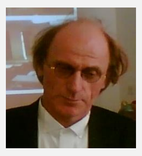
WSEAS International Conference on FUZZY SYSTEMS
Sofia, Bulgaria, May 2-4, 2008

Code Evaluation Using Fuzzy Logic



Zikrija Avdagic Dusanka Boskovic Aida Delic

Faculty of Electrical Engineering
University Sarajevo

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.

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.

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.

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.

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.

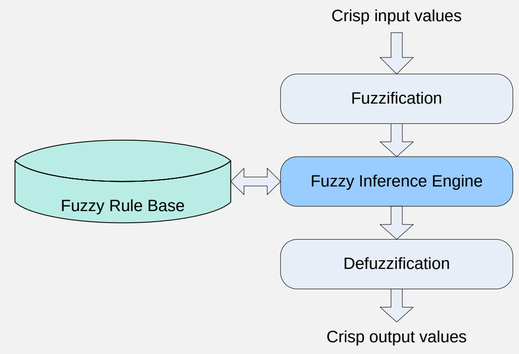
Application of Fuzzy Logic to Code Evaluation

In this chapter our approach for evaluating the maintainability of classes based on bad smells by using fuzzy logic is described.

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 the next Table, and Figure shows the model that defines the fuzzy inference process.

Table

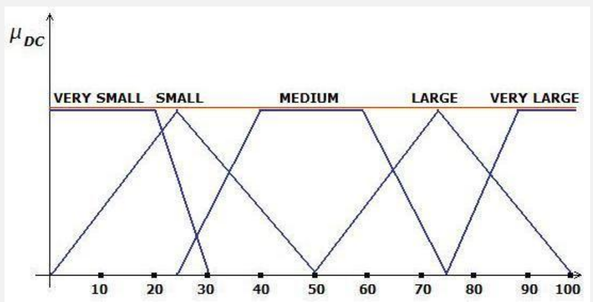
Metrics	Description
LOC	Lines of code
V(G)	McCabe cyclomatic complexity used to quantify method's complexity
NOP	Number of parameters



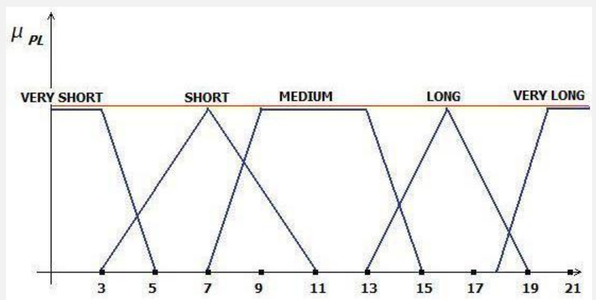
Input Values and Membership Functions

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

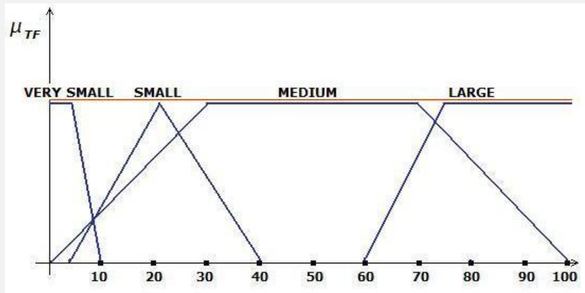
Duplicated code is represented by five membership functions, as a combination of left and right shoulder, triangle and trapezoid shape of function. The definitions of very small, small, medium, large and very large pieces of duplicated code are defined in terms of number of line of codes as illustrated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



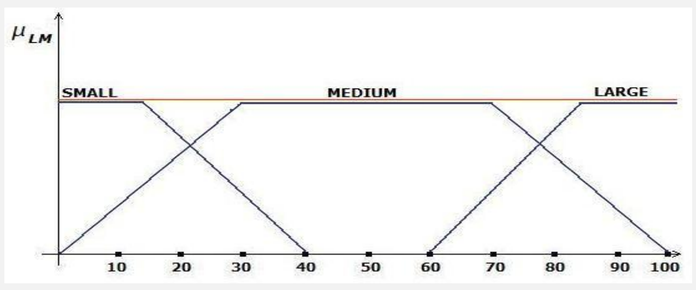
Long parameter *list* bad smell is represented by five membership functions: very short, short, medium, long and very long which are represented by left and right shoulder, triangle and trapezoid shape of membership functions as shown in the next Figure.



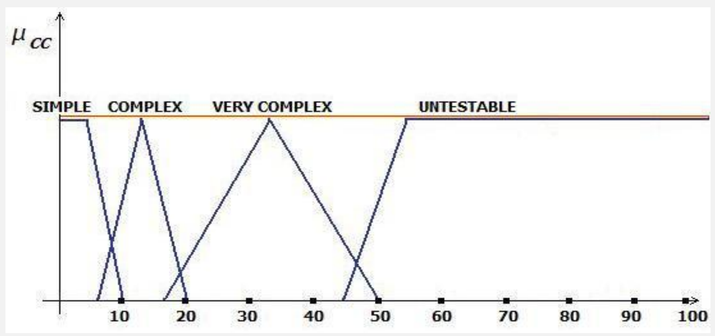
Temporary field is a member variable in a class used only occasionally. It is represented by four membership functions, as a combination of left shoulder, triangle, trapezoid and right shoulder shape of function as graphical representation of a scale on a defined universe of discourse as illustrated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



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 the next Figure.

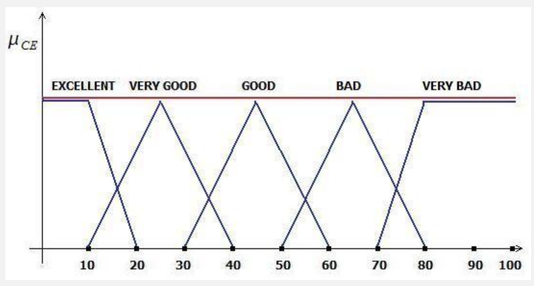


Cyclomatic complexity is usually represented by well defined ranges but in order to get better outcome it is defined with overlapping membership functions. This input is defined by four membership functions: simple code, complex, very complex and untestable code. The next Figure shows the cyclomatic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



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



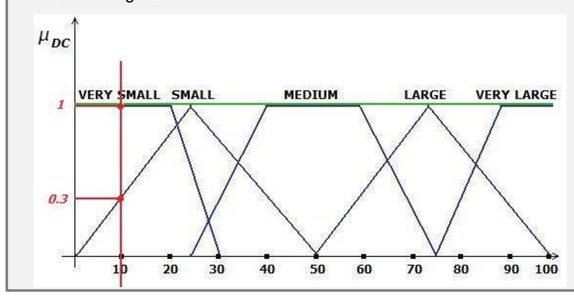
Fuzzy Rule Base Expansion

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.

Fuzzyfication, Inference and Defuzzification

The first step in an inference process is fuzzyfication. Since the membership function of each input value has been defined, it was easy to fuzzyfy the given values. Fuzzyfication 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 fuzzyfication, as we can see in the next Figure.



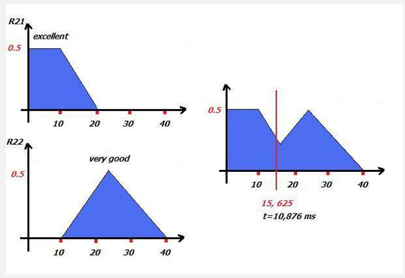
Implementation

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. 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.

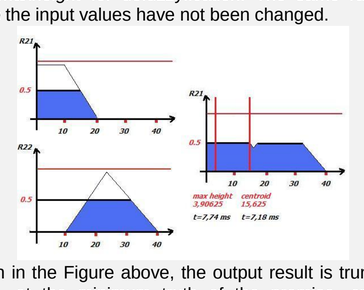
To demonstrate an example of code evaluation let us suppose that the evaluation of one method is based on the following inputs:

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

With the given values only two rules will fire, Rule 21 and Rule 22. In that case we have a situation as shown as on Figure below. 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.



Next methods are applied in the second example: the *minimum* method for fuzzyfication, *minmax* for inferencing and *maxheight* for defuzzyfication. The same rules fire, since the input values have not been changed.



As it can be seen in the Figure above, 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 defuzzyfication 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.

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 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.

Literature

[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