johnson	Source Code, Fingerprinting	[11]
Duploc	Source Code, Substring Comparison/ Dot Plot Scatter Plot	[12]
NICAD	Source Code, LCS	[13]
Token Based detection Tools		
Dup	Tokens, Suffix Tree	[14]
CCFinder	Tokens, Suffix Tree	[15]
CP-Miner	Tokens, Frequent Itemset Mining	[17]
Parse Tree Based Tools		
CloneDr	AST, LSH/Dynamic Programming	[18]
Deckard	PDG/ Parse Tree, LSH/ Subtree Comparison	[19]
Wahler	AST, XML representation	[20]
Graph Based Tools		
Duplix	PDG, k- length patch matching	[21]
PDG-DUP	PDG, Program Slicing	[22]
Scorpio	PDG, Program Slicing	[23]
Metrics Based Tools		
CLAN	AST, Metrics	[24]
Patenaude	Source Code, Metrics	[25]
Lanubile	Source Code, Metrics/ Feature Vector Clustering	[26]
Hybrid Tools		
Kodhai	Textual, Metrics	[27]
CloneMiner	Data Mining, Clustering	[28]

Table 2.2 shows the comparison between all the code clone detection techniques which are derived from the findings of Bellon et al. [29] who compared six clone detectors, all belonging to different detection techniques. It is analyzed that the text and token based code clone detection tools behave somewhat similar and give higher

recall. The tree based detection tools give higher precision and the graph based tools are sensible for Type III only.

Table-2.2: Comparison of Code Clone Detection Techniques.

Clone Detection Technique	Type of detected clones	Portability	Efficiency	Integrity
Text Based	Syntactic (Type I)	Good	High	Depends on the algorithm
Token Based	Syntactic (Type I and II)	Average	Low	Good
Tree Based	Syntactic (Type I, II and III)	Poor	High	Depends on the algorithm
Graph Based	Syntactic and Semantic (Type IV)	Poor	High	Medium
Metric Based	Syntactic	Good	High	Medium

2.2.2 Clone Documentation

The results of different clone detectors differ from each other in various forms. Some detectors yield clone pairs, others give clone groups. Also the format of the resultant clone information differs from tool to tool such as html, xml or plain text. This makes difficult for the detection tools to exchange information. So Koschke et al. [30] proposed a schema based format, namely Rich Clone Format (RCF), for storing and exchanging clone data. Clone Region Descriptor (CRD) [31] has also been developed which returns clone regions within procedures.

2.2.3 Clone Analysis

Once the clones are detected in the source code of a program, a thorough analysis is done in finding the potential candidates for refactoring. CeDAR [32] has been developed as an Eclipse plugin which embeds a clone detection tool, uses its results, does the required amount of analysis and displays the clone properties in an IDE irrespective of the fact that different clone detection tools record different type of clone information. CeDAR itself does not support clone management but Eclipse can extend its support for simultaneous refactoring of detected clones.

The detected clones can be analyzed in a number of steps:

2.2.3.1 Analysis of Clone Evolution

Clones evolve each time a newer version of a program containing clones is released. Thus it becomes essential to analyze the clone evolution from version to version as it describes the nature of clones and helps in making refactoring decisions. A recent survey has been conducted on clone evolution by Tairas et al. [33] which revealed the presence of change patterns over the evolving source code. These change patterns can propound techniques for enhancing clone management including refactoring and removal.

Clone evolution can be visualized using certain tools. One of the tools is SoftGuess developed by Adar and Kim [34], a system which explores clone evolution and offers a simple visualization of clone evolution using three different views. It consists of a dependency graph where nodes represent clones and describes how the nodes (package, class or method) within a version evolve from other nodes and how they have evolved in the next version. Recently, a tool called CYCLONE has been developed by Harder and Gode [35]. It can be considered as advancement over the previous one as it is a multi-perspective tool for clone evolution analysis. Saha et al. [36] used the popular scatter plot for visualization of clone evolution.

These different tools for visualization thus aid in clone evolution analysis, which further helps in clone management.

2.2.3.2 Tracking

The evolving software systems might introduce inconsistency in the source code either by formation of new clones or by invalidating the existing ones. Thus it becomes necessary to track the continuous evolution of the software system and make the corresponding updates in the respective documentations.

Koschke et al. [37] attempted to make an incremental clone detector, iClone, which detects clones in subsequent versions of a system by using a suffix tree algorithm. JSync [38] is regarded as one of the most efficient tool for tracking clones and is available as a plugin to SVN. Basically it is a clone detection tool which uses the change information, provided by the SVN, and detects clone in an incremental manner. This makes it an efficient and elegant tool.

2.2.3.3 Clone visualization

Clone visualization is important to analyze the properties of the detected clones and to study their distribution throughout the code. The main motivation behind the development of clone visualizing tools was to address the problem of handling the huge textual data obtained from the clone detectors in the form of clones.

To visualize the textual similarity between files, Johnson [39] used the Hasse diagram which represents the similarity using a directed acyclic graph. Reiger et al. [40] introduced a set of polymetric views which abstracts the clone information into a clone class family and allows the visualization of a number of clone metrics. A well known clone visualization tool is Gemini [16] which first detects clones using CCFinder and then visualizes the resultant clones using a scatter plot. The scatter plot and the metric values, thus obtained, represent the clone relationships. Clone Miner, a clone detection tool, can work with an eclipse plugin Clone Visualizer [41] which not only supports visualization of clones but also provides query based filtering.

2.2.4 Clone Correction

Refactoring of clones for their removal can act as a peculiar corrective measure. Refactoring refers to modifications done in a program, with respect to the code clones, in order to make it more efficient and less complex without changing the behavior of the program. The refactoring technique intends to correct or remove code clones, without affecting the functionality.

The suggested techniques can be consistent renaming, software refactoring patterns, and synchronized modifications of code clones.

2.2.4.1 Consistent Renaming

It refers to a kind of modification where identifiers in the cloned code are renamed depending on the destination's context. CReN [42] has been developed as an Eclipse plugin that helps in consistent renaming of identifiers. It also aids in keeping track of clones. The idea behind the tool was further extended to CnP [44] which managed the clones proactively [43] on their creation on the basis of copy-paste clipboard activities. Cnp has also been developed as a plugin to Eclipse.

2.2.4.2 Refactoring Patterns

An archive has been maintained by Fowler [3] containing ninety-three refactoring patterns. Some of the refactoring patterns found to be the most suitable ones are:

- **Extract method** where similar blocks of code (code clones) are replaced by calls to an extracted generalized method.
- **Move method** can be used to merge identical methods within classes.
- **Pull-up method** introduces a generalized method in a common superclass by removing the similar methods in the subclasses.
- **Extract superclass** is appropriate where classes having similar methods do not have a common superclass, thus a new common superclass is introduced and pull-up method is applied.
- **Extract utility-class** introduces a new class including a generalized method as the classes containing the similar functions are not fit to have a common superclass.

Other than the above mentioned, certain other not so popular, refactoring patterns also exist which can help in removing near miss clones. These include identifier renaming, function parameter re-ordering, modifying type declarations, splitting of loops, substitution of conditionals, loops and relocation of methods [45].

2.2.4.3 Synchronized Modification

Synchronized modification consists of simultaneous or linked editing that can be applied to multiple segments of identical text to synchronize changes made in one fragment with other fragments. As an example, Miller [46] incorporated the feature of simultaneous editing in a text processing system, where simultaneous editing can be carried out by selecting multiple regions of text.

Mondan et al. [49] emphasized on refactoring of only important code clones by ranking them. Here ‘important’ refers to those code clones which belong to the same clone class and tend to change together while software evolution. The main idea behind their approach is to identify code clones that change according to a prespecified pattern and mine association rules among them for which they developed

MARC (Mining Association Rules among Clones). The rules are then ranked by the tool based on their confidence values.

Venkatasubramanyam et al. [50] also prioritized the code clones for their refactoring in order to deal with large set of code clones detected by clone detection tools. They studied the impact of code clones with respect to the code quality, maintenance efforts and refactoring costs, based on which prioritization is done. At present this approach is used at project level in their company and a generalized tool has not been developed yet.

Chapter 3

Problem Statement

3.1 Research Gap

Code cloning has emerged as an active area for research since last decade. Presence of code clones indicates higher maintenance efforts. Numerous code clone detection tools and techniques have been developed for the same. But the work does not end here. Only locating the code clones in the source code does not solve the problem. Clones are needed to be removed or corrected and for that purpose, appropriate clone management activities should be in place.

Code clone management involves a deep analysis of the existing code clones, tracking their evolving nature, documentation of the code clone information and making appropriate decisions for their removal. On the basis of all the information thus gathered, the code clones are either refactored or can be ignored. This decision depends on the sensitivity of the detected clones. Certain short lived code clones can be ignored but on the other hand, the long lived code clones requires rigorous refactoring operations. A number of other factors also contribute towards making an appropriate refactoring decision.

It has been concluded by Kapser and Godfrey [47] that all the code clones can't be avoided and it is not always possible to remove all the detected code clones. Code clones are thus considered as a bad smell and needed to be managed in a more effective and efficient way. Therefore a technique is needed that might decide the order of removal of code clones.

3.2 Problem Identification

i) A better clone management process is needed.

ii) Clone management requires a prescreening step in order to filter the clones on the basis of their relevancy.

iii) All the detected clones need not be removed to improve the maintainability of the software. Infact, an approach is needed that refactors or removes the more critical clones first to make the system more efficient.

iv) A tool is needed that calculates the criticality of clones and prioritize them in the decreasing order of their criticality for their removal or refactoring.

3.3 Methodology

This thesis presents an approach to prioritize code clones for their removal. It works by detecting code clones and ranking them on the basis of their criticality or severity. The technique makes use of maintenance overhead metrics for calculating the severity of clones. The calculation of maintenance overhead, in advance, would estimate the efforts that would be required if the clones were not fixed. Thus the clones having higher maintenance overhead would get a higher priority as compared to others and would be removed first.

Chapter 4

Proposed Work and Implementation

Code clone prioritization is basically a clone management activity which makes clone removal much easier and efficient. Clone management covers all aspects of code cloning starting from clone detection to their correction. So clone prioritization acts as a bridging activity between clone detection and their removal. It enhances the management process by prioritizing the clone detection results. The prioritized list of code clones thus obtained can be presented for removal or refactoring.

The clone prioritization system works on the clone information that has been obtained from the clone detection tool. It works on cloned code fragments and makes use of results of various maintenance overhead metrics which decide the criticality of the fragments.

Initially a clone detection tool is used which detects the cloned code fragments. Now the next important step becomes to remove or refactor those detected clones in order to make the code base more efficient and less vulnerable to updation anomalies. In case of large code base, numerous (in hundreds) code snippets could be detected which might have been cloned. This makes difficult for the user to tackle each and every cloned copy. So, to manage the clones more efficiently, they are ranked. This ranking is needed in order to decide on which clone set action is needed to be taken first.

The major disadvantage of having code clones is that they increase the maintenance efforts. This implies that if any bug is detected in any piece of code, it has to be corrected in all of the cloned copies. Hence system's maintainability decreases. This maintenance effort, if estimated previously, would help in determining the criticality of clones on the basis of which their urgency of removal could be decided. So if any duplicated piece of code is having more length, more cyclomatic complexity and more no of repetitions as compared to other duplicated fragments it would surely incur more cost and effort at the time of maintenance if not fixed.

4.1 Solution Design

The Figure 4.1 shows the flow of activities in the proposed system. Figure 4.2 shows the architecture of the described approach.

4.1.1 Flow Diagram of the Proposed Work

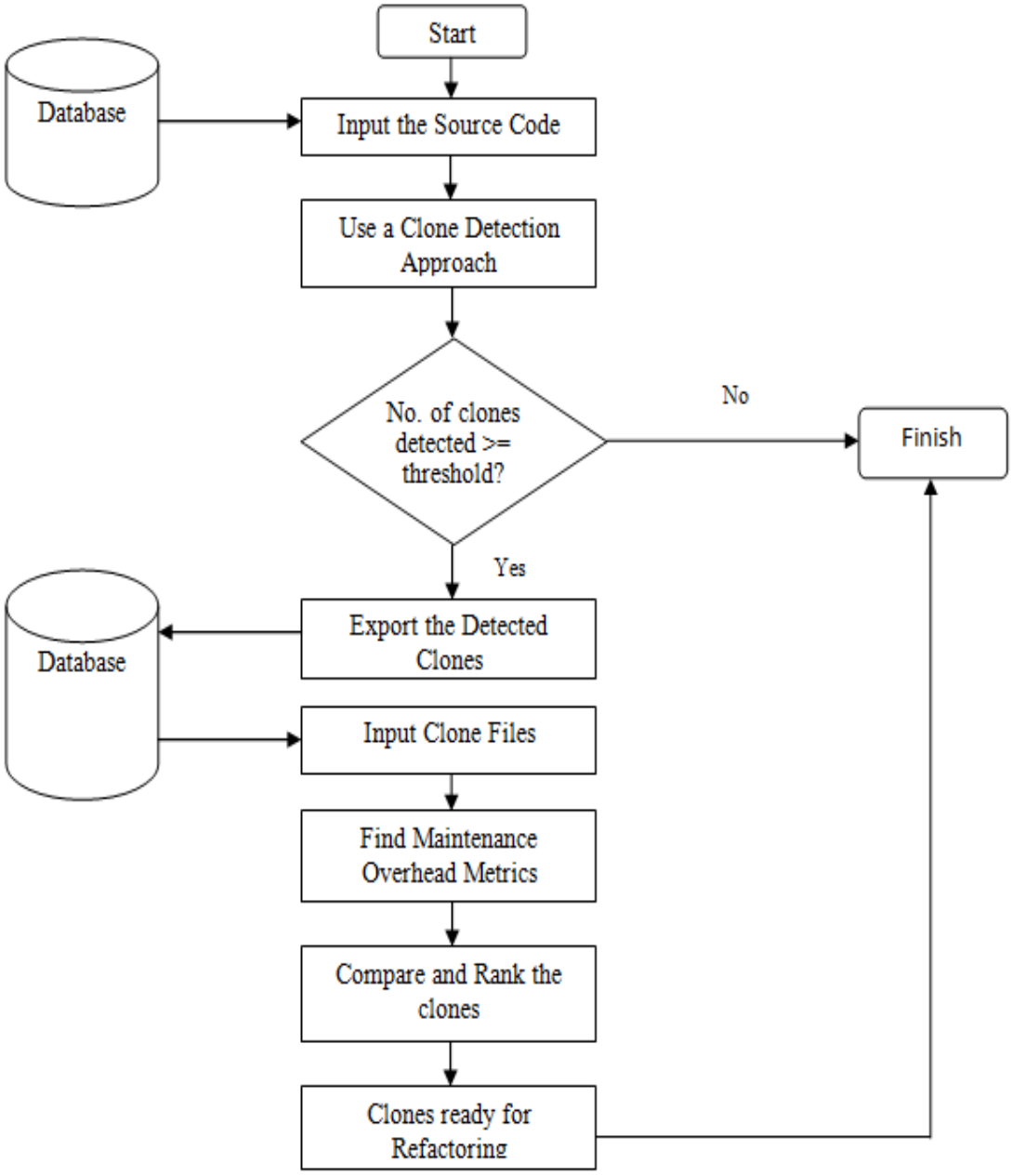


Figure 4.1: Flow Diagram of the Proposed System.

4.1.2 Architecture of the Proposed Work

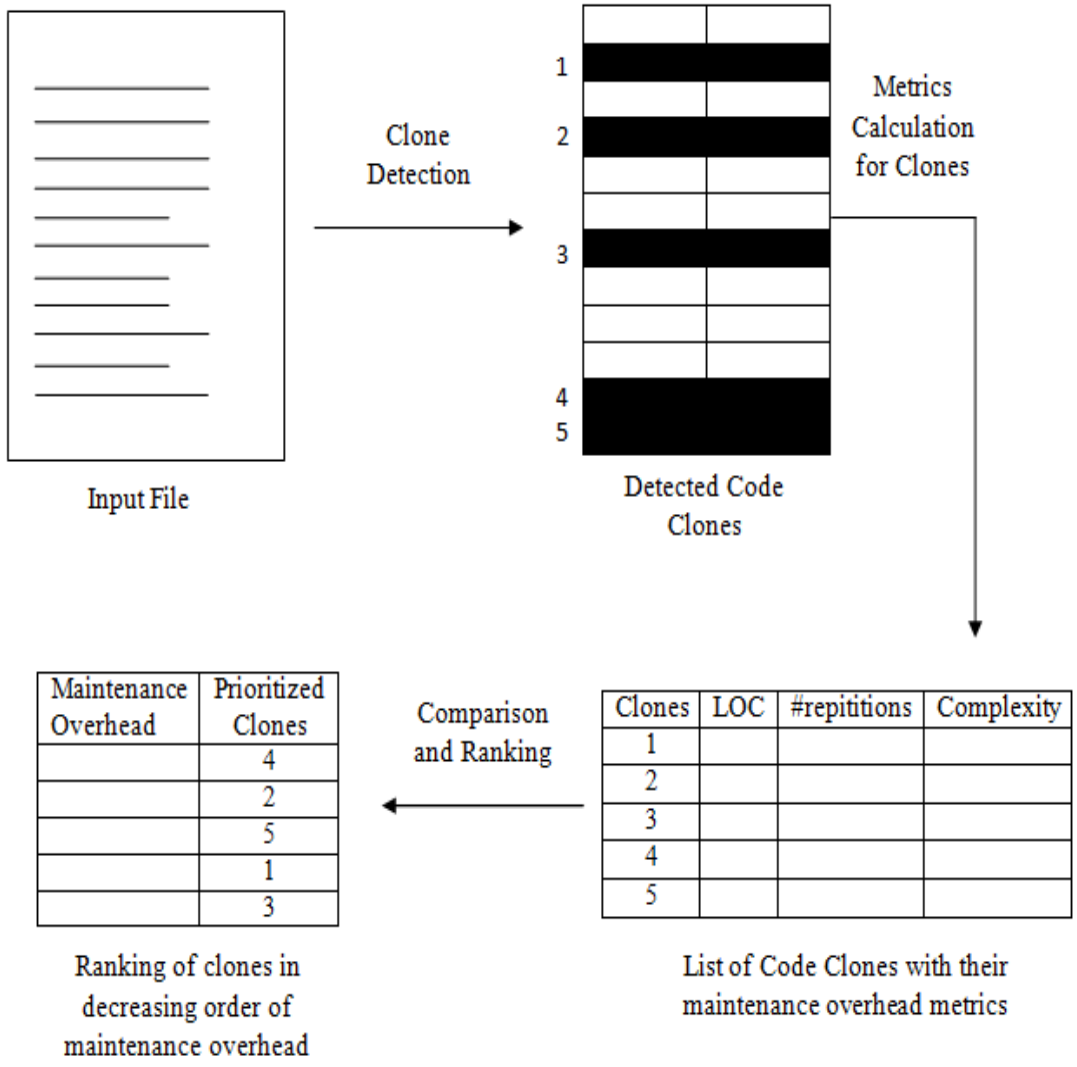


Figure 4.2: Architecture of the Proposed Work.

4.1.3 Algorithm: Code Clone Prioritization

The token based code clone detection tool named CCFinder [15] detects the code clones in the given input file. Each code clone denoted as C_i , where (i ϵ 1 to n), belongs to the set $C_s = \{C_1, C_2, C_3, \ldots C_n\}$, where C_s is the code clone set. The detected code clones acts as input to the Algorithm 1 which calculates certain metrics for each of the code clone in C_s and is as follows:

Algorithm 1: Calculate_Metrics (C_s)

Input: Set of clones detected by CCFinder.

Output: Calculated Metrics Value.

1. if (| C_s | >= threshold_value)
2. for each $C_i \in C_s$, do
3. calculate LOC_{C_i}, // Algorithm 1.1
4. $Freq_{C_i}$, // Algorithm 1.2
5. $Cyclo_{C_i}$. // Algorithm 1.3
6. end for
7. end if

Thus the results obtained are metrics values calculated for each code clone fragment. Here the threshold_value denotes a predefined value which must be satisfied. Unless the number of clones reaches the threshold, their metrics will not be calculated. For each code clone in the clone set, three metrics are calculated where,

LOC_{C_i} = length of code clone i,

$Freq_{C_i}$ = frequency of occurrence of code clone i in the input file,

$Cyclo_{C_i}$= McCabe's cyclomatic complexity of code clone i.

These metrics are calculated by using some coding conventions and the algorithms for the same are given as follows.

Algorithm 1.1: Length_of_Clone (C_i)

Input: Clone file

Output: Length of code clone

1. read file f
2. if (f == null)
3. return 0
4. end if
5. while (f != EOF)
6. if ('\n')

7. lines++
8. end if
9. end while
10. return lines

Algorithm 1.2: Frequency_of_Repetition (C_i, original file)

Input: Clone file and Original file

Output: Number of occurrences of clone in original file

/* C_i [j] is the array containing the clone. */

1. read original file
2. while (!EOF)
3. fgets (str, size of str, fp)
4. char *dd = C_i [j]
5. if (strstr (str, dd))
6. j++
7. allfound++
8. end if
9. if (allfound = LOC_{C_i} - 1)
10. occurrence++
11. end if
12. end while
13. return occurence

Algorithm 1.3: Cyclomatic_Complexity (C_i)

Input: Clone file

Output: Cyclomatic complexity

/* key is an array of keywords which define decision point which are if, while, for, case, ?:, ?., &&, || */

1. read file f

2. if (f == null)
3. return 0
4. end if
5. while (f != EOF)
6. for i → 0 to length of array
7. search for key[i] in f,
8. if found, sum++
9. end for
10. end while
11. return (sum + 1)

These three metrics altogether define the maintenance overhead for a particular code clone. When they are assigned some fixed weights, their summation gives an approximate value for the overhead cost that would be incurred in their maintenance. Following algorithm 2 returns the normalized value for each code clone.

Algorithm 2: Normalize_Metrics (LOC_{C_i}, $Freq_{C_i}$, $Cyclo_{C_i}$)

Input: Calculated metrics value of each clone.

Output: Maintenance Overhead.

1. for each $C_i \in C_s$, do
2. $N\_LOC_{C_i}$= 0.5 * LOC_{C_i},
3. $N\_Freq_{C_i}$= 0.3 * $Freq_{C_i}$,
4. $N\_Cyclo_{C_i}$= 0.2 *$Cyclo_{C_i}$.
5. $Maintenance\_Overhead_{C_i}$= $N\_LOC_{C_i}$ + $N\_Freq_{C_i}$ + $N\_Cyclo_{C_i}$
6. return $Maintenance\_Overhead_{C_i}$
7. end for
8. end

Here fixed weights are assigned to the three metrics based on their role in contributing towards maintenance overhead which tends to calculate a normalized value for each code clone, where,

$N\_LOC_{C_i}$ = Normalized value for LOC_{C_i},

$N\_Freq_{C_i}$ = Normalized value for $Freq_{C_i}$,

$N\_Cyclo_{C_i}$ = Normalized value for $Cyclo_{C_i}$.

Now each clone C_i in the clone set C_s is arranged in decreasing order of their $Maintenance\_Overhead_{C_i}$ calculated above. Algorithm 3 is given below for the same.

Algorithm 3: Prioritize_Clones ($[MO_{C_s}]$)

Input: Maintenance Overhead.

Output: Prioritized list of clones.

1. for each $MO_{C_i} \epsilon MO_{C_s}$
2. set C_i.n_value = MO_{C_i};
3. set C_i.index = i;
4. end for
5. for i → 0 to n, do // Number of code clones
6. set max = C_i.n_value
7. id = C_i.index
8. compare max for j → (i+1) to n
9. if (C_j.n_value > max)
10. swap (C_j.n_value, max)
11. swap (C_j.index, C_i.index)
12. end if
13. end for
14. end for
15. for i → 0 to n, do
16. return C_i.n_value and C_i.index
17. end for

As a result, we get a prioritized list of code clones where C_i.n_value denotes the normalized value of maintenance overhead for code clone C_i and C_i.index denotes the

corresponding code clone file id. The algorithm follows the procedure of selection sort.

4.2 Maintenance Overhead Metrics

The evolving nature of software makes it more complex and therefore difficult to manage. As software evolves, the corresponding source code becomes lengthy and lacks in proper abstraction mechanisms. This not only leads to introduction of clones but also increases their maintenance overhead. It has been already discussed in section 1.4 that introduction of clones leads to increase in maintenance effort and cost.

The proposed system makes use of certain metrics which altogether estimates the maintenance overhead of the system. This is the maintenance effort incurred due to the presence of code clones.

Each of the metric used in the proposed work can be described as follows:

i) **Size of Code Clone (LOC**): This metric indicates number of source code lines in a clone fragment. Larger size of the cloned fragment indicates higher maintenance. Increase in code size increases complexity. Therefore clone size becomes the foremost criteria to calculate the maintenance overhead.

ii) **Frequency of Clone Repetition**: It indicates the number of times a cloned copy has been used. The more a clone fragment is used, the more time and effort it would require during maintenance and higher is the urgency to remove it.

iii) **Cyclomatic complexity**: It defines the logical complexity for a piece of code in terms of number of independent paths through that piece of code. It can be calculated using the number of decision points in the graphical representation of the source code. More complex code implies higher maintenance efforts.

Determining each of the above metric against each cloned file and calculating a normalized value for every file quantifies each of the cloned fragments in terms of maintenance effort required if that particular clone is not refactored. The

normalization is done on the basis of fixed weights assigned to the metrics depending on their severity.

4.3 Implementation Details

The proposed system can be divided into a number of phases and can be well interpreted from Figure 4.3.

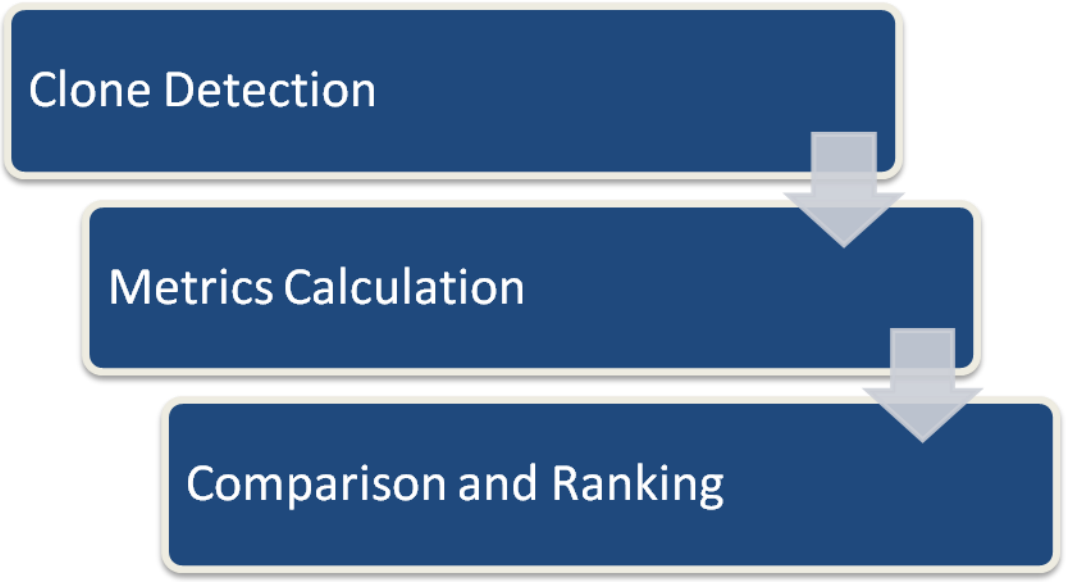


Figure 4.3: Steps in the Proposed Approach.

i) Clone Detection Phase

In this phase, the source code file is parsed for the detection of duplicated fragments. A number of tools exist based on different clone detection approaches having their own features. Here we can use any detection technique as we are concerned only with the detected cloned fragments. The selection of the detection technique depends on the fact that which type of clones is needed to be detected. The detected clones act as input for the next phase.

This thesis makes use of CCFinder: a token-based code clone detection tool which is capable of efficiently detecting Type I and II clones.

ii) Metrics Calculation Phase

Here the detected clones are extracted and maintenance overhead metrics are calculated for each clone file. All the metrics collectively quantifies the maintenance overhead in terms of effort and cost required to fix those clones.

Therefore a normalized value is calculated for each clone file. This normalization process gives a unified value for all the metrics involved and further helps in characterization of individual clone file.

iii) Comparison and ranking phase

All the files are compared against each other for the normalized value and the clones are ranked in the decreasing order of their maintenance overhead.

This technique prioritizes the clones for their removal and thus makes the refactoring more efficient.

Chapter 5

Experimental Results and Discussion

The proposed approach is not limited in prioritizing any one type of clone. It can be applied to all types of clones. Also there is no language constraint. But it depends on the user, whether he wants to detect syntactic or semantic clones, based on which the code clone detection tool can be selected. Moreover the code clone detection tools are not language independent and have to be precisely selected.

The first step includes detection of code clones using CCFinder: a token-based code clone detection tool [15]. This tool detects type II code clones and works well for various languages including cpp, csharp, java, cobol and plain text.

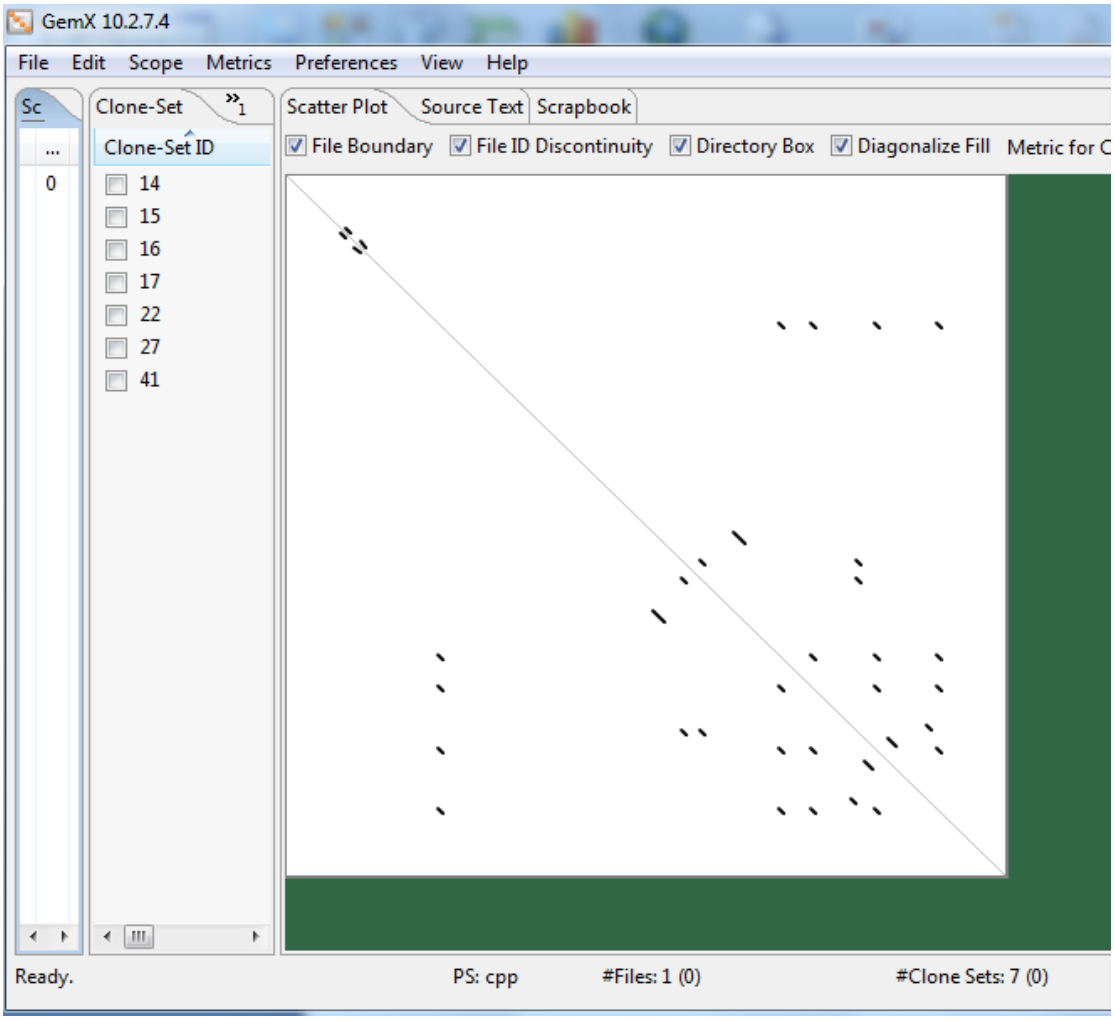


Figure 5.1: Scatter Plot of Detected Code Clones.

Figure 5.1 shows the scatter plot for the detected code clones in a cpp program of 'operations on Linked List'. The tool detects seven clone sets in one file. Figure 5.2 shows the corresponding source code with highlighted portions as the detected code clones.

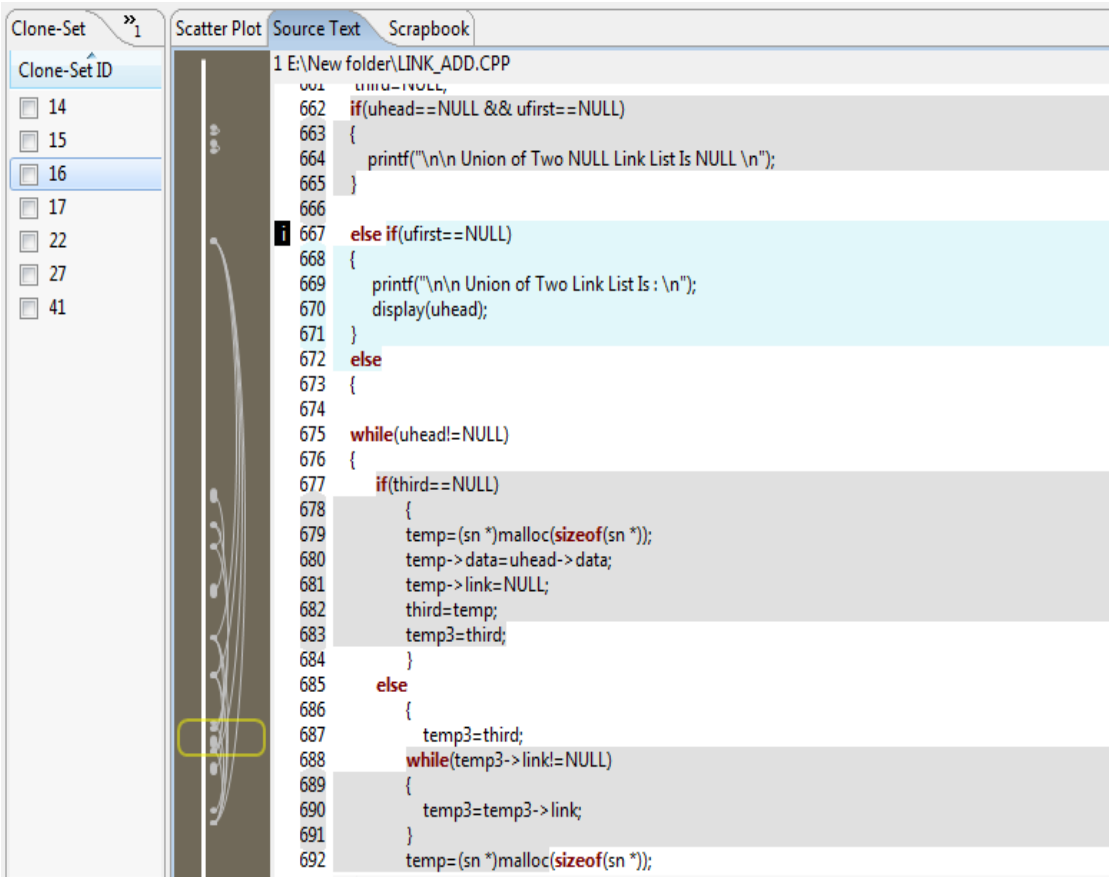


Figure 5.2: Source Code showing the Code Clones.

Once the code clones are detected, all of them are exported to make separate individual clone files. These clone files are further uploaded for prioritization process. Following are the screenshots showing the step by step procedure for the same.

Figure 5.3 shows the startup page which shows the calculation of maintenance overhead metrics of all the seven uploaded code clone files, as the given example includes seven clone sets.

Figure 5.4 displays the screenshot for calculation of length of code clones in terms of lines of code in the particular code clone fragment.

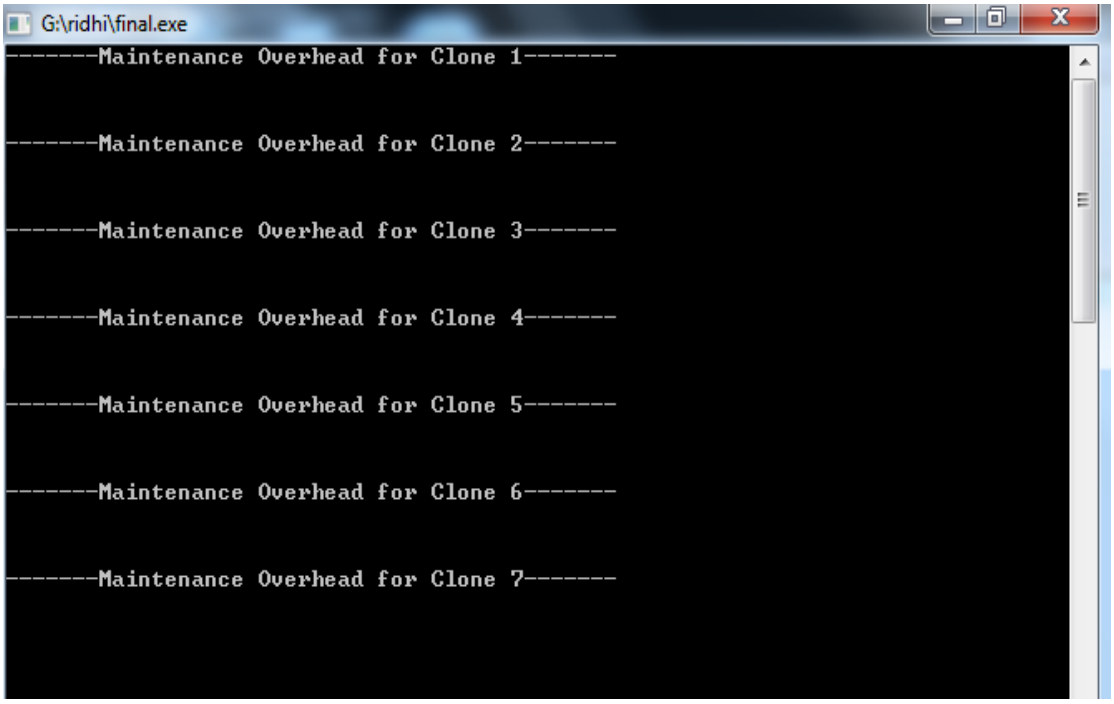


Figure 5.3: Start-up window for Calculation of Maintenance Overhead.

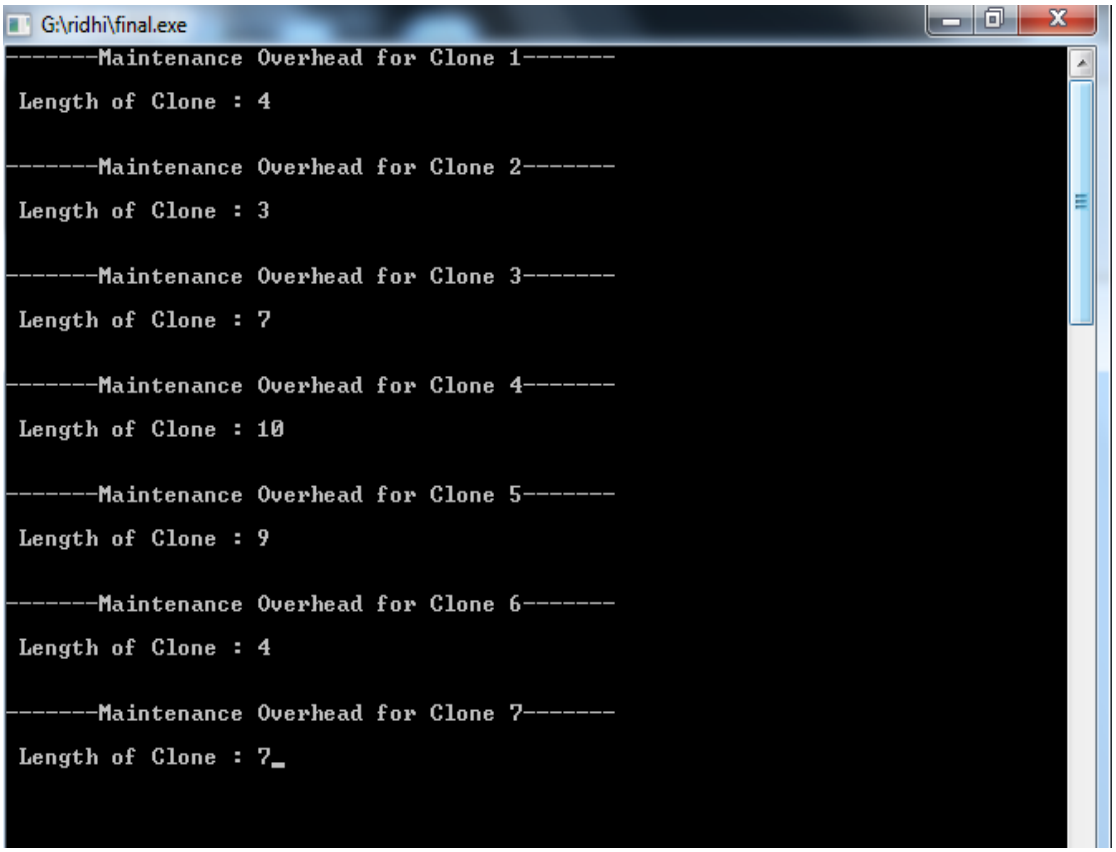


Figure 5.4: Calculation of Code Clone Length.

Figure 5.5 displays the Number of Repetitions for each code clone which states the frequency of occurrence of that particular code clone in the original program.

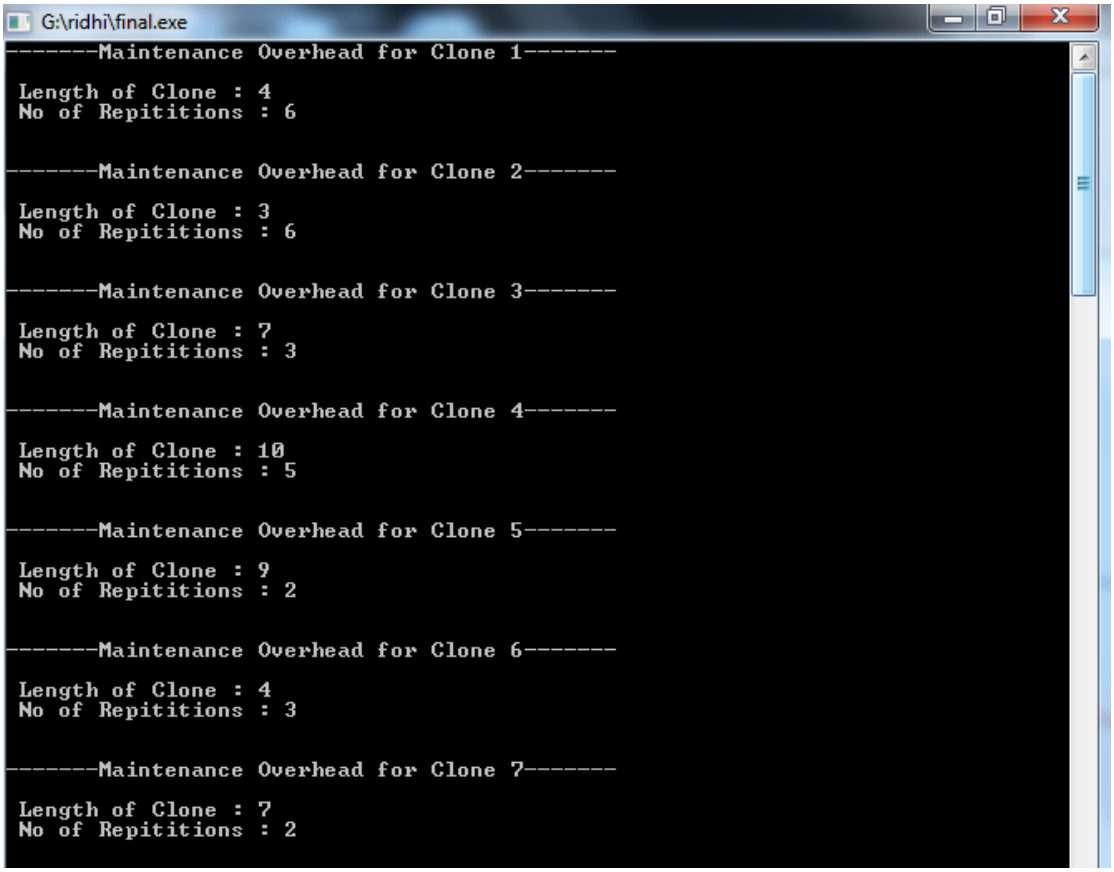


Figure 5.5: Calculation of Frequency of Repetition.

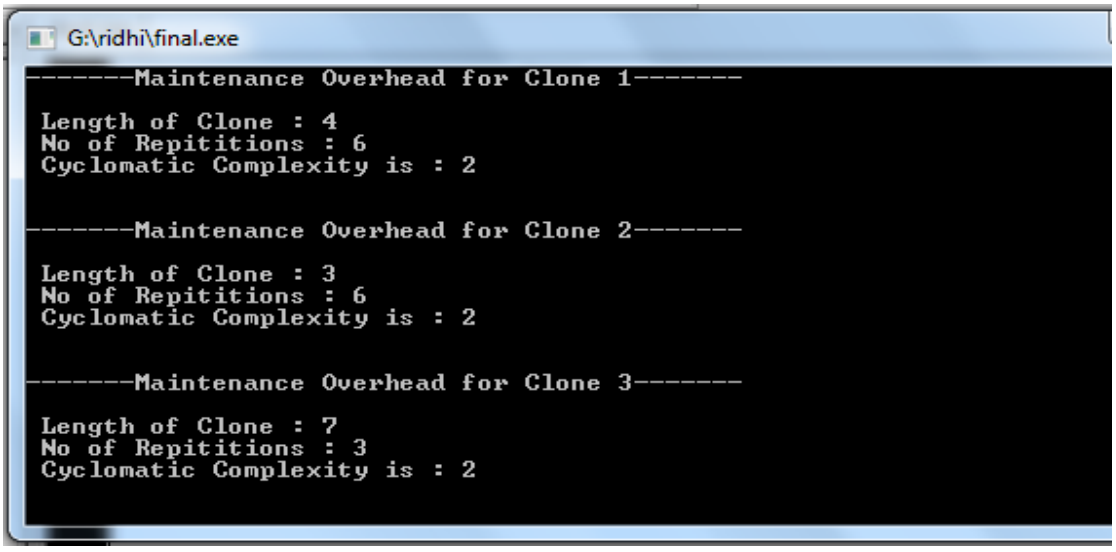


Figure 5.6(a): Calculation of Cyclomatic Complexity.

Figure 5.6(a), 5.6(b) adds to the list of maintenance overhead metrics by calculating the McCabe's cyclomatic complexity for each code clone file.

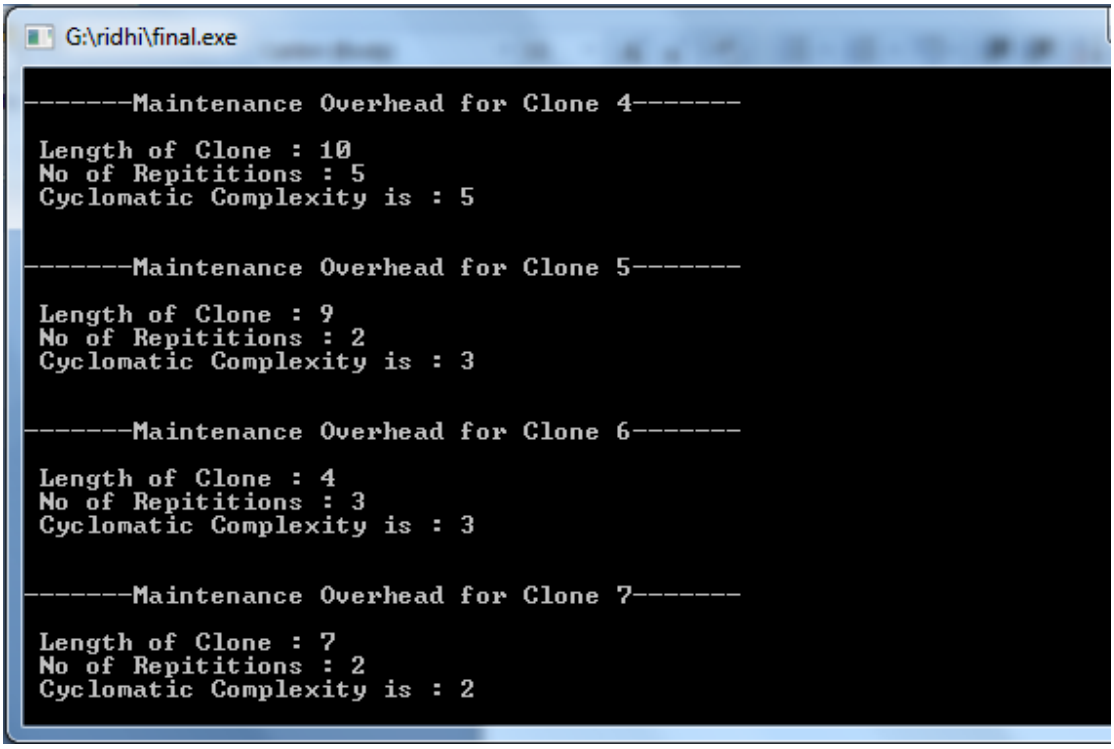


Figure 5.6(b): Calculation of Cyclomatic Complexity.

Figure 5.7(a), 5.7(b) calculates a normalized value which unifies the three metrics and

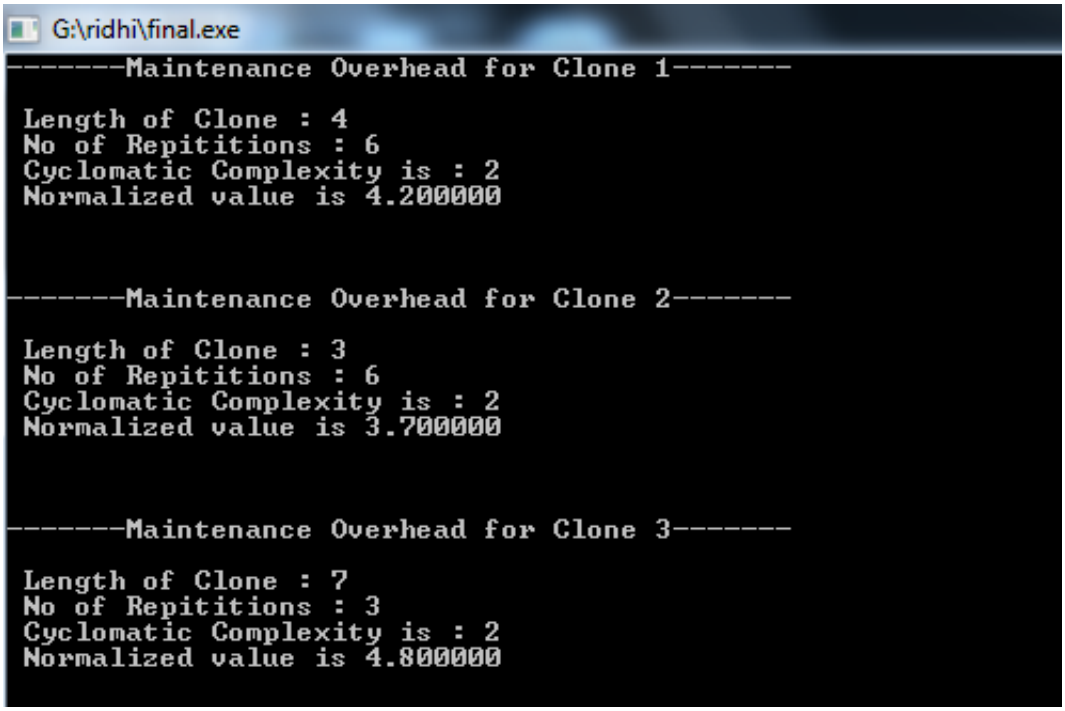


Figure 5.7(a): Calculation of Normalized value for Maintenance Overhead.

gives a single value determining the maintenance overhead of a particular code clone.

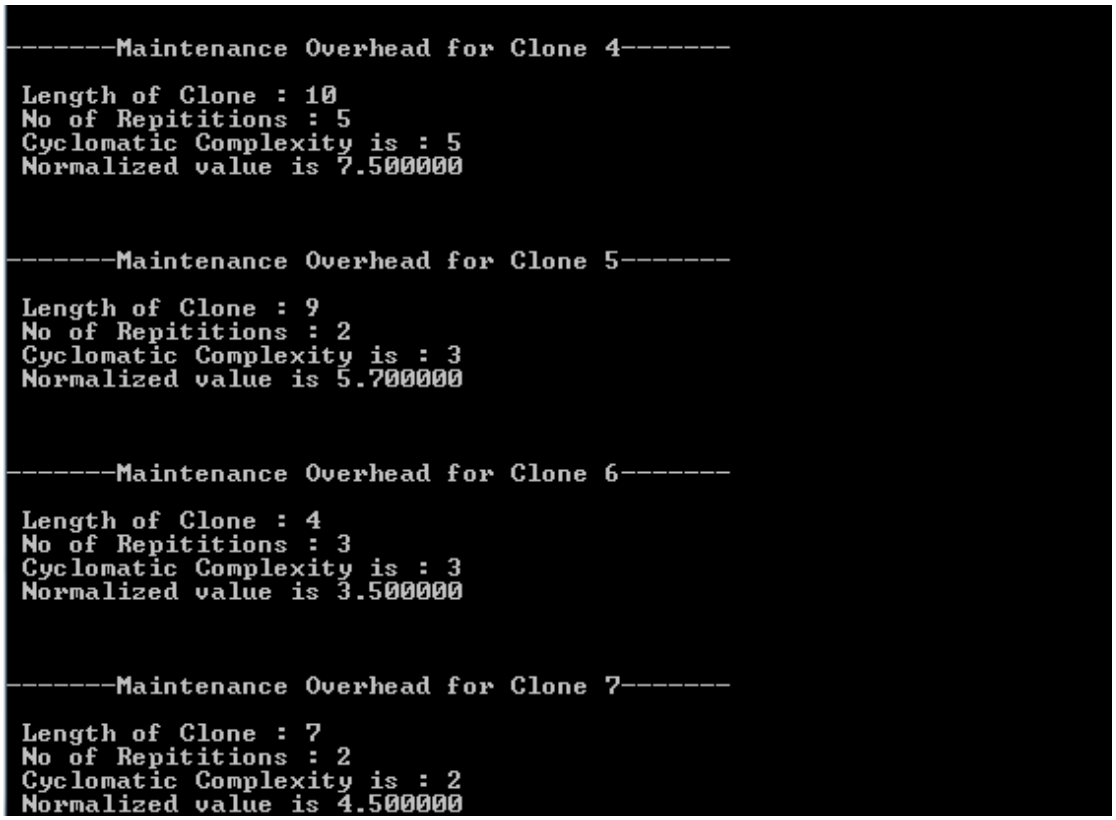


Figure 5.7(b): Calculation of Normalized value for Maintenance Overhead.

Following table shows the weights being assigned to the three factors used in calculating the normalized value for maintenance overhead.

Table 5.1: Weight Assigning Matrix.

Factors	Clone Length	Frequency of repetition	Code complexity
Weights assigned	0.5	0.3	0.2

The weights are assigned based on the impact factor the code clone have on the maintenance of that particular code clone.

The normalized value is calculated using the following formula:

Normalized Maintenance Overhead for a code clone = (0.5 * Length of Clone) + (0.3 * No. of Repetitions) + (0.2 * Cyclomatic Complexity)

The results obtained above can be summarized in Table 5.2 showing every code clone with its corresponding metrics value and the normalized maintenance overhead.

Table 5.2: List of code clones with their corresponding maintenance overhead metrics.

Clone id	Length of Clone	Frequency of Repititions	Cyclomatic complexity	Normalized value for Maintenance Overhead
1	4	6	2	4.2
2	3	6	2	3.7
3	7	3	2	4.8
4	10	5	5	7.5
5	9	2	3	5.7
6	4	3	3	3.5
7	7	2	2	4.5

The code clones can now be prioritized in the decreasing order of their maintenance overhead as shown in Figure 5.8.

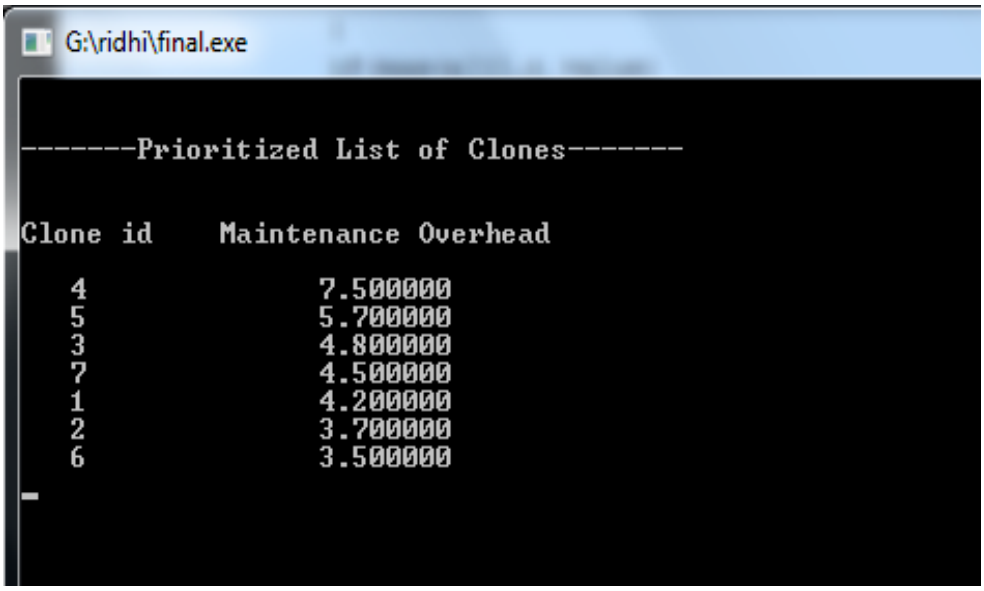


Figure 5.8: List of Prioritized Code Clones.

Table 5.3 gives a tabular representation of the code clones prioritized in an order to be presented for refactoring.

Table 5.3: Prioritized List of Code Clones.

Estimated Maintenance Overhead in decreasing order	**Prioritized List (Clone id)**
7.5	4
5.7	5
4.8	3
4.5	7
4.2	1
3.7	2
3.5	6

The resultant prioritized list of code clones generates an order in which the code clones should be removed or refactored. The inference that can be drawn from here is that the clone 4 has been estimated with the highest maintenance overhead and is needed to be removed first as compared to others. Moreover the clone 6 tends to have the lowest overhead value and thus can be removed in the last.

Although the demonstration given above does the ranking for seven code clone files only but the technique of prioritization is meant for large number of code clones as it becomes somewhat difficult and time consuming to deal with them. The prioritization process defines an order of urgency in which the code clones are required to be removed. Therefore it makes their refactoring efficient. It saves time also as sometimes the code clones with least priority can be ignored too in case they are not important.

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

The proposed technique enhances the code clone management process to a great extent. As a known fact, code clone detection is the essence of clone management, yet the detected code clones require a sophisticated treatment. They can either be removed or refactored within the code base as removing the code clones is not always possible. Refactoring the entire code clone set is also not necessary; some of the less harmful clones could be ignored.

The problem stated in Chapter 3 has been solved in Section 4.1 as an approach is introduced to prioritize the code clones for their removal. The tool detects the code clones and ranks them on the basis of their criticality or severity. It makes use of maintenance overhead metrics namely size of code clone, their frequency of repetition and cyclomatic complexity for calculating the severity of clones. The calculation of maintenance overhead, in advance, estimates the maintenance efforts that would be required if the clones were not fixed. Thus the proposed tool gives a higher priority to the code clone with higher maintenance overhead. The resultant prioritized list of code clones can be further presented for refactoring.

This would make the refactoring of code clones more efficient and much easier as the clones with higher maintenance efforts will be corrected first and the ones with least maintenance efforts could even be ignored. In terms of performance, it would reduce the overhead of rigorously refactoring each and every clone pair.

6.2 Summary of Contributions

i) The thesis proposed a code clone management technique of prioritizing the code clones that acts as a bridging activity between clone detection and removal and prevents the overhead of rigorously refactoring the entire clone set.

ii) It demonstrated the idea of ranking the code clones on the basis of their criticality so that the most critical ones could be refactored first.

6.3 Future Scope

A few more maintenance metrics can be incorporated within the system in order to enhance the accuracy of the proposed approach. In case of big industrial projects, static code analysis can also return several maintenance overhead metrics such as the age of code clones, how frequent changes are being made to them etc. Other than maintenance overhead, other factors can also be considered in determining the quality of code clones. One of the factors can be the refactoring costs that can potentially help in prioritizing code clones.

References

[1] H. Mili, A. Mili, S. Yacoub and E. Addy, “Reuse Based Software Engineering: Techniques, Organization, and Controls”, Wiley-Interscience, 2001.

[2] R. Koschke, “Survey of research on software clones,” In proceedings of Duplication, Redundancy, and Similarity in Software (DRSS), 2006, pp 1–24.

[3] M. Fowler, “Refactoring: Improving the Design of Existing Code,” Addison Wesley, 2000.

[4] H. Basit, S. Jarzabek, “Detecting higher-level similarity patterns in programs,” In Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/SIGSOFT FSE '05), Lisbon, Portugal, 2005, pp. 156-165.

[5] S. Giesecke, “Generic modelling of code clones,” In proceedings of Duplication, Redundancy, and Similarity in Software (DRSS), 2007, pp. 1–23.

[6] C. K. Roy, M. F. Zibran, R. Koschke, “The Vision of Software Clone Management: Past, Present, and Future (Keynote Paper),” In IEEE Conference of Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week, 2014, pp.18-33.

[7] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen, “Clone management for evolving software,” In IEEE Transaction on Software Engineering, 2012, pp 1008–1026.

[8] M. F. Zibran and C. K. Roy, “IDE-based real-time focused search for near-miss clones,” In Proceedings of the 27th Annual ACM Symposium on Applied Computing, 2012, pp 1235–1242.

[9] C. K. Roy and James R. Cordy, “A Survey on Software Clone Detection Research,” In Technical Report No. 2007-541, School of Computing Queen's University at Kingston Ontario, Canada, September 26, 2007.

[10] Yue JIA, “Clone Detection Using Dependence Analysis and Lexical Analysis,” PhD Thesis, King's College London, 2007.

[11] J. H. Johnson, “Identifying Redundancy in Source Code Using Fingerprints,” In Proceeding of the 1993 Conference of the Centre for Advanced Studies Conference (CASCON’ 93), Toronto, Canada, October 1993, pp. 171–183.

[12] S. Ducasse, M. Rieger and S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code,” In Proceedings of the 15th International Conference on Software Maintenance (ICSM’99), September 1999, pp. 109-118.

[13] C. K. Roy and J. Cordy, “NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization,” In Proceedings of 16th IEEE International Conference on Program Comprehension (ICPC’2008), 2008, pp. 172-181.

[14] B. Baker, “On finding duplication and near-duplication in large software systems,” In Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE'95), Toronto, Ontario, Canada, 1995, pp. 86-95.

[15] T. Kamiya, S. Kusumoto, K. Inoue, “CCFinder: A multi-linguistic token-based code clone detection system for large scale source code,” In IEEE Transactions on Software Engineering, 2002, pp. 654-670.

[16] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, “Gemini: Maintenance support environment based on code clone analysis”, In Proceedings of the 8th IEEE Symposium on Software Metrics (METRICS'02), Ottawa, Canada, 2002, pp.67-76.

[17] Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou, “CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code,” In IEEE Transactions on Software Engineering, vol. 32, March 2006, pp. 176-192.

[18] I. D. Baxter, A. Yahin, L. Moura, M. SantAnna, L. Bier, “Clone detection using abstract syntax trees,” in Proceedings of the 14th International

Conference on Software Maintenance (ICSM ‟98), Bethesda, Maryland, USA, 1998, pp. 368-378.

[19] V. Wahler, D. Seipel, J. Gudenberg and G. Fischer, “Clone Detection in Source Code by Frequent Itemset Techniques,” In Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation (SCAM), Chicago, IL, USA, September 2004, pp. 128-135.

[20] L. Jiang, G. Misherghi, Z. Su, S. Glondu, “DECKARD: Scalable and accurate tree-based detection of code clones,” In Proceedings of 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 2007, pp. 96-105.

[21] J. Krinke, “Identifying similar code with program dependence graphs,” In Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01), Stuttgart, Germany, 2001, pp. 301-309.

[22] R. Komondoor, S. Horwitz, “Using slicing to identify duplication in source code,” In Proceedings of the 8th International Symposium on Static Analysis (SAS'01), Vol. LNCS 2126, Paris, France, 2001, pp. 40-56.

[23] Y. Higo, S. Kusumoto, “Code clone detection on specialized PDG‟s with heuristics,” In Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11), Oldenburg, Germany, 2011, pp. 75-84.

[24] J. Mayrand, C. Leblanc, E. M. Merlo, “Experiment on the automatic detection of function clones in a software system using metrics,” In Proceedings of the 12th International Conference on Software Maintenance (ICSM'96), Monterey, CA, USA, 1996, pp. 244-253.

[25] J. F. Patenaude, E. Merlo, M. Dagenais, B. Lague, “Extending software quality assessment techniques to Java systems,” In Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99), Pittsburgh, PA, 1999, pp. 49-56.

[26] F. Lanubile, T. Mallardo, “Finding function clones in web applications,” In

Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03), Benevento, Italy, IEEE, 2003, pp. 379-386.

[27] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, "Detection of Type-1 and Type-2 Clone Using Textual Analysis and Metrics," In proceedings of IEEE International Conference on Recent Trends in Information, Telecommunication and Computing (ITC), Kerala, India, March 2010, pp. 241-243.

[28] H. Basit, S. Jarzabek, "A data mining approach for detecting higher-level clones in software," In IEEE Transactions on Software Engineering, 2009, pp. 497-514.

[29] Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, "Comparison and evaluation of clone detection tools," In IEEE Transactions on Software Engineering, 2007, pp. 577-591.

[30] J. Harder and N. Gode, "Efficiently handling clone data: RCF and cyclone," In Proceedings of the 5th International Workshop on Software Clones (IWSC), ACM, 2011, pp. 81-82.

[31] E. Duala-Ekoko and M. Robillard, "Clone region descriptors: Representing and tracking duplication in source code," In ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 20, June 2010.

[32] R. Tairas and J. Gray, "Get to know your clones with CeDAR," In Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA), 2009, pp. 817–818.

[33] J. Pate, R. Tairas, and N. Kraft, "Clone evolution: a systematic review," In Journal of Software: Evolution and Process 25, vol. 3, 2013, pp. 261–283.

[34] E. Adar and M. Kim, "SoftGUESS: Visualization and exploration of code clones in context," In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), vol. 7, 2007, pp. 762 –766.

[35] J. Harder and N. Gode, “Effciently handling clone data: Rcf and cyclone,” In Proceedings of the 5th International Workshop on Software Clones (IWSC), ACM, 2011, pp. 81–82.

[36] R. K. Saha, C. K. Roy, and K. A. Schneider, “Visualizing the evolution of code clones,” In Proceedings of the 5th International Workshop on Software Clones (IWSC), ACM, 2011, pp. 71-72.

[37] N. Gode and R. Koschke, “Studying clone evolution using incremental clone detection,” In Journal of Software Maintenance and Evolution: Research and Practice, 2010, pp. 1-28.

[38] T. Nguyen, H. Nguyen, J. Al-Kofahi, N. Pham, and T. Nguyen, “Scalable and incremental clone detection for evolving software,” In IEEE International Conference on Software Maintenance, (ICSM), IEEE, 2009, pp. 491-494.

[39] J. Johnson, “Visualizing textual redundancy in legacy source,” In Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research (CASCON), IBM Press, 1994, pp. 32-41.

[40] M. Rieger, S. Ducasse, and M. Lanza, “Insights into system-wide code duplication,” In Proceedings of 11th Working Conference on Reverse Engineering (WCRE), IEEE, 2004, pp. 100-109.

[41] Y. Zhang, H. Basit, S. Jarzabek, D. Anh, and M. Low, “Query-based filtering and graphical view generation for clone analysis,” In IEEE International Conference on Software Maintenance (ICSM), IEEE, 2008, pp. 376 –385.

[42] P. Jablonski and D. Hou, “CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE,” In Proceedings of the 2007 OOPSLA workshop on eclipse technology exchange (ETX), ACM, 2007, pp. 16-20.

[43] P. Jablonski, “Managing the copy-and-paste programming practice in modern IDE‟s,” In Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, Montreal, Quebec, Canada, 2007, pp. 933-934.

[44] D. Hou, P. Jablonski, F. Jacob, “CnP: Towards an environment for the proactive management of copy-and-paste programming,” In Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, BC, Canada, 2009, pp. 238-242.

[45] M. F. Zibran and C. K. Roy, “A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring,” in 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2011, pp. 105–114.

[46] R. Miller and B. Myers, “Interactive simultaneous editing of multiple text regions,” In USENIX Annual Technical Conference on Interactive Simultaneous Editing of Multiple Text Regions USENIX, USENIX Association, 2001, pp. 161–174.

[47] C. Kapser and M. W. Godfrey, ““Cloning Considered Harmful" Considered Harmful,” In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06), IEEE, 2006, pp. 19-28.

[48] D. Rattan, Rajesh Bhatia, and Maninder Singh, “Software clone detection: A systematic review,” Information and Software Technology, vol. 55, no. 7, 2013, pp. 1165-1199.

[49] M. Mondal, C. K. Roy and K. A. Schneider, “Automatic Ranking of Clones for Refactoring through Mining Association Rules,” In IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), Software Evolution Week-IEEE, 2014, pp. 114-123.

[50] R. D. Venkatasubramanyam, S. Gupta, H. K. Singh, “Prioritizing Code Clone Detection Results for Clone Management,” In Proceedings of 7th International Workshop on Software Clones (IWSC), IEEE, San Francisco, USA, 2013, pp. 30-36.

Publications

Published/Accepted

[1] Ridhi Garg and Rajkumar Tekchandani, "Analysis of Existing Literature on Software Code Clone Management", In Proceedings of Global Summit on Computer & Information Technology (GSCIT), IEEE, Tunisia, 2014.

[2] Ridhi Garg and Rajkumar Tekchandani, "An Approach to Rank Code Clones for Efficient Clone Management," In IEEE International Conference on Advances in Electronics, Computers and Communications (ICAECC), IEEE, India, 2014.

Communicated

[1] Ridhi Garg and Rajkumar Tekchandani, "Code Clone Prioritization Approach for Efficient Clone Management", In Proceedings of Seventh International Conference on Contemporary Computing (IC3), IEEE, India, 2014.