An Approach for Source Code Classification Using Software Metrics and Fuzzy Logic to Improve Code Quality with Refactoring Techniques

Pornchai Lerthathairat and Nakornthip Prompoon

Software Engineering Lab, Center of Excellence in Software Engineering,
Department of Computer Engineering,
Faculty of Engineering, Chulalongkorn University, Thailand
pornchai.lerthathairat@thomsonreuters.com,
Nakornthip.S@chula.ac.th

Abstract. The problem of developing software quality is developer's experience which has different style and does not care about coding with principle that causes error or even bad smell. To reduce the risk of causing bad smell, the developer should concern with a good design principle and coding well. In addition, knowing the qualification and the characteristic of code is also important to promptly support verifying, recovering bad smell and improving them to be a good code. This research presents an approach for source code classification using software metrics and fuzzy logic to improve code quality with refactoring techniques. Our approach composed of 3 main sections; Source code definition with metrics and evaluation to classify source code type, Source code classification with fuzzy logic and Source code improvement with refactoring. The result of our approach is able to classify source code in type correctly and improve bad smell, ambiguous code to be a clean code.

Keywords: Code Quality Improvement, Clean Code, Fuzzy Logic, Refactoring, Software Metrics.

1 Introduction

In order to get software quality by Phillip Crosby's principle "Conformance to requirement"[15], design and programming process is an important step about software development but the problems occur repeatedly during developing, for example, analysis requirements inaccurately, time pressure or even the developer's experience that coding in different style. These problems generate poor design and bad smell [2] that needs to correct them before delivering to the users. One technique of Bad smell resolution is refactoring which is a method of elimination bad smell.

In fact, the developer tries to design and do programming with good code or clean code. Clean code can help internal software working effectively and indicate a good design. By the characteristic of clean code that Robert C. Martin [16] explained; it should be simple, well-written, easy to enhance. When the developer measure and evaluate the quality, the result will show the different between clean code and bad

J.M. Zain et al. (Eds.): ICSECS 2011, Part III, CCIS 181, pp. 478–492, 2011.
© Springer-Verlag Berlin Heidelberg 2011

smell that the developer should concentrate on coding with clean code from the start rather than eliminate bad smell at the end.

This research presents an 0061pproach for source code classification using software metrics and fuzzy logic to improve code quality with refactoring techniques that divided into 3 main sections; first section is source code definition with metrics and evaluation, the method classifies source code with software metrics to categorize the code types. After classifying, we have found an ambiguity between clean code and bad smell that we define as ambiguous code. The second section is source code classification with fuzzy logic because the ambiguous code cannot classify or measure with metrics. We have to bring fuzzy logic to clarify the ambiguity by using knowledge's expert to set rule base, for example, bad smell gets rule base from analysis Mika Mäntylä's research [12]. The last section is source code improvement with refactoring. The method improves bad smell and ambiguous code with appropriated refactoring techniques until they are verified to be clean with clean code criteria. The scope of this research is only classifying source code and selects the improvement techniques suitably in order to be clean code with quality but does not measure the value of quality in each code type.

Section 2 briefly summarizes background knowledge relevant to this approach, section 3 is related works, and section 4 is the approach and the methodology. The last section provides a conclusion and future works.

2 Knowledge Background and Related Works

2.1 Design Principle and Source Code Classification

According to a good design and programming [1, 19, 24] the developer should understand that each object in software can interact and collaborate of each other. Object-oriented programming (OOP) is a programming paradigm base on the idea of using the flexible objects work together to enhance programming capability with quality. To accomplish the objectives, there are five important principles of programming [16]; Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Dependency Inversion Principle (DIP) and Interface Segregation Principle (ISP). Therefore, to write code with the design principle will be possible to conduce a clean code but if it does not follow the principles. They would produce a bad smell instead. Our approach uses the principle for defining clean code and the rule base for defuzzication in a classification process.

The source code classification with software metrics defines the code type into 2 groups; bad smell and clean code. However after doing a research, we have found another type which is an ambiguity. It lines between clean code and bad smell. We introduce it as an ambiguous code. Its definition is in the following details.

2.1.1 Bad Smell

This kind of code does not work in the right qualification. [12, 25] The reason of causing bad smell for example, the developer's experience which is different styles, requirements changed frequently and the external facts which constrain the design and programming process more shorten. The developer does not have time to think about

coding with the design principle, finally the complexity of code and bad smell that cause software quality degradation. The developer must pay an attention to verify and find a solution to correct it. There are 22 bad smell types for an example, Long method has large size and takes many responsibilities that hard to understand and modify, Large Class is a class that tries to do many jobs in software. These classes have too many instance variables or methods.

Our approach uses the characteristic of bad smell type to define bad smell criteria and the rule base for defuzzication in the classification process then improves it to be clean code with refactoring techniques.

2.1.2 Clean Code

A kind of code that helps software working smoothly because of its characteristic follows the design principle [16], for example, readability, none of the duplication. In additional, clean code help decreasing risk when the requirement changed, guiding the developer to get a good design and programming to reduce the amount of corrections. There are more 80 clean code types for an example; Function should be small and do one thing, it is easy to understand and flexible to change, Meaningful Names is the name of a variable, function, or class, should answer all the purpose. It should tell the developer why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.

So, our approach uses the characteristic of clean code to define clean code criteria and the rule base for defuzzication in the classification process.

2.1.3 Ambiguous Code

After classifying source code with metrics, we have found an unspecified code type. It cannot measure by metrics same as clean code and bad smell but the ambiguity between clean code and bad smell need to clarify and categorize it into the correct group because it is necessary to improve bad smell and some ambiguous code to be clean code

Therefore, classifying three groups of source code, our approach uses software metrics to present the qualification of each code and also ensure that classification method works precisely.

2.2 Software Metrics

Our research uses software metrics because it is a tool of software measurement with Mathematics and science [22, 13]. The approach uses McCabe, Kamerer and Halstead [11, 17, 18, 20] to set the specification of outlier to measure and to classify source code. But there is the limitation of using metrics is not able to classify the ambiguity between clean code and bad smell because of high complexity and does not support non-ambiguous then we have to find other techniques to clarify the ambiguity.

2.3 Fuzzy Logic

Fuzzy logic [7, 8, 26] is derived from fuzzy set theory to deal with reason and imitate the expert decision where binary set two-valued logic. Fuzzy logic ranges in degree (Set membership values) between 0 and 1. The approach uses Lotfi Zadeh concept

and techniques [5, 6, 7, 8, 9] to classify ambiguous code. When the method uses software metrics to measure and classify source code. There is some ambiguous code, which cannot classify because its qualification is very complex. We need to clarify and make a good decision under complicated conditions with If-Then form that conforms to human logic. The result from using fuzzy logic does not value only true or false but show the relation between true and false.

3 Our Approach

Our approach with aim to classify source code with software metrics and fuzzy logic to improve code quality with refactoring techniques is divided into 3 main sections as shown in Figure 1.

First section is source code definition with metric measurement and evaluation to define the code criteria for classification with software metrics. Second section is source code classification with fuzzy logic to classify three groups of code with rule base and clarify the ambiguity with fuzzy logic. The third section is source code improvement with refactoring to improve bad smell and ambiguous code with refactoring techniques until they are clean. Then we verify all the method for affirmative. The detail of each section is described in following detail:

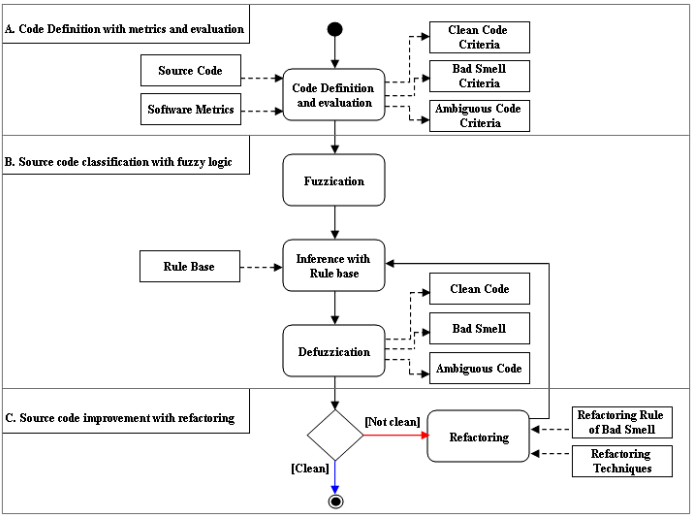


Fig. 1. An approach for source code classification using software metrics and fuzzy logic to improve code quality with refactoring techniques

3.1 Code Definition with Metric Measurement and Evaluation

In this section, to classify bad smell, clean code and ambiguous code. We have to prepare the classification rules with software metrics and the specification of outlier.

The method organizes two groups of source code sample; first set of source code for setting fuzzy rules and second set of source code for verification.

3.1.1 Bad Smell Selection Criteria

To select bad smell for our approach, we use Mäntylä's suggestion [12] which ranked bad smell group from level 0-5. Level 0 means bad smell detection is very complex that cannot use metrics to detect. This level needs only the expert to detect them. Level 5 means bad smell detection is easy to detect with few metrics. Our approach selects level 4 – 5 which focus only the Bloaters group. The Bloater is something that has grown large that it cannot be effectively handled.

Therefore, by the bad smell criteria; we detect it with an semi auto-system that does not count on any expert. So bad smell which qualified are the Long method, Large class and Long parameter list.

3.1.2 Clean Code Selection Criteria

To select clean code for our approach, we impose five conditions;

- Code is explicit and readable. The developer can understand the qualification of code, where code link to and the output.
- Code can be explained with software metrics that means code is written with OOP principle [16] and can be measurable [13, 22].
- Code does not develop from the engine. The developer should analyze and plan. They should not count on only the machine because when there is any problem happened. They should know the cause immediately.
- Code is flexible and portability that will accelerate the software cycle.
- Code is not duplicated because one object should respond for only one responsibility. If it is repeated, the developer should re-categorize to prevent duplication.

For our approach, we choose Function, Classes and Comment because function and class are important in programming. It has to be well-structure and comment is necessary involved to describe the responsibility purpose of function and class.

3.1.3 Source Code Measurement with Software Metrics

To select metrics for Source code measurement, we use basic software metrics [11, 17, 18, 20]. We set the specification of outlier to classify source code, as the following tables. (Software metrics Acronym is available at Appendix)

3.1.3.1 Metrics and the Specification of Outlier of Bad Smell. According to the bad smell criteria, we bring Mäntylä's suggestion [12] to set the specification of outlier is shown in the table 1. In the table, we use NLOC to measure Long Method. The specification of outlier is equal to or more than 60. If a source code is evaluated and NLOC more than or equal to 60 that means it is in a bad smell type.

Table 1. Metrics and the specification of outlier of Bad smell

Bad Id	Description	Metrics (m)	The specification of outlier (x)
1	Long Method	NLOC(**1**)	x>=60
		NILI(**2**)	x>=200
		CC(**3**)	x>=30
		ILCC(**4**)	x>=60
2	Long Parameter List	NOP(**5**)	x>7
3	Large Class	LCOM(**8**)	x>0.8
		LCOM-HS(**9**)	x>1.0
		NFD(**7**)	x>20
		NOM(**6**)	x>20

3.1.3.2 Metrics and the Specification of Outlier of Clean Code. The metrics for clean code classification, we analyze from the idea proposal by Robert and software experts [16] to choose metrics accord with qualified clean code by criteria as the table 2. In the table, we measure Small Function by using NLOC and the specification of outlier that equal 20 or less. If source code is evaluated and NLOC is less than or equal to 20 that means it is in a clean code type.

Table 2. Metrics and the specification of outlier of Clean code

Clean Id	Description	Metrics (m)	The specification of outlier (x)	Clean Id	Description	Metrics (m)	The specification of outlier (x)
1	Comment	PCC	x>=20	9	Class Organization (Cont.)	NOIM	x<20
2	Small Functions	NLOC(**1**)	x<=22			NMI	x<1
		NILI(**2**)	x<=50	10	Encapsulation	NPF	x<1
		CC(**3**)	x<=15	11	Classes Should Be Small	NOP	x<=5
		ILCC(**4**)	x<=40			NOV	x<=8
3	Do One Thing	RC	1.5<=x<3.5			NOO	x<=6
4	One Level of Abstraction per Functions	ABL	x<1.5			NOM(**6**)	x<=10
5	Switch Statements	ILND	x<=4			NFD(**7**)	x<=10
6	Function Arguments	NOP(**5**)	x<=5	12	Maintaining Cohesion	LCOM(**8**)	x<=0.5
7	Have No Side Effects	NOI	x<1.4			LCOM-HS(**9**)	x<=0.8
8	Structured Programming	ISS	x=1	13	Organizing for Change	ACML	x>0
9	Class Organization	NOC	x<6			ECML	x<=50
		DIT	x<6				

3.1.3.3 Metrics and the Specification of Outlier of Ambiguous Code. The method uses software metrics to help measuring and classifying source code as the previous section. We analyzed the specification of outlier and found some codes, which cannot describe what type of code. We defined them as an ambiguous code and try to use some metrics from table 1 and 2 to classify but the result is still ambiguity. In the table 3, we measure Ambiguity in method by using NLOC and the specification of outlier which evaluated more than 20 but less than 60 that mean it is in an ambiguous code because it does not present clean value and bad smell value.

Table 3. Metrics and the specification of outlier of Ambiguous code

Amb Id	Description	Metrics (m)	The specification of outlier (x)	Amb Id	Description	Metrics (m)	The specification of outlier (x)
1	Ambiguity in Comment	PCC	x<20	9	Ambiguity in Class Organization (Cont.)	NOIM	x>=20
2	Ambiguity in Method	NLOC(**1**)	22<x<60			NMI	x>1
		NILI(**2**)	50<x<200	10	Ambiguity in Encapsulation	NPF	x>1
		CC(**3**)	15<x<30	11	Ambiguity in Classes	NOP	x>5
		ILCC(**4**)	40<x<60			NOV	x>8
3	Ambiguity in One Thing	RC	1.5>x>3.5			NOO	x>6
4	Ambiguity in Abstraction per Functions	ABL	x>=1.5			NOM(**6**)	10<x<=20
5	Ambiguity in Switch Statements	ILND	x>4			NFD(**7**)	10<x<=20
6	Ambiguity in Arguments	NOP(**5**)	5<x<=7	12	Ambiguity in Cohesion	LCOM(**8**)	0.5<x<=0.8
7	Ambiguity in Side Effects	NOI	x>=1.4			LCOM-HS(**9**)	0.8<x<=1.0
8	Ambiguity in Structured Programming	ISS	x<1	13	Ambiguity for Change	ACML	x<0
9	Ambiguity in Class Organization	NOC	x>=6			ECML	x>50
		DIT	x>=6				

After using metrics with first set of source code, the results show in the table 1, 2 and 3. The classification result is shown in the table 4. There is an ambiguity in code and cannot specify by using metric from bad smell or clean code.

Table 4. Classification Evaluation (Pre-Test)

No.	Metrics(m)	Clean	Ambiguous	Bad Smell
1	NLOC	16.4	42.1	78.4
2	NILI	45.8	176.5	344.1
3	CC	14.1	27.8	44.2
4	ILCC	32.1	51.3	112
5	NOP	2.6	4.2	5.5
6	NOM	7.6	10.2	12.3
7	NFD	6.4	9.2	11.9
8	LCOM	0.4	0.9	0.9
9	LCOM-HS	0.6	0.8	1.1

From the table 4, to measure clean code, bad smell and ambiguous code, we use nine metrics, which is shared among three groups of source code. The measurement result is from initiation source code samples that point out an ambiguous code section cannot be judged or put into bad smell type because there is clean code qualification combines in ambiguous criteria.

This point indicates that metrics is not troubleshooting for non-ambiguous, it is necessary to use another measuring technique that can decide and extend the clarity by using expert's knowledge.

3.2 Source Code Classification Method with Fuzzy Logic

This part is for applying the fuzzy logic to classify bad smell, clean code and ambiguous code, there are following 3 subsections;

3.2.1 Fuzzication of Input Variable

This section brings the specification of outlier from bad smell, clean code and ambiguous code to create the relation by Sigmoidal membership function [3] to implicate membership input set. We also use Triangular membership function and Trapezoidal membership function to present the qualification of clean code, bad smell and ambiguous code. Then we use Mamdani-type inference to implicate the input set into 2 types; the first type is divided into 2 ranges are called less and more, the second type is divided into 3 ranges are called low, moderate and high as shown in Figure 2:

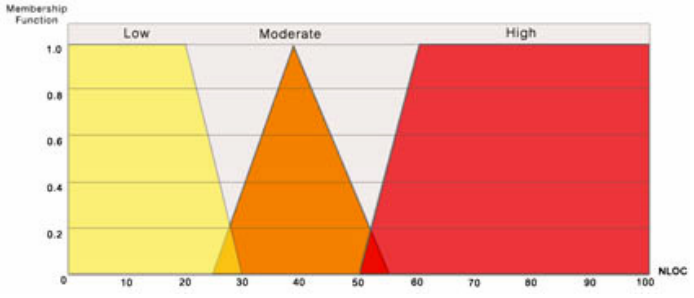


Fig. 2. Example of fuzzication input variable: NLOC

From the figure 2; NLOC is from all metrics and the specification of outlier and μx is Sigmoidal membership function, the fuzzication process converts the specification of outlier into input variable. This example presents input variable that shows the value as second type that divided into 3 ranges which are low moderate and high.

The fuzzication conduces to rule base which is a tactic in fuzzy model. The programming expert creates rule bases by forming the condition [3] which is completely true. We set the rule base by our programming experience more than 10 years.

3.2.2 Inference with Rule Base and Applying with Fuzzy Logic

The classification with fuzzy logic starts applying with rule base, as the following detail:

3.2.2.1 Bad Smell. This section analyzes and implicates bad smell with rule base; Bad smell type refers from rule condition as the table 5. In the table, Long method uses bad smell rule (NLOC is high) OR (NILI is high) OR (CC is high) OR (ILCC is high); according to the specification of outlier in table 1, the variable equal to 60 or more must be “High” and “OR” is an operation to present the maximum of variable along fuzzy logic that implicate as bad smell as show in the figure 2.

Table 5. Bad Smell Rules Formation

Rule No	Bad Smell Type	Rule condition
1	Long Method	(NLOC is high) OR (NILI is high) OR (CC is high) OR (ILCC is high)
2	Long Parameter List	(NOP is high)
3	Large Class	(LCOM is high) OR (LCOM-HS is high) OR (NFD is high) OR (NOM is high)
4		(NFD is high)
5		(NOM is high)

3.2.2.2 Ambiguous Code. When an ambiguous code is found, the process will analyze and also implicate with rule base. Ambiguous type refers from rule condition in the table 6. In the table, we use ambiguous rule condition to measure Small function; (NLOC is moderate) AND (NILI is moderate) AND (CC is moderate) AND (ILCC is moderate); according to the specification of outlier in table 3, the variable is between 20 and 60 is called "Moderate" that implicate as ambiguous code in the figure 2.

Table 6. Ambiguous Code Rules Formation

Rule No	Ambiguous Type	Rule condition	Rule No	Ambiguous Type	Rule condition
1	Comment	(PCC is less)	10	Encapsulation	(NPF is more)
2	Small Function	(NLOC is moderate) AND (NILI is moderate) AND (CC is moderate) AND (ILCC is moderate)	11	Classes Should Be Small	(LCOM is moderate) OR (LCOM-HS is moderate) OR (NFD is moderate) OR (NOM is moderate)
3	Do One Thing	(RC is low) OR (RC is high)	12		(NFD is moderate)
4	One Level of Abstraction per Functions	(ABL is more)	13		(NOM is moderate)
5	Switch Statements	(ILND is more)	14		(NOV is more)
6	Function Arguments	(NOP is moderate)	15		(NOO is more)
7	Have No Side Effects	(NOI is more)	16	Maintain Cohesion	(LCOM is moderate) OR (LCOM-HS is moderate)
8	Structured Programming	(ISS is less)	17	Organizing for Change	(ACML is less)
9	Class Organization	(NOC is more) AND (DIT is more) AND (NOIM is more) AND (NMI is more)	18		(ECML is more)

3.2.2.3 Clean Code. Clean code classification, the section is similar to previous section. Clean code will be analyzed and implicated with clean code rule base. Clean code Type is clean without any bad smell. If the process still finds some bad smells in source code, it should be clean up until it is clearly clean. The clean code type refers from rule condition as show in the table 7. In the table, we use clean code rule condition (PCC is Moderate) in comment; according to the specification of outlier in table 2, the variable equal to 20 or more must be "Moderate" that implicate as clean code as Figure 2.

Table 7. Clean Code Rules Formation

Rule No	Clean Code	Rule condition	Rule No	Clean Code	Rule condition
1	Comment	(PCC is more)	10	Encapsulation	(NPF is less)
2	Small Function	(NLOC is low) AND (NILI is low) AND (CC is low) AND (ILCC is low)	11	Classes Should Be Small	(LCOM is less) OR (LCOM-HS is less) OR (NFD is low) OR (NOM is low)
3	Do One Thing	(RC is moderate)	12		(NFD is low)
4	One Level of Abstraction per Functions	(ABL is less)	13		(NOM is low)
5	Switch Statements	(ILND is less)	14		(NOV is less)
6	Function Arguments	(NOP is low)	15		(NOO is less)
7	Have No Side Effects	(NOI is less)	16	Maintain Cohesion	(LCOM is less) OR (LCOM-HS is less)
8	Structured Programming	(ISS is more)	17	Organizing for Change	(ACML is more)
9	Class Organization	(NOC is less) AND (DIT is less) AND (NOIM is less) AND (NMI is less)	18		(ECML is less)

3.2.3 Defuzzication

After classifying a source code and apply with all rules, which are 5 rules of bad smell, 18 rules of ambiguous code and 18 rules of clean code. The next section is defuzzication by setting input set and applying with Center of gravity (COG) [3]. From the figure 3, it presents three possible situations of the qualification for each code as the following detail.

3.2.3.1 Bad Smell. From the Figure 3, the gravity value is represented on the right that means bad smell; first the method must improve bad smell with refactoring techniques until it is clean and then reduces ambiguous code.

3.2.3.2 Ambiguous Code. From the Figure 3, the gravity value is represented on the middle that means ambiguous code; it should be improved by refactoring techniques until it is verified to be clean. After the classification is completed, bad smell and ambiguous code have to be improved with refactoring to be clean code.

3.2.3.3 Clean Code. From Figure 3, after doing defuzzication, this method can approve a group of clean code is absolutely clean but permitted to have some ambiguous code. All bad smell and ambiguous code need to improve with refactoring techniques.

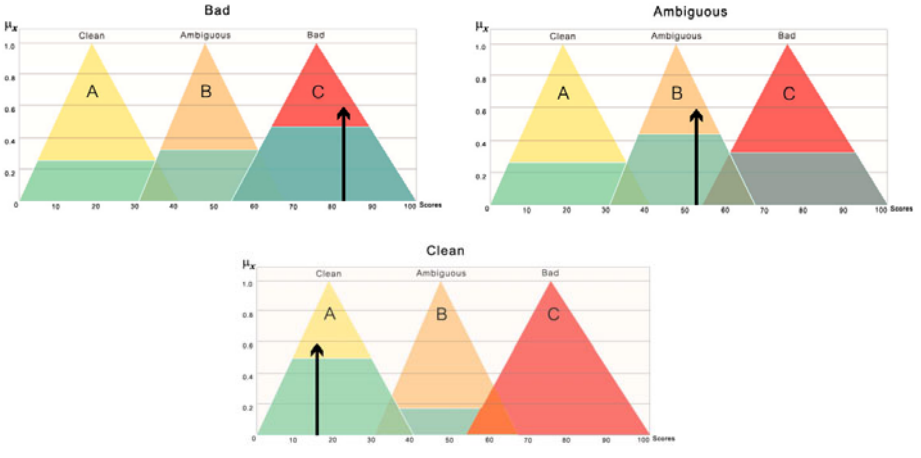


Fig. 3. Center of gravity value: Bad smell, Ambiguous Code and Clean Code

3.3 Source Code Improvement with Refactoring

This process emphasized on source code improvement. Every bad smell and ambiguous code has to improve with appropriated refactoring techniques by expert.

3.3.1 Improving Code by Clean Code Rules

To improve bad smell, we do it as the algorithm as the following;

Algorithm

```
Procedure ImproveSourceCodeToCleanCode(sourcecode)
begin
repeat
    codetype ← ClassifySourceCode(sourcecode);
    if (codetype is Bad Smell) then
        badsmell_type ← CheckBadsmellType(sourcecode);
        technique ← UseRefactoringTechniquefromTable(badsmell_type);
        sourcecode ← DoRefactoring(sourcecode, technique);
    else if (codetype is Ambiguous Code) then
        ambigous_type ← CheckAmbigousType(sourcecode);
        technique ← SuggestionFromProgrammingExpert(ambigous_type);
        sourcecode ← DoRefactoring(sourcecode, technique);
    end if
until (codetype is Clean Code or under acceptance criteria is true)
return {sourcecode}
end
```

By the algorithm, we classify source code. When we find bad smell, we use the refactoring techniques in the table 8 to improve it to be clean code. In additional, when we find source code as ambiguous code, we bring the suggestions from expert to select the appropriated refactoring techniques same as bad smell improvement until it is clean or the developer will decide to stop doing refactoring.

Table 8. Bad Smell Improvement with Refactoring Techniques

Refactoring Technique	Large Class	Long Method	Long Parameter List
Decompose Conditional		✓	
Duplicate observed data	✓		
Extract Class	✓		
Extract Interface	✓		
Extract Method		✓	
Extract Subclass	✓		
Introduce Parameter Object		✓	✓
Preserve Whole Object		✓	✓
Replace Method with Method Object		✓	
Replace Parameter with Method			✓
Replace Temp with Query		✓	

3.3.2 The Experiment Example of Our Approach

The experiment example of our approach presents source code improvement with appropriated refactoring techniques by expert which are represented in Section II Knowledge Background to bring bad smell on the right qualification as well as clean code.

From the table 9, to improve bad smell as the example; first step classify source code with bad smell rule base and we have found bad smell which are Long method and Long parameter list, so we select the suitable refactoring technique from table 8.

Second step presents the result after refactoring and classifying source code; we still have found bad smell which is Long parameter list that we have to improve it with refactoring again.

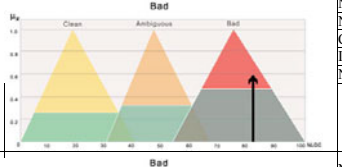
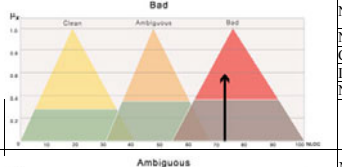
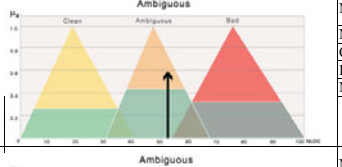
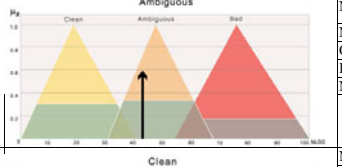
Third step presents that we can eliminate all bad smell but ambiguous code appear, they are Small Function, Do One Thing, Function Arguments and Have No Side Effects, so we have to select the appropriated techniques to manage them to be clean.

Fourth step presents that we can improve source code much better than the original source code but they are not clean as clean code criteria then we have to improve again by using knowledge's expert and different refactoring techniques which is Replace Parameter with Method.

The last step is measuring and evaluating source code. The result presents that almost source code is clean as clean code criteria. There is some of ambiguous code. We can accept because they do not impact to clean code qualification.

Therefore, refactoring can use for source code improvement. The method continuously cleans up bad smell and ambiguous code until they are verified to be a clean code without bad smell but the limitation of our approach is not able to count the refactoring cycle because some types of bad smell or ambiguous code are very complicated. Some cases, when improving bad smell might cause another bad smell. This is reason why cannot specify refactoring cycle certainly.

Table 9. The experiment example of source code improvement with refactoring techniques

Step	Classification Result	Metrics					Classification types	Refactoring techniques
1		NLOC	80.4	NOM	5.4		Long method	Extract Method
		NILI	311	NFD	5.5		Long parameter list	-
		CC	48	LCOM	0.3		-	-
		ILCC	121	LCOM-HS	0.5		-	-
		NOP	6.1				-	-
2		NLOC	44.7	NOM	5.4		Long parameter list	Introduce Parameter Object
		NILI	180	NFD	5.5		-	-
		CC	25.5	LCOM	0.3		-	-
		ILCC	48.7	LCOM-HS	0.5		-	-
		NOP	6.1				-	-
3		NLOC	42	NOM	5.4		Small Function	Replace Method with Method Object
		NILI	178	NFD	5.5		Do One Thing	-
		CC	22.2	LCOM	0.3		Function Arguments	-
		ILCC	40.1	LCOM-HS	0.5		Have No Side Effects	-
		NOP	5.4				-	-
4		NLOC	14.3	NOM	6.4		Function Arguments	Replace Parameter with Method
		NILI	48.2	NFD	6.2		Have No Side Effects	-
		CC	14.8	LCOM	0.3		-	-
		ILCC	34.7	LCOM-HS	0.5		-	-
		NOP	5.4				-	-
5		NLOC	14.0	NOM	6.6		Have No Side Effects	-
		NILI	47.8	NFD	6.2		-	-
		CC	14.2	LCOM	0.3		-	-
		ILCC	31.4	LCOM-HS	0.5		-	-
		NOP	2.4				-	-

3.4 Verification and Conclusion Process

This part is for analyzing the method and concluding a classification testing, there are 2 subsections as following;

Table 10. Comparison Source code: Pre-test and Post-test

Metrics (m)	The specification of outlier	Clean	Ambiguous		Bad Smell	
			Pre-test	Post-test	Pre-test	Post-test
NLOC	x <=22	16.4	42.1	**19.2**	78.4	**19.4**
NILI	x <=50	45.8	176.5	**68.3**	344.1	**88.2**
CC	x <=15	14.1	27.8	**16.5**	44.2	**15.8**
ILCC	x <=40	32.1	51.3	**35.4**	112	**33.3**
NOP	x <=5	2.6	4.2	**3.1**	5.5	**2.9**
NOM	x <=10	6.4	10.2	**7.6**	12.3	**8.1**
NFD	x <=10	6.6	9.2	**8.7**	11.9	**7**
LCOM	x <=0.5	0.4	0.8	**0.5**	0.9	**0.5**
LCOM-HS	x <=0.8	0.6	0.8	**0.6**	1.1	**0.6**

3.4.1 Verifying a Source Code

This section takes a source code for verification to affirm the correction of classification and source code improvement method.

After source code improvement process, every source code samples have to be compared between pre-test and post-test evaluation and present the code quality value which passes the classification method and refactoring. The value comparison is shown in the figure of source code: Pre-test and Post-test is shown in the table 10.

All 40 source code samples are classified with classification rules. The approach inspects and classifies clean code, bad smell, also specify an ambiguous code explicitly. Moreover bad smell and ambiguous code can be improved to be clean code but the quality cannot be better than developing code with good design and programming from the start.

4 Conclusion and Future Work

After, we classify source code with software metrics and fuzzy logic. We prove that we can classify three groups of source code; bad smell, ambiguous code and clean code. We can also improve bad smell and ambiguous code to be more quality by the programming expert. However, there is a limitation of improving source code with refactoring techniques that do not do it automatically by tool. The future work, hopefully to improve source code with refactoring automatically and use more source code samples to measure code quality with software metrics under software standards to build the code quality in depth.

References

1. Kay, A.C.: The Early History of Smalltalk, pp. 69–95 (1993)
2. Boehm, B.W., Basili, V.R.: Software Defect Reduction Top 10 List. IEEE Computer 34(1), 135–137 (2001)
3. Meesad, P.: Fuzzy Logic. Fuzzy systems and Neural Networks Lecture. Faculty of Information Technology, King Mongkut's University of Technology North, Bangkok
4. Stroggylos, K., Spinellis, D.: Refactoring-Does It Improve Software Quality? In: Stroggylos, K., Spinellis, D. (eds.) 5th International Workshop on Software Quality, vol. 10, IEEE Computer Society, Los Alamitos (2007)
5. Zadeh, L.A.: Fuzzy Sets. Information and Control 8, 338–353 (1965)
6. Zadeh, L.A.: Is There a Need for Fuzzy Logic? Fuzzy Information Processing Society, Annual Meeting of the North American 178(13), 2751–2779 (2008)
7. Zadeh, L.A.: Toward a Perception-Based Theory of Probabilistic Reasoning with imprecise probabilities. Special Issue on Imprecise Probabilities, Journal of Statistical Planning and Inference 105, 233–264 (2002)
8. Zadeh, L.A.: Toward a theory of fuzzy information granulation and its centrality in human reasoning and fuzzy logic. Fuzzy Sets Systems 90(2), 111–127 (1997)
9. Zadeh, L.A., Klir, G.J., Bo, Y.: Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems. In: Advances in Fuzzy Systems - Applications and Theory, vol. 6, World Scientific Pub Co. Inc, Singapore (1996)
10. Fowler, M.: Refactoring: Improving the Design of Existing Code. XP/Agile Universe (2002)
11. Halstead, M.H.: Elements of Software Science. In: Operating and programming systems series, Elsevier Science Inc., New York (1977)
12. Mäntylä, M., Vanhanen, J., Lassenius, C.: A Taxonomy and an Initial Empirical Study of Bad Smells in Code. In: International Conference on Software Maintenance, pp. 381–384 (2003)
13. Fenton, N.E.: Software Metrics, A Rigorous Approach. Chapman & Hall, London (1991)
14. Drucker, P.: Innovation and Entrepreneurship. Collins (1985)
15. Crosby, P.: Quality is Free. McGraw-Hill, New York (1979)
16. Martin, R.C.: Clean Code. A Handbook of Agile Software Craftsmanship. Prentice-Hall, Englewood Cliffs (2008)
17. Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design. IEEE Trans. Software Eng. 20(6), 476–493 (1994)
18. Chidamber, S.R., Kemerer, C.F.: Towards a Metrics Suite for Object Oriented Design. In: OOPSLA, pp. 197–211 (1991)
19. Feldman, S.I., Kay, A.C.: A conversation with Alan Kay. ACM Queue 2(9), 20–30 (2004)
20. McCabe, T.J.: A Complexity Measure. IEEE Trans. Software Eng. SE-2(4), 308–320 (1976)
21. Ruhroth, T., Voigt, H., Wertheim, H.: Measure, Diagnose, Refactor: A Formal Quality Cycle for Software Models. In: 35th EUROMICRO-SEAA, pp. 360–367 (2009)
22. DeMarco, T.: Controlling Software Projects: Management, Measurement, and Estimates. Prentice-Hall, Englewood Cliffs (1986)
23. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Trans. Software Eng. 30(2), 126–139 (2004)
24. Alan Curtis Kay, `http://en.wikiquote.org/wiki/Alan_Kay`
25. Code smell, `http://en.wikipedia.org/wiki/Code_smell`
26. Fuzzy Logic, `http://en.wikipedia.org/wiki/Fuzzy_logic`
27. Software Quality, `http://en.wikipedia.org/wiki/Software_quality`

Appendix: Software Metrics Acronym

Acronym	Description	Acronym	Description
ABL	Abstraction Level	NLOC	Number Line of Code
ACML	Afferent Coupling at Method Level	NMI	Number of Methods Inherited
CC	Cyclomatic Complexity	NOC	Number of Children
DIT	Depth of Inheritance Tree	NOI	Number of Immutable
ECML	Efferent Coupling at Method Level	NOIM	Number of Interface with many Methods
ILCC	InLine Cyclomatic Complexity	NOM	Number of Methods
ILND	InLine Nesting Depth	NOO	Number of Overloads