Method-Level Code Clone Modification using Refactoring Techniques for Clone Maintenance

E. Kodhai[1], S. Kanmani[2]

[1] Research Scholar, Department of CSE, Pondicherry Engineering College , Puducherry, India.
kodhaiej@yahoo.co.in
[2] Department of IT, Pondicherry Engineering College, Puducherry, India.
kanmani@pec.edu

ABSTRACT

Researchers focused on activities such as clone maintenance to assist the programmers. Refactoring is a well-known process to improve the maintainability of the software. Program refactoring is a technique to improve readability, structure, performance, abstraction, maintainability, or other characteristics by transforming a program. This paper contributes to a more unified approach for the phases of clone maintenance with a focus on clone modification. This approach uses the refactoring technique for clone modification. To detect the clones 'CloneManager' tool has been used. This approach is implemented as an enhancement to the existing tool CloneManager. The enhanced tool is tested with the open source projects and the results are compared with the performance of other three existing tools.

KEYWORDS

Code clone, Refactoring, Software maintenance.

1. Introduction

It is generally said that code clone is one of the factors that make software maintenance more difficult [6]. Code clone is a code fragment that is identical or similar to another. Code clones are introduced for various reasons like reusing code by 'copy-and-paste'. Code clone detection techniques can be categorized based on the types of clones they can detect [4] [36]. Many clone detection approaches that can uncover duplication in large scale software systems have been proposed [24- 28][30-31][37].

Information retrieval technique proposed in [18] is to detect trends and associations among the clustered clone classes and determine if they provide further comprehension to assist in the maintenance of clones. Nguyen et al. [20] have developed a clone management tool *JSync* to notify developers change and its inconsistency of code clones in source files. Sandro et. al [23] approach is to take into account detailed code clone analysis and classification as well as how the analysis results are presented to the user in order to guide an interactive removal process.

One technique that helps to process the code clones is Refactoring. Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure

DOI : 10.5121/acij.2013.4202

without changing its external behavior [6]. By making refactoring efforts on a set of code clones, they can be merged into a method [5, 15], a component [7], or an aspect [9].

Program refactoring is not a very simple technique to apply. Firstly, it is often very difficult to identify which part of the program should be refactored without understanding the basic notion of software quality. Secondly, software in practice is so large in scale that developers have to spend a decent amount of time to inspect the whole target software. Finally, program refactoring potentially introduces a new bug in the source code because it does require a source code modification.

Although, numerous techniques and tools have been proposed for code clone detection [30] and [31], only little has been known about which detected code clones are appropriate for refactoring and how to extract code clones for refactoring. Roy [22] proposed an Integrated Development Environment (IDE) based clone management system to flexibly detects, manage, and refactor both exact and near-miss code clones.

Rajlich [16] use different ways of restructuring functions to remove clones. Komondor [32] and Liu [33] suggested approaches to extract a function from the clones, which is similar to *Extract Method*. These works provide extensive mechanisms for function extraction in procedural languages. Li and Thompson [34] proposed code clone removal for the functional language Erlang. This technique is integrated within a refactoring environment for the language. The detection process is limited to the associated detection tool for Erlang.

The refactoring functions are performed one at a time on each duplicate code detected. Robert Tairas [19] conclude that sub-clone refactoring should be considered to augment refactoring performed on the entire clone. Popular development environments such as Eclipse, semi automatic refactorings [17] implementing Visual Studio or Squeak the programmer specifies which refactoring patterns to be applied and where to be applied.

In this paper, we show that the existing refactoring patterns [6] can be used to modify code clones and we propose an approach to support refactoring process by applying code clone detection techniques. Furthermore, a tool CloneManager [38] is extended to support our proposed approach. The functionality of this tool is to find certain code clones to which the refactoring patterns can be applied.

Existing refactoring patterns has different levels of elements for identifying refactoring opportunities such as field, method, class, objects, etc. As the detected clones by the CloneManager tool are methods, we need to go for identifying the method level refactoring opportunities. Moreover, as we are extending the CloneManager tool, it becomes capable of both detection and modification of clones as by its name.

This paper is presented in four major sections. Section 2 presents the related work. Section 3 presents the basics of refactoring. Section 4 describes the existing CloneManager tool [38]. Section 5 describes the implementation of the proposed approach. Section 6 discusses the results obtained using our proposed approach. Finally, section 7 concludes the paper.

2. Related Work

The tool Aries developed by Yoshiki et. al [3][10] supports removal of code clones from source code. This tool applies the characterization of code clones by some metrics, which suggest how to remove them. A refactoring support tool was developed by them to remove code clones. This tool is developed for detecting only two types of refactoring patterns namely extract method and pull up method. They have evaluated the tool with the open source software ant 1.6.0.

The RefactoringCrawler tool developed by Danny Dig et. al. [8] detects refactoring pattern or performs refactoring during component evolution, by combining Shingles encoding with traditional semantic analyses, and by iterating the analyses until a fixed point was discovered. They detect over 85% of the refactoring opportunities. This is mainly useful for clone evolution than maintenance.

The CeDAR which stands for Clone Detection, Analysis, and Refactoring is developed by Robert Tairas [2]. A unified process where the phases of clone maintenance with a focus on clone removal (i.e., detection, analysis, and refactoring) are streamlined together within the programmer's working environment. In this work, the extract method refactoring pattern alone has been developed.

The existing refactoring tool RefactoringCrawler is developed for clone evolution and not for clone maintenance. The tools Aries and CeDAR are tools developed for clone maintenance but with limited refactoring patterns applied such as extract method and/or pull up method. Whereas our proposal has applied three refactoring patterns such as extract method, move method and pull up method.

3. Program refactoring

Refactoring is the process of changing the structure of a program while maintaining all of its functionality. There are many types of refactoring patterns such as renaming a class, changing a method signature, extracting some code. With each refactoring patterns, a number of steps are carried out to keep the code consistent with the original code. Each refactoring patterns includes both a description of a refactoring opportunity i.e., a set of code fragments that should be refactored and the corresponding procedure to perform refactoring.

Martin Fowler et al.[6] introduced a catalog for refactoring patterns where they used a standard format to represent frequently needed refactoring process. They initially introduced 72 refactoring patterns and later the number has been increased to 93[35]. In this work we apply only 3 refactoring patterns as our detected clones are methods. They are extract method, move method and pull up method. The following subsections will explain each of the refactoring patterns in detail.

3.1 Extract Method Refactoring

"Extract Method Refactoring" is intended for "a code fragment that can be grouped together"[6]. Refactoring turns this fragment into a method whose name explains the purpose of

the method. The rationale is that if there is a very large method that does require some added comments to understand its purpose, turning a fragment of code into method will increase the understandability. Fowler also emphasizes the importance of selecting an appropriate short name for the extracted method refactoring.

The "Extract Method" is classified into three types. They are as follows:

- Clone sets that can be removed only by extracting them and making a new method in the same class.
- Clone sets that can be removed by extracting them and making a new method with setting the externally defined variables as parameters of it because such variables are used in the clone.
- Clone sets that can be removed by extracting them and making a new method with setting the externally defined variables as parameters of it and with adding parameters of return statement to deliver the results to the variables used in the caller.

To put it plainly, "Extract Method" means extraction of a part of existing method as a new method, and the extracted part is replaced by a new method caller shown in Figure 1. In general, this pattern is applied to the case that there is a too long method. In applying the pattern to code clones, a new method, that is a code fragment of code clone, is defined and the original code clones are replaced by the new method caller. As a result, we can modify the code clones.

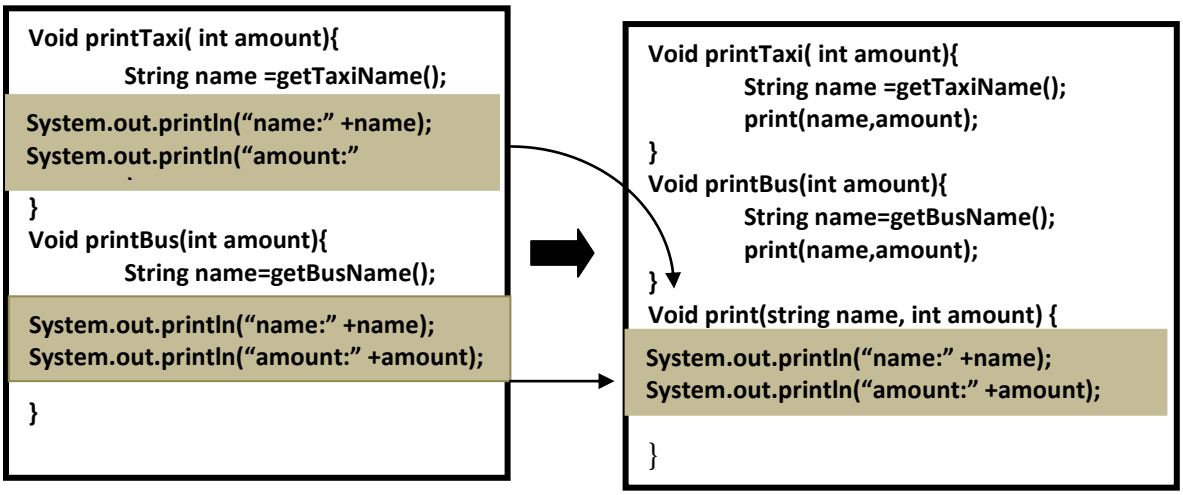

Figure 1. One Scenario for Extract Method Refactoring

3.2 Move Method Refactoring

A method is used or will be used by more features of another class than by its own. It can be moved from one class to another. As one of the most elementary refactor operations, move method aids the reduction of a system's complexity by moving functionality from classes suffering too much behavior or strong coupling. By moving methods around, it can make the classes simpler and they end up being a crisper implementation of a set of responsibilities. An illustration of move method refactoring is shown in Figure 2.

Moving a method actually consists of two actions: (1) removing the method from the original class and (2) adding the removed method to the new class.

The move method from the source class to a target class is performed as follows:
1. The tag indicating to which class the method belongs is changed from source class to target class.
2. The entity sets of all methods accessing the method are updated according to the new tag.
3. The entity sets of all attributes that are being accessed by the method are updated according to the new tag.
4. The method is removed from the entity set of the source class.
5. The method is added to the entity set of the target class.

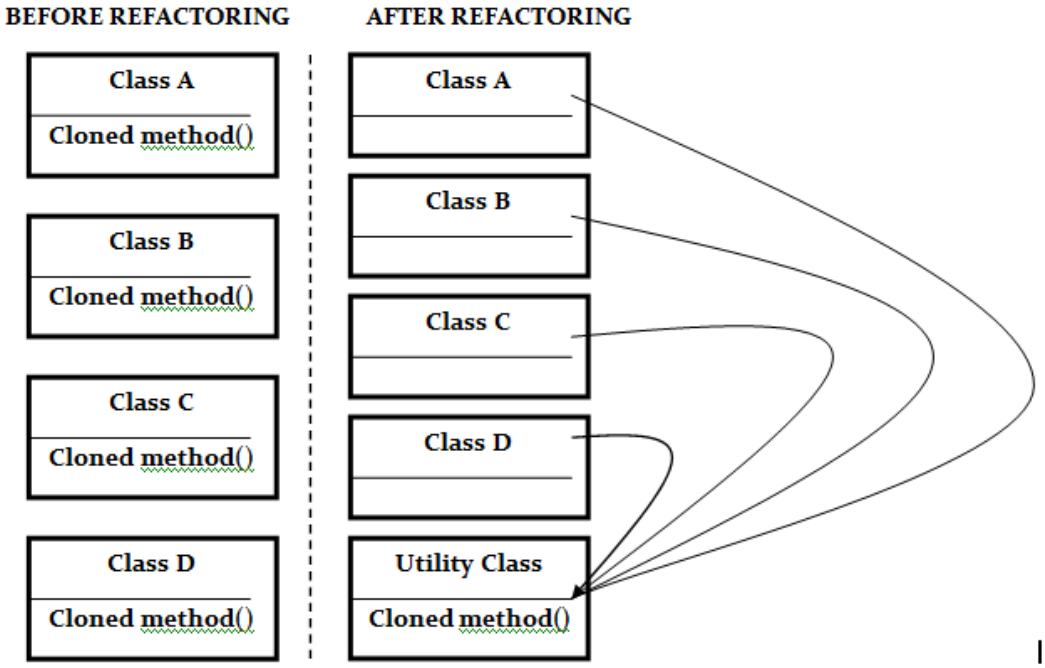

Figure 2. One Scenario for Move Method Refactoring

3.3 Pull Up Method Refactoring

"Pull Up Method Refactoring" means moving identical methods in derived classes to the base class, so it is necessary that the derived classes have common base class. Therefore, we measure the position and distance of clones in the class hierarchy. If the base class has several derived classes and some of them have the same method (that is, code clone), pulling up the method can modify the code clone.

The easiest case of using Pull Up Method Refactoring occurs when the methods have the same body, implying there's been a copy and paste. The two methods in different classes can be parameterized in such a way they end up as essentially the same method. In that case the smallest step is to parameterize each method separately and then generalizes them. A special case of the

need for Pull Up Method Refactoring occurs when there is a subclass method that overrides a superclass method yet does the same thing.

If all or most of a class's subclasses declare the same method, then it should be pulled up into the base class. This way it is only defined once in a central location instead of being defined multiple times. If many of the subclasses have the same method and it should not be pulled up into the base class, then extract superclass can be used. Refer Figure 3.

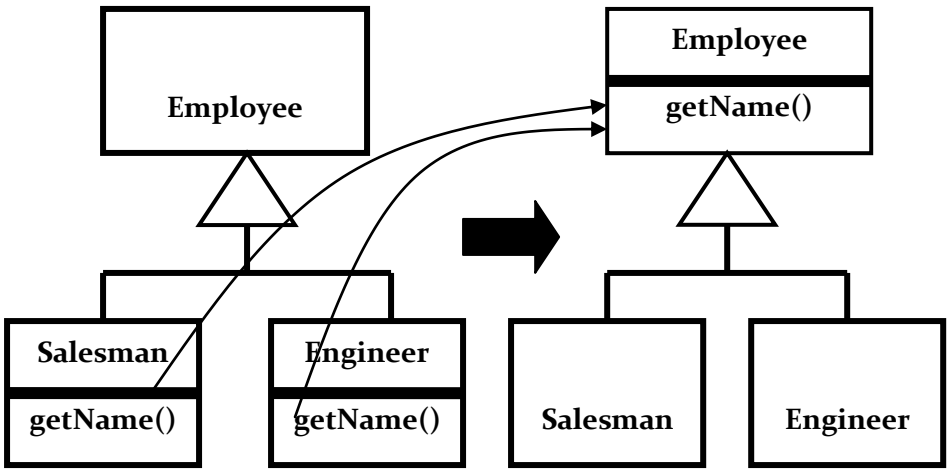

Figure 3. One Scenario for Pull Up Method Refactoring

4. Clone Manager

The existing clone detection tool CloneManager [38] is used, to detect the clones. It is well suited as it is a metric-based code clones because they can be the target of refactoring operations in their entirety. The tool CloneManager is developed for clone detection to effectively and accurately detect all four types of code clones at method level in C and Java source codes through Lexical Analysis and Metrics. The tool is implemented in Java language.

The various types of clones detected by the tool must then be classified and clustered as clone clusters and given as output. The granularity level for clone detection process of the tool CloneManager are methods. It means that the tool detects the functions or the methods of the software system as clones. The tool requires users to select the project containing the necessary files that are to be analyzed. The user is also given the choice of selecting the types of clones that he/she wants to be detected and analyzed.

The result of the code clone detection tool is given as clone pairs. Clone Pair (CP) is the pair of code portions / fragments which is identical or similar to each other. Having detected the clone pairs in each type, the results need to be given in a suitable form for review and analysis. The cloned methods detected in each type are clustered into groups called clone clusters (CC). Every clone pair is commutative and hence in a clone cluster all the members are associative, i.e. every member of the clone cluster is a clone of every other member of the clone cluster. Thus the detected clones and clusters are stored in the text files.

4.1 Final Result Set And Coverage Output

i. **Duplicates.txt:** Contains list of all duplicates grouped by similarity which will be very useful for a quick overview of results and in the selection of 'similarity groups' for closer review, etc. The duplicates are listed as in the clustered order for easier verification. It also contains details of the no. of lines in the cloned method, cluster no., etc.

ii. **Duplicates_summary.txt:** A summary version of the 'Duplicates.txt' containing information about the clone pairs and clusters without any source code. The clusters are represented as a set of cloned method names with the method name and line no. in the file specified along with it.

5. Proposed approach

The detected code clones can be modified automatically using refactoring technique. The refactoring process is performed as follows. The clones are detected using the tool CloneManager[38] and the results are documented in a text file. To perform a refactoring process, the element(s) selected are methods in the source code. The results of the CloneManager are method-level clones which are appropriate for refactoring process. The three types of refactoring discussed in section 3 are implemented by our approach. This approach has been used to enhance the clone detection tool to extend the environment to realize the three refactoring methods.

The main window has two parts. The original source code will be displayed in the left side part and the results of the detected clones by CloneManager tool will be displayed in the right side part. Figure 4 shows the overall system architecture.

The following are the steps to do clone modification by applying refactoring patterns in the proposed approach.

- Identify the possible refactoring opportunities and highlight those clones in the right side window
- This clone in the left side window will be highlighted automatically
- Determine which refactoring pattern should be applied to the identified places
- Apply the refactoring pattern
- Assess the effect of refactoring process
- If the developer wants to reflect the changes, then the changes are store permanently in the original code itself. Otherwise, it will not reflect the changes in the original code

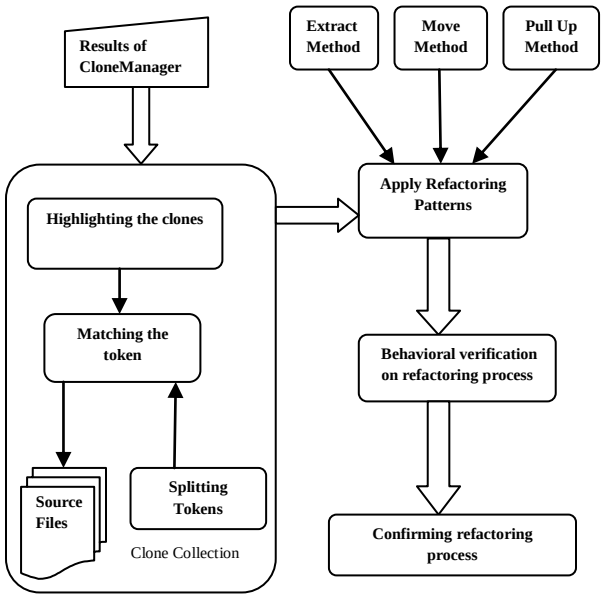

Figure 4: Extended CloneManager System Architecture

The implementation of proposed approach is carried out in four phases as follows

1. Clone Collection
2. Applying refactoring patterns
3. Behavioral verification on refactoring process
4. Confirming refactoring process

5.1 Clone Collection

Using the existing combination of clone metrics proposed in [1][21], the clone clusters are selected from the text file resulted from the tool CloneManager. These clone metrics helps us to identify the clone clusters that are appropriate for refactoring process. After selection of clone clusters, the exact locations of the clones in the source code are located using the string matcher technique.

Clone metrics namely LEN(S), POP(S) and RNR(S) are used for this purpose. Each of them characterize a clone cluster (i.e. an equivalent of code clone) S:

- LEN(S) - The average number of token sequence of code clones in a clone cluster S
- POP(S) - The number of code clones in a clone cluster S
- RNR(S) - The ratio of non-repeated token sequence of code clone in a clone cluster S

The definition of LEN(S) and POP(S) are explicit as they are the simple count metrics. Higher LEN(S) values mean that each code clone in a clone cluster S consists of more token sequences. Clone Metric LEN(S) eliminates small size code clones detected.

A clone cluster with a high value of *POP*(*S*) means that the code clones appears more frequently in many places. A clone set who's higher POP(S) apply good motivation for refactoring to developers because to perform refactoring to code clones in a clone cluster appear more frequently in software improves the maintainability.

The definition of RNR(S) metric is defined by Equation (1). If clone set *S* includes *n* code clones, c_1; c_2 : : : ; c_n, $LOS_{whole}(c_i)$ is the Length of the whole token Sequence of code clone c_i. $LOS_{non\text{-}repeated}(c_i)$ is the Length of non-repeated token Sequence of code clone c_i, then,

$$RNR(S) = \frac{\sum_{i=1}^{n} LOS_{non\text{-}repeated}(c_i)}{\sum_{i=1}^{n} LOS_{whole}(c_i)} \times 100 \qquad (1)$$

Higher RNR(S) values mean that each code clone in a clone cluster S consists of more non-repeated token sequences. It eliminates types of if (or if-else) blocks. It also eliminates language dependent code clones.

The following combination of the clone metrics are used for identifying the clone cluster
1. Clone cluster whose LEN(S) and RNR(S) is the highest
2. Clone cluster whose LEN(S) and POP(S) is highest
3. Clone cluster whose RNR(S) and POP(S) is highest
4. Clone cluster whose LEN(S), RNR(S), and POP(S) is highest

These are the identified clone clusters for applying refactoring mechanisms. The clone collector helps us to find the presence of code clones in the source code. The exact location of the clones in the source code can be determined by clone collector. Using the string matcher technique the clone collector matches the similarity between the source code and the cloned codes.

After the above steps in the browser, the code clones are highlighted using a different background color such as yellow to show the exact location and presence of the clones in the original source code. The remaining part of the code is left with the white background color as such. After highlighting the clones the clone files are integrated together into a single file. The integrated file contains all the copies of the cloned codes from the source code. This type of clone collection is used in future by the user. i.e programmer who refer to the clones can decide which type of refactoring pattern will be made to the clones to correct them or to remove them. Figure 5 illustration of highlighting clones.

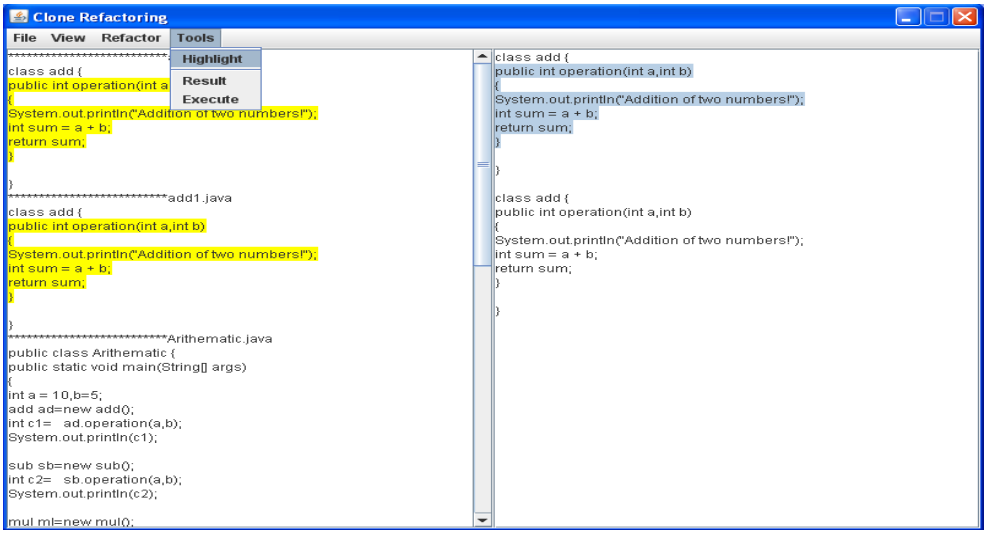

Figure 5. Illustration of Highlighting the Clone Clusters

5.2 Applying Refactoring Patterns

In this phase the refactoring patterns extract method, move method and pull up method for modifying the clones are applied. The steps in each of the methods are explained in the following subsections.

5.2.1 Extract method

Make a list of all methods in the selected clone cluster. For each method, count the lines of code and the number of statements in the method's body. If both of those counts exceed their corresponding thresholds, then the method should probably be extracted into multiple smaller methods. The statement count is important because it prevents false positives from methods with statements that span many lines. For example, a method may have just a few statements, but one of those statements can contain a conditional expression that spans one or more screens. The line count is less important; however, it is useful for users those who want to enforce strict policies on how many lines a method may span.

α:= 70,β := 50
for *m ϵ methods[clone set]* ***do***
if *lines-of-code[m] >= α ^|statements[m]| >= β* ***then***
report ("Extract Method", m)

Figure 6. Algorithm for Extract Method

The Extract Method algorithm is based on the assumption that a method with a large body can most likely be split up into smaller methods. This is done by extracting chunks of functionality from the large method into separate small methods. This algorithm also makes the assumption that the larger the method body is, the more likely it is to have multiple chunks of code that are mostly independent of each other. Those independent chunks of code are easily

extracted. This is not always the case; sometimes a method is so interconnected and cohesive that there is no multiple functionality. It cannot easily be extracted into separate methods.

5.2.2 Move Method

A data class is a class that holds data but does not make use of the data itself; it merely holds the data for other classes to use. For each class or struct in the list, count the number of attributes and methods declared. If there is a low ratio of methods to attributes, then the assumption is that the class is a data class and targets it as a candidate class for move method.

Once a candidate class has been found, for each attribute in the class determine the set of methods referencing the attribute by using the forward reference chains previously built. If a foreign method makes too many references to a number of distinct fields, then move the method or some of its functionality into the data class. The additional check on the number of distinct attributes is needed since some methods may reference a single attribute many times, but in fact a temporary variable could have been used to reference the attribute only once.

α := 0.40,β := 4, μ := 2
for *c ϵ classes[program]* ***do***
if *|fields[c]| > α·|methods[c]|* ***then***
for *f ϵ fields[c]* ***do***
for *r ϵ references[f]* ***do***
m := containing-method(r)
fields-referenced[m] := fields-referenced[m] ∪ { f }
total-references[m] := total-references[m]+1
for *m ϵ fields-referenced* ***do***
if *total-references[m] >= β^|fields-referenced[m]| >=μ* ***then***
report("move method",m,c)
delete *fields-referenced, total-references*

Figure 7.Algorithm for Move Method

If there are duplicated methods in different classes that have no common base class, it is difficult to apply refactoring using class hierarchy to them. In such case, the "Move Method" is a good solution for identifying refactoring opportunities for removing code clones as shown in figure 2.

5.2.3 Pull Up Method

This algorithm makes the assumption that multiple instances of any method in a class hierarchy with the same method signature will be used in very similar ways. It also assumes that methods with the same method signature and roughly the same number of lines are more likely to have a similar purpose.

β := 20
for *m ϵ methods[clone set]* ***do***
common-methods-count[m] := common-methods-count[m]+1

if *common-methods-max-lines[m] < lines-of-code[m]* ***then***
common-methods-max-lines[m] := lines-of-code[m]
if *common-methods-min-lines[m] > lines-of-code[m]* ***then***
common-methods-min-lines[m] := lines-of-code[m]
for *m ϵ common-methods-count* ***do***
if *common-methods-max-lines[m]−common-methods-min-lines[m] <= β* ***then***
report("pull up method",m)

Figure 8: Algorithm for Pull Up Method

5.3 Behavioral Verification on Refactoring Process

The definition of behavior preservation states that, for the same set of input values, the resulting set of output values should be the same before and after refactoring process. So, we need to check the behavior of the system. For this the given input software are executed before and after to check the output of the system. (i.e) to test the results are same after refactoring process. The tool allows the user to test that the files in the project all compile correctly before a refactoring process is performed. This is an essential feature since one of the assumptions made by the tool is that the code compiles correctly. Thus it is automated to check the external behavior.

The user can test whether refactoring patterns can be applied without actually applying the refactoring patterns. The JavaCompiler interface from the framework is used to provide a method for the user to compile files in the project. All files specified in the project are compiled according to the various settings specified in the project file.

To check the external behavior, the refactored source code file is converted into executable file. Hence the file can be run on a machine to produce output file. The code file is saved for the future reference and for the case study that is done on the source code.

5.4 Confirming Refactoring Process

If, the refactored codes do preserve the same external behavior as before, then according to the requirement of the software developer, the software can be made to change completely and permanently in the original source code. The tool allows output conditions to be checked by testing. If the conditions hold, apply the refactoring to the current source code. This confirms the refactoring.

The user interface window has two options with button named *doractoring* and *cancel*. The user is enabled to select the dorefactoring option, if the user selects that option the code is modified as per refactoring process and it is replaced and saved as such. If the developers don't want to make any changes, then the changes will not be done in original source code. For this the user has to select the cancel option.

The tool provides facilities for undoing changes made by refactoring process. However it leaves the management of these changes up to the user so that they can be represented as desired.

It is necessary for the tool to use its own mechanisms to undo the refactoring process, for example by storing a textual representation in its own structure.

6 Experiments and Results

6.1 Experimental Setup

The proposed method is implemented and experimented with seven Java Projects. We have chosen the dataset which have been already evaluated in the literature, so that it will be helpful for us to do comparison easier. We compared our results with three of the existing tools.

To measure the accuracy of our proposed work, we use precision and recall, two standard metrics.

- *Precision* is the ratio of the number of relevant refactoring opportunities found to the total number of irrelevant and relevant refactoring opportunities found.

- *Recall* is the ratio of the number of relevant refactoring opportunities found to the total number of actual refactoring opportunities in the component.

Ideally, precision and recall should be 100%. Finally, we also cross checked the results by manual inspection of the open source projects.

6.2 Datasets

We have analyzed with a medium sized program called JHotDraw 5.3 which is for structured drawing editors of approximately 27,000 lines and to the large size program called Eclipse which is the java text editor along user interface with 352,000 lines. Table 1 lists the features of open source projects which are taken for the performance analysis of our CloneManager tool.

Table 1: Projects chosen as dataset for Clone modification using refactoring

S.No	Project	version	Size KLOC	#Methods	#Classes
1	Eclipse UI	3.0	352	15894	1735
2	Struts	1.2.4	97	6044	469
3	JHotDraw	5.3	27	2038	195
4	JFreeChart	1.0.10	76	1847	436
5	JEdit	4.2	51	5418	426
6	Ant	1.6.0	180	9947	994
7	Ant	1.7.0	67	4231	778

In Table 1, the second column is the list of open source program names as the input project. The third column is the version of the project. The fourth column is the no. of lines in the source code in thousands. The fifth column is the no. of methods in each project. The last column is the no. of classes in each project. Table 2 shows the results for all chosen dataset.

6.3 Results

In Table 2, the 2, 3 & 4 column shows the no. of refactoring opportunities identified and refactoring patterns applied for clone modification. The last column shows the total no. of refactoring opportunities.

From the results, we observed, that even though eclipse UI 3.0 is the largest open source in size, the total of number of the refactoring opportunities is very less. Next, we observed that the total no. of refactoring opportunities in ant 1.6.0 is 87, where as in ant 1.7.0 the number is only 62. This shows that the number of refactoring opportunities reduced a lot in the next version of ant. Finally, we observed that the number of pull-up method is nearly 20 in average for all the projects, where as the extract method and move method varies a lot among the projects.

Table 2. No. of refactoring opportunities detected for each project

Project	Extract Method	Move Method	Pull Up Method	Total Refactoring opportunities
Eclipse UI 3.0	3	10	12	25
Struts 1.2.4	6	21	1	28
JHotDraw 5.3	5	0	26	31
JFreeChart 1.0.10	65	58	21	144
JEdit 4.2	20	58	23	101
Ant 1.6.0	35	29	23	87
Ant 1.7.0	32	27	03	62

6.4 Evaluation of the tool

The table 3 shows the results produced for all the datasets to evaluate our tool. The column 3 holds [M] the number of manually detected refactoring opportunities for all the datasets. The Column 4 holds [D] the number of refactoring opportunities detected by our tool CloneManager. The column 5 holds [C] the number of refactoring opportunities detected correctly by our tool. These values are used to calculate the two parameters precision and recall for evaluation. The formula to calculate Precision = [C]/ [D] * 100 and Recall = [D]/ [M] * 100.

Table 3: Results produced for evaluation of the tool

Project	Refactoring patterns	Manually detected Refactoring opportunities [M]	Detected Refactoring opportunities [D]	Correctly Detected Refactoring opportunities [C]
Eclipse UI 3.0	Extract Method	2	3	2
	Move Method	10	10	9
	Pull Up Method	12	12	12
Struts 1.2.4	Extract Method	6	6	5

	Move Method	23	21	21
	Pull Up Method	1	1	1
JHotDraw 5.3	Extract Method	5	5	5
	Move Method	0	0	0
	Pull Up Method	26	26	26
JFreeChart 1.0.10	Extract Method	66	65	65
	Move Method	57	58	56
	Pull Up Method	20	21	20
JEdit 4.2	Extract Method	19	20	19
	Move Method	58	58	58
	Pull Up Method	24	23	22
Ant 1.6.0	Extract Method	36	35	35
	Move Method	31	29	29
	Pull Up Method	23	23	22
Ant 1.7.0	Extract Method	32	32	32
	Move Method	25	27	25
	Pull Up Method	03	03	03

From the results produced, the precision and recall parameters are calculated for each refactoring patterns for all the chosen datasets. We observed that the precision percentage is above 85 for all the datasets. Even though it is not feasible to get 100 percent for all methods, we had many 100% results. .

We also observed that there is more number of 100 percentages in recall than precision. Moreover the minimum percent is 90 for recall. From this we come to know that our system performance is even higher in recall when compared to precision. This shows that almost all refactoring opportunities detected by our tool are correct. Thus our tool proves to provide high precision and recall, which are the best parameters for the evaluation of code clones tools.

Table 4: Evaluation parameters for all the datasets

Project	**[D]**	**[C]**	**[A]**	**Precision in %**	**Recall in %**	**Time in sec**
Eclipse UI 3.0	25	26	25	92	100	98
Struts 1.2.4	28	29	30	97	93	15
JHotDraw 5.3	31	31	31	100	100	05
JfreeChart 1.0.10	144	147	145	98	99	51
Jedit 4.2	101	103	103	98	98	45
Ant 1.6.0	87	88	90	99	97	56
Ant 1.7.0	62	64	62	97	100	32

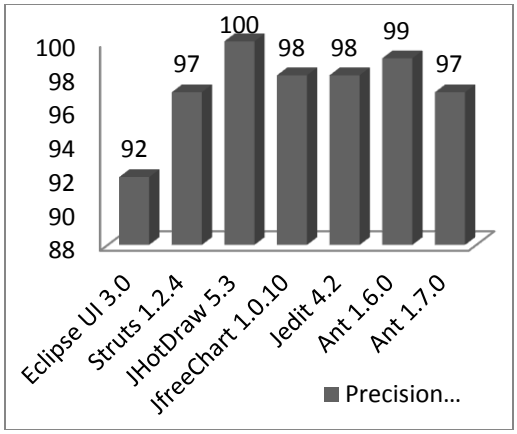

Figure 9. Precision values for all datasets.

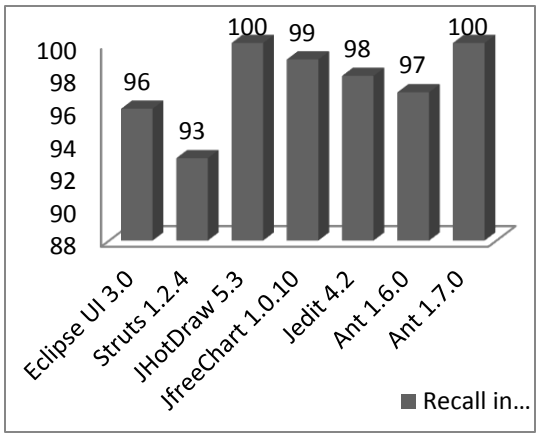

Figure 10. Recall values for all datasets.

The system is also evaluated with overall detected refactoring opportunities for all datasets. The parameters precision, recall and time taken in seconds are given for all datasets. All these values are tabulated in Table 4 respectively. The overall Precision and recall is above 90%. Figure 9 & 10 shows the precision and recall values in graph for all datasets.

The maximum time taken is not even 2 minutes. This shows that our system is very fast. Finally we are able to get results for the Eclipse UI system also which is larger in size. This proves that our system is also scalable.

6.5 Comparison of the tool

Having computed the results for all three types of refactoring patterns and modified the clones in all datasets, we compared our results with three of the existing tools. Table 5 list outs the existing refactoring tool along with their data.

Table 5: Existing tools data considered for analysis

Tool Name	**Refactoring patterns Used**	**Projects considered**
Aries [3]	Extract Method	Ant 1.6.0
	Pull Up method	
RefactoringCrawler [8]	Move Method	Eclipse UI 3.0
	Pull Up Method	Struts 1.2.4
		JHotDraw 5.3
CeDAR [2]	Extract Method	JfreeChart 1.0.10
		Jedit 4.2
		Ant 1.7.0

a) The first tool considered for analysis is Aries for refactoring process. *Aries* developed by Yoshiki et. al [3] supports extract method and pull up method for code clones. Yoshiki et. al has tested the tool with only one open source program Ant 1.6.0.

b) The second tool we considered for analysis is the RefactoringCrawler developed by Danny Dig et. al. [8] an algorithm that detects refactorings opportunities performed during component evolution.

c) The third tool we considered for analysis is the CeDAR which stands for Clone Detection, Analysis, and Refactoring is developed by Robert Tairas [2] . The information about a selected clone group can be forwarded to the Eclipse refactoring engine for the purpose of refactoring process.

Table 6: comparison with the existing tool Aries for Ant 1.6.0

Refactoring Patterns	No. of Refactoring Opportunities		
	Aries	**CloneManager**	**Manual**
Extract Method	32	35	36
Pull Up method	20	22	23
Total	52	57	59
Time in sec	120	56	-

From the table 6, we compared our tool results; which shows more no. of refactoring opportunities identified in both refactoring patterns. This reveals that our tool is able to find and do refactoring process better than Aries. The time taken to detect refactoring opportunities is also lesser as shown in the table.

Table 7: Comparison with the existing tool RefactoringCrawler

Project	RefactoringCrawler			CloneManager			Manual		
	Move Method	**Pull Up Method**	**Total**	**Move Method**	**Pull Up Method**	**Total**	**Move Method**	**Pull Up Method**	**Total**
Eclipse UI 3.0	8	11	19	9	12	21	10	12	22
Struts 1.2.4	20	1	21	21	1	22	23	1	24
JHotDraw 5.3	0	26	26	0	26	26	0	26	26

Table 8: Comparison of the evaluation parameters with RefactoringCrawler

Project	RefactoringCrawler			CloneManager		
	Precision%	**Recall%**	**Time**	**Precision%**	**Recall%**	**Time**
Eclipse UI 3.0	90	86	16.38	92	96	1.38
Struts 1.2.4	100	86	4.55	97	93	0.15
JHotDraw 5.3	100	100	0.37	100	100	0.05

From table 7, we observed that the value in pull up method for struts 1.2.4 and the values for JHotdraw 5.3 project are same. Other values are higher than the refactoringCrawler tool. This shows that our tool is able to detect even more refactorings opportunities which are left by the

refactoringCrawler tool. The values given are correctly detected by our tool[c]. The last column list outs the values detected manually for each project.

Moreover, the precision and recall for their tool is only above 85%, where as our tool is above 90%.Finally, the last column in each tool gives the time taken in minutes and seconds. This shows that our tool is faster than RafactoringCrawler.

Table 9: comparison with the existing tool CeDAR for Eixtract Method

Project	No. of refactoring patterns		
	CeDAR	CloneManager	Manual
JfreeChart 1.0.10	62	65	66
Jedit 4.2	20	19	19
Ant 1.7.0	28	32	32

The CeDAR tool detects for Extract method alone. The value for Jedit 4.2 is the same. The remaining two projects JfreeChart 1.0.10 and Ant 1.7.0 detect more refactoring opportunities than CeDAR. The last column list outs the values detected manually for each project.

7 Conclusion

Thus we have proposed a method for clone modification using refactoring technique, for the existing clone detection tool CloneManager. This proposed method is implemented as an added feature for CloneManager tool. We used the existing refactoring patterns for clone modification. The refactoring patterns extract method, move method, pull up method are used as given in the literature. We have shown that our system is higher in the precision and recall and in terms of speed.

References

[1] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, Tateki Sano, Extracting Code Clones for Refactoring Using Combinations of Clone Metrics, International Workshop on Software Clones (IWSC) – 2011.

[2] Robert Tairas, Representation, Analysis, and Refactoring Techniques to Support Code Clone Maintenance, Ph.D. Thesis – 2010.

[3] ARIES: Refactoring Support Tool for Code Clone, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, Proceeding 3-WoSQ Proceedings of the third workshop on Software quality, ACM New York, NY, USA.2005.

[4] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo, "Comparison and Evaluation of Clone Detection Tools," IEEE Transactions on Software Engineering, Volume 33, Number 9, September 2007, pages 577 - 591.

[5] M. Balazinska, E. Merlo, M. Dagenais, B. Lag¨ue, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. Proc. of the 7th IEEE International Working Conference on Reverse Engineering, pages 98–107, Nov 2000.

[6] M. Fowler, Refactoring: improving the design of existing code, Addison Wesley, 1999.

[7] S. Jarzabek and L. Shubiao. Eliminating redundancies with a "composition with adaptation" metaprogramming technique. Proc. of ESEC-FSE'03 European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of SoftwareEngineering, pages 237–246, Sep 2003.

[8] Danny Dig, Can Comertoglu, Darko Marinov, Ralph Johnson: "Automated Detection of Refactorings in Evolving Components", Proceedings of European Conference on Object-Oriented Programming (ECOOP'06), pp 404-428, Nantes, France.2006.

[9] M. Bruntink, A. V. Deursen, T. Tour´we, and R. V. Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In Proc. of the 20th IEEE International Conference on Software Maintenance, pages 200–209, Sep 2004.

[10] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support environment based on code clone analysis. In Proc. of 8th IASTED InternationalConference on Software Engineering and Applications, pages 222–229, Nov 2004.

[11] Eclipse Java Development Tools, http://www.eclipse.org/jdt.

[12] jEdit, http://www.jedit.org.

[13] FreeChart, http://www.jfree.org/jfreechart.

[14] JHotDraw, http://www.jhotdraw.org.

[15] Apache Ant, http://ant.apache.org.

[16] Richard Fanta and Vaclav Rajilich, "Removing Clones from the Code," Journal of Software Maintenance and Evolution: Research and Practice, Volume 11, Number 4, August 1999, pages 223 - 243.

[17] Emerson Murphy-Hill and Andrew Black, "Refactoring Tools: Fitness for Purpose," IEEE Computer, Volume 25, Number 5, September/October 2008, pages 38 - 44.

[18] Robert Tairas and Jeff Gray, "An Information Retrieval Process to Aid in the Analysis of Code Clones," Empirical Software Engineering, Volume 14, Number 1, February 2009, pages 33 - 56.

[19] Robert Tairas and Jeff Gray, "Sub-clones: Considering the Part Rather than the Whole," International Conference on Software Engineering, Research,and Practice, Las Vegas, Nevada, July 2010.

[20] Hoan Nguyen, Tung Nguyen, Nam Pham, Jafar Al-Kofahi, Tien Nguyen, Clone Management for Evolving Software, IEEE Transactions on Software Engineering, vol. 38, Issue:5, oct 2012, pages 1008-1026.

[21] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, Tateki Sano, Finding Code Clones for Refactoring with Clone Metrics: A Case Study of Open Source Software, IEICE Technical Report – 2011.

[22] Minhaz Zibran, Chanchal K. Roy, Towards Flexible Code Clone Detection, Management, and Refactoring in IDE, International Workshop on Software Clones (IWSC) – 2011.

[23] Sandro Schulze, Martin Kuhlemann, Advanced Analysis for Code Clone Removal, GI-Workshop on Software Reengineering – 2009.

[24] R. Koschke, "Survey of research on software clones," Dagstuhl seminar 06301. ISSN 1682–4405 - Duplication, Redundancy, and Similarity in Software, 2006.

[25] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's University, Kingston, Canada, Technical Report. 2007-541, 2007.

[26] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," IEEE Transaction Software Engineering, vol. 32, no. 3, pp. 176–192, 2006.

[27] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," The 16th IEEE International Conference on Program Comprehension '08, 2008.

[28] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective – a workbench for clone detection research," International Conference on Software Engineering '09, 2009.

[29] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and Implementation for Investigating Code Clones in a Software System. Information and Software Technology, 49(9-10):985{998, 2007.

[30] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," International Conference on Software Engineering '07, 2007.

[31] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," IEEE Transaction on Software Engineering, vol. 28, no. 7, pp. 654–670, 2002.

[32] Raghavan Komondoor and Susan Horwitz, "Effective, Automatic Procedure Extraction," International Workshop on Program Comprehension, Portland, Oregon, May 2003, pages 33 - 42.
[33] Yidong Liu, "Semi Automatic Removal of Duplicated Code," Diploma Thesis, University of Stuttgart, Germany, 2004.
[34] Huiqing Li and Simon Thompson, "Clone Detection and Removal for Erlang/OTP within a Refactoring Environment," Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Savannah, Georgia, January 2009, pages 169 - 178.
[35] M. Fowler. Refactoring Catalog, http://refactoring.com/catalog/
[36] C. Roy, J. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Science of Computer Programming, vol. 74, no. 7, pp. 470–495, 2009.
[37] N. Gode and R. Koschke, "Incremental clone detection", 13th European Conference on Software Maintenance and Reengineering '09, 2009.
[38] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika and B. Vijaya Saranya, CloneManager : A tool for detection of type1 and type2 code clones, International Conference on Recent Trends in Business Administration and Information Processing, Springer digital library, Trivandrum, Kerala, India, March 26 & 27, 2010.