

Implementing a 3-Way Approach of Clone Detection and Removal using PC Detector Tool

Ginika Mahajan

Dept of Computer Science & Engineering
Central University of Rajasthan
Ajmer, India
ginika.5aug@gmail.com

Meena Bharti

Dept of Computer Science & Engineering
Thapar University
Patiala, India
meenabharti89@gmail.com

Abstract— Software Systems are evolving by adding new functions and modifying existing functions over time. Through the evolution process, copy paste programming and other processes leads to duplication of data resulting in model clones or code clones. Since clones are believed to reduce the maintainability of software, several code clone detection techniques and tools have been proposed. This paper proposes a new clone detection technique to outwit the hindrance of clones by applying a 3-way approach of detecting and removing the clones. The 3-way approach for cloning integrates the three aspects of software engineering: Model Based Visual Analysis, Pattern Based Semantic Analysis and Syntactical Code Analysis. The process is automated by developing a tool that requires no parsing yet is able to detect a significant amount of code duplication.

Keywords—Software Cloning; Model clones; Code clones; PC Detector

I. INTRODUCTION

Software Clones affects software maintenance and other engineering activities. Hence clone detection and removal has grown as an active area in software engineering research community yielding numerous techniques, various tools and other methods for clone detection and removal [6][10][13]. From requirement analysis till software maintenance, it is important to consider various factors that affect its quality, reuse and maintenance. In this respect, software cloning plays an important role. Detecting and removing clones and redundant data will improve the overall efficiency of the software and specifically will ease the maintenance and reuse of the components from the repositories.

Cloning works at the cost of increasing lines of code without adding to overall productivity. Same software bugs and defects are replicated that reoccurs throughout the software at its evolving as well its maintenance phase. It results to excessive maintenance costs as well [1][14]. So cut paste programming form of software reuse raise the number of lines of code without expected reduction in maintenance costs associated with other forms of reuse. So, to eliminate code clones, is a promising way to reduce the maintenance cost in future.

In this paper three different aspects of software engineering are considered and are integrated to detect and possibly remove the clones. First, Model based Visual Analysis. Model clones are segments of models that are similar according to some definition of similarity [5]. During software development complete system is modeled by UML diagrams. The model clones can be detected within UML diagrams at the initial phase of development. The removal of model clones will prevent further penetration of model clones as code clones.

Second aspect is pattern based semantic analysis. Refactoring patterns are used to find the cloned codes. Refactoring pattern improves the code by detecting clones and then removing them [11]. Here four cloning patterns are discussed namely extract, pull up, template and strategy pattern. If the same code structure occurs in more than one place, it's sure that code will become better if it is unified. The simplest duplicated code problem is having the same expression in two methods of the same class. Here Extract pattern can be used and the code is invoked from both places. Similarly Pull Up, Template and Strategy patterns [6] are used to detect and remove the unnecessary code clones. Also if all the cloned fragments of a same source fragment can be detected, the functional usage patterns of that fragment can be discovered.

Third one is syntactical code analysis. Clones can be detected based on the degree of similarity that varies from an exact superficial copy to some near miss similar regions of code or structurally similar regions of code [2][7][9].

So, we need a better approach for the analysis of clone detection and its removal. Also some cloning concepts need to be revised and flaws need to be eliminated. An approach that could integrate cloning at or before design as well as code level for the effective maintenance is needed. An automated system or tool is required, that can automatically detect and remove clones. This will definitely contribute to reduce the maintenance efforts. This work contributes a methodology by integrating Model-Based Visual Analysis, Pattern Based Semantic Analysis and Syntactical Analysis of Code for better detection of clones not only at code level but also at other

levels of software development. And it thereby improves the code, design, quality and maintenance of software as a whole.

II. PROPOSED MODEL

Embracing software cloning is the cornerstone of software development and maintenance. Being able to detect clones more efficiently will improve software quality, increase its reuse and will also reduce the overall cost of maintenance [4]. A 3-way approach is proposed that will first detect the model clones, then semantic clone patterns and finally syntactical code clones. The proposed work is implemented by a tool PC Detector for detecting and removing the clones.

The proposed model as shown in figure 1 is divided into three phases: Model Based Visual Analysis, Pattern-Based Semantic Analysis and Syntactical Code Analysis. At design level, the clones are detected and removed through UML model-based visual approach. After analyzing the unfinished model, clones are detected and removed. The code is then generated from the finished model using Rational Rose. At code level, clone detecting patterns and their solutions are applied to remove the clones. Even after Pattern Based Semantic Analysis some syntactically similar code clones are left that are further detected by the tool. Pattern Clone Detector (PC Detector) tool is developed to automate the task of pattern and syntactical clone detection.

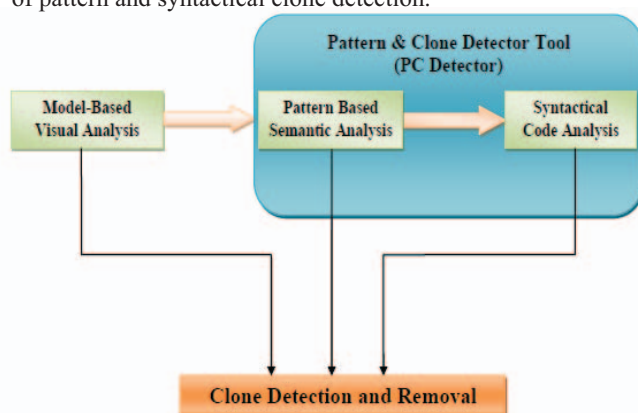


Figure 1. A 3-Way Approach for Clone Detection and Removal

A. Model Based Visual Analysis through UML

Model clones through unfinished modeling are needed to be detected and removed. Some clones may be characterized as the remains of unfinished modeling [5]. There are occurrences of common methods and variables among classes. It can't be said just looking at the view that they are clones or not. Use case diagram are used to view overall behavior of system, class diagram directly helps in implementing the code, and activity diagrams are used to show various actions step-by-step [12].

During initial phases of software development, process modeling of system is done using UML diagrams. With the help of diagrams software structure and its behavior can easily be understood and by analysis clones are detected at initial

levels only, thus avoiding further penetration of clones. Here a small analysis-level model is expressed in following UML diagrams in order to show the clones. To show this analysis a simple case study of Hospital Management System (HMS) is taken where clones are visualized in following diagrams.

- Use-case Diagram
- Class Diagram
- Activity Diagram

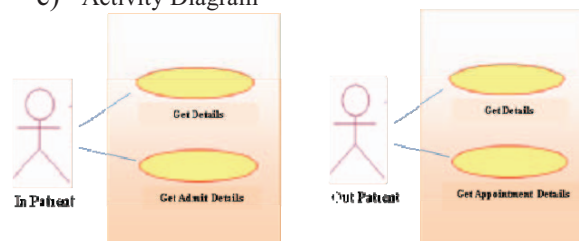


Figure 2 (a). Use Case Diagram showing Unfinished Model

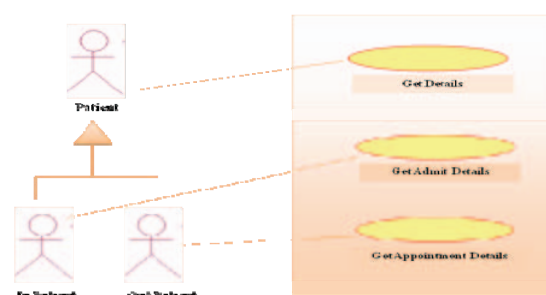


Figure 2 (b). Use Case Diagram showing Finished Model

Figure 2 (a) shows the use case diagram of the category of patients- InPatients and OutPatients. It can be seen that inpatients and outpatients have common behavior and attributes. Visualizing these we can see that they are clones to some extent. These clones can be easily removed as shown in the figure 2 (b). As getDetails() is a common method of InPatients and OutPatients, it can be pulled up thus removing the common data, i.e. clones.

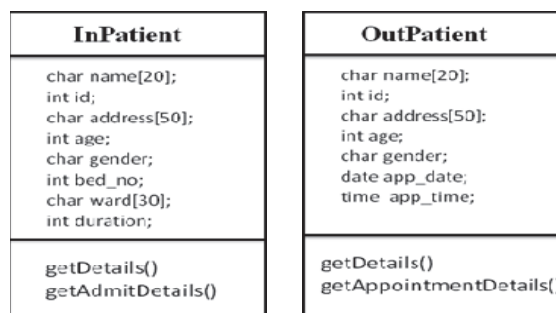


Figure 3 (a). Class Diagram showing Unfinished Model

Figure 3(a) shows a schematic representation of a part of the HMS highlighting the class structures. It is seen in the figure 3(a) that there are occurrences of common methods and variable in both classes named InPatient and OutPatient in the diagram. The method getDetails() and various attributes like name, id, age, contact, and others that are identical in both

classes InPatient and OutPatient. They could be factored out into a class Patient which is a common superclass to both InPatient and OutPatient, yielding the model shown in Figure 3(b).

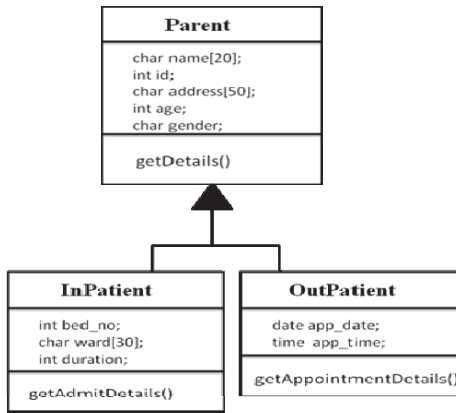


Figure 3 (b). Class Diagram showing Finished Model

Activity diagram shows series of actions performed. This will not only remove the outer clones but also the clones among the same class or of same actor. As shown in figure 4 (a), the input of personal details of patients is common in both the methods of getDetails() and getAdmitDetails(). So the detection of clones is further analyzed and thus they are removed. So the common entry of personal details is extracted and made common among both methods thus removing the clones as shown in figure 4 (b).

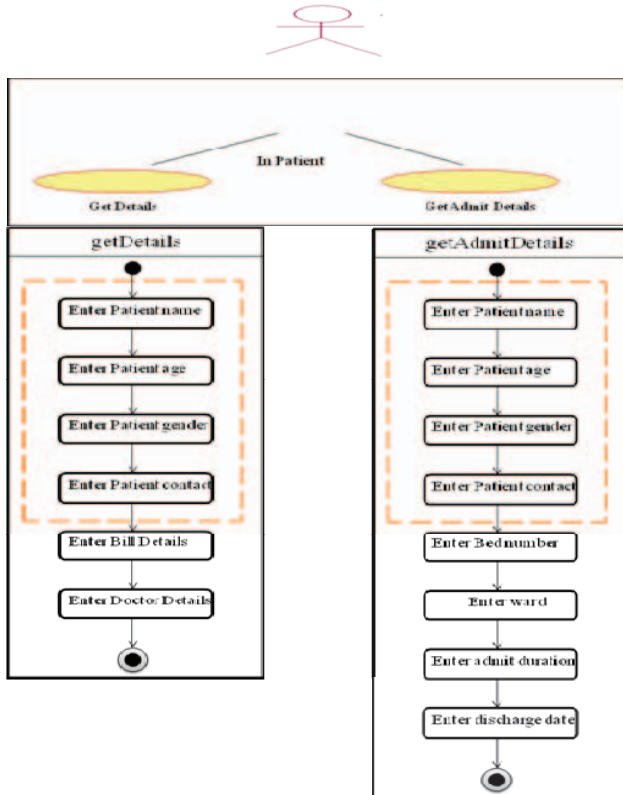


Figure 4(a). Activity Diagram showing Unfinished Model

B. Pattern Based Semantic Analysis

Pattern based semantic analysis detects cloning patterns and remove them. There are various refactoring patterns where clones can be detected. Automating this task of recognizing the patterns and applying appropriate solution will detect the clones and also remove them with greater accuracy. Various refactoring patterns are used to remove code clones like “Extract Method” and “Pull Up Method” that are related to the code clone [6][11]. Here four cloning patterns are discussed namely Extract, Pull up, Template and Strategy. If the same code structure is in more than one place, it’s sure that code will become better if it is unified. The simplest duplicated code problem is having the same expression in two methods of the same class. Then Extract pattern is used and the code is invoked from both places [3].

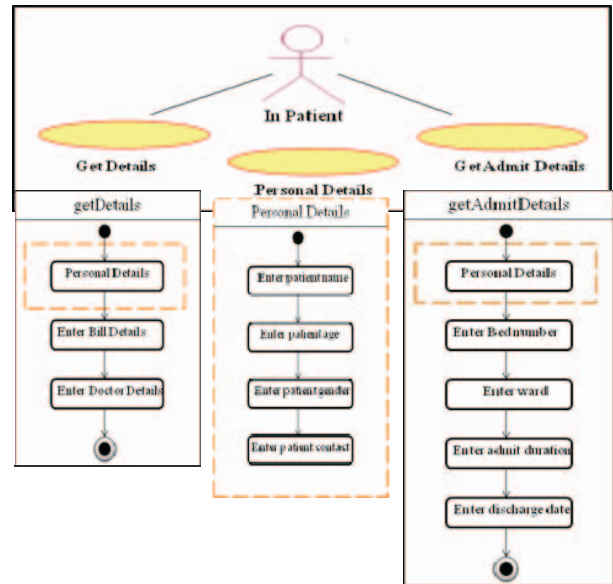


Figure 4 (b). Activity Diagram showing Finished Model

Another common duplication problem is having the same expression in two sibling subclasses. This duplicity is removed by using Extract pattern in both classes then Pull Up Field. If the code is similar but not the same then Extract pattern is used to separate the similar bits from the different bits. If the methods do the same thing with a different algorithm Strategy pattern can be used [11]. Here Pull Up pattern is automated using P C Detector tool. The tool will detect common data members and member functions among the classes. It outputs a new parent class and makes other classes its child classes. The process is explained with the help of flowchart shown in figure 5 followed by its algorithm.

Algorithm of Pattern Detection Process

Input: Source code

Step 1 Detect classes from the given source code.

- Step 2 Compare the classes detected from the source code for the commonalities (i.e. common data members and member functions).
- Step 3 If there are no common data members and member functions present, then pattern cannot be applied.
- Step 4 If common data members and member functions are present, then create a parent class file.
- Step 5 Put the identified common data and member functions in the newly created parent file and delete from child classes.
- Step 6 Display the output.

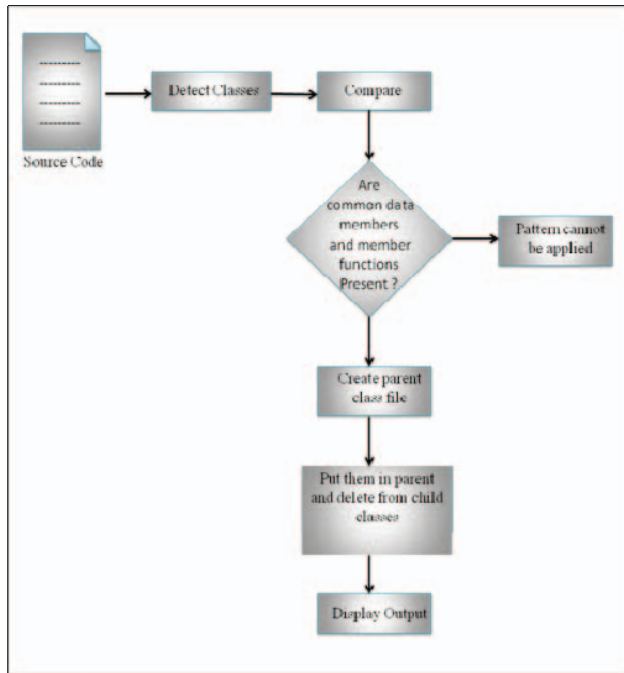


Figure 5. Flowchart of Pattern Detection

C. Syntactic Detection of Clones

Detecting and removing clones through modeling and further through patterns, there are some syntactic code clones that are needed to be detected. Code clones are categorized into Type 1, Type 2, Type 3 and Type 4 clones.

1) Flaw in Defining Type 2 Clone

According to definition [8][9], Type 2 clones are syntactically identical fragments, except for variations in identifiers, literals, types, whitespace, layout and comments. Two code fragments having same data structure and same number of literals with different name can be considered as clones of each other because they can replace each other as their outputs are same. But if two code fragments have renamed literals and different data structures, e.g. one code fragment has integer data type and second one has double data type, these two code fragments cannot be considered as clones as they can never generate same output.

This clause shows a flaw in the definition of type 2 clones and is proved by schematic diagram shown in figure 6. There

are two code fragments – source code 1 and source code 2. In order to detect Type 2 clones the variables are converted into similar tokens. The tokenized code shows that they are clones of each other but as shown in figure 6 they are producing different outputs. Hence during clone detection and comparison data types are needed to be considered.

2) Clone Detection Process

The flowchart in figure 7 shows the process of Clone detection of Type1 and Type 2 and gives the output in the form of matrix. In between the code is normalized, filtered and tokenized. The process is explained in the following algorithm.

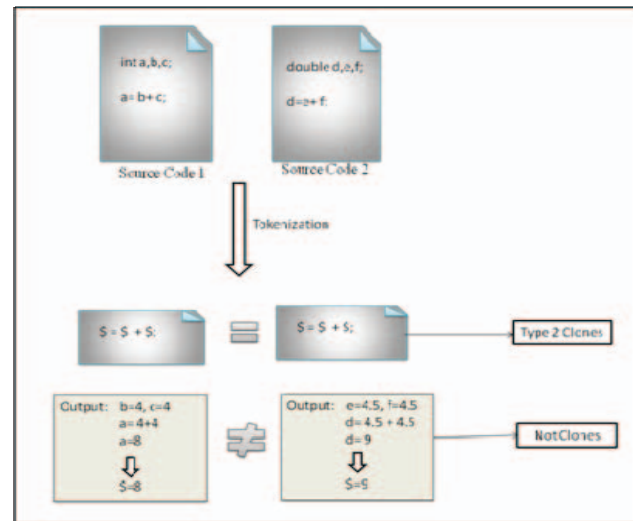


Figure 6. Example explaining the Flaw in Type 2 Clones with respect to Data types

Algorithm of Clone Detection Process

Input: Two source codes

- Step 1 Take two source codes in which clones are to be detected.
- Step 2 Normalize both the source codes by removing the comments and white spaces.
- Step 3 Filtrate the source code obtained after normalization by removing the elements of the code such as void main(), header files, getch(), etc.
- Step 4 Compare the two files after filtrations to check whether they are same or not.
 - a. If they are not same, it means there are no clones in the codes and exit.
 - b. If similarities are found, it means code contains Type I clones.
- Step 5 After the identification of Type I clone, convert the literals present in the code into some type of token (a \$ sign is taken here for this purpose).
- Step 6 Compare the tokens of the two codes.
- Step 7 If match is found, it means Type II clones are present in the two codes.

Step 8 Represent the identified clones of Type I and Type II in the form of a matrix.

Output: Matrix Representation of the identified clones.

III. TOOL IMPLEMENTATION

PC Detector tool is implemented in C# using development environment of . Net. The tool is based on (a) simple string matching and parameterized string matching, (b) refactoring pattern detection and removing the clones by synthesizing the duplicate information found, and (c) matrix representation as a helpful means in visualizing the duplication.

The tool generates a token sequence from the input code after normalization and filtration. The purpose is to transform code portions in a regular form to detect Type 2 clone code portion. Another purpose is to filter out code portions with specified keywords. Representing a source code in matrix format enables us to detect clones within same and different files also. Using this tool it is easy to identify duplicated code between several files and within same file. The tool detects object oriented base Pull Up pattern from C++ source file and outputs the clone free enhanced code. The tool also extracts clones from C and C++ source files. It detects clone in same file or among different file and report its output in matrix format. The working of tool is basically divided into two modules (a) Pattern detection (b) Clone detection.

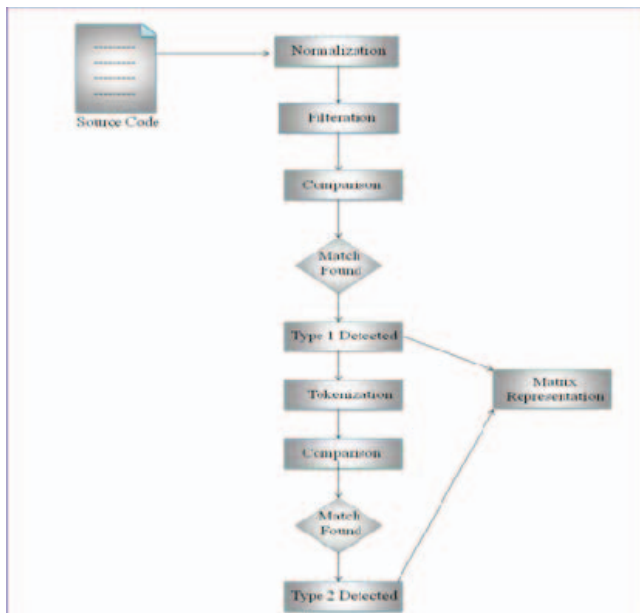


Figure 7. Flowchart of Clone Detection Type1 and Type 2

1) Pattern Detection Process

In pattern detection the tool detects Pull Up pattern in source code and refactor the code by removing clones. The tools take C++ file as input by browsing it from location where it is placed. On detection of Pull Up, a new class is created as 'Parent' class which contains all the common data member and member function of the classes. The common

data members and member function are removed from respective classes and extracted up in the parent class.

2) Clone Detection Process

In clone detection module, the tool detects syntactical clones. This tool detects Type 1 and Type 2 clones. We are working on this tool to extend its functionality to detect Type 3 and Type 4 clones. Type 1 clones remove white spaces, comment or Layout in the code fragment. Type 2 clones detects names of user defined identifiers - variables, constant, classes, methods etc and rename them as \$ token. The clones can be present in the same file or among different files. Technique used for clone detection is string matching and parameterized String matching on the token stream of input code. The clones are detected in various stages as shown in figure 7 of flow chart. The clone detection has further two sub modules, one for detection of clone in same file and other for detection of clones in different files. The option is provided for the user to detect the clone from any file.

In this process the input are source files and the output is matrix showing which lines are having clones. The entire process of our clone detecting technique is shown in Figure 7. The process consists of four steps. Once the source file is input in the form of C or C++ file it goes through following stages for detection of Type 1 and Type 2 clones.

```

Source Code File 1
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    //Declaration of variables.
    int aa,b,c;
    /*Addition of two numbers */
    c=aa+b;
    cout<<c;

    getch();
}

Source Code File 2
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    //Declaration of variables.
    int d,e,f;
    cout<<d;
    /*Addition of two numbers */
    d=e+f;

    getch();
}
  
```

Figure 8. Sample Codes

Step 1: Normalization

In this phase the code (shown in figure 8) is normalized by removing all the comments or whitespaces. The code obtained is normalized (shown in figure 23) and is input to next stage of filtration.

```

Normalized Code File 1
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    int aa,b,c;
    c=aa+b;
    cout<<c;
    getch();
}

Normalized Code File 2
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    int d,e,f;
    cout<<d;
    d=e+f;
    getch();
}
  
```

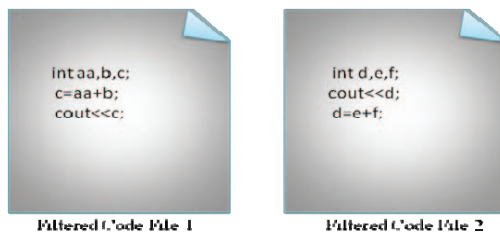
Figure 9. Normalized Code

Step 2: Filtration

This phase is an important phase as it filters the codes from the common data like main function, header files that are commonly found in all codes yet not considered as clones as their detection does not matter in case of clone detection. Also other common functions like `getch()`, `clrscr()`, etc are also filtered from the code.

Step 3: Comparison

This phase inputs the filtered code that is to be compared for clone detection. The textual comparison technique is used to find if any text matches. If any match is found it means they are clones of each other. Thus type 1 clones are detected.



```
int a,b,c;
c=aa+b;
cout<<c;
```

Filtered Code File 1

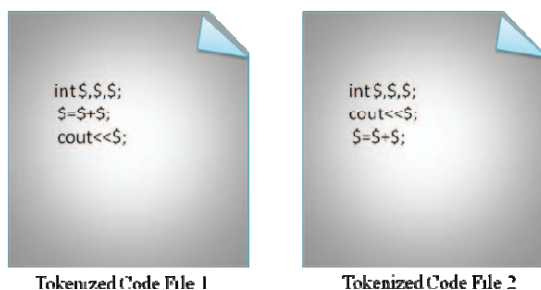
```
int d,e,f;
cout<<d;
d=e+f;
```

Filtered Code File 2

Figure 10. Filtered Code

Step 4: Tokenization

Each line of source file is divided into tokens corresponding to a lexical rule of the programming language. For detection of Type 2 clones we need to tokenize the variables, constants, class, method names as a special token '\$' so as to check if they are same or not. The code is converted so as to detect type 2 clones that are different by names yet are clones of each other.



```
int $,$,$;
$=$+$;
cout<<$;
```

Tokenized Code File 1

```
int $,$,$;
cout<<$;
$=$+$;
```


Tokenized Code File 2

Figure 11. Tokenized Code

Step 5: Matrix Representation

The clones detected are represented in the form of matrix. The row of matrix represents line number of first file. The column of matrix represents line number of second file which is compared with first file to detect if clones are present. In case of detecting clones in same file, both row and column are represented by line numbers of same file. If a match

occurs, it is represented in the matrix by value 1 or else by 0 values as shown in figure 12.



		File 1		
		1	2	3
File 2	1	1	0	0
	2	0	0	1
	3	0	1	0

Figure 12. Matrix representing Clones present in two files

IV. CONCLUSION

The proposed methodology addresses the problem of clone detection and removal. The clone detection and removal is done using the 3-way approach of integrating Model Based Visual Analysis, Pattern Based Semantic Analysis and Syntactically Code Analysis to detect Type1 and Type 2 clones using P C Detector.

Since removal of code clones is believed to reduce the maintainability of software, several code clone detection techniques and tools have been proposed. This paper proposes a new Pattern and Clone Detection approach implemented by PC Detector tool that works on the transformation of input source text into normalized code, then filtering the code, comparing and detecting Type 1 clones. Also a token-by-token comparison is done on the filtered code to detect type 2 clones. Summarizing the work

- An effective 3-way approach is proposed for the analysis of clones, their detection, and removal thus enhancing the quality and maintenance of software.
- Analyzing the flaw in definition of Type 2 clones.
- Detecting the clones at initial levels of software development by using UML diagrams and hence removing them.
- Applying patterns at code level to detect and remove semantic clones. After semantic code removal, the code is further analyzed to detect syntactical clones.
- Developing Pattern and Clone Detection tool and hence automating the task of pattern and syntactical clone detection.

V. FUTURE WORK

PC Detector detects Type 1 and Type 2 clones. This filtered token based clone detection method, is very efficient and scalable to large software systems. This can further be extended to detect Type3 and Type4 clones. The tool can easily detect Refactoring Patterns. Presently, it is working on Pull Up Pattern only but it can be extended to detect other cloning patterns like Extract, Template, and Strategy Patterns. Cloning at design level can further be enhanced by automating the Model Based Analysis. Currently PC Detector detects clones in C and C++ source files. This tool can be adapted to other Object Oriented Languages like Java, C#, etc. The most

important analysis is that software cloning can be integrated in Agile Experiments which can improve its maintenance phase.

REFERENCES

- [1] Mondal M., Rahman M., Saha R., Roy C., Krinke J. and Schneider K., "An Empirical Study of the Impacts of Clones in Software Maintenance", in *Proceedings of IEEE International Conference on Program Comprehension*, pp. 242-245, 2011.
- [2] Rahman F., Bird C., Devanbu P., "Clones: What is that smell?", accepted to *Empirical Software Engineering*, an International Journal 2011 Springer-Verlag.
- [3] Choi E., Yoshida N., Ishio T., Inoue K. and Sano T., "Finding Code Clones for Refactoring with Clone Metrics: A Case Study of Open Source Software", in *Proceedings of The Inst. Of Electronics, Information and Communication Engineers (IEICE)*, pp. 53-57, July 2011.
- [4] Mondal M., Rahman M., Saha R., Roy C., Krinke J. and Schneider K., "An Empirical Study of the Impacts of Clones in Software Maintenance", in *Proceedings of IEEE International Conference on Program Comprehension*, pp. 242-245, 2011.
- [5] Störrle H., "Towards Clone Detection in UML Domain Models", in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pp. 285-293, 2010.
- [6] Mahajan G., and Ashima., "Software Cloning in Extreme Programming Environment", *International Journal of Research in IT and Management*, vol. 2, no. 2, pp. 1906-1919, February 2012.
- [7] Lakhotia A., Li J., Walenstein A. and Yang Y., "Towards a Clone Detection Benchmark Suite and Results Archive", in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pp. 285-286, 2003.
- [8] Kamiya T., Kusumoto S., and Inoue K., "CCFinder: A MultiLinguistic Token-Based Code Clone Detection System for Large Scale Source Code", *Journal IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654-670, July 2002.
- [9] Kapser C. and Godfrey M., "Cloning Considered Harmful", in *Proceedings of the 13th Working Conference on Reverse Engineering*, pp. 19-28, 2006.
- [10] Rysselberghe F. V., Demeyer S., "Evaluating Clone Detection Techniques" in *Proceedings of 19th IEEE International Conference on Automated Software Engineering (ASE)*, pp. 336-339, September 2004.
- [11] Fowler M. and Beck K., "Refactoring: Improving the Design of Existing Code", The Addison Wesley Object Technology Series, Addison Wesley, Boston, 2000.
- [12] Kelter U., Wehren J., and Niere J., "A Generic Difference Algorithm for UML Models", in *Proceedings of Natl. Germ. Conf. Software-Engineering 2005 (SE'05)*, pp. 105-116, 2005.
- [13] Roy C. K., Cordy J. R. and Koschke R., "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", *Science of Computer Programming*, vol. 74, no. 7, pp. 470-495, May 2009.
- [14] Jablonski P. and Hou D., "Aiding Software Maintenance with Copy-and-Paste Clone-Awareness", in *Proceedings of IEEE 18th International Conference on Program Comprehension*, pp. 170-179, 2010.