

Improving Object-Oriented Methods by Using Fuzzy Logic

Francesco Marcelloni and Mehmet Aksit

Abstract-- Object-oriented methods create software artifacts through the application of a large number of rules. Rules are typically formulated in two-valued logic. There are a number of problems on how rules are defined and applied in current methods. First, two-valued logic can capture completely neither method developers' intuition nor software engineers' perception of artifact types. Second, artifacts are generally produced based only on a subset of relevant properties. Third, two-valued logic does not model explicitly contextual factors, which can affect the validity of methodological rules. Fourth, no means is supplied to deal with multiple design alternatives and to measure the quality of each alternative during the development process. High loss of information, early elimination of artifacts and process iterations are some of possible fastidious effects. To reduce these problems, this paper proposes fuzzy logic-based methodological rules. Thanks to its ability to cope with uncertainty and imprecision, and to compute with real-world linguistic expressions, fuzzy logic appears to be a natural solution for improving current methods.

Index terms-- Object-oriented methods, fuzzy logic-based reasoning, quantization error, adaptable design models, development environments.

I. INTRODUCTION

Today's software projects are typically characterized by overrunning schedules and budget. In addition, software products in general suffer high maintenance costs. To make software development and maintenance processes manageable, several methods have been proposed [1][2]. Methods create software products (or artifacts) through the application of a number of heuristic rules. Rules are derived from the intuition and experience of method developers and are generally defined by using two-valued logic. For example, in object-oriented methods, a tentative class is identified by applying the following rule: if an entity in a requirement specification is relevant and can exist autonomously in the application domain then select it as a tentative class.

The expressive ability of two-valued logic seems to be quite

poor in capturing completely both method developers' intuition in defining an artifact type and software engineer's perception in identifying an artifact. For instance, when method developers define a tentative class, they intuit that a partial relevant entity should be selected as a partial class, or an entity more relevant than another should be closer to the concept of class than the other. Nevertheless, when expressing their intuition in terms of two-valued logic-based rules, they are forced to quantize relevance into "relevant" and "irrelevant" in such a way as to create a sharp boundary between instances and non-instances of artifact type tentative class. Also, while developing a software system, software engineers can perceive different grades of relevance of an entity in their turn, but they are obliged to quantize their perception to match input values permitted by rules. The two quantization processes result in loss of information because the information about the partial relevance of an entity is not modeled and therefore cannot be considered explicitly in the subsequent phases of the development process.

Further, relevance and autonomy are only a subset of the properties, which characterize artifact type class. Before accepting or rejecting an entity as a class, all the relevant properties should be collected. In the mean time, the possible conflicting design alternatives have to be maintained. On the contrary, current methods impose to each property characterizing an artifact type to perform an abrupt classification.

Finally, the validity of a rule may largely depend on contextual factors such as application domain, changes in user's interest and technological advances. Unless these relevant contextual factors are not modeled explicitly, the applicability of that rule cannot be determined.

In this paper, we firstly intuitively discuss the problems that affect current software development methods. Then, to reduce these problems we propose fuzzy logic based-methodological rules. Fuzzy logic provides a sound framework to define a language, to associate a meaning with each expression of the language and to compute these expressions [3]. Method developers can describe their intuition of an artifact type using their natural language and this intuition can be modeled and maintained along the overall development process. Also, software engineers' perception is not repressed by the necessity to fit restrictive

F. Marcelloni is with the Dipartimento di Ingegneria della Informazione, University of Pisa, Via Diotisalvi, 2-56126, Pisa, Italy, email: france@iet.unipi.it.

M. Aksit is with the TRESE project, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands, email: aksit@cs.utwente.nl

input values imposed by methodological rules. Capturing as much as possible method developers' intuition and consequently software engineers' perception reduces the loss of information and improves the quality of the overall development process. Also, the influence of contextual factors on the validity of methodological rules can be controlled by adapting the meaning associated with each linguistic expression. Finally, fuzzy reasoning allows managing a number of design alternatives and associating a measure with each alternative. Measures prove to be particularly useful in selecting the best alternative in a set of possible conflicting design alternatives.

II. OBJECT-ORIENTED METHODS

Methods are typically described in terms of artifacts, rules for identifying, defining and transforming artifacts, and a notation to represent artifacts. Essential artifact types in object-oriented methods, for instance, are *entities*, *classes*, *attributes*, *operations*, and *aggregation* and *inheritance relations*.

To identify or eliminate an artifact, and relate an artifact to other artifacts, methods provide heuristics. In most methods, heuristics are defined informally using textual forms in a natural language (see, for instance, [2] [4]). Artifact types may have some casual order among each other. The heuristics implicitly express how an artifact type is casually related to other artifact types. For example, to define a tentative class, first an entity must be identified.

To exemplify the problems addressed in this paper, in this section we will introduce some heuristics extracted from [2]. Heuristics are expressed using conditional statements in the form **IF** <antecedent> **THEN** <consequent>. Assume that the following rule *Tentative Class Identification* is used to identify tentative classes:

IF AN ENTITY IN A REQUIREMENT SPECIFICATION IS RELEVANT AND CAN EXIST AUTONOMOUSLY IN THE APPLICATION DOMAIN THEN SELECT IT AS A TENTATIVE CLASS.

Here, *Relevant* and *Autonomously* are the *input values* for the first and second conditions of the rule, respectively. If the conditions are true, then the result of this rule is the classification of an entity in a requirement specification as a tentative class. After identifying tentative classes, redundant classes can be eliminated for instance by using rule *Tentative Class Elimination*:

IF TWO TENTATIVE CLASSES EXPRESS THE SAME INFORMATION THEN DISCARD THE LEAST DESCRIPTIVE ONE.

If an entity is not considered to be sufficiently autonomous, it is discarded as a tentative class, but it could be classified as a tentative attribute by rule *Tentative Attribute Identification*:

IF AN ENTITY IN A REQUIREMENT SPECIFICATION IS RELEVANT AND IT CANNOT EXIST AUTONOMOUSLY THEN IDENTIFY IT AS A

TENTATIVE ATTRIBUTE.

In case tentative classes and attributes are not eliminated or converted, they are selected as classes and attributes, respectively. The application of each rule investigates different properties of an artifact type. The values of these properties can cause the elimination of an artifact (see, for instance, rule *Tentative Class Elimination*), or the conversion of an artifact from an artifact type to another. For example, a class may be converted into an attribute by rule *Class to Attribute Conversion*:

IF A CLASS IS NOT RESPONSIBLE FOR THE REALIZATION OF ANY FUNCTION THEN CONVERT THE CLASS TO AN ATTRIBUTE.

III. PROBLEMS IN CURRENT OBJECT-ORIENTED METHODS

Using the heuristics introduced above, in this section we list and discuss some of the main problems that affect current methods.

A. Lack of expressive ability

Current methodological rules are based on the classic concept of category. A category is represented as a container, with an interior (containing the members), an exterior (containing the nonmembers), and a boundary. The boundary is sharp and does not have any interior structure. Each category is defined by a set of properties shared by the members of the category. Based on values of properties collected so far, rules decide whether a development process element can be classified as a member of the category identified by an artifact type or not. For example, rule *Tentative Class Identification* gathers values of properties Relevance and Autonomy to conclude whether an entity can be selected as a tentative class or not. When method developers define rules, they intuit, for instance, that entities can be partially relevant, and a partially relevant entity should be selected as a partial member of artifact type Tentative Class. Nevertheless, they are forced to quantize their intuition of partial relevance in such a way as to create a sharp boundary between instances and non-instances of an artifact type. There exists a semantic gap between method developers' intuition of an artifact type and actual representation of this intuition by means of two-valued logic-based rules.

Also, classic concept of category does not allow capturing completely software engineers' perception of an artifact. Software engineers, for instance, can perceive different grades of relevance of an entity, but they are required to quantize their perception in order to match input values permitted by rule *Tentative Class Identification*. There exists a semantic gap between the software engineers' perception and the input required by the rule.

As observed in Lakoff [5], some categories appear *graded categories*, that is, they show a fuzzy boundary whose “width” is defined by a linear scale of values between 0 and 1, with 1 at the interior and 0 at the exterior. Artifact types seem to be graded categories rather than classic categories: a development process element such as an entity in a requirement specification can be an instance of an artifact type at a certain degree.

The most evident effect produced by the quantization process caused by the lack of expressive ability of methodological rules is quantization error. To make the concept of quantization error clear, we can refer to the area of digital signal processing. Here, quantization process consists of assigning the amplitudes of a sampled analog signal to a prescribed number of discrete *quantization levels*. Quantization error is defined as the difference between an analog and the corresponding quantized signal sample. If the amplitude distribution of the signal is known, then the root mean square (RMS) value of the quantization error can be computed [6]. To provide the reader with a numerical example, suppose that over a long period of time all possible amplitude values of the signal appear the same number of times. Let us assume that N is the number of quantization levels and A is the maximum value of the signal. Then, the RMS value $\bar{\epsilon}$ of the quantization error is computed as:

$$\bar{\epsilon} = \frac{A}{2(N-1)} \cdot \sqrt{\frac{1}{3}} \quad (1)$$

It is clear from (1) that RMS value of the quantization error decreases with the increase of the number of quantization levels.

In current software development methods, high quantization errors arise from the fact that rules adopt only two quantization levels. Suppose that the relevance of an entity in the requirement specifications can assume values between 0 and 1. If, for example, the software engineer concludes that an entity in a requirement specification is substantially relevant, then substantially relevant has to be approximated to either 0 or 1. This means that there is a loss of information about the relevance of the entity. Here, the quantization error is the difference between the software engineers’ perception and the quantization levels imposed by the methodological rules. Similarly, while defining methodological rules, quantization errors arise from the difference between the method developers’ intuition and the quantization levels adopted in rules.

Assume that if the relevance value is between 0 and 1/2, and 1/2 and 1, it is approximated to 0 and 1, respectively. Suppose that over a long period of time all possible relevance values appear the same number of times. We can compute the RMS value of relevance by applying formula (1). This computation results in 0.289. Under the hypothesis of uniform distribution, therefore, each input to a rule is

affected by 0.289 RMS value of quantization error. Although this example is quite far from reality because inputs generally are not uniformly distributed, it gives however an idea of the possible dramatic effects of quantization process. Further, it can be proved that the result produced by the execution of a rule with multiple conditions in the antecedent part is affected by a quantization error higher than the one computed for each single input [7]. Considering that methodological rules are chained, that is, the result of a rule is the input to another rule, the quantization error can assume intolerable values especially in the last stages of the development process.

We point out that the RMS value of quantization error does not indicate that the resulting object-oriented model will have the same percentage of error. The measurement of error in the resulting object model requires detailed semantic analysis of the requirement specification and the object model. It is however expected that loss of information will eventually cause errors in the resulting object-oriented model.

B. Classification of artifacts based on partial views

Artifact types are characterized by a set of properties. The application of each rule acquires values for some of these properties. For instance, rule *Tentative Class Identification* collects relevance and autonomy values to decide whether an entity is an instance of artifact type Tentative Class. If an entity is not selected as tentative class, then this entity will not be considered by the rules, which apply to tentative classes. Further, if all the rules, which are applicable to an entity in the requirement specification, reject the entity, then the entity is practically discarded. Actually, a software engineer could perceive that the entity is slightly relevant and classify the entity as slight member of artifact type Tentative Class.

The application of subsequent rules and the consequent acquisition of new property values could revalue the entity as class. In current methods, this revaluation process is not possible because a non-relevant entity is discarded and therefore is not considered anymore in the development process. Here, property Relevance, which is only a partial view of artifact types as Class and Attribute, determines the membership of the entity to these artifact types and consequently the possible elimination of the entity. Early elimination reduces the complexity of the development process, but can result in high loss of information and excessive restriction of the design space.

C. Lack of explicit modeling of context

Contextual factors may influence validity of the result of a rule in two ways. Firstly, the input of a rule can be largely context dependent. In rule *Tentative Class Elimination*, for

instance, the elimination of a class is based on the perception of software engineers whether they find a tentative class more descriptive than an equivalent tentative class.

Secondly, validity of a rule may depend on contextual factors such as application domain, changes in user's interest and technological advances. Let us consider the following rule *Inheritance Modification* extracted from [8]:

IN THE CLASS HIERARCHY, IF THE NUMBER OF IMMEDIATE SUBCLASSES SUBORDINATED TO A CLASS IS LARGER THAN 5, THEN THE INHERITANCE HIERARCHY IS COMPLEX.

If this rule concludes that the inheritance hierarchy is complex, then the hierarchy may be modified. The success of this rule heavily depends on the type of application. For example, in graphics applications, it appears natural that many classes inherit directly from class Point. This is because class Point represents a very basic abstraction in a graphic processing system.

D. Inability to manage design alternatives

During the development process an element can be an instance of only one out of conflicting artifacts types, that is, artifact types which are characterized by the same properties, but with different values. For instance, an entity can be an instance of one of the two conflicting artifact types Class and Attribute. If alternative design solutions exist, however, these should be preserved to allow further refinements along the development process. Elimination of alternatives results in loss of information and may consequently degrade the quality of the overall development process. Current methods do not provide a means to deal with multiple design alternatives and to measure the quality of each alternative during the development process. For conflicting artifact types, whenever a property value is collected, current methods have to provide rules to convert artifacts from the one to the other of the conflicting artifact types. Rule *Class to Attribute Conversion* is an example of this conversion.

If software engineers realize that the resulting object model is not satisfactory, there are two possible options: improving the model by applying subsequent rules and/or by iterating the process. The application of subsequent rules may not adequately improve the model because of the loss of information due to quantization errors. The iteration of the process still suffers from the problems discussed in this section. Further, managing iterations remains as a difficult task.

IV. FUZZY LOGIC-BASED METHODS

To reduce the aforementioned problems, a new expressive form rather than two-valued logic has to be investigated. Such a form has to be able to capture as much as possible

the method developers' intuition and software engineers' perception. To this aim, three requirements are strongly demanded: i) capability to manage graded categories, ii) similarity to the natural language typically used by method developers and software engineers, and iii) capability to reason on the linguistic expressions to deduce conclusions and conduct the development process.

Fuzzy logic looks to be the ideal solution. Fuzzy logic relies on the definition of fuzzy set. A fuzzy set S of a universe of discourse U is characterized by a membership function $\mu_S : U \Rightarrow [0,1]$ which associates with each element y of U a number $\mu_S(y)$ in the interval $[0,1]$ which represents the grade of membership of y in S [9]. In fuzzy logic, graded categories are implemented by fuzzy sets.

Let us consider properties Relevance and Autonomy of an Entity. Suppose that values of these properties can be expressed as real numbers in the interval $[0,1]$. In current methods, properties Relevance and Autonomy have to be quantized to, respectively, the values relevant (1) or irrelevant (0), and autonomous (1) or non-autonomous (0). This is because the result of the application of rule *Tentative Class Identification* has to be an abrupt classification of the entity as a member of category Tentative Class. In fuzzy set theory, an entity can be an instance of artifact type Tentative Class at some grade. So, properties Relevance and Autonomy of an Entity can assume each value in the interval $[0,1]$ and these values determine the grade of membership of the entity to category Tentative Class.

To express numerical values for properties such as Relevance and Autonomy may, however, be very hard and the reliability of these values may be quite questionable. To deal with this problem, fuzzy logic introduces the concept of linguistic variable: A linguistic variable is a variable whose values, called *linguistic values*, have the form of phrases or sentences in a natural language [9]. Each linguistic value is associated with a fuzzy set that represents its meaning. Properties Relevance and Autonomy can be represented as linguistic variables. Meaningful linguistic values for these two properties might be: *weakly*, *slightly*, *fairly*, *substantially* and *strongly relevant* for property Relevance and *dependent*, *partially dependent* and *fully autonomous* for property Autonomy. The meaning associated with the linguistic values of Relevance and Autonomy is shown in Fig. 1. Here, the X and Y axes indicate the universe of discourse and the membership values, respectively.

The meaning associated with linguistic values could be defined by using ordinary sets. A characterization based on ordinary sets, however, would require method engineers to fix a value to separate, for instance, the linguistic value *weakly* from *slightly*. But the point of discontinuity, which then would appear in the membership function, would not coincide with the conception that there is a smooth transition. The transition from a set to another set in

domains such as those involved in software engineering appears gradual rather than abrupt. Gradual transition reduces the sensitivity of the membership functions. With respect to ordinary sets, slight variations in software engineers' perception do not result in coarse differences. Fuzzy sets, therefore, allow increasing the robustness of the overall development process.

Introducing a linguistic representation of properties is not however sufficient to solve expressiveness problems. We need a means for reasoning on linguistic values. To this aim, fuzzy logic introduces the concept of fuzzy rules. A fuzzy rule is expressed as: IF X IS A THEN Y IS B , where X and Y are linguistic variables and A and B are linguistic values. The antecedent part of methodological fuzzy rules expresses conditions on linguistic values of properties. The consequent part defines how much a development process element is a member of a category identified by an artifact type. For instance, rule *Tentative Class Identification* could be expressed in linguistic terms as:

IF AN ENTITY IN A REQUIREMENT SPECIFICATION IS RELEVANCE VALUE RELEVANT AND CAN EXIST AUTONOMY VALUE IN THE APPLICATION DOMAIN, THEN IT IS MEMBERSHIP VALUE A TENTATIVE CLASS.

Here, an entity and a tentative class are the artifact types to be reasoned, Relevance and Autonomy are the properties, relevance value and autonomy value indicate the domains of these properties, instance value denotes the domain of membership grades of an entity to artifact type Tentative Class. These membership grades are expressed by means of linguistic values. We consider being reliable values *weakly*, *slightly*, *fairly*, *substantially* and *strongly*. Their meaning is analogous to the one shown in Fig. 1-A. Each combination of relevance and autonomy values of an entity has to be mapped into one of the five membership values. The mapping is based on method developers' intuition of what a tentative class is. Here, the intuition is that the more relevant

and autonomous an entity is, the more the entity is a member of artifact type Tentative Class. This means that the membership value increases with the increase of relevance and autonomy. To represent the mapping, we adopt a tabular representation. The resulting 15 *sub-rules* are shown in Table I. Each element of the table, shown in italics, represents the output value of a sub-rule, that is, the membership value of an entity to artifact type Tentative Class. For example, if the relevance and autonomy values of an entity are respectively *strongly* and *fully autonomous*, then the entity can be considered to be *strongly* a Tentative Class. We selected these output values based on our intuition and knowledge of object-oriented methods. Similarly, methodological rules used in current object-oriented methods can be transformed into more intuitive fuzzy logic-based rules. We believe that this transformation is quite natural. Indeed, the method developer's intuition which each rule is based on is more fuzzy than crisp. Our experimentation in transforming methodological rules of well-known object-oriented methods confirms this our feeling.

To reason on fuzzy rules, fuzzy logic provides fuzzy inference tools. Given a fact and a rule, one of the most known fuzzy inference tools, the *generalized modus ponens*, allows deducing a fuzzy conclusion [10]. A conclusion is expressed as a fuzzy set. If we are interested in a crisp value, we must defuzzify the conclusion by a defuzzification strategy [9]. A defuzzification strategy aims to produce the crisp value which best represents the fuzzy set. At present, the commonly used strategies may be described as the *mean of maxima* and the *center of area*. The crisp value produced by the mean of maxima strategy represents the mean value of the elements, which belong to the fuzzy set characterizing the conclusion with maximum grade. The center of area strategy produces the center of gravity of the fuzzy set characterizing the conclusion.

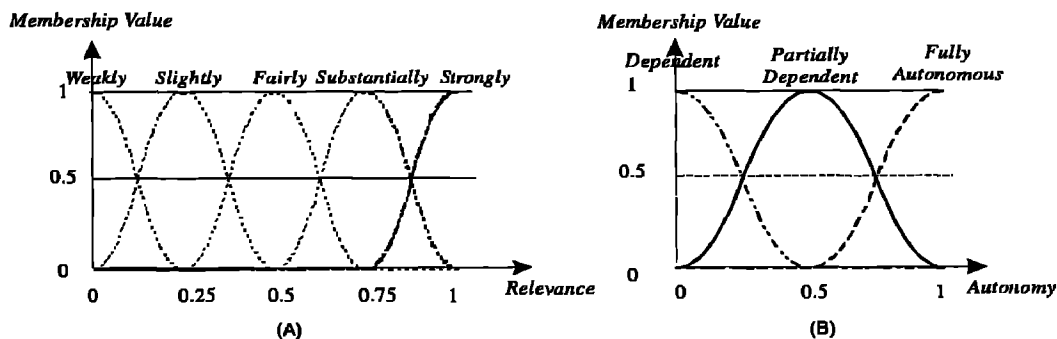


Fig. 1. Meaning of the linguistic values of Relevance (A) and Autonomy (B).

TABLE I.
SUB-RULES OF RULE *TENTATIVE CLASS IDENTIFICATION*.

Entity: Relevance					
TentativeClass: Membership	Weakly	Slightly	Fairly	Substantially	Strongly
Dependent	<i>Weakly</i>	<i>Weakly</i>	<i>Weakly</i>	<i>Weakly</i>	<i>Weakly</i>
Partially Dependent	<i>Weakly</i>	<i>Slightly</i>	<i>Slightly</i>	<i>Fairly</i>	<i>Fairly</i>
Fully Autonomous	<i>Weakly</i>	<i>Slightly</i>	<i>Fairly</i>	<i>Substantially</i>	<i>Strongly</i>
Entity: Autonomy					

V. ADVANTAGES OF USING FUZZY LOGIC-BASED METHODS

A. Increased expressive ability

Fuzzy sets allow method developers to describe artifact types as graded categories. Thus, an entity, which is relevant and partially autonomous, can be classified as a partial class and a partial attribute at the same time. Methodological heuristics can be expressed using linguistic expressions. So, method developers can represent their intuition of an artifact type by means of expressions in natural language. Further, the possibility to express gradation in their intuition permits method developers to highlight peculiarities which cannot be represented by two-valued logic.

Let us consider rule *Class to Attribute Conversion* as defined in Section 2. Here, only the intuition about lack of responsibility is captured. In the reality, method developers also reason on the presence of responsibility. It is evident that the more a class is responsible for at least a function, the more this class is an instance of artifact type Class. This

intuition can easily be modeled in fuzzy logic by fuzzy version of rule *Class to Attribute Conversion*.

IF *P* IS MEMBERSHIP VALUE A CLASS AND *P* IS RESPONSIBILITY VALUE RESPONSIBLE FOR THE REALIZATION OF FUNCTIONS THEN *P* IS MEMBERSHIP VALUE A CLASS AND *P* IS MEMBERSHIP VALUE AN ATTRIBUTE

Table II shows the definition of the rule. Each element of the table represents the output value of Membership for artifact types Class and Attribute. Finally, the use of linguistic variables allows input required by methodological rules to be closer to software engineers' perception.

The increased expressive ability, therefore, decreases the quantization error both during the definition and the use of methodological rules. Consider rule *Tentative Class Identification*. A software engineer can provide both linguistic values and crisp values as input to this rule. For instance, the relevance of an entity might be defined as *Strongly* relevant or 0.95 relevant. Although the reliability of crisp values is quite questionable for some rules, in the following we will examine the quantization error both in case of linguistic values and in case of crisp values.

TABLE II
SUB-RULES OF RULE *CLASS TO ATTRIBUTE CONVERSION*.

Class: Membership					
Class, Attribute: Membership	Weakly	Slightly	Fairly	Substantially	Strongly
Weakly	<i>Weakly</i> <i>Weakly</i>	<i>Weakly</i> <i>Slightly</i>	<i>Weakly</i> <i>Fairly</i>	<i>Weakly</i> <i>Substantially</i>	<i>Weakly</i> <i>Strongly</i>
Slightly	<i>Weakly</i> <i>Weakly</i>	<i>Weakly</i> <i>Slightly</i>	<i>Slightly</i> <i>Fairly</i>	<i>Slightly</i> <i>Fairly</i>	<i>Slightly</i> <i>Substantially</i>
Fairly	<i>Slightly</i> <i>Weakly</i>	<i>Slightly</i> <i>Weakly</i>	<i>Fairly</i> <i>Slightly</i>	<i>Fairly</i> <i>Slightly</i>	<i>Fairly</i> <i>Fairly</i>
Substantially	<i>Fairly</i> <i>Weakly</i>	<i>Fairly</i> <i>Weakly</i>	<i>Substantially</i> <i>Weakly</i>	<i>Substantially</i> <i>Slightly</i>	<i>Substantially</i> <i>Slightly</i>
Strongly	<i>Substantially</i> <i>Weakly</i>	<i>Substantially</i> <i>Weakly</i>	<i>Strongly</i> <i>Weakly</i>	<i>Strongly</i> <i>Weakly</i>	<i>Strongly</i> <i>Weakly</i>
Class: Responsibility					

Consider the membership functions defined in Fig. 1-A. Here, 0.125, 0.375, 0.625 and 0.875 correspond to the X-axis values of the crossing points of the membership functions. If the software engineer selects a linguistic value

such as *slightly*, the actual value can be at any point defined by the membership function *slightly*. To simplify the formulation of the quantization error, however, we make the following assumption: Software engineer's perception is

likely to be restricted to the values between the crossing points. For example, if a software engineer assumes that the actual value is less than 0.125, probably he or she would have used *weakly* instead of *slightly*. Therefore, we assume that software engineers mentally limit the fuzzy set to the range of values at which the membership function associated with the fuzzy set takes higher values than the other membership functions. This means, for example, that *slightly* is considered to be between 0.125 and 0.375. We would like to stress that this assumption is only taken to formulate the quantization error, and during the application of the fuzzy rules, no such restriction is applied.

Assume that the crossing points are the threshold values. The amplitude of a quantization level can be considered as the defuzzified value of the corresponding fuzzy set. Referring to Fig. 1-A, if the mean of maxima defuzzification strategy is used, then the quantization levels will be 0, 0.25, 0.5, 0.75 and 1. Assuming over a long period of time all possible X values appear the same number of times, by using formula (1) of Section 3.1, with $A = 1$ and $N = 5$, the RMS value of the quantization error is computed as 0.072. With respect to the RMS value computed in case of two-valued logic, the RMS value of the quantization error is sensibly decreased.

As the RMS value of the quantization error is inversely proportional to the number of quantization levels (see formula (1)), a method developer might think to increase this number at will. We recall that fuzzy logic-based methodological rules have been introduced to improve the expressive ability of two-valued logic and therefore to capture method developers' intuition as much as possible. The number of quantization levels has to be fixed by method developers' intuition and not by the quantization error. Quantization error can however be used to improve design methods by reorganizing design rules [7].

Instead of linguistic values, the software engineer may provide crisp values as input. A crisp value indicates a single point at the X axis. This means that the input is determined with certainty. Providing a crisp value, however, may be very difficult especially for the early phases of software development process. On the other hand, inputs to some rules can be estimated with an acceptable certainty. For example, consider the fuzzy version of rule *Inheritance Modification* as defined in Section 3.3.

The software engineer can precisely determine the input crisp value of the linguistic variable *Immediate Subclasses*. Having a crisp value as input makes it possible to determine the grade of membership of the crisp value with respect to the fuzzy set associated with each linguistic value. This is similar to knowing the distance between the actual signal and the amplitude of the quantization level. Thus, the quantization error introduced by the rule can be negligible. This is considered as an additional advantage of using fuzzy logic with respect to two-valued logic.

B. Classification of artifacts based on complete information

If fuzzy logic-based methodological rules are applied, development process elements can be instances of artifact types at different grades. Let us consider rule *Tentative Class Identification*. In current methods, if a software engineer considers an entity as slightly relevant in a requirement specification, he/she is likely to quantize slightly relevant to irrelevant and therefore reject the entity as a tentative class. Once an entity has been rejected as a tentative class, it is not considered anymore by the rules that apply to tentative classes. Values of other properties could revalue the entity as tentative class. These properties, however, cannot be investigated since a partial view discarded the entity as tentative class. In fuzzy logic-based methods, rule *Tentative Class Identification* accepts the entity as a slight tentative class. Therefore, rules, which apply to tentative classes, can be applied and the new information can revalue the entity as a tentative class. In fuzzy logic-based methods the decision whether an entity should be rejected or accepted as a class is taken only when all the relevant information is collected. The fuzzy logic-based method can be considered as a *gradual learning process*; a new aspect of the problem being considered is learned after the application of each rule. Clearly, software development through learning creates very adaptable and reusable design models.

C. Explicit modeling of context

Contextual factors can affect inputs of rules and compromise the validity of rules themselves. In fuzzy logic-based approach, the first effect is reduced by increasing the number of possible input values for a given property. Consider rule *Tentative Class Identification*. Selection of an entity as a tentative class is based on the software engineer's perception of relevance. This perception can be different from software engineer to software engineer. In current methods, a little difference in perception can cause contradictory results. For example, assume that the same entity in a requirement specification is considered differently by two software engineers, one as *slightly* and the other as *substantially relevant*. In case of a two level quantization process, it is likely that the first software engineer would reject and the second one would accept the entity as a tentative class. By increasing the number of "quantization levels", the difference between the input values is not amplified.

The effect of contextual factors on the validity of a rule can be reduced by modeling the influence of the context explicitly. The validity of a rule is determined by the validity of its conditions. For instance, let us consider rule *Inheritance Modification* as defined in Section 3.3. The condition of this rule may not be valid for certain kinds of applications.

Our solution to this problem is to adapt the meaning of linguistic values based on the contextual factors. Consider the fuzzy logic version of *Inheritance Modification*:

IN THE CLASS HIERARCHY, IF THE NUMBER OF IMMEDIATE SUBCLASSES SUBORDINATED TO A CLASS IS IMMEDIATE SUBCLASSES, THEN THE INHERITANCE HIERARCHY IS COMPLEXITY.

Table III shows the three intuitive sub-rules derived from this rule. *Low*, *Medium* and *High* are the values in the domains immediate subclasses and complexity.

The validity of this rule depends on the meanings associated with linguistic values *Low*, *Medium* and *High* of linguistic variable *Immediate Subclasses*. Different contexts may associate different meanings with a linguistic value: membership functions have to be adapted to the context. A membership function can be adapted by translating, compressing and dilating it. Translation operation is used to shift the membership function along the X axis. Fig. 2 shows a linear dilation function. Here, initial membership functions associated with linguistic values of *Immediate Subclasses* are dilated with factor 2. This result can be obtained by dilating the universe of discourse. In practice,

fuzzy sets are initially defined on a reference universe of discourse. Then, compression or dilation of fuzzy sets is achieved by reducing or expanding the reference universe of discourse. The compression or dilation function has to be related to contextual factors. In general, it is difficult to formalize this relation by analytical functions and therefore heuristic rules have to be adopted. Since rules defining the effect of contextual factors are typically expressed in terms of linguistic expressions, fuzzy logic is again used for implementing these rules.

D. Capability of managing design alternatives concurrently

Fuzzy logic allows managing a number of design alternatives and associating a measure with each alternative. During the development process, a software engineer can judge an entity as a *substantial* tentative class and a *slight* attribute. Even though artifact types Class and Attribute are conflicting artifact types, fuzzy logic allows maintaining these conflicting alternatives along the overall development process.

TABLE III
SUB-RULES OF RULE INHERITANCE MODIFICATION.

	Class: Immediate Subclasses		
	Low	Medium	High
Inheritance: Complexity	Low	Medium	High

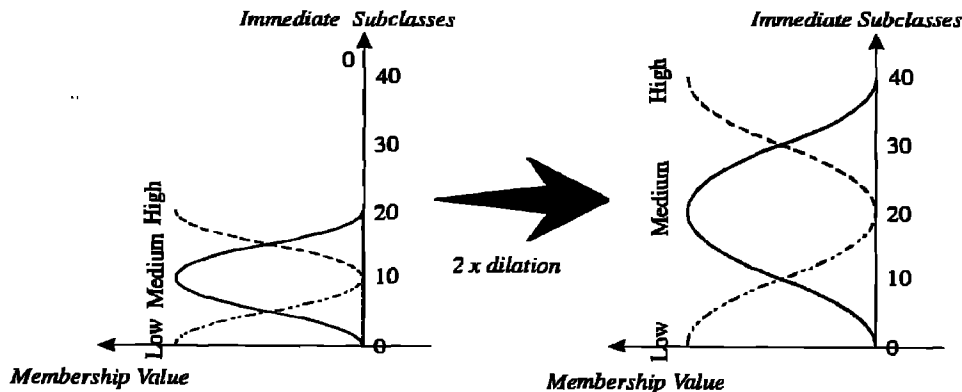


Fig. 2. Adapting to context through dilation.

The memberships of an entity to Class or Attribute are updated whenever a new value of a relevant property is acquired. This reduces the loss of information and practically eliminates the necessity to iterate the development process. When the final product has to be delivered, conflicts have to be solved. The meanings univocally associated with the linguistic expressions provide a valid support to conflict resolution. Each meaning can be

considered as a measure of each alternative. Conflict resolution can be therefore reduced to select the artifact type with the maximum defuzzified value of membership. For instance, if a development process element is 0.8 member of class and 0.3 member of attribute after defuzzification, then the element is implemented as class.

As observed in the previous sections, current methods adopt

classic categories. In such categories, all the objects that have a common set of properties form a category. Intuitively, however, not all the properties may have the same relevance in characterizing a category. Some properties are perceived to be more relevant than others. Current methods cope implicitly with different relevance of properties through conversion rules. Let us consider rules *Tentative Attribute Identification* and *Class to Attribute Conversion*. The first rule identifies an entity which is relevant and non-autonomous in the application domain as a tentative attribute. The second rule converts a class to an attribute if the class is not responsible for any function. Here, the property *Responsibility* is considered more important than *Autonomy* in determining an attribute. Indeed, if an entity is autonomous, but it is not responsible for the realization of any function, it is selected as an attribute. Current methods can only implement the intuition of different relevance of properties by means of conversions from an artifact type to another: they cannot appropriately combine the values of differently relevant properties.

In fuzzy logic-based methods, the concept of graded categories allows introducing degrees of relevance of a property in determining a concept. The relevance of a property can be expressed by a real number between 0 and 1. The membership of an artifact to a type depends, therefore, on both the actual values of the properties and the relevance of these properties in determining the type. In identifying an attribute, the property *Responsibility* has a relevance degree higher than property *Autonomy*.

In the CASE tool we implemented in order to support fuzzy logic based-methods, different relevance of properties is implemented by associating weighting factors with the results of the rules. For instance, fuzzy rule *Tentative Attribute Identification* has a weighting factor less than the one associated with fuzzy rule *Class to Attribute Conversion*. Weighting factors are used in aggregating the several conclusions obtained for the same linguistic variable. Thanks to the weighting factors, the different relevance of properties in determining a type is considered within fuzzy logic reasoning.

Weighting factors can be used also to cope with another issue which is ignored by the current methods: the influence of accurateness and preciseness of the input values on the conclusions inferred by the rules. It is likely that input values of rules used in later phases of the development process are more accurate and precise than the ones of the rules in early phases. This is because information requested in the later phases of the development process is less dependent on the perception and experience of the software engineer and more based on objective evaluations. For instance, rule *Inheritance Modification* requests providing a number as input and this input value is certainly objective.

VI. RELATED WORK

Similarly to our approach, in architectural and mechanical design, the use of fuzzy logic and fuzzy sets has been investigated to cope with changing requirements. In [11], fuzzy models are used to study the transformations of architectural objects during design. In [12], imprecise preliminary design information is modeled and manipulated by fuzzy sets. Further, a method is provided to map design imprecision alternatives (*design variables*) into design performance aspects (*performance variables*).

In [13], fuzzy logic techniques are used to manage inconsistent requirements, which are expressed using fuzzy sets. By using the so-called *satisfaction functions*, alternatives are evaluated and selected. Our approach is, however, different in that we model the heuristic rules as defined in well-known object-oriented methods. Fuzzy values are generated and evaluated by fuzzy logic-based production rules. This provides a better integration with the current methods and CASE environments, which are defined around artifacts, rules and processes. To the best of our knowledge, fuzzy logic-based reasoning has not been applied to software development methods before. On the contrary, the use of fuzzy sets has been investigated in object-oriented models by introducing the notion of fuzzy objects.

Fuzzy objects can have attributes that contain fuzzy sets as values and there may be a *partial inheritance* relation between classes [14]. For example, class *ToyVehicle* can be defined to inherit the property *Cost* with a grade of 0.9 from class *Toy* and with a grade of 0.3 from class *Vehicle*. In class *ToyVehicle* the property *Cost* is initialized to *Low*, whereas in class *Vehicle* it is initialized to *High*. By using the fuzzy union of the fuzzy sets determined by *Low* and *High*, the value of property *Cost* of class *ToyCar* can be obtained. In [15], fuzzy objects are defined to represent complex objects with uncertainty in the attribute values. Further, the operator *merge* is introduced to combine two objects into a single object, provided that the predefined value levels are achieved. In relational databases, such a merge operator makes it possible to retrieve data based on uncertain information.

Our work presents several similarities with the methods proposed in the literature to defer elimination of alternatives in software development. The similarity relates to the wish of coping with and tolerating alternative solutions. The Demeter system, for instance, defers certain design decisions so that software may be adapted to the changing context [16]. For example, functionality of software can be developed independent of the class structure. Only when the executable software has to be generated, the operations are allocated with the appropriate classes. In [17], inconsistent data are automatically marked by means of *pollution markers*. A pollution marker makes the inconsistent data known to procedures or human agents, which are responsible for solving the inconsistency. Further, it protects

the inconsistent data from the action of other procedures sensitive to the inconsistency. In [18], inconsistency handling in multi-perspective specifications is studied by using the ViewPoint framework. Each developer specifies the system by means of a representation language and a development process according to his/her own viewpoint. The consistency rules are expressed in terms of two-valued logic and represent some of the integrity constraints used in controlling and coordinating a set of viewpoints. A meta-language based on linear-time temporal logic is used to specify the actions necessary to cope with inconsistency.

VII. CONCLUSIONS

This paper has highlighted some problems which affect current software development methods. First, two-valued logic typically used to express methodological rules is not able to capture completely both method developers' intuition in defining an artifact type and software engineers' perception in identifying an artifact. Second, current methods classify artifacts based on partial views. Third, two-valued logic does not model explicitly contextual factors. Contextual factors may influence validity of the result of a methodological rule. Fourth, current methods do not provide a means to deal with multiple design alternatives and to measure the quality of each alternative during the development process. Some effects produced by these problems, such as high quantization error, early elimination of artifacts and fastidious iterations of the development process have been analyzed and discussed.

To minimize these problems, fuzzy logic-based methodological rules have been proposed. It has been shown that fuzzy logic can capture the method developers' intuition and the software engineers' perception more appropriately than two-valued logic thanks to its ability to compute with real-word linguistic expressions. This allows reducing quantization error and is useful in adapting design rules with respect to changing contexts.

A small fuzzy logic-based method has been implemented using our experimental CASE environment and tested on example problems [19]. For the method developer, the CASE environment provides tools for defining linguistic variables, values and fuzzy rules. The software engineer basically interacts with tools that represent design rules.

VIII. REFERENCES

- [1] G. Booch, *Object Oriented Design With Applications*, The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.
- [2] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [3] L.A. Zadeh, "Fuzzy Logic = Computing with Words", *IEEE Transactions on Fuzzy Systems*, vol. 4, n. 2, pp. 103-111, 1996.
- [4] A. Riel, *Object Oriented Design Heuristics*, Addison-Wesley, 1996.
- [5] G. Lakoff, *Women, Fire, and Dangerous Things*, The University of Chicago Press, 1987.
- [6] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, 1991.
- [7] M. Aksit and F. Marcelloni, "Reducing Quantization Error and Contextual Bias Problems in Object-Oriented Methods by Applying Fuzzy Logic Techniques", University of Twente, Report, 1997.
- [8] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design", *IEEE Transactions on Software Engineering*, vol. 20, n. 6, pp. 476-492, 1994.
- [9] L.A. Zadeh, "Outline of a New Approach to the Analysis of Complex Systems and Decision Processes", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-3, n. 1, pp. 28-44, 1973.
- [10] B. Lazznerini and F. Marcelloni, "Some Considerations on Input and Output Partitions to Produce Meaningful Conclusions in Fuzzy Inference", *Fuzzy Sets and Systems*, vol. 113, n. 2, pp. 221-235, 2000.
- [11] L. Mudri, P. Perny and P. Chauvel, "An Approach to Design Support using Fuzzy Models of Architectural Objects", in *Artificial Intelligence in Design*, J.S.Gero and F. Sudweeks (eds.), pp. 697-714, 1994.
- [12] W.S. Law and E.K. Antonsson, "Multi-Dimensional Mapping of Design Imprecision", in *Proceedings of the 1996 ASME Design Engineering Technical Conference and Computers in Engineering Conference*, Irvine, California, pp. 18-22, 1996.
- [13] J. Yen and W.A. Tiao, "A Systematic Trade-off Analysis for Conflicting Imprecise Requirement", in *Proc. of the 3rd IEEE Symposium on Requirement Engineering*, pp. 87-96, 1997.
- [14] I. Graham, *Object-Oriented Methods*, 2nd. Edition, Addison-Wesley, 1994.
- [15] A. Yazici, R. George, B.P. Buckles and F.E. Petry, "A survey of conceptual and logical data models for uncertainty", in *Fuzzy Logic for the Management of Uncertainty*, L.A. Zadeh, and J. Kacprzyk, (eds), John Wiley & Sons, Inc., pp. 281-295, 1992.
- [16] K. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method*, PWS Publishing Company, 1996.
- [17] R. Balzer, "Tolerating Inconsistency" in *Proceedings of 13th International Conference on Software Engineering*, Austin, Texas, pp. 158-163, 1991.
- [18] A.C.W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer and B. Nuseibeh, "Inconsistency Handling in Multiperspective Specifications", *IEEE Transactions on Software Engineering*, vol. 20, n. 8, pp. 569-578, 1994.
- [19] M. Aksit and F. Marcelloni, "Deferring elimination of design alternatives in object-oriented methods", *Concurrency - Practice and Experience*, John Wiley & Sons, Inc., to be published.