Fuzzy Based Refactoring Cost Resembler (FRCR) Model for Object Oriented Systems

Ankit Desai and Amit Ganatra

***Abstract*—Successful software systems must be prepared to evolve or they will die. Although object-oriented software systems are built to last, over time they degrade as much as any legacy software system. As a consequence, one may identify various reengineering patterns that capture best practice in reverse- and re-engineering object-oriented legacy systems. Software re-engineering is concerned with re-implementing older systems to improve them or make them more maintainable, while Refactoring is re-engineering with-in an Object-Oriented context. In this paper, given object-oriented refactoring opportunities, the cost of refactoring is resembled using FRCR. The opportunities are class misuse, violation of the principle of encapsulation, lack of use of inheritance concept, misuse of inheritance, misplaced polymorphism.**

***Index Terms*—Fuzzy, FRCR, refactoring.**

I. Introduction

Software re-engineering is the transformation from one representation from to another at the relative abstraction level, while preserving the systems' external behavior. Reengineering a software system has two key advantages over more radical approaches to system evolution i. reduced risk: There is a high risk in redeveloping software, which is presently an essential backbone of the organization. Errors may be made in system specification; development problems; financial risk may be high; etc. ii. Reduced cost: The cost of re-engineering is significantly less than the cost of developing new software. Refactoring is reengineering within the object oriented context. Software refactoring can be defined as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [1]-[3].

In our previous paper various refactoring opportunities were discussed in detail. For the completeness here all five opportunities are redefined. i. Class misuse (CM): The most fundamental mistake done in developing a program in OOP context is not designing a class properly. Even if one class is not designed properly the entire program is spoiled. One bad class design has a cumulative effect on all other classes in the entire software. So the first step in developing good software is to design the basic construct, i.e. the class, correctly. The class has to be designed taking into consideration of UML (Unified Modeling Language) concepts. Moreover its members, i.e. data and behaviors to be defined adequately, the interfaces properly laid down and the relationships between different classes correctly defined. ii. Violation of the principal of encapsulation (VPE): Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interface and misuse. If the design is poor then objects of other class may peep into a poorly designed class. iii. .Lack of use of Inheritance concept (LUIC): Inheritance may not be properly used. Programmers tend to define their classes right from the scratch. So the same code gets duplicated in more than one class. If any change has to be made to any one method, then the changes have to be made in all the classes which contain the code, thus duplicating effort. If inheritance has been implemented, then the changes will have to be made in only one of the classes in which it is defined. As the concept of inheritance can be extended to many generations, this code replication can be avoided. iv. Misuse of Inheritance (MI): A base class should be inherited by the derived class when all the contents of the base class are used fully and at the same time distinct in the derived class. Often inheritance is implemented for code reuse rather than polymorphism. Inheritance is more to achieve Polymorphism rather to code reuse. v. Misplaced Polymorphism (MP): Polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation. Function overloading is the process of using the same name for two or more functions. The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters. Functions performing the same action on different data types should be given the same name. Polymorphism helps reduce complexity allowing the same interface to be used to access a general class of actions. Table 1. Shows the conclusion derived in [4].

Fuzzy logic can be defined as "It is super-set of conventional (Boolean) logic that has been extended to handle the concept of partial truth." Central notion of fuzzy systems is that truth values (in fuzzy logic) or membership values (in fuzzy sets) are indicated by a value on the range [0.0, 1.0], with 0.0 representing absolute Falseness and 1.0 representing absolute Truth [5]. It is developed to deals with real world vagueness.

These fuzzy system has an ability to take the linguistic values as an input then to convert it in to the fuzzy values, this process of converting crisp values in to fuzzy set values is termed fuzzification, where as once the fuzzy inputs are converted in to fuzzy outputs, these output is also converted in to crisp values by defuzzifiction process [6].

The objective of this paper is to use both of these concepts

Manuscript received February 17, 2012; revised March 31, 2012.

Ankit B. Desai is with the M. Tech (Computer Engineering) at Dhramsinh Desai University, Nadiad and Assistant Professor at Charotar University of Science Technology (CHARUSAT), Education Campus, Changa, Gujarat, India (e-mail: desaiankitb@gmail.com).

Amit P. Ganatra is with the Charotar University of Science Technology (CHARUSAT), Education Campus, Changa, Gujarat, India (e-mail: amitganu@yahoo.com).

and to build a Fuzzy based Refactoring cost resembler, FRCR model with the use of the fuzzy set theory and to show how actually refactoring opportunities can be intended from the given project code samples or UML documentations [7]-[8]. Subsequently opportunities will be transformed in to the required format i.e. count of each, supplied as an input to resemble the actual cost to demonstrate the presumption model in practice[9]-[11].

Refactoring deals with code changes in any software system. These changes do not necessarily make any functional changes to the corresponding code. However, it improves it internal structure [12].

Refactoring is basically a transformation process which can be applied in series of small but behavior preserving transformations. Do not necessarily each transformation bring a significant restructuring, but each refactoring does a little. Advantage of such small refactoring (transformation) is, since each of them is small, it's less likely to go wrong. This process once applied the system will behave more robust as it has been restructured. To apply refactoring many tools exist to automate the process [13].

Refactoring, if applied on the working software than it ameliorates the performance of the support contrasting with the principal of "If Its Working Don't Change". Refactoring can be applied on the poorly working program that is having some bugs to be fixed. If you have a poorly factored program that does what the customer wants and has no serious bugs, then you may feel not to apply refactoring on it. When you need to fix a bug or add a feature, you Refactor mercilessly the code that you encounter in your efforts. Thus, Refactor mercilessly can live in harmony with "If It Is Working Don't Change".

"If It Is Working Don't Change" applies to the maintenance programmers, too. If the first spike winds up being some ugly piece of spaghetti, for whatever reason, then we should be allowed to untangle it when the opportunity presents itself. Code stops working when the customer changes their mind. If we have an agile process, the artifacts can change quickly and frequently. 'Refactoring' is about some ways this is done correctly.

Tool support for refactoring is highly desirable because checking the preconditions for a given refactoring often requires nontrivial program analysis, and applying the transformations may affect many locations throughout a program. In recent years, the emergence of light-weight programming methodologies such as Extreme Programming has generated a great amount of interest in refactoring, and refactoring support has become a required feature in modern-day IDEs. The key insight is that it's easier to rearrange the code correctly if you don't simultaneously try to change its functionality. The secondary insight is that it's easier to change functionality when you have clean (refactored) code. The existing tools provides only the feature of transformation from existing to the new design where as the proposed model in this paper can be used along with the tool support for the calculation of the cost (amount of efforts) required to perform refactoring provided that we have proposed five opportunities to be refactored.

TABLE I: EFFECT OF VARIOUS OPPORTUNITIES OF REFACTORING ON COMPLEXITY MEASURES

	Complexity Measures Before Applying Refactoring			**Refactoring Cost**
Refactoring Opportunities	**Space**	**Time**	**Design**	
CM	Very High	High	High	Low
VPE	Not Applicable	Very High	Very High	Very High
LUIC	Very High	Medium	Very High	High
MI	Not Applicable	High	High	Medium
MP	High	Very Low	High	Very Low

II. FUZZY BASED REFACTORING COST RESEMBLER (FRCR) MODEL FOR OBJECT ORIENTED SYSTEM

An approach to build the refactoring cost resembler model is to get proposed parameter's values as an input from the user, then processes the input and estimates the cost of refactoring based on the pre-identified rule base. The fuzzy model can be constructed using following steps: 1. Model input and output Membership Functions: identify the refactoring opportunities and identify the particular opportunity's vague values by analyzing the projects or studying the projects on which there is a possibility of applying refactoring. An example of such identification; i.e. consider a refactoring opportunity named number of class misused (NCM). NCM is studied over more than fifty medium and large size projects and we have concluded that on an average, probability of class misuse found is high but at the same time it is very easy to identify such misused classes from the project's documents, UML analysis or even from the code; provided that the system is medium sized. This leads us to conclusion that the cost of refactoring will be less as identification is faster and less complex compared to other opportunities and at the same time we can estimate the input in terms of fuzzy vagueness i.e. for NCMC is divided into five vague values in accordance to its effect on refactoring cost. NCMC = {VL, L, M, H, VH}. These set elements individually are called as membership function. To sum up, the inputs are crisp non-fuzzy numbers limited to a specific range. 2. Model rule base: Once the membership functions for all opportunities are identified, next is to identify individual membership function's effect on the output. This process in summed up in terms of the rule base; rule base is the rules on membership function to form the output. These rules most commonly take the form 'if-then-else'. In our case input values are 'and' with each other. An example of such rule formation; i.e. if NCMC is VL and NVPEC is L and

NLUIC is VL and NMIC is VL and NMPC is L then RC is VL. This rule is formed by simply identifying the final cost of the project when all MFs take the input which is considered according to the given range for MF.

In FRCR model we have considered a few assumptions i.e. apart from five opportunities considered in FRCR as an input there are other factors which may affect the cost of refactoring but we are showing cost estimation based on these selected five.

The preferred inference method to implement this proposed system is mamdani – type fuzzy inference method; as the output to this system does not take the linear form; if it would have been linear then we would have chosen Sugeno - Type Fuzzy Inference method. Furthermore membership functions are proposed to be of the shape triangular and sigmoid; i.e. considering the example cited in previous paragraph step 1. where NCM takes five MFs in its set. To justify the shape consider the figure 2. where VL is taken triangular because it seems more beneficial to use linear membership function and it is already been normalized with log function: log (N / n). Same reasoning is applicable to other triangular membership functions used in the system. To consider the positive infinity values of inputs sigmoid shape of MF is most suitable. For the purpose of "and" MFs product can be used by default. Aggregation of the inputs to form the one output "sum" is used moreover defuzzification method is centroid.

III. Cost Computing Using FRCR Model

The FRCR (Figure 1) is implemented in MATLAB using the fuzzy logic toolbox. This toolbox allows for the development of input membership functions, fuzzy control rules, and output membership functions.

To implement this system we need to have five different inputs: number of class misuse count (NCMC), number of violation of the principle of encapsulation count (NVPEC), number of lack of use of inheritance count (NLUIC), number of misuse of inheritance count (NMIC), number of misplaced polymorphism count (NMPC). These five inputs will then be processed by a fuzzy logic controller that will output a percentage cost of refacoring (RC). This degree of refactoring is then decoded into one of five possible outputs: very low, low, medium, high and very high.

The first input membership function for number of class misuse count (Figure 2) will have five different Membership functions: very low (0-5-10), low (5-10-15), medium (10-15-20), high (15-20-25), and very high (25-inf). The second input membership function for number of number of violation of encapsulation (Figure 3) will have two different Membership functions: low, high. Where low is considered between [0 3] violation and beyond that any number of violations found is considered as high. The third input membership function for number of number of lack in use of inheritance (Figure 4) will have five different Membership functions: very low (0-1-2), low (1-2-3), medium (2-3-4), high (3-4-5), and very high(5-inf). The fourth input membership function for number of misuse of inheritance (Figure 5) will have five different Membership functions: very low (0-1-2), low (1-2-3), medium (2-3-4), high (3-4-5), and very high (4-inf). The ranges of these functions are 0 to 10; these are the possible input values. The very high/high membership function continues on to infinity in positive direction to include any number of faults found. The output has four membership functions; very low, low, medium, high, very high (Figure 7). These membership functions are all triangular and are spread evenly on a range of 0 to 1.Once all of the input and output membership functions have been defined the heart of the control can now be defined; the rules. The fuzzy rules are in the form of if-then statements. These statements look at both inputs and determine the desired output. In this system increase in number, of any of five inputs will lead to gradual increase in cost. The rules defined for this system are in Table 2.The simulink modeled can be prepared in MATLAB for the FRCR; an example model is shown in figure 8. and the result graph for rule base is along with the test cases is shown in Table3.

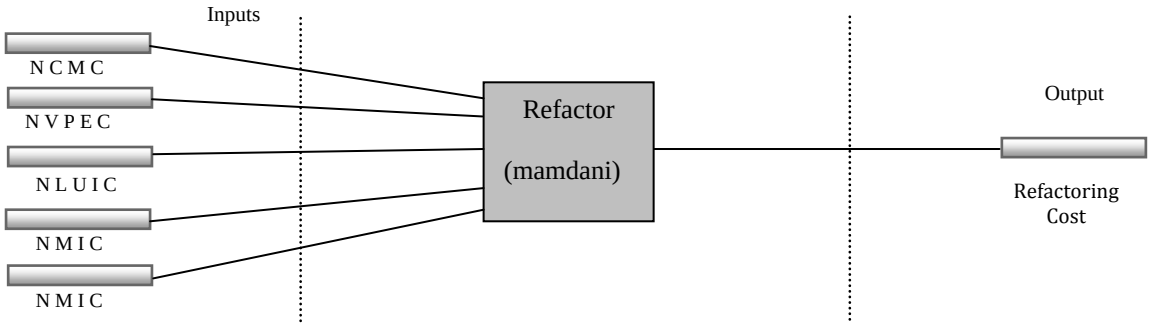


Fig. 1. FRCR – Fuzzy model

The inputs for this example system have been presented in Table 3. Column TC, they are randomly generated data within a valid range of the rule R_i. The system is simulated using such valid range of inputs for each rule R_i. Particular test case takes a specific format i.e. [a b c d e] where a, b, c, d and e are NCMC, NVPEC, NLUIC, NMIC and NMPC are respectively. In table cost column is the resembled cost for the particular test case. It can be interpreted as e.g. if cost is resembled 40% by our FRCR Model then the refactoring cost of the project is considered as 40% cost of its original development cost.

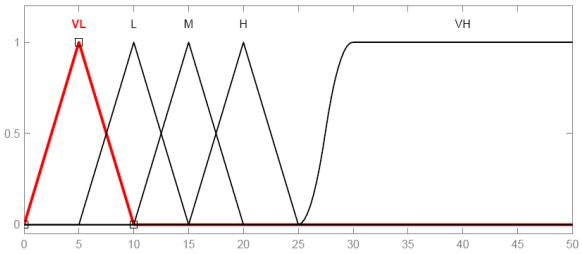


Fig. 2. NCMC – input membership function.

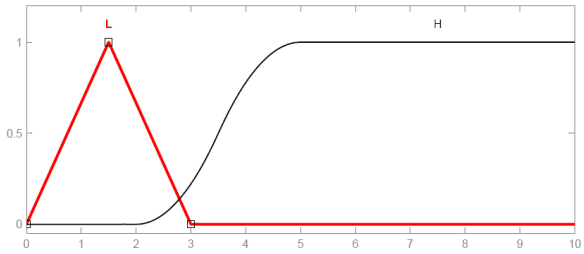


Fig. 3. NVPEC – input membership function

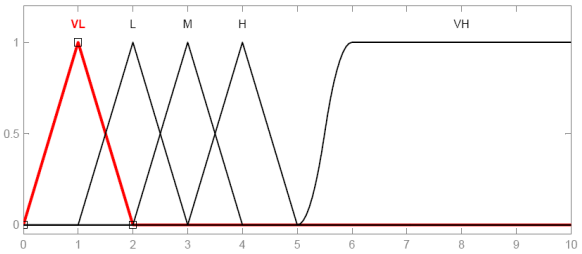


Fig. 4. NLUIC – input membership function

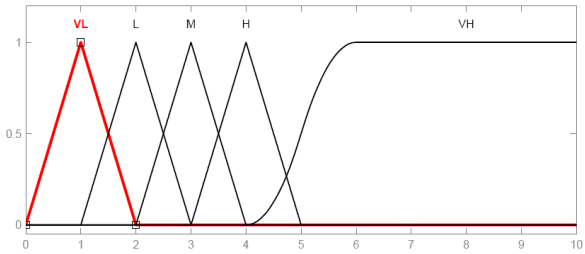


Fig. 5. NMIC – input membership function

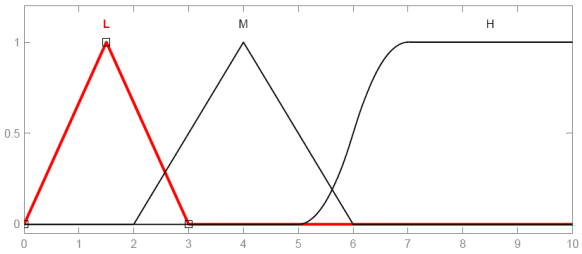


Fig. 6. NMPC – input membership function

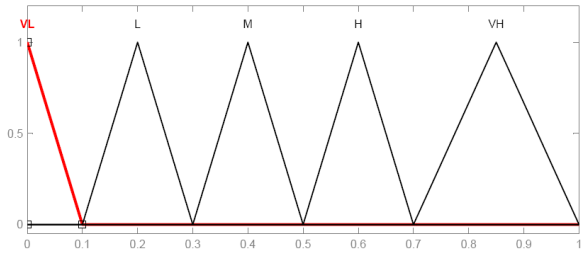


Fig. 7. RC – output membership function

TABLE II: MEMBERSHIP RULES

NCMC	NVPEC	NLUIC	NMIC	NMPC	RC
VL	L	VL	VL	L	VL
VL	H	VL	VL	M	L
VL	L	VL	VL	H	L
VL	H	VL	VL	L	M
VL	L	VL	VL	M	VL
L	H	L	L	H	M
L	L	L	L	L	VL
L	H	L	L	M	L
L	L	L	L	H	M
L	H	L	L	L	M
M	L	M	M	M	M
M	H	M	M	H	H
M	L	M	M	L	L
M	H	M	M	M	M
M	L	M	M	H	M
H	H	H	H	L	H
H	L	H	H	M	M
H	H	H	H	H	VH
H	L	H	H	L	M
H	H	H	H	M	VH
VH	L	VH	VH	H	H
VH	H	VH	VH	L	VH
VH	L	VH	VH	M	H
VH	H	VH	VH	H	VH
VH	L	VH	VH	L	H

(VL = Very Low; L = Low; M = Medium; H = High; VH = Very High. NCMC: Number of class misuse count; NVPEC: Number of violation of the principle of encapsulation count; NLUIC: Number of lack of use of inheritance concept count; NMIC: Number of misuse of inheritance count; NMPC. Number of misplaced polymorphism count; RC: Refactoring cost)

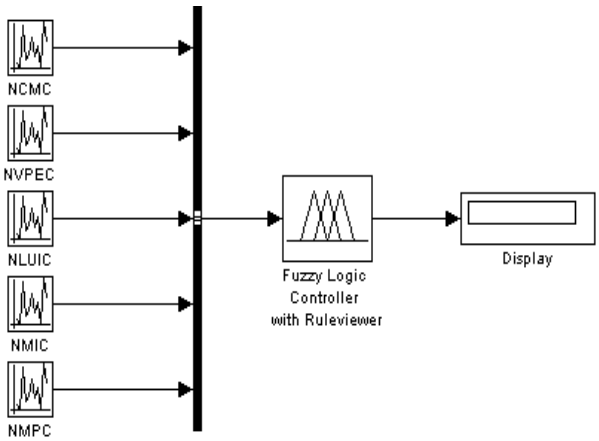


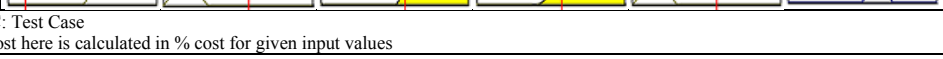
Fig. 8. Simulink Model for FRCR

IV. EXAMPLE SHOWING WORKING OF FRCR MODEL

For the sake of simplicity here we have consider a simple Employee Management System (EMS) for analysis and a few snippets of the code written in java programming language are taken as a part of example along with their UML specifications. If one wants to apply same kind of logic in other object oriented language than it can be applied without much modification. Furthermore, applicability of these concepts is not limited to the examples stated in this section

as examples of these five refactoring opportunities acquire a wide scope[14]-[16].

TABLE III: SIMULATION OF THE RULE VIEWER WITH TEST CASES AND RESEMBLED COST USING FRCR

R_i	NCMC	NVPEC	NLUIC	NMIC	NMPC	RC	TC*	Cost#
1							[1 1 1 1 1]	4.26
2							[2 3 1 1 3]	20
3							[3 1 1 1 6]	20
4							[4 4 1 1 2]	40
5							[5 1 1 1 4]	3.34
6							[6 4 2 1 6]	40
7							[7 1 2 2 1]	23.4
8							[8 3 2 3 4]	22
9							[10 1 2 2 1]	42
10							[12 3 2 2 1]	45.29
11							[15 1 3 2 3]	50
12							[16 3 3 3 6]	60
13							[13 1 3 3 2]	20
14							[17 4 2 3 4]	52
15							[14 1 3 2 6]	43
16							[20 5 4 4 1]	62
17							[21 1 4 4 4]	41.9
18							[18 3 4 4 5]	85
19							[22 1 4 4 1]	44.86
20							[18 3 4 4 3]	85
21							[25 1 7 9 6]	62.7
22							[26 5 7 5 1]	85
23							[27 4 8 7 4]	70.3
24							[30 3 6 5 6]	85
25							[32 1 9 8 1]	63.4

*TC: Test Case
Cost here is calculated in % cost for given input values

CM: Consider Employee Management System (EMS) relating to an organization. CM is applied between Employee, EmpAddress, and EmpName of the prescribed code. The basic mistake in design of above mansion classes is that all the information pertains to a single Employee is spread across many classes. All the information should be stored in a single class. If an employee leaves the organization and his records is to be destroyed, in the above design three different objects pertaining to three different classes have to be destroyed instead of one single class object. Besides, if information about a particular employee is to be obtained, then one has to access three different objects instead of one single object. Class misuse instances are wonderful opportunities for refactoring. Inheriting all the classes into a single class often does not solve all the problems as between example classes shown. Since some of the variables might be repeated in more than one class and this leads to compile time errors during implementation, e.g. variable employee_id Moreover, if the function for printing the class values is defined three times and one has to print the values in the inherited class then three function calls are mandatory, i.e. print(), printk(), printl(). This leads to poor functionality. Hence a new function has to be defined to print all the inherited class attributes. This leads to repetition of coding. Only option in this case is to redesign the classes and make a single class in place of three with all the functionality in it.

VPE: When classes are not designed correctly, reflection has to be used. Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. For example, it is possible for a Java class to obtain the names of all its members and display them [2], [17]-[19]. As shown in CM, instead of defining a single class, when there are many classes and one has to access private members of these different defined classes (which otherwise would have been in a single class), then one has to use reflection concept.

LUIC: EMS comprises of class Employee. In case of bulky projects there exists a possibility of defining class HourlyPaidEmp as in snippet 4. which actually should be build upon class Employee in snippet 3.

MI: EMS system defines a class Employee as in snippet 1. and IncomeTax as defined in sinppet 5. There exists a chance of refactoring in this snippet 5 because designation is not used in class IncomeTax. Class IncomeTax, when inherited from class Employee, there will be an extra variable 'designation' which is not necessary for class IncomeTax. Such dangling variables are dangerous. This poses a problem

for future maintenance of the software. This extra variable will not be present in the specification and has to be initialized correctly to some initial value; else it might lead to bizarre error.

MP: Snippet 6 of EMS defines class Salary with three different calls to calcSal(TypeOfEmp). Because of the polymorphism, only one set of names calcSal() should be defined which is used for all three specific versions of these functions, one for each type of Employee, later compiler will automatically select the right function based upon the data being used [18]. The individual version of these functions defines the specific implementations for each type of data. If the class is developed initially for MonthlyEmp salary calculation only and much later the class is modified to include for HourlyPaidEmp and WeeklyPaidEmp types then for the same calcSal() operation there exist a possibility to define a class as in snippet 7. These results in similar code (not the same code) in different names provide same interface for different types. Such cases provide a wonderful chance for refactoring.

Calculation of the cost of refactoring from the above analysis; NCMC = 2; as there are two classes in the system which are misused. NVPEC = 2; as there are places in the class structure where we need to introspect the properties of the other class. NLUIC = 1; as there exist one class which comes under LUIC. NMIC = 1; as IncomeTax class has misused the inheritance concept. At last NMPC = 3; because three methods in class Salary performs in correct use of polymorphism. Which leads us to the input to the FRCR as [2 2 1 1 3]. Once supplying this input we get output as 17% which means that if the original cost of the project is e.g. 1,00,000 units then the refactoring charges are 17,000 units.

Snippet 1:

```
class Employee{
private:
    int employee_id;
    String designation,
dept_name;
    float da, basic, gross;
public:
    print();
…….
}
class EmpAddress{
private:
   int employee_id;
   String  apartment_no,
flat_name, street_name;
public:
    printk();
…….
}

class EmpName
{
private:
    int employee_id;
    String first_name,
middle_name, last_name;
public:
    printl();
…….
}
```

Snippet 2:

```
class Employee{
private:
    int employee_id,
dept_name;
public:
   void setInfo();
   void showInfo();
}
```

Snippet 3:

```
class  HourlyPaiedEmp
extends Employee{
private:
     int hours;
public:
     void showHours();
}
```

Snippet 4:

```
class HourlyPaiedEmp{
private:
     int employee_id,
dept_name, hours;
public:
     void setInfo();
     void showInfo();
     void showHours();
}
```

Snippet 5:

```
class IncomeTax extends Employee{
private:
     float totalTax;
public:
     void showTax();
}
```

Snippet 6:

```
class  Salary{
……..
public:
   int calcSal(MonthlyEmp);
   int calcSal(HourlyPaidEmp);
   int calcSal(WeeklyPaidEmp);
}
```

Snippet 7:

```
class  Salary{
……..
public:
   int calcSalMonthlyPaid();
   int calcSalHourlyPaid();
   int calcSalWeeklyPaid();
}
```

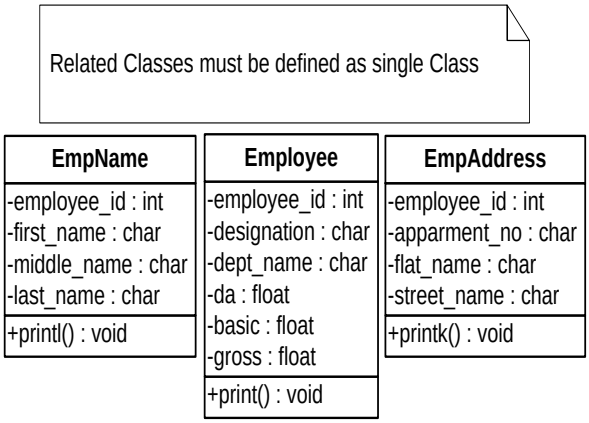
V. Conclusion

This paper presents a methodology to analyze cost of refactoring projects using a fuzzy logic based system (FRCR). The examples simulated indicate the potential for using such a procedure for analyzing the cost of complex systems and performing a meaningful evaluation and/or analysis of the cost. Analysis based on the rules mansion in

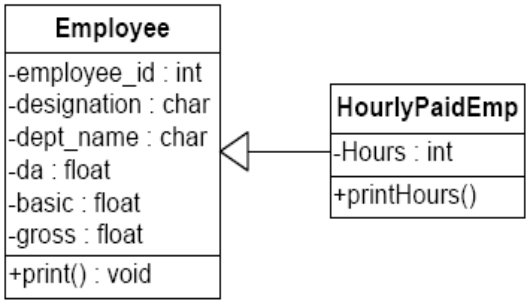
Table 2. leads a conclusion that for these opportunities as an input one may estimate the cost of a project, which will be based on the number of faults count, the range of cost measured in simulation for random numbers is found to be 3-85% of the original project cost.

The cost impact analysis is performed on these five inputs and conclusion is derived that the major cost is encured due to NVPEC and NMPC because from our experience of studying projects, when concluded, gives us the idea that it is difficult to find more instances of NVPEC and NMPC but when found, even if they are very few in count, impacts highly on RC. On the other hand, NCMC can be found easily in large count but at the same time its impact is negligible on RC. NLUIC and NMIC leaves very close impact on RC which is intermediate compared to other three.

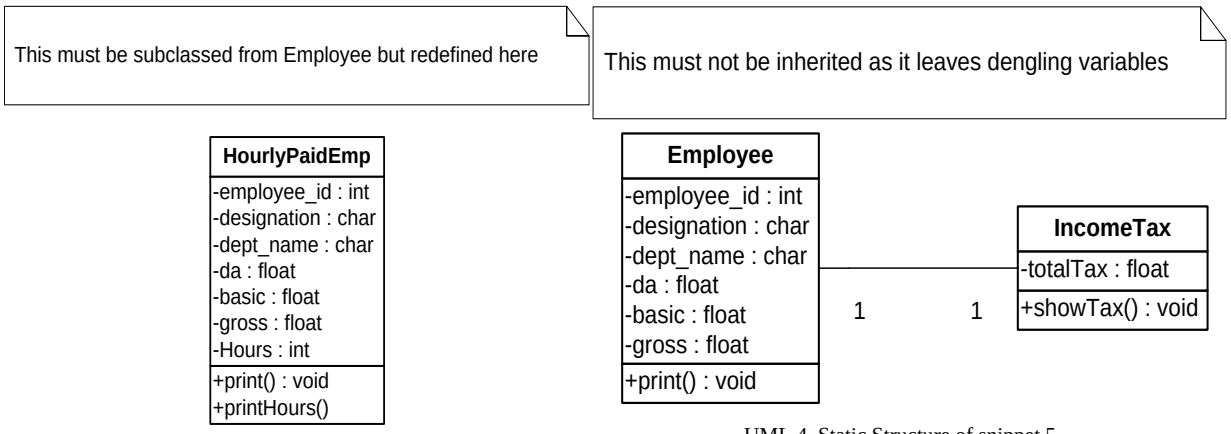
FRCR if used then the homogeneous cost of refactoring can be anticipated. The companies which find difficulty in persuasive to their clientele to justify their charges for making the changes in the software; especially when the software system is refactored; as in refactoring the clientele may argue, when they will not see changes in the outlook of the system. So such a system may be useful for various software companies who undertakes refactoring projects and where it is difficult to analyze the final cost of the refactored project.



UML 1. Static structure of snippet 1.

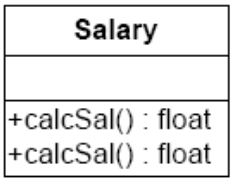


UML 2. Static Structure of snippet 2 – 3.

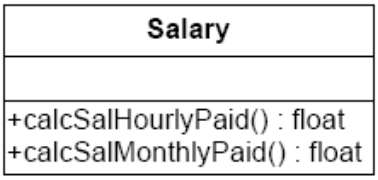


UML3. Static structure of snippet 4.

UML 4. Static Structure of snippet 5.



UML 5. Static structure of snippet 6.



UML 6. Static structure of snippet 7.

ACKNOWLEDGMENT

This work is incomplete if few people are not acknowledged. The first and the most important person who motivated and boost us to do this contribution in research community is Mr. J. T. Lalchandani, we thank you for the great help. We also thank CHARUSAT, our university which is always ready to support us in research in terms of funding and providing useful resources.

REFERENCES

[1] U. Zdun, "*Using Split Objects for Maintenance and Reengineering Tasks*," 8th European Conference on Software Maintenance and Reengineering (CSMR), Tampere, Finland, March, 2004.

[2] Serge Demeyer, Stéphane Ducasse, Kim Mens, Adrian Trifu, Rajesh Vasa, and Filip Van Rysselberghe, "*Object Oriented Reengineering"*, ISBN 978-3-540-22405-1, June, 2004.

[3] Oscar Nierstrasz, Stéphane Ducasse and Serge Demeyer, "*Object Oriented Reengineering Patterns an Overview*," ISBN 978-3-540-29138-1, October, 2005.

[4] Ankit Desai, Jaimin Chavda, Amit Thakkar, Amit Ganatra, and ypkosta, "*Refactoring Software Projects Using Object Oriented Concepts*," presented at ICICCA, Bangaluru 2010.

[5] Bryan Klingenberg, "A Time-Varying Harmonic Distortion Diagnostic Methodology Using Fuzzy Logic," july2004.

[6] L. A. Zadeh, "Fuzzy Sets, Information and Control," 1965.

[7] E. Piveta, J. Araujo, M. Pimenta, A. Moreira, P. Guerreiro, and R. T. Price, "*Searching for Opportunities of Refactoring Sequences: Reducing the Search Space*," Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International.

[8] Soodeh Hosseini and Mohammad Abdollahi Azgomi, "*UML Model Refactoring with Emphasis on Behavior Preservation*," august - 2008.

[9] Hojjat Salehinejad and Siamak Talebi, "*Dynamic Fuzzy Logic-Ant Colony System-Based Route Selection System*," Hindawi Publishing Corporation Applied Computational Intelligence and Soft Computing, vol. 2010.

[10] Jim Murtha, "Applications of fuzzy logic in operational meteorology," June, 2010.

[11] José M. Alonso and Luis Magdalena, "An Experimental Study on the Interpretability of Fuzzy Systems," IFSA-EUSFLAT 2009.

[12] Miguel P. Monteiro and João M. Fernandes, "*Refactoring a Java Code Base to AspectJ: An Illustrative Example*," Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005.

[13] Dave Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella, "*Automated Refactoring of Object Oriented Code into Aspects*," Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005.

[14] M. Delgado and F. Cofré, "*A Fuzzy model for work performance assessment*," EUSFLAT - LFA 2005.

[15] M. Oussalah and A. Eltigani, "Personalized Information Retrieval system in the Framework of Fuzzy Logic," EUSFLAT - LFA 2005.

[16] M. Hellmann, "Fuzzy Logic Introduction."

[17] A. Kiezun, M. Ernst, F. Tip, and R. Fuhrer. "*Refactoring for parameterizing Java classes*," In the Proceedings of International Conference on Software Engineering (ICSE) 2007.

[18] Raffi Khatchadourian, Jason Sawin, and Atanas Rountev, "Automated Refactoring of Legacy Java Software to Enumerated Types." ICSM, 2007.
[19] N. Raj Kiran and V. Ravi, "*Software reliability prediction by soft computing techniques*," The Journal of Systems and Software, 2007.

Ankit B Desai is a student of Master of Technology in computer Engineering at Dharmsinh Desai University, Nadiad, Gujarat, India. He is also an Assistant Professor at U and P U. Patel Department of Computer Engineering at Charotar University of Science and Technology, Changa, Dist. Anand, Gujarat, India. He has received his B.E. degree from Charotar Institute of Technology, Changa, Gujarat, India in 2007. He has joined M. Tech. at Dharmsinh Desai University, Nadiad, Gujarat, India in 2010. His current research interest includes Refactoring, Soft-computing and Data Mining Classification (Cost-Sensitive Boosters).

Amit P. Ganatra (B.E.-'00-M.E. '04-Ph.D.* '11) has received his B.Tech. and M.Tech. degrees in 2000 and 2004 respectively from Dept. of Computer Engineering, DDIT-Nadiad from Gujarat University and Dharmsinh Desai University, Gujarat and he is pursuing Ph.D. in Information Fusion Techniques in Data Mining from KSV University, Gandhinagar, Gujarat, India and working closely with Dr. Y. P. Kosta (Guide). He is a member of IEEE and CSI. His areas of interest include Database and Data Mining, Artificial Intelligence, System software, soft computing and software engineering. He has 11 years of teaching experience at UG level and concurrently 7 years of teaching and research experience at PG level, having good teaching and research interests. In addition he has been involved in various consultancy projects for various industries. After spending almost a year in C.U.Shah college of Engineering, Wadhwan, Gujarat, he joined CITC as a faculty member in 2001. His general research includes Data Warehousing, Data Mining and Business Intelligence, Artificial Intelligence and Soft Computing. In these areas, he is having good research record and published and contributed over 70 papers (Author and Co-author) published in referred journals and presented in various international conferences. He has guided more than 90 industry projects at under graduate level and 47 dissertations at Post Graduate level. He is concurrently holding Associate Professor (Jan 2010 till date), Headship in computer Engineering Department (since 2001 to till date) at CSPIT, CHARUSAT and Deanship in Faculty of Technology-CHARUSAT (since Jan 2011 to till date), Gujarat. He is a member of Board of Studies (BOS), Faculty Board and Academic Council for CHARUSAT and member of BOS for Gujarat Technological University (GTU). He was the founder head of CE and IT departments of CITC (now CSPIT).