Refactoring Code Clone Detection

1st Zhala Sarkawt OTHMAN
Software Engineering
Firat University
Elazig, Turkey
zhala.sarkawt@gmail.com

2nd Mehmet KAYA
Electrics Engineering
Adiyaman University
Adiyaman, Turkey
m.kaya@adiyaman.edu.tr

***Abstract*— Refactoring duplicate code is an important issue and is one of the most important smells in software maintenance. There is an important relationship between clones and code quality. Most programmers use clones because they are cheaper and faster than typing the program code.**

A cloning code is created by copying and pasting the existing code fragments of the source code with or without slight modifications. A major part (5% to 10%) of the source code for large computer programs consists of copy codes. Since cloning is believed to reduce the possibility of software maintenance, many techniques and cloning detection tools have been recommended for this purpose.

The basic goal of clone detection is to identify the clone code and replace it with a single call to the function, where the function simulates the behavior of one instance of the clone group. This research provides an overview of the refactoring IDE. The aspects of cloning and detection of cloning are explained. In the copy detection algorithm, the source code is created in XML format.

Keywords*— *Refactoring, Code Clone, Clone Detection

I. Introduction

Clones impact a software's internal quality. Excessive use of code clones reduces the maintainability of software. Refactoring duplicate code is an important issue and is one of the most important smells in software maintenance. There is a statistically significant relationship between clones and code quality, but there are not many studies on this subject. Most of the programmers use them because it is cheaper and quicker to use than writing the code of the software. Code clones, which are defined as parts of code similar enough to be duplicated or considered for the same function, are problematic. In spite of the conviction that code ought not ever to be reordered and speedy decisions that all copy must be expelled, there are regularly great subtleties to reorder the code. [1]. We provide an overview of the copy detection function that enables detection of code cloning along with the detection of phases and reports the experiences of their application. Code cloning or duplication code is created by copying and pasting the existing code portions of the source code with or without a simple modification. As a result, this duplication of execution logic often leads to the need to modify multiple areas of code in a consistent manner, meaning that there are copies of the code in a system. Cloned versions of codes are a major source of software defects.

A code clone is an undesirable original sin because it causes difficulties in the maintenance of the software and increases the risk of bugs, which increase system complexity and maintenance costs. A major part (5% to 10%) of the source code for large computer programs consists of copy codes. Since clone detection is believed to reduce the frequency of software maintenance, numerous methods and instruments for clone detection have been proposed. Removal of clones can ensure decreased software maintenance and reduced cost. Various tools can automatically detect duplicate sets of codes. Therefore, a major part of the research on this subject depends on the discovery of cloning of the code via various programs, approaches, techniques, and tools proposed to provide accurate results. The basic goal of clone detection is to identify the clone code and replace it with a single call to the function, where the function simulates the behavior of one instance of the clone group.

II. Code Clone

Duplication usually occurs when many programmers work on different parts of the same program at the same time. Since they are working on different tasks, they may not be aware that their colleague has already written a similar symbol that can be re-employed for their own needs. There is also a more refined frequency when certain parts of the code look different but perform the same function. It may be difficult to find this type of duplication and fix it.

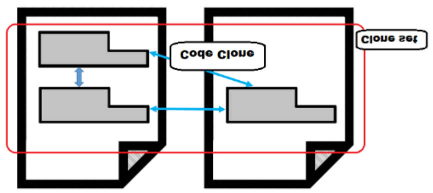


Fig. 1. Code Clone

A single report may additionally include multiple copies of the equal code clone. Except detecting replica code, it may find algorithms and cloning tools for copying different makes use of. CCFinder became used to determine whether a company stole the supply code. One organization filed fit in opposition to another corporation for taking its code and submitted evidence based totally on CCFinder. CCFinder is unique that more than 50% of the documents contained clones that were added within the other business enterprise's supply code. Clone detection gear is also similar to plagiarism detectors used to locate pupil fraud [2].

Maximum codes comprise statements that occur frequently. It's far likely that 300 copies of print (f) will now not be reproduced in different locations repeatedly via the developer. In most instances, it will no longer be useful to extract these phrases within the commonplace manner. If that is a mystery that needs to be hidden, it can be taken into consideration a new clone to be marked. Therefore, algorithms should be configurable so that builders can choose what constitutes a new machine image, encryption conventions, or challenge handily. The tool will not be beneficial if the developer has to search for many false positives. The clone detection set of rules should also be capable of understanding clone replica to locate large clones in place of the smaller clones. Small strings may match pieces of code. If those strings are a part of larger strings and are transcripts, the strings which might be contained by the larger strings must now not be considered as reproduced in themselves. Therefore, clone detection algorithms help find the minimum number of clones that still capture all the character symbols within the code [3].

A. *Type of code clone*

It is worth noting that the standards in the case of naming are still lacking, resulting in unique ratings by way of exceptional researchers. We listing the basic kinds of clones. Use of open-source software and variations of code reuse have improved. The current code can be modified to meet new requirements consequently facilitating and growing open source development. The urgent want to find out software program variations has inspired the invention of software as an active studies area [4]:

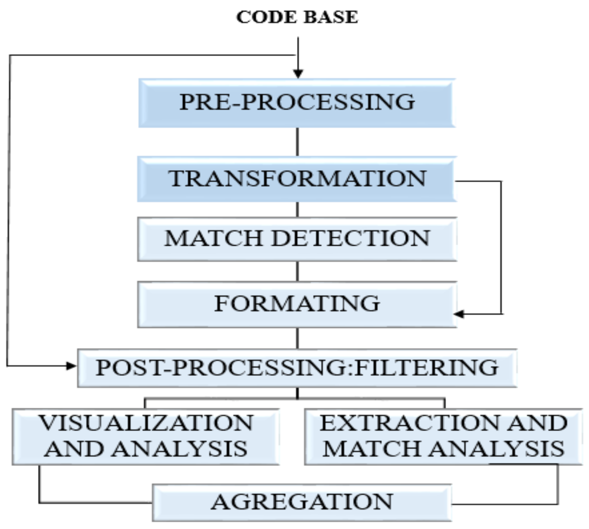
- Class 1 (precise clones): software components are equal besides for variations in white spaces and feedback.
- Class 2 (renamed/parameters clones): program components which can be structurally/constructively similar besides for adjustments in identifiers, characters, bureaucracy, layout, and feedback.
- Class 3 (close to omitting clones): software components are copied with further modifications along with insert/delete statements as well as changes in IDs, numeric characters, types, and designs.
- Category four (semantic clones): software-like parts of this system that are not textually identical.

III. CODE CLONE DETECTION

Code cloning detection refers to the automatic process to look for similarities between symbol sections. Cloning increases the size of the program unnecessarily. Because many software maintenance efforts are related to program size, this increases maintenance efforts. It requires developers to find and update multiple parts while changing any module. In code analysis, source code analysis is the automatic testing of the source code to debug a computer program or application before it is distributed or sold. The source code is the most permanent form of software, although it is possible to modify, improve, or upgrade the software later. Source code analysis can be either static or dynamic.

A. *Clone Detection Process*

A clone detector should locate pieces of code of excessive similarity in a machine's supply textual content. A clone is detected through static analysis of source code.


CODE BASE
PRE-PROCESSING
TRANSFORMATION
MATCH DETECTION
FORMATING
POST-PROCESSING:FILTERING
VISUALIZATION AND ANALYSIS
EXTRACTION AND MATCH ANALYSIS
AGREGATION

Flowchart 1. Code clone base

The main trouble is that it isn't recognized which elements of the code can be duplicated. For that reason, the detector ought to evaluate each feasible part with each other possible part. On this segment, a comprehensive summary of the vital steps might be furnished within the reproduction discovery procedure. This full-scale picture allows us to examine and evaluate detection tools [6]. It is also possible to assess their level of support for these steps [5]:

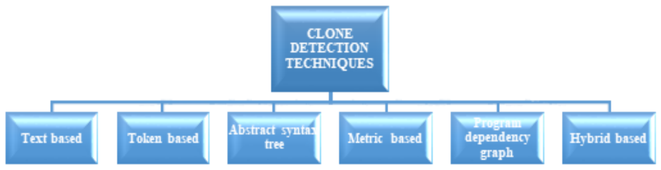
- Pre-Processing: At the start of any clone detection technique, the supply code is damaged down, and the scope of the evaluation is decided.
- *Transformation*: as soon as the similarities are determined, if the evaluation technique is non-textual, the source code of the evaluation devices may be transformed right into a suitable proxy illustration.
- *Extraction*: Extraction transforms the source code to the precise shape as an input for the real contrast set of rules.
- Pretty-printing of source code (source code printing): pleasant printing is a simple way to rearrange the source

code into a preferred model that eliminates variations in layout and spacing.

- Match Detection: The converted code is then inserted into the assessment algorithm in which the transformed devices are in comparison to every different to discover matches. The output of healthy detection is a listing of fits in the transformed code which is represented or aggregated to shape a fixed of candidate clone pairs.
- Formatting: in this section, the clone pair list for the changed code obtained by the comparison algorithm is transformed into a corresponding clone pair list for the original code base.
- Post-processing: At this stage, cloning is classified or filtered the usage of guide evaluation or auto detecting.
- Aggregation: despite the fact that a few types of equipment specify instructions of cloning without delay, most of them most effectively bring about the cloning of pairs as an end result.

B. *Clone Detection Tools and techniques*

A clone detection device is something that wishes to be started and used one at a time. Some of the maximum sophisticated tool requirements for clones make it less complicated for the developer to pick clones they want to their supply and survey the most precious copies for refactoring. CCFinder affords statistics about clone training or sets of clones. The maximum length of any clone inside the magnificence is given in LOC or programming language codes. It affords the range of clones within the clone magnificence. It offers deflation, which is expected to be internet LOC removed by way of changing all times of cloning with a popular characteristic call and including phrases to the brand new position of all instances of clone index. We identified one-of-a-kind styles of software program variations, and the factors leading to software program cloning. We summarize the disadvantages and benefits of cloning software program [1].



Flowchart 2. Clone detection techniques

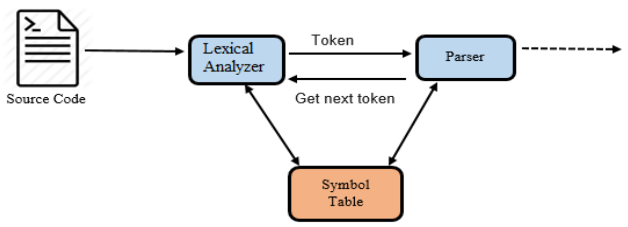
Moreover, the existence of a redundant law not only complicates the design but reduces understanding, hindering improvements and modifications. In the long term, the program may become too complicated to make simple changes. The cloning of the code increases the size of the software system, resulting in pressure on system resources. It degrades the overall performance in terms of compilation/execution time and space requirements.
There are many of Clone detection techniques?

A. *Text/String-based techniques*

In this strategy, the supply sections are examined as a subsequence of content material. The two portions are contrasted. A few analysts have proposed various string/content-primarily based techniques for clone detection. It applied a completely unique parameter for spotting clones (that have diverse elements names). However, it becomes not geared up to differentiate clones written in an alternate fashion and it could not bolster research of the direction between the duplicated codes [7, 8].

B. *Lexical/token-based techniques*

The token-based totally methodology is referred to as the lexical approach. In this system, the whole supply code is separated into tokens by way of a lexical exam and after that, every token is shaped into many token sequences. Sooner or later, the succession is checked for distinguishing the copied code. One of the important apparatus in token-primarily based methodology called CCFinder become proposed by using Kamiya [4].



Flowchart 3. Lexical analyzer

C. *Syntactic/Tree-based techniques*

On this approach, this system is represented inside the shape of an AST in place of growing tokens for every statement, and with the assist of the tree-matching set of rules, a comparable subtree is searched inside the identical tree. One of the preliminary AST-primarily based tools named CloneDR was proposed by Baxter. It creates AST with the help of a compound generator and then compares its subnet using scales that depend upon the hash capabilities. But, it became now not capable of hit upon similar clones [9]. To triumph over this trouble, Bauhaus supplied the ccdiml tool with the aid of averting using defragmentation and similarity measures. However, it turned into incompetent in verifying the identification. Yang added one of the grammar-based totally processes [10]. Its miles used to fill grammatical variations among copies of the equal program via growing their personal analysis tree and then making use of a dynamic programming approach to become aware of a similar subtree [11].

D. *Syntactic/Metric-based techniques*

With metric-based strategies, disparate measurements are gathered, for example, various capacities and numbers of lines, from code portions and those measurements are assessed regardless of appraisals of source code specifically. The plausible match's identified by a unique example apparatus [12, 13, 14].

E. Semantic/Hybrid approach

There are numerous combination strategies for code clone detection. The crossbreed technique is an accumulation of a few methodologies and it very well may be classified based on previous techniques. The tree and token-based 1/2-and-half of method become proposed by way of Koschke for finding kind-I (exact) and type-II code clones [7]. On this methodology, the creators produced a suffix tree for serialized AST hubs that are consecutive within the preorder traversal. A closely equivalent to technique was proposed with the aid of Greenan with the various Code Clone Detection techniques and gear 661 arrangement, coordinating the calculation for the detection of technique-stage clones [15].

IV. CLONE DETECTION VIA VISUAL OBSERVATION

For clone detection, we believe that we can compromise the precision of clone detection algorithms and only find potentially problematic code fragments that are similar enough to be presented to the user as potential clones. Two pieces of information will be enough to pinpoint code fragments which may constitute clones: scope tree structure and the variables and their reference counts.To identify potential code fragments as extract method refactoring suggestions, we evaluate and test them for clone detection purposes. Before we move to our discussion on how to detect potential clones and display them to the user, we shall define some terminologies that we use in the algorithm [1].

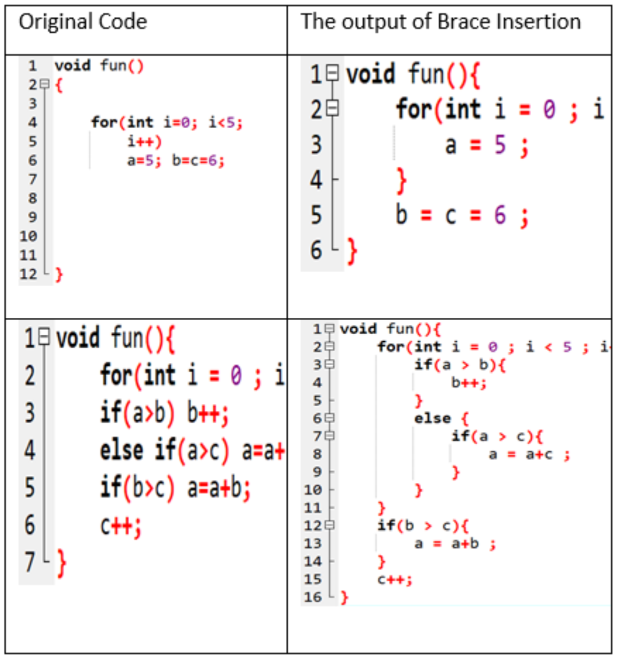
- *Scope*: This is a particular code block that is usually enclosed in braces. An example of this could be an entire function, a loop or control structure, or a pair of anonymous opening and closing braces, i.e. '{'and '}'.
- *Variable Span:* This defines all of the statements between a variable's declaration and its last reference in a given code fragment. The last reference here is identified using static analysis. Therefore, the last reference would be the highest numbered line that contains a reference to the variable inside the given function.

The outline of our algorithm is given below:

- Brace insertion
- Scope Detection
- Variable Span Detection
- Variable Span Expansion
- Variables Reference Counts
- Final Visualization

The most important thing that we do here is that we identify control and loop structures that are missing explicit scope identification characters e.g. braces. We created a preprocessed version of the input source file in which all scopes are explicitly marked using scope identification characters.

This is not a trivial operation. When the scopes get a deeply nested structure, tracking the actual scope boundaries and maintaining the original code semantics become crucial. Figure 5.1 shows two samples run for the brace insertion.



Original Code

```
void fun()
{

    for(int i=0; i<5;
        i++)
        a=5; b=c=6;





}
```

The output of Brace Insertion

```
void fun(){
    for(int i = 0 ; i
        a = 5 ;
    }
    b = c = 6 ;
}
```

Original Code

```
void fun(){
    for(int i = 0 ; i
    if(a>b) b++;
    else if(a>c) a=a+
    if(b>c) a=a+b;
    c++;
}
```

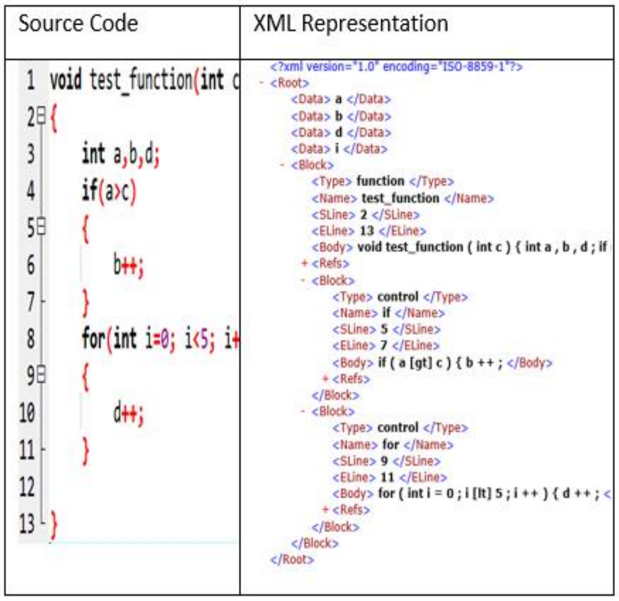
The output of Brace Insertion

```
void fun(){
    for(int i = 0 ; i < 5 ; i
        if(a > b){
            b++;
        }
        else {
            if(a > c){
                a = a+c ;
            }
        }
    }
    if(b > c){
        a = a+b ;
    }
    c++;
}
```

Fig. 2. Two samples run for the brace insertion

When we compare the original code with the output of our brace insertion operation, it is clear that a certain level of improvements has already been achieved as a result of this preprocessing.

The second step is to detect the scope boundaries in the code. This step is straightforward thanks to the brace insertion step which marks these boundaries explicitly. An intermediate representation of the source code is created in XML format for this step to be used in the visualization phase. In Figure 5.2, a sample function and its XML representation are provided.



Source Code

```
1  void test_function(int c
2  {
3      int a,b,d;
4      if(a>c)
5      {
6          b++;
7      }
8      for(int i=0; i<5; i+
9      {
10         d++;
11     }
12
13 }
```

XML Representation

```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <Root>
    <Data> a </Data>
    <Data> b </Data>
    <Data> d </Data>
    <Data> i </Data>
  - <Block>
      <Type> function </Type>
      <Name> test_function </Name>
      <SLine> 2 </SLine>
      <ELine> 13 </ELine>
      <Body> void test_function ( int c ) { int a , b , d ; if
    + <Refs>
    - <Block>
        <Type> control </Type>
        <Name> if </Name>
        <SLine> 5 </SLine>
        <ELine> 7 </ELine>
        <Body> if ( a [gt] c ) { b ++ ; </Body>
      + <Refs>
      </Block>
    - <Block>
        <Type> control </Type>
        <Name> for </Name>
        <SLine> 9 </SLine>
        <ELine> 11 </ELine>
        <Body> for ( int i = 0 ; i [lt] 5 ; i ++ ) { d ++ ; <
      + <Refs>
      </Block>
    </Block>
</Root>
```

Fig. 3. Sample function and XML representation

As seen in Figure 5.2, the XML provides information about the code such as where each scope starts and ends, its name and type, and all variable references within it. This gives us a basis to create the scope tree for clone detection.

Next, we detect the variable spans and expand them based on their interactions with the control blocks and other variable spans. The variable span is also represented in XML format. A portion of the variable span XML for the code in Figure 5.2. This XML provides valuable information regarding the variable spans such as where each one starts originally and where it should be expanded for method extraction.

In the next step, this information from the intermediate XML representations is visualized. In Figure 5.4, the visualization for the test function in Figure 5.2 is provided.

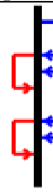
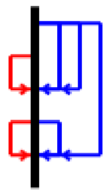
Original scopes and variable spans	Expanded versions of the variable spans
	

Fig. 4. Visualization for the test function

Notice that in Figure 5.2, variable d is declared at line 2 and its first and only use appears at line 10. Thus, from the visualization in Figure 5.4, this situation is perceived as the interaction between that variable's span add the other three variables' spans (a, b, and i) as well as the two control blocks (if and for). Therefore, the span for variable d will always unnecessarily cover two other variable spans (a and b) and a control block (if). Yet, as it can be observed from the code, it has real interactions with only the span of variable I and the scope of a loop.

We introduce another view where the user can observe the references for each variable directly from the visual representation by clicking the variable span lines. In Figure 5.5, we show the resulting views after clicking each line.

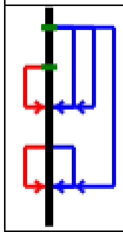
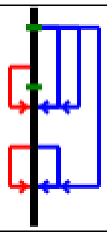
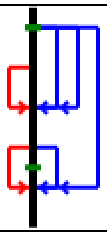
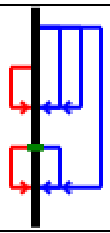
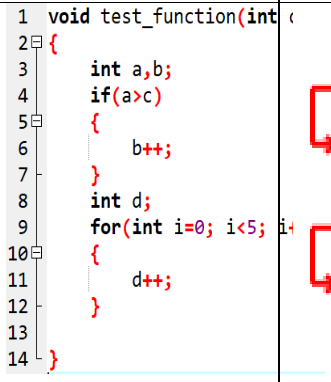
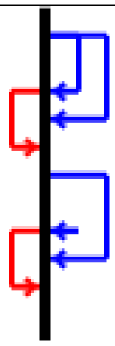
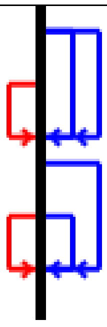
References for a	References for b	References for d	References for i
			

Fig. 5. Show the resulting views after clicking each line

It can be observed from Figure 5.5 that variable d has not been used in the two variables' scopes, and the 'if control' scope is following its declarations. This indicates that variable d's declaration can be moved after the block to eliminate this misrepresented interaction. In Figure 5.6, we show how these views will change after moving the declaration for b to the said location.

Modified Code	Original Scopes and Var. Spans	Expanded versions of the variable spans
		

```
void test_function(int (
{
    int a,b;
    if(a>c)
    {
        b++;
    }
    int d;
    for(int i=0; i<5; i+
    {
        d++;
    }

}
```

Fig. 6. Views will change after moving the declaration

This view utilizes the scopes and variable reference counts and gives each scope a unique color based on the variable that is referenced most inside that scope. For the sample function in Figure 5.2, this box view provided in Figure 5.7.

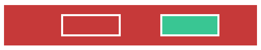


Fig. 7. Box view

In this view, the color indicates the potential extract method opportunities. Here, we show how it can be used to suggest potential code clones. For this, we will use the following sample code in which we find and display all prime numbers in three different vectors provided as arguments.

The original view created in this method for extract method opportunities is given below. We have modified this view to better display code regions that might be cloned. After our modifications, the new view is structurally the same, but it is easier to see the code clones. The two views are provided in Figure 5.9.

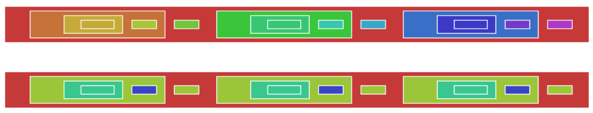
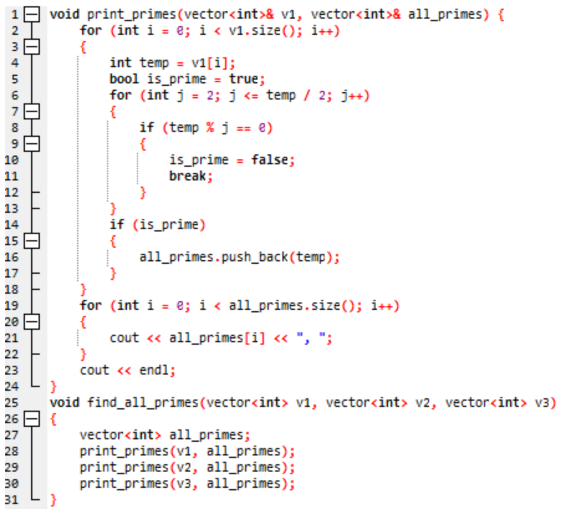


Fig. 8. Code clones

This new view clearly shows the clones using the size and modified color properties of the scopes. When we look at the view, we can easily identify three clones of the same code. We can extract that as a new method. Here, after the extraction, the code eliminates the clone and improves the code quality drastically.



```
void print_primes(vector<int>& v1, vector<int>& all_primes) {
    for (int i = 0; i < v1.size(); i++)
    {
        int temp = v1[i];
        bool is_prime = true;
        for (int j = 2; j <= temp / 2; j++)
        {
            if (temp % j == 0)
            {
                is_prime = false;
                break;
            }
        }
        if (is_prime)
        {
            all_primes.push_back(temp);
        }
    }
    for (int i = 0; i < all_primes.size(); i++)
    {
        cout << all_primes[i] << ", ";
    }
    cout << endl;
}
void find_all_primes(vector<int> v1, vector<int> v2, vector<int> v3)
{
    vector<int> all_primes;
    print_primes(v1, all_primes);
    print_primes(v2, all_primes);
    print_primes(v3, all_primes);
}
```

Fig. 9. Clone code after detection

Therefore, with this new method, we are able to detect the clones very efficiently and remove them. As can be observed from Figure 5.10, the new code is a lot smaller in size and more maintainable and reusable.

V. CONCLUSION

A software program is seldom written below best conditions. Limitations of programmer's abilities and time constraints inhibit right software evolution [3, 16]. We provide an outline of the clone detection characteristic that enables detection of cloned codes alongside detection levels and records the studies with their software. Code cloning or code duplication involves copying and pasting existing code fragments of a supply code, without *or with minor modulations* [17]. The consequences of this duplication of execution logic frequently lead to the need to adjust more than one areas of code in a consistent manner, which means that there are copies of code in a gadget, so the code that achieves the equal or comparable common sense is not in a single vicinity. Maximum programmers use clones due to the fact the use of clones is less expensive and quicker than writing new software code. Code cloning is an authentic sin as it usually causes difficulties in the upkeep of software and results in an excessive risk of insects, growing gadget complexity and preservation costs [17, 18]. The want for detecting reproduction codes has resulted inside the improvement of diverse gear that may mechanically find reproduction units of codes [19].

REFERENCES

[1] Kaya, M, "Identifying Extract Class and Extract Method Refactoring Opportunities through Analysis of Variable Declarations and Uses" (2014). Dissertations - ALL. Paper 53.

[2] Schleimer, S, Wilkerson, D.S and Aiken, A.,Winnowing: local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, 2003 June, pp. 76-85.

[3] Godfrey, M.W and Zou, L, Using origin analysis to detect merging and splitting of source code entities. IEEE Transactions on Software Engineering, 2005, 31(2), pp.166-181.

[4] Kamiya, T, Kusumoto, S and Inoue, K, CCFinder: a multilinguistic token-based code clone detection system for large-scale source code. IEEE Transactions on Software Engineering, 2002, 28(7), pp.654-670.

[5] Walenstein, A, Jyoti, N, Li, J, Yang, Y. and Lakhotia, A, Problems Creating Task-relevant Clone Detection Reference Data. In WCRE, 2003 November, Vol. 3, p. 285.

[6] Roy, C.K and Cordy, J.R, Scenario-based comparison of clone detection techniques. In The 16th IEEE International Conference on Program Comprehension, 2008 June, pp. 153-162. IEEE.

[7] Koschke, R, Falke, R. and Frenzel, P, Clone detection using abstract syntax 5suffix trees. In Reverse Engineering, 2006. WCRE'06. 13th Working Conference on, 2006 October, pp. 253-262. IEEE.

[8] Roy, C.K and Cordy, J.R, A mutation/injection-based automatic framework for evaluating code clone detection tools. In IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2009 April, pp. 157-166. IEEE.

[9] Baxter, I.D, Yahin, A, Moura, L, Sant'Anna, M. and Bier, L, Clone detection using abstract syntax trees. In Software Maintenance, Proceedings, International Conference on, 1998 November, pp. 368-377. IEEE.

[10] Yang,W,Identifying syntactic differences between two programs. Software: Practice and Experience, 1991, 21(7), pp.739-755.

[11] Wahler, V, Seipel, D, Wolff, J. and Fischer, G, Clone detection in source code by frequent itemset techniques. In Source Code Analysis and Manipulation, Fourth IEEE International Workshop on, 2004 September, pp. 128-135.

[12] Mayrand, J, Leblanc, C. and Merlo, E, Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In icsm, 1996 November, Vol. 96, p. 244.

[13] Kontogiannis, K, Galler, M. and DeMori, R, Detecting code similarity using patterns. In Working Notes of 3rd Workshop on AI and Software Engineering, 1995 August, Vol. 6.

[14] Di Lucca, G.A, Di Penta, M, Fasolino, A.R and Granato, P, Clone analysis in the web era: An approach to identify cloned web pages. In Seventh IEEE Workshop on Empirical Studies of Software Maintenance, 2001 November, pp. 107-113.

[15] Greenan, K, Method-level code clone detection on transformed abstract syntax trees using sequence matching algorithms. Student Report, University of California-Santa Cruz, 2005 Winter.

[16] Kapser, C. and Godfrey, M.W, "Cloning considered harmful" considered harmful. In Reverse Engineering WCRE'06. 13th Working Conference on, 2006 October, pp. 19-28.

[17] Roy, C.K and Cordy, J.R, A survey on software clone detection research. Queen's School of Computing TR, 2007 ,541(115), pp.64-68.

[18] Bari, M.A and Ahamad, D.S, Code Cloning: The Analysis, Detection and Removal. International Journal of Computer Applications, 2011, 20(7), pp.34-38.

[19] Kamiya, T, Kusumoto, S. and Inoue, K, CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 2002 ,28(7), pp.654-670.