Code Evaluation Using Fuzzy Logic

ZIKRIJA AVDAGIC, DUSANKA BOSKOVIC, AIDA DELIC[1]
Faculty of Electrical Engineering
University Sarajevo
Zmaja od Bosne bb, Sarajevo
BOSNIA AND HERZEGOVINA
[zikrija.avdagic, dusanka.boskovic]@etf.unsa.ba, aida.delic@mdh.se

Abstract: - This paper presents application of a fuzzy logic based system to automatically evaluate the maintainability of code. Code evaluation is accomplished by rating its quality provided with bad smells in code as inputs. Straightforward bad smells with existing software metrics tools are selected as inputs: duplicated code, long methods, large classes having a high cyclomatic complexity, or a large number of parameters and temporary fields. Removing these bad smells can result in significant code improvements concerning readability and maintainability. However, the precise definition of attributes like small, long, large or high is not clear, and their identification is rather subjective. Fuzzy logic values are suitable for capturing partial correspondence to attributes and fuzzy rules model have been used to describe the relation between bad smells and code quality. Model supporting the experimental evaluation of the fuzzy based code evaluation is implemented in Java.

Key-Words: - Fuzzy Logic, Reasoning Systems, Refactoring, Bad Smells

1 Introduction

Software maintenance mainly deals with understanding and changing pieces of code. Understanding is often hampered by code which is written without proper documentation and bad programming style, expressed by so-called bad smell patterns.

In the 1960s refactoring was introduced, as a technique for source code improvement without changing its semantics. The focus was on the replacement of goto statements and refining case statements. Structured programming guidelines were used as the basis. The objective was to replace bad smells, and to make code more understandable and maintainable.

Refactoring became more popular in 1999, when Martin Fowler published his book [1]. In [1], Fowler describes bad smells as the pieces of code that make code understanding and maintenance much more complicated than necessary. Code evaluation is needed to identify potential problems and also to reveal bad smells within the code during software development.

In the process of evaluation it is hard to define single quantitative value corresponding precisely to attributes such as small, long, large or high. For example, a quantitative value is defined as indicator of a "large" method. However a problem occurs, when we try to define attributes for predecessor or successor values. The question is whether these values should also be defined as "large".

Fuzzy logic is the answer to this question. Fuzzy logic helps us to deal with uncertainty and imprecision problems. In fuzzy logic values can have partial correspondence to attributes, i.e. single quantitative value belongs to several attributes with certain degree of truth [2]. For example, if we take a closer look at the long method definition, we can say that a method, which has up to 40 lines of code, can be defined as a small method and the one that has more than 60 lines can be defined as large method, and for values 40 to 60 lines of code we can define membership function matching to attribute "large" method.

This paper describes code evaluation approach based on the fuzzy reasoning system. The output of described model is estimation of the quality of the code written for a certain class according to certain input parameters. All input values are represented in a term of membership functions among a defined universe of discourse. The model provides developers with possibility to identify overall quality of a given class.

The Section 2 of this paper describes bad smells and gives a short overview on some examples of bad smells. In the Section 3 the use of fuzzy logic and its application to the code evaluation process is described. Section 4 provides the discussion of the results and Section 5 present the conclusions and future work.

[1] Aida Delic was a student at Faculty of Electrical Engineering during the work on this paper.

2 Bad Smells

Bad smells are considered as the "errors" in the code that make the code syntax harder to understand. Refactoring itself will not bring the full benefits, if we don't know when it is appropriate to apply it. To make it easier for developers to decide if software needs refactoring, Fowler and Beck proposed a list of bad smells. The list made by Fowler and his associates includes 22 possible bad smells that can be found in [1].

Identifying what to refactor is rather subjective, but to achieve quality in object-oriented programming, is to have short and understandable methods, with a clear responsibilities. Hence, input values used for code evaluation by fuzzy reasoning model described in this paper are following four bad smells: duplicated code, long method, long parameter list and temporary field. Additional input used is a metric that precisely measures complexity of code by counting the distinct paths through a method - cyclomatic complexity.

Duplicated code assumes the same code is written in more than one place, so it is important to find a better way to implement the code functionality without repeating the written code. The simplest problem of code duplication is the occurrence of the same expression in two methods that are part of the same class. The more complicated is to detect code duplicated in its meaning and accomplished result.

Long method assumes a method so long that it is difficult to understand, change or extend. As already mentioned, object-oriented programs are best to understand if only short methods are used.

Long parameter list indicates that a method has too many parameters, what makes it difficult to understand, since almost everything is passed as a parameter. Objects do not make it necessary to pass every parameter to a method, only the values really needed for the operation.

Temporary field – a member variable in a class used only occasionally and it is considered redundant to allocate resources for this member. Most often the temporary field is a variable put in the class scope instead of in method scope, thus violating the information hiding principle.

Cyclomatic complexity is integer-based metric appropriately representing method complexity. As the objects of our evaluation are classes, it is important to define class complexity in a term of method complexity. Cyclomatic complexity is a measure of the number of linearly independent paths of a program module as defined in [3]. This measure provides a single ordinal number that can be compared to the complexity of other programs. When the metric is exceeding empirically defined threshold values [4] can be used for detecting the *large class* bad smell.

Bad smells can be considered as a measure of software maintainability; since the code could become less maintainable without knowing that the code is written with pieces that are considered as bad code smells. Presence of any bad code smell can disturb normal code maintainability. It is important to recognize the concept of bad smells as a particular compromise between the vague programming aesthetics and the precise source code metrics.

3 Application of Fuzzy Logic to Code Evaluation

In this chapter our approach for evaluating the maintainability of classes based on bad smells by using fuzzy logic is described. We define a fuzzy model that will rate the maintainability of a class within a scale from very bad to excellent. The inputs in the model are crisp values entered as numerical values for duplicated code and temporary field bad smells or obtained from software metric tools as presented in Table 1.

Table 1. Metrics used

Metrics	Description
LOC	Lines of code
V(G)	McCabe cyclomatic complexity used to quantify method's complexity
NOP	Number of parameters

Fig. 1 shows the model that defines the fuzzy inference process. In the first step, those inputs are fuzzified, which means that they are represented in terms of fuzzy logic. As in any problem defined using fuzzy systems it is important to correctly choose the inputs, the crisp values that are in the first step converted into linguistic variables. It means that each crisp input value is transformed into grades of membership for linguistic terms of fuzzy sets. The membership functions are used to associate a grade to each linguistic term.

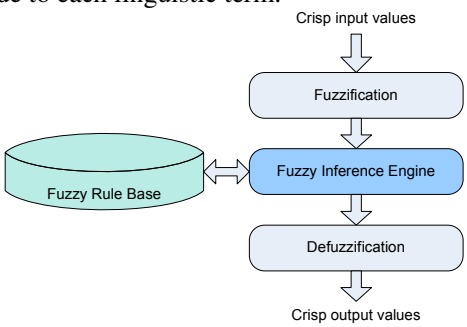

Fig. 1. Fuzzy logic inference system

The next step utilizes a fuzzy rule base and the facts obtained from the fuzzification are combined with the rule base and the fuzzy reasoning process is conducted. The rules whose preconditions satisfy input values are fired, and correlation, inference and defuzzification methods are applied. At the output we get a grade of membership that describes result value. It is important to transform this new fuzzy set into a crisp value, and the process of transformation is called defuzzification. It is not unique operation as different approaches are possible.

As a result we are able to tell how maintainable and "smelly" evaluated methods are and what can make us aware of possible problems in advance. The following part describes given inputs and outputs, an example of a rule base and the used fuzzification and defuzzification algorithms.

3.1 Input Values and Membership Functions

To provide code evaluation as the input values next bad smells are chosen to be inputs, and all are represented as crisp values:

Duplicated code is represented by five membership functions, as a combination of left and right shoulder, triangle and trapezoid shape of function. The definitions of very small, small, medium, large and very large pieces of duplicated code are defined in terms of number of line of codes as illustrated in Fig. 2.

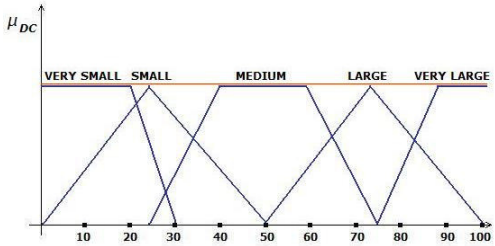

Fig. 2. Duplicated code membership functions

Long method is represented by a left shoulder shape, which defines the smallest possible long method area, a trapezoid, that defines medium membership functions and a right shoulder that is a graphical representation of a large membership function as can be seen in Fig. 3.

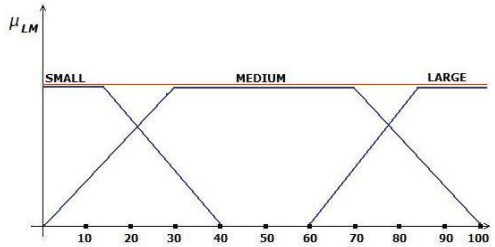

Fig. 3. Long method membership functions

Long parameter list bad smell is represented by five membership functions: very short, short, medium, long and very long which are represented by left and right shoulder, triangle and trapezoid shape of membership functions as shown in Fig. 4.

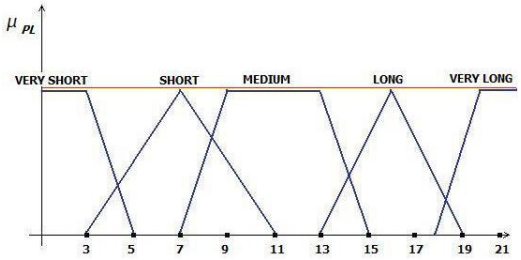

Fig. 4. Long parameter list membership functions

Temporary field is a member variable in a class used only occasionally. It is represented by four membership functions, as a combination of left shoulder, triangle, trapezoid and right shoulder shape of function as graphical representation of a scale on a defined universe of discourse as illustrated in Fig. 5.

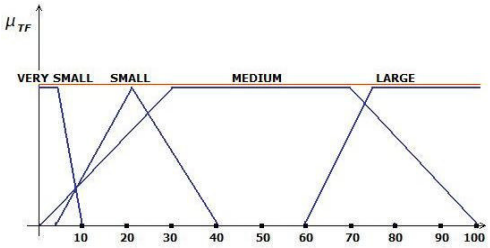

Fig. 5. Temporary filed membership functions

Cyclomatic complexity is usually represented by well defined ranges but in order to get better outcome it is defined with overlapping membership functions. This input is defined by four membership functions: simple code, complex, very complex and untestable code. Fig.6. shows the cyclomatic complexity in terms of membership functions.

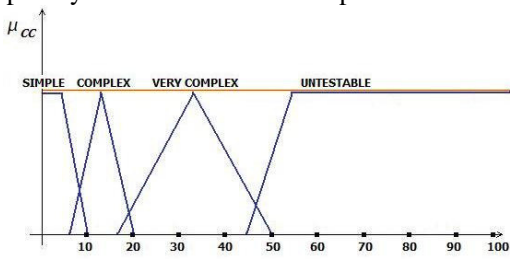

Fig. 6. Cyclomatic complexity membership functions

3.2 Class Quality Definition

The overall quality definition of the code of a class is represented by five membership functions; excellent, very good, good, bad and very bad. Left and right shoulder and triangle are used for the graphical representation as shown in Fig. 7. One

region or a combination of several regions, which are represented here, are the outcome of the inference process.

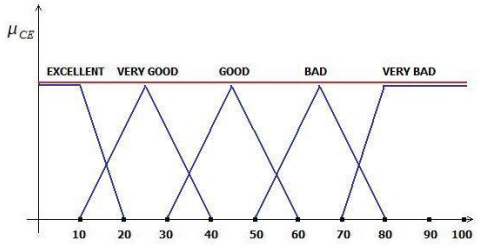

Fig. 7. Code quality in terms of membership functions

3.3 Fuzzy Rule Base Expansion

Fuzzy rules are used to define code quality depending on given input values. Developed application includes about one hundred rules so far, accomplishing input variable values involvement in the inference process and assuring that all output membership functions can be reached as possible result. All rules are written in fuzzy implication form, using the AND operator between the input values. Here is an example of a rule written in the rule base:

IF duplicatedCode IS small AND
longMethod is small AND
cyclomaticComplexity IS simple
AND parameterList IS veryShort
AND temporaryField IS verySmall
THEN codeEvaluation IS excellent.

3.3 Fuzzy Rule Base Expansion

The fuzzy model, explained earlier, has been applied to the class level, because the intention was to develop a model that is able to evaluate methods in a class and to get quality values to be used in further research about evaluating complete programs.

The fuzzy approach requires typically a large number of rules and it is a tedious task to obtain a full set of rules. The larger the number of rules provided by the user, the better the prediction accuracy of the fuzzy model. As the number of rules required increases, the simplicity of using the model decreases since the user has to define a lot of rules to adequately model all quality attributes and their dependencies.

Since every single input is important for the evaluation and makes it more reliable, the increase of inputs leads to the rule base expansion. Code evaluation is performed with five inputs which would result in only one output, and for each input three to five membership functions have been defined resulting in more than thousand rules.

Without any kind of automatic rule generator, this process is time consuming and requires a lot of effort to define the whole rule base. The number of rules therefore was reduced significantly for initial experiments and further on expanded.

3.4 Fuzzification, Inference and Defuzzification

The first step in an inference process is fuzzification. Since the membership function of each input value has been defined, it was easy to fuzzify the given values. Fuzzification means that each value gets a description in terms of a membership function. Let us suppose, for example, that the input value for a variable duplicated code is 10. That means that this value is 100% in a very small area and 30% in a small area after the fuzzification, as we can see in Fig. 8.

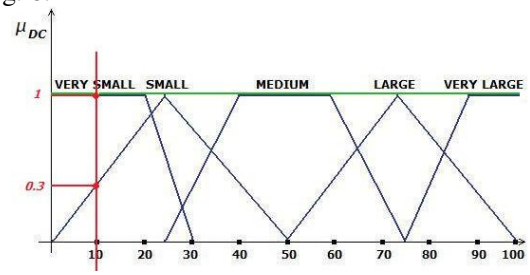

Fig. 8. Fuzzification process

For this input it is possible to write: DC = {vs, s, m, l, vl} = {1, 0.3, 0, 0, 0}. This procedure is applied to all inputs. The next step includes passing through all rules, and forming sets of values for each rule. By using this step, we are able to determine which rule or rules are able to fire and which of them satisfy the input values.

After deterring which rule(s) to fire, the next step requires the application of a correlation method. In this example two correlation methods are implemented:

Product method specifies that the membership value of the consequent fuzzy region is the product of the fuzzy region and the truth of the premise. The effect is that the corresponding fuzzy region is scaled, preserving its shape;

Minimum method specifies that the membership value of the corresponding fuzzy region is the minimum of the fuzzy region and the truth of the premise. The effect is that the corresponding fuzzy region is truncated at the truth of the premise, creating a plateau.

As a result of the correlation process we get one or more corresponding fuzzy regions, which again depend on the number of rules which have been activated at the beginning of the process.

The next step is inferencing. It is important to make sure to use a certain inference method with a

particular correlation method; otherwise, the results wouldn't be reliable. In this example the following inference methods are implemented:

Fuzzy add method specifies that the fuzzy solution set is updated by adding the minimum truth value of the consequent fuzzy region, bounded by 1.0. This method is generally used with a product correlation method;

Minmax method specifies that the fuzzy solution set is updated by using the maximum of the minimum truth value of the corresponding fuzzy set. This method is generally used with the minimum correlation method.

The combination of the minimum correlation method with the minmax inference method is also known as Mamdani fuzzy inference method. Another combination of methods (product and fuzzy add) is named Larsen product inference.

The fuzzy inference process requires one step more, and that step is the defuzzification process. As a result of the Mamdani min implication or the Larsen product, it is possible to get fuzzy result, which is represented in terms of membership functions. For the majority of users, the result, written in these terms, doesn't make any sense, so it was needed to provide a defuzzification process, which includes a conversion of a fuzzy result into a quantifiable value. The output of this process isn't always the same value, since it depends on the method applied. An optimal defuzzification method does not exist; it should be selected based on the problem that we are trying to solve. There are many different methods, but in this model two of them are implemented:

Centroid is calculated as the weighted mean or center of gravity of the output fuzzy region. It is the most commonly used method;

Maxheight specifies that a fuzzy set's crisp value is calculated from the point that has the highest truth value, also known as composite maximum.

4 Implementation

In this Section we present Java application implementing fuzzy-logic based inferencing and demonstrates two examples of code evaluation. There are only few attempts of building current fuzzy logic based systems in Java [5]. However, such systems are either built on an ad-hoc basis without utilizing object oriented features as generality and code reusability, or they are restricted to provide learning environment support.

Fuzzy notions are modeled using several classes namely FuzzyValue, FuzzyVariable, FazzySet, FazzyRule and FuzzyRuleBase supporting fuzzification, inferencing and defuzzification.

Input values are taken from the metrics that automatically calculates bad smells, and can be applied to any individual class as well as to a complete program.

Metric values are exported to an XML file, where the values are sorted by examined bad smells for each method in class. It is important to point out that inputs for duplicated code and temporary field are not supported by the metric tools, and therefore the values are entered by user.

This approach gives us the opportunity to take only the parameters, that are part of our model, and to apply an inference process to them. In case that we want to expand the input set, it is easier just to connect that input(s) with corresponding data in XML file, but in this case we need to be aware of the rule base expansion.

Quality evaluation of a class can be either by using the same input values for all methods in a class, or the provided XML file could contain individual values for different methods. In any case, if we need more reliable results, we can apply the inference process, to each method individually.

If we evaluate only one set of values the result is displayed in a dialog window and includes values used for the evaluation (named badSmells) and the elapsed time of evaluation. If we evaluate the whole class, then the results will be written in a text file. The evaluation value and the elapsed time are given for each evaluated method.

To demonstrate an example of code evaluation let us suppose that the evaluation of one method is based on the following inputs:

duplicated code = 15
long method = 10
cyclomatic complexity = 6
parameter list = 2
temporary field = 22

The user also needs to select the strategies underlying the inference process as described in the last section. Let us suppose that the *product* method is selected for fuzzification; the *fuzzy add* method for inferencing and the *centroid* method for defuzzification.

With the given values only two rules will fire, Rule 21 and Rule 22. In that case we have a situation as shown as in Fig.9. The minimum value in terms of the membership function for given values is 0.5 which is used to scale the output fuzzy result for each rule, since a product method is chosen to be the one for correlation. First rule activates excellent area as fuzzy result and other very good area.

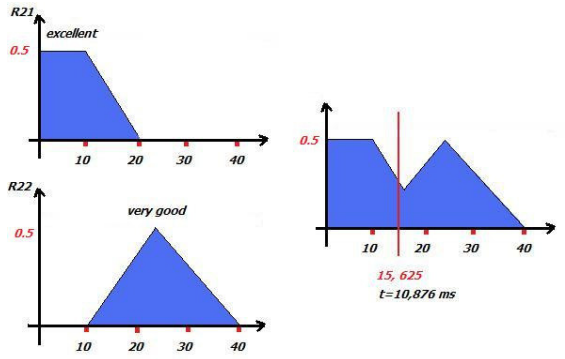

Fig. 9. Code evaluation – Example 1

As a result of the whole process, namely fuzzification, inferencing and defuzzification the resulting output value is 15. 625, which means that the given method is about 35% in excellent area and about 30% in very good area. Calculation has been done in 10.876 ms.

In the second example next methods are applied: the *minimum* method for fuzzification, *minmax* for inferencing and *maxheight* for defuzzification. The same rules fire, since the input values have not been changed.

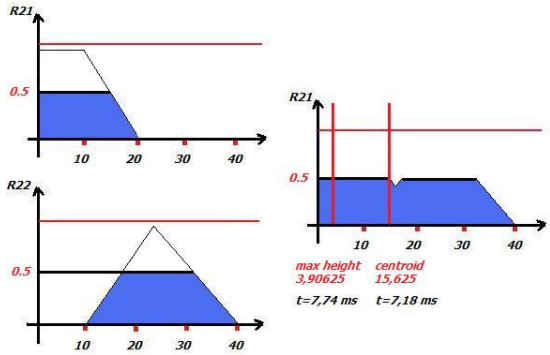

Fig. 10 Code evaluation – Example 2

As it can be seen in Fig. 10, the output result is truncated at the minimum value, at the minimum truth of the premise, creating a plateau. We can also see that if we used *centroid* as defuzzification method, we would get the same result as in the example before; the only difference is in time needed for the calculation. When using *maxheight* method, it will use one randomly chosen value, since there is no specific maximum.

It is not recommended to use the *maxheight* defuzzification method in this combination of methods, since the algorithm is not able to find the maximum and it takes one randomly chosen value from the given set of values.

5 Conclusion

Fuzzy logic is suitable for this area of research because it provides a great range of possible values for each input in terms of membership functions. It is applicable to complex problems such as code evaluation, since it is able to deal with the subjective human analysis involved with software engineering decision making.

The future work goes in two different directions. The first direction is the expansion of an existing model, which would include an automatic evaluation at program level. That means that the existing outputs from the evaluation of each method could be the input to the next level, whereby it would be possible to automatically evaluate how "smelly" a whole program is. The approach can be used as a preliminary step of the pattern based reengineering process presented in [6] to identify smelly classes, which are then searched for concrete smell or anti pattern instances and subsequently improved by refactoring.

On the other hand, automatic rule base generation has to be addressed. As mentioned above, writing all required rules manually to cover all combinations of smells in a given piece of code does not scale in practice.

References:

[1] M. Fowler, Refactoring, *Improving the Design of Existing Code*, Addison-Wesley Publishing house, 2000

[2] Z. Avdagic, *Artificial Intelligence & Fuzzy-Neuro-Genetic*, Grafoart, 2003

[3] T.J. McCabe, A Complexity Measure, *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, 1976, pp. 308-320

[4] M. Mäntylä, C. Lassenius, Subjective evaluation of software evolvability using code smells: An empirical study, *Journal of Empirical Software Engineering*, Vol. 11, No. 3, 2006, pp 395-431

[5] J.P. Bigus, J. Bigus, *Constructing Intelligent Agents Using Java*, John Wiley & Sons, Inc., 2001

[6]M. Meyer, Pattern-based Reengineering of Software Systems, *Proceedings of the 13th Working Conference on Reverse Engineering - WCRE*, 2006, pp 305-306