# Automating Software Development Process Using Fuzzy Logic

Francesco Marcelloni[1] and Mehmet Aksit[2]

[1] Dipartimento di Ingegneria dell'Informazione, University of Pisa, Via Diotisalvi, 2-56122, Pisa, Italy, E-mail: f.marcelloni@iet.unipi.it.
[2] Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands, E-mail: aksit@cs.utwente.nl.

Today's software projects are typically characterized by overrunning schedules and budget. In addition, software products in general suffer from high maintenance costs. To make software development and maintenance processes manageable, a number of methods [7, 12, 14, 17, 31] and Computer-Aided Software Engineering (CASE) environments have been proposed [6, 8, 25, 27]. CASE environments aim to automate the tedious and labor intensive part of the design process, and allow software engineers to concentrate only on the tasks that require human intuition.

Compared to electronic and mechanical Computer Aided Design (CAD) environments, however, current CASE environments provide a much lower-level semantic support for guiding the development process. Indeed, CAD environments offer not only drafting tools for engineering diagrams, but also support for engineering tasks such as computing the weight and strength of materials or the frequency response of an electronic circuit [1, 15]. On the contrary, most successful efforts in software design automation are related to modeling structure and behavior of artifacts. Automating the design process has not been addressed adequately yet. To enhance the level of design automation, not only the artifacts but also design knowledge must be available in a computable format. Since designing is a human-intensive activity, the formalism must adopt designer's language as well. In addition, it must be able to represent and cope with multiple design alternatives and uncertainty.

In this chapter, we aim to highlight how fuzzy logic can be a valid expressive tool to manage the software development process. We characterize a software development method in terms of two major components: artifact types and methodological rules. Classes, attributes, operations, and inheritance and part-of relations are examples of object-oriented artifact types. Each type is characterized by a set of properties whose values determine the membership of an artifact to other types. The relation between the property values and the membership values is defined by the heuristics that are typically expressed informally using textual forms in a natural language. The causal order among artifacts identifies the software process. Especially in the early phases of the development process, property values correspond to software engineer's perceptions [28, 29]. For instance, when defining a tentative class in object-oriented analysis, the software engineer is required to evaluate the relevance of an entity in the requirement specification. To automate the software development process, we need a tool to express perceptions and reason on them. As claimed by L.A. Zadeh, fuzzy logic provides a unique foundation for a computational theory of perceptions: a theory which may have an important bearing on how humans make perception-based rational decisions in an environment of imprecision, uncertainty and partial truth [34]. This is the typical environment which the software engineer has to cope with especially in the first phases of the development process.

We model properties of artifacts as linguistic variables and define the methodological heuristics in terms of fuzzy rules. Artifact types are represented as fuzzy categories. The application of each rule collects property values and infers a value of membership of an artifact to an artifact type. Artifacts can be instances of multiple (possible conflicting) artifact types at different extent. For instance, in object-oriented methods, an entity in the requirement specification can be considered slightly a class and strongly an attribute. These multiple design alternatives can be automatically and concurrently managed along the overall software development. The membership of an artifact to an artifact type can be considered as a measure of the alternative: this measure is updated whenever a new property of the artifact type is investigated. When a product has to be released, appropriate conflict resolution strategies are applied.

Finally, we describe our CASE environment which, on the one hand, allows method engineers to define easily the methodological rules using their natural language and, on the other hand, guides software engineers to develop their software applications. The CASE environment is basically a fuzzy expert system: during the development process, the fuzzy inference engine decides the rules to be investigated and whether instances of artifact types can be created. A repository is associated with each artifact type and contains all the instances of the type. A same artifact can belong to different types with different membership values. Following the sequence of instantiations is possible to trace the story of the artifact evolution. Methodological rules are implemented as objects. This allows storing the evolution of an artifact using the same paradigm with which the artifact will be implemented, so as to improve software maintainability.

## Software Methods

In the last years, the increasing complexity of software systems has given rise to several software methods aimed at helping software engineers develop high quality software [7, 12, 17, 29, 31]. In general, a method can be characterized in terms of two major components: artifacts and rules for identifying, defining and transforming artifacts. Artifacts are the (partial) results of each step of the development process. In object-oriented methods, essential artifacts are, for instance, *entities*, *classes*, *attributes*, *operations*, and *aggregation* and *inheritance relations*. Rules reason on properties of artifacts, denoted *pre-artifacts* in the following, to generate new artifacts, denoted *goal-artifacts*. For instance, to identify a tentative class, several object-oriented methods apply 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*. Here, *entity* is the pre-artifact necessary to generate the goal-artifact *tentative class*, and relevance and autonomy are the two properties significant to conclude whether the entity should be also instance of artifact type *tentative class*. Rules implicitly express how an artifact, the goal-artifact, is casually related to other artifacts, the pre-artifacts. For example, to define a tentative class, first an entity must be identified. A goal-artifact of a rule is generally a pre-artifact of other rules. Once a goal-artifact is instanced, then all rules which have the goal-artifact type as pre-artifact type can be activated. This activation determines the paths of the software development process.

More formally, denote each artifact type as [T, ($P_1$, $D_1$), ($P_2$, $D_2$),..., ($P_n$, $D_n$)], where $T$ is the artifact type name, $P_i$ is a property of $T$ and $D_i$ is the definition domain of $P_i$. An example of an artifact type is [Entity, (Relevance, {*true, false*}), (Autonomy, {*true, false*})]. Here, *true* and *false* are the only two values that Relevance and Autonomy can assume in current methodological rules. An artifact is an instantiation of its type and can be expressed as Name $\leftarrow$ [T, ($P_1$: $V_1$), ($P_2$: $V_2$),..., ($P_n$: $D_n$)], where $T$ is the artifact type, *Name* is the name of the artifact, and $V_i$ is a value defined on domain $D_i$ of property $P_i$. Properties of an artifact can be partially instantiated. For instance, property $P_n$ is not instantiated for artifact *Name.* In the following, when we will refer to a partially instantiated artifact, we will show only the instantiated properties. Values of pre-artifacts' properties are gathered as rules are inferred. For example, when the rule to identify tentative classes is applied, properties Relevance and Autonomy of the entity under consideration are investigated. If both these values are *true*, then the entity is also an instance of artifact type Tentative Class. Rules can be expressed in the form $A \Rightarrow C$, where $A$ and $C$ are the antecedent and the consequent of the rule, respectively, and $\Rightarrow$ is the classical implication operator. Using the notation introduced to represent artifacts, a rule is represented as:

$$N \leftarrow [\mathbf{Pre}_1, (\mathbf{P}_{1,1}: V_{1,1}),,\ldots,(\mathbf{P}_{1,n_1}: V_{1,n_1}),\ldots., \mathbf{Pre}_P, (\mathbf{P}_{P,1}: V_{P,1}),,\ldots,(\mathbf{P}_{P,n_P}: V_{P,n_P})] \Rightarrow$$
$$N \leftarrow [\mathbf{Goal}]$$

Here, $N$ indicates a variable, which is instantiated to the artifact being reasoned on, $Pre_1$, …, $Pre_P$ denote the pre-artifact types and *Goal* indicates the goal-artifact type of the rule. Using this notation, rule *Tentative Class Identification* can be formulated as:

*N* ← ***[Entity**, (**Relevance**: true), (**Autonomy**: true)]* ⇒
*N* ← ***[Tentative Class]***

For simplicity, we will refer to rules with only a proposition in the consequent. To be an instance of an artifact type, each artifact has to satisfy the conditions fixed in the antecedents of rules which have the artifact type in their consequent. For instance, to be an instance of Tentative Class, an entity has to satisfy the conditions imposed by rule *Tentative Class Identification.* Let $C_T(T)$ be the set of conditions defined by the antecedents of rules which have $T$ as goal-artifact type.

Then, the set of instances $x$ of $T$ is the set $\{x : C_T(x)\ \textit{holds}\}$. Conditions $C_T(x)$ are typically expressed as logical expressions of properties of one or more pre-artifact types of $T$. For instance, the set of instances $x$ of artifact type Tentative Class is $\{x : x \leftarrow$ Entity*,* (Relevance*: true*)*,* (Autonomy*: true*)]$\}$. The set of instances of an artifact type is a classical set: an artifact is an instance or is not an instance of the artifact type, but not partially both.

Artifacts can be instances of different artifact types. Consider a generic methodological rule. An artifact is certainly instance of the pre-artifact types in the antecedent of the rule and, if the rule is satisfied, of the goal-artifact type in the consequent. By storing the sequence of instantiations of an artifact, we can trace the evolution of the artifact during the development process. For example, before applying rule *Tentative Class Identification*, an artifact $x$ is only instance of artifact type Entity. If the rule is satisfied, artifact $x$ becomes also instance of artifact type Tentative Class. The software development process is therefore a sequence of instantiations of artifacts to artifact types: the order of the sequence is fixed by the methodological rules. The software engineer follows this sequence and, at each step of the development process, decides whether the artifact should be instantiated to an artifact type by providing the inputs required by the antecedents of the methodological rules.

There exist, however, some artifact types which cannot share the same instance. These artifact types are conflicting. More precisely, we define two artifact types $A$ and $B$ to be conflicting if there exists no artifact which can be an instance of both [24]. Let $\{x_A : C_A(x_A)\ \textit{holds}\}$ and $\{x_B : C_B(x_B)\ \textit{holds}\}$ be the sets of instances of artifact types $A$ and $B$. Then, $A$ and $B$ are conflicting if there exists no artifact $x$ such that both $C_A(x)$ and $C_B(x)$ hold. As an example, consider the following rule *Tentative Attribute Identification* used to identify tentative attributes in some popular object-oriented methods: if an *entity* in a requirement specification is relevant and cannot exist autonomously in the application domain then select it as a *tentative attribute*. This rule can be expressed in our notation as:

*N* ← ***[Entity***, (**Relevance**: *true*), (**Autonomy**: *false*)] ⇒
*N* ← ***[Tentative Attribute]***

Observing the two rules *Tentative Class Identification* and *Tentative Attribute Identification*, we note that the antecedents of the two rules cannot be satisfied at the same time. The value of property Autonomy in rule *Tentative Class Identification* is opposite of the value in rule *Tentative Attribute Identification*. Thus, we can conclude that Tentative Class and Tentative Attribute are two conflicting artifact types. The input value provided by the software engineer for property Autonomy determines to which type the artifact should be instanced.

By the closed-world assumption, if no rule which contains a specific goal-artifact type can be satisfied for an artifact, then it can be concluded that the artifact is not an instance of the goal-artifact type. This implies that all rules which have the goal-artifact type as pre-artifact type will not be activated for the artifact.

In conclusion, a software method can be formalised in terms of sets of instances of artifact types and rules which determine the boundaries of these sets. The development process can be carried out using the modus ponens, one of the most popular classical reasoning tool: Given a fact which matches the antecedent of a rule, then the consequent of the rule can be inferred. In practice, if the antecedent of a rule is true, then also the consequent is true. The consequent of the rule can be considered as a fact in its turn and, thanks to the modus ponens, can activate one or more rules. Typically, the consequent consists of instantiating the artifact to be reasoned on to a goal-artifact type. This instantiation triggers the activation of the rules which have the goal-artifact type as pre-artifact type. The resulting "forward chaining" leads a software engineer to visit all paths relevant to the artifact taken into consideration.

The automation of the software method can be carried out by using a rule-based expert system [18, 26]. We recall that an expert system is a computer application that emulates the problem-solving behaviour of human experts. An expert system consists basically of a knowledge base, an inference engine and a user interface. The knowledge base stores the experience on the domain, i.e., all the information which allows the domain expert to take decisions. The inference engine exploits specific tools, such as the modus ponens, to reason with both the expert knowledge stored in the knowledge base and data provided run-time by the user and specific to the particular problem being solved. The user interface manages on the one hand the interaction between the expert and the computer during the expert system development phase, and on the other hand the interaction between the user and the computer during the consulting phase. In this phase, the user interface guides the user in the insertion of data and shows the conclusions reached during the inference process. In our case, the conclusions of the inference process are the instantiations of artifacts to artifact types. A repository is associated with each artifact type and is delegated to store and control the instances of that artifact type. A same artifact can be maintained in different repositories: analysing the presence of the artifact in the different repositories is possible to trace the evolution of the artifact. In the following, for illustrative purposes, we introduce a very simple method and describe as rules are related with each other.

## An example method

We composed our example method by extracting a few rules from some popular object-oriented methods [28, 29]. The rules were chosen in such a way to highlight the aspects discussed in the previous section.

**R(1)** *Tentative Class Identification:*

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

**R(2)** *Tentative Attribute Identification:*

**IF** AN ENTITY IN A REQUIREMENT SPECIFICATION IS RELEVANT **AND** CANNOT EXIST AUTONOMOUSLY IN THE APPLICATION DOMAIN **THEN** SELECT IT AS A TENTATIVE ATTRIBUTE.

**R(3)** *Class to Attribute Conversion:*

**IF** A TENTATIVE CLASS IS NOT RESPONSIBLE FOR THE REALIZATION OF ANY OPERATION **THEN** CONVERT THE TENTATIVE CLASS TO ATTRIBUTE.

**R(4)** *Attribute to Class Conversion:*

**IF** A TENTATIVE ATTRIBUTE IS RESPONSIBLE FOR THE REALIZATION OF OPERATIONS **THEN** CONVERT THE TENTATIVE ATTRIBUTE TO CLASS.

**R(5)** *Aggregation Identification:*

**IF** CLASS A CONTAINS CLASS B, **THEN** CLASS A AGGREGATES CLASS B.

**R(6)** *Inheritance Identification:*

**IF** CLASS A IS A KIND OF CLASS B, **THEN** CLASS A INHERITS FROM CLASS B.

The casual dependencies between these rules are shown in Fig. 1. Here, using the closed world assumption, if an artifact cannot be instanced to a goal-artifact type using all pertinent rules, then the artifact is classified as non-instance of the artifact type. Thus, for instance, since only rule *Tentative Class Identification* is used to identify tentative classes, an entity that is not relevant and autonomous is classified as non-tentative class. Further, to explicitly show that an entity cannot be a tentative attribute and a tentative class at the same time, we have represented that inputs to rule *Tentative Attribute Identification* are taken from the Non-Tentative Class repository.

The example method takes the requirement specification as input and produces *classes*, *attributes*, and *inheritance* and *aggregation relations* as output. The method has to evaluate various rules before generating a model. For example, to identify an entity in a requirement specification as a class, the corresponding rules must be evaluated in the following order. First, rule *Tentative Class Identification* must accept the entity as instance of tentative class. Second, rule *Class to Attribute Conversion* must reject the tentative class. Alternatively, first rule *Tentative Attribute Identification* must accept the entity as an attribute. Second, rule *Attribute to Class Conversion* must accept the tentative attribute as an instance of class. To identify an entity as an attribute, rule *Tentative Attribute Identification* must accept the entity as a tentative attribute and rule *Attribute to Class Conversion* must reject the tentative attribute. Alternatively, rule *Tentative Class Identification* must accept the entity as a tentative class and rule *Class to Attribute Conversion* must accept the tentative class. Obviously, artifacts that are classified both as non-tentative classes and non-tentative attributes are practically discarded in the development process. Indeed, they cannot infer any further rule.

Analyzing the links between the rules, we can note that artifacts cannot be instances of conflicting artifact types at the same time. Consider, for example, the artifact types Tentative Class and Tentative Attribute. An entity can be an instance of Tentative Attribute only if it has been rejected by rule *Tentative Class Identification*. For conflicting artifact types, whenever a property value is collected, current methods 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.

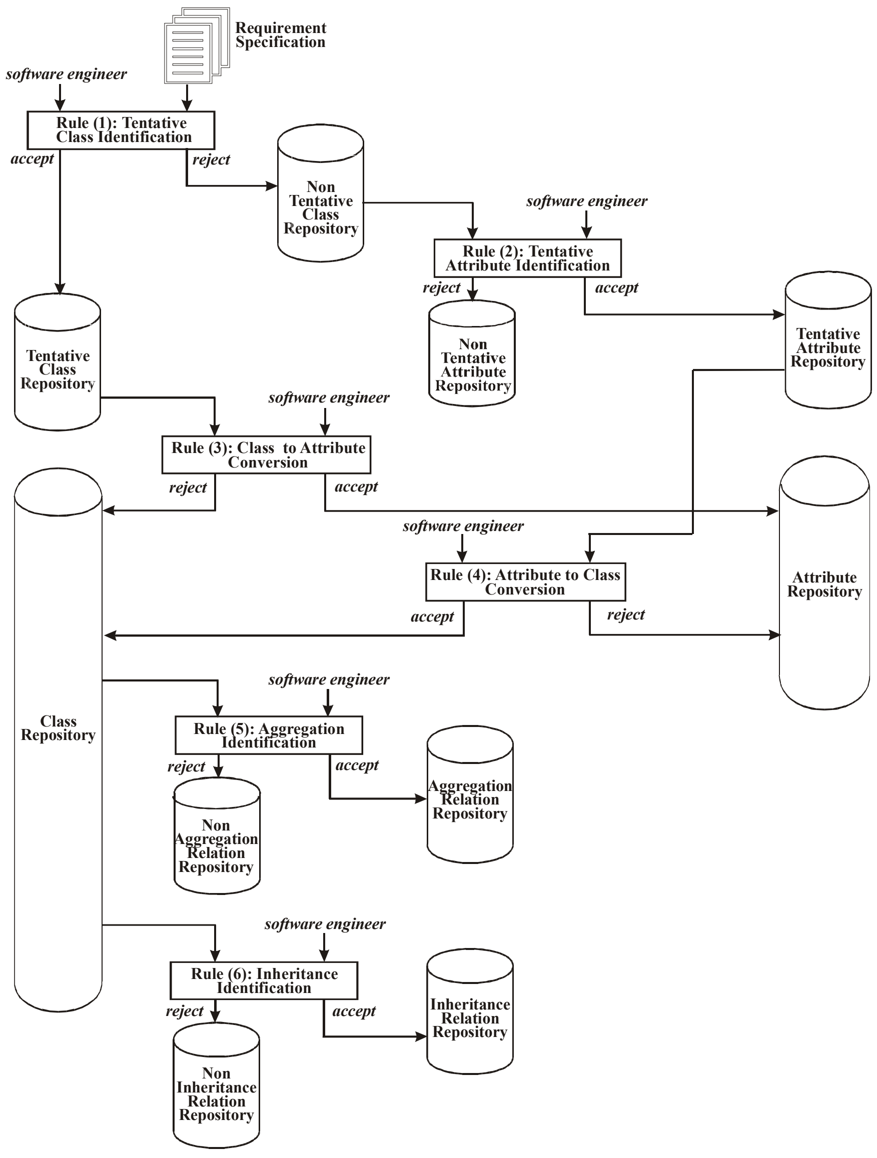
Requirement Specification
software engineer
Rule (1): Tentative Class Identification
accept
reject
Non Tentative Class Repository
software engineer
Rule (2): Tentative Attribute Identification
reject
accept
Non Tentative Attribute Repository
Tentative Attribute Repository
Tentative Class Repository
software engineer
Rule (3): Class to Attribute Conversion
reject
accept
software engineer
Rule (4): Attribute to Class Conversion
accept
reject
Attribute Repository
Class Repository
software engineer
Rule (5): Aggregation Identification
reject
accept
Non Aggregation Relation Repository
Aggregation Relation Repository
software engineer
Rule (6): Inheritance Identification
reject
accept
Inheritance Relation Repository
Non Inheritance Relation Repository

**Fig. 1.** The casual dependencies between the rules of the example method.

Despite reduced number of rules, the example method shown in this section highlights how methodological rules are chained to each other, that is, the output of a rule is input to another rule. This implies that bad decisions taken in the first levels of the rule chain have repercussions on the subsequent levels. For instance, when identifying inheritance or aggregation relations, if entities have been misclassified as classes or as non-classes, inheritance and aggregation relations will not be defined correctly in their turn.

In the later phases of the development process, when the final structure of the software is almost defined, heuristics can be based on more precise and objective inputs than in the first phases. The application of these heuristics can validate the design choices or trigger a reevaluation of these choices.

## Modelling software development process

As formalised in the previous section, current software methods consider the set of instances of an artifact type as a classical set. A classical set behaves as a container, with an interior (containing the members), an exterior (containing the non-members), and a boundary. The boundary is sharp and does not have any interior structure. Each set is defined by a group of properties shared by the members of the set. Based on values of properties collected so far, rules decide whether an instance of one or more pre-artifact types can be classified as a member of the set identified by the goal-artifact type. 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. The use of classical sets and consequently of rules based on two-valued logic appears however to be inadequate to model a human activity such as the software development process [34]. While developing a system, software engineers typically adopt linguistic expressions to assign values to properties of artifacts. These expressions represent perceptions and are qualitative and imprecise by their nature. Nevertheless, current methods force the software engineer to transform these expressions into exact evaluations so as to determine whether an artifact can or cannot be a member of the set of instances of an artifact type. This approach adopted by current methods gives rise to a loss of information which shows itself both in the definition of the methodological rules and in their activation [22, 23]. Indeed, 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, the modeling adopted in current methods of an artifact type as a classical set forces method engineers 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. Thus, 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.

Similarly, when developing a software system, software engineers 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.

The loss of information is therefore not due to the inadequateness of the method engineer's intuition or of the software engineer's perception, but rather to the tools used to model artifact types and methodological rules which are not able to capture the peculiarities of the human reasoning. Thus entities perceived by a software engineer as partially relevant are typically considered to be equivalent to non-relevant entities and therefore are selected neither as tentative classes nor as tentative attributes. Further, if all the rules, which are applicable to an entity in the requirement specification, reject the entity, then the entity is not considered further in the development process. Actually, a software engineer could perceive that the entity is slightly relevant and classify the entity as a 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 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 [2]. A similar problem occurs when dealing with conflicting artifact types. During the development process an artifact can be an instance of only one out of a set of conflicting artifacts types. 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. As current methods do not manage design alternatives concurrently, they do not provide a means to deal with multiple design alternatives and to measure the quality of each alternative during the development process [24]. 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.

In conclusion, we observe that the problems described above originate from the lack of expressive power of the classical sets and classical logic in modeling a human-intensive activity such as the software development process. Software engineers perform the task of developing a software system using their perception and experience. To effectively automate the software development process we need a methodology for reasoning and computing with perceptions rather than measurements. To this aim, we adopt the computational theory of perceptions (CTP) proposed by L.A. Zadeh in [34]. CTP is based on the methodology of computing with words (CW). In CTP, perceptions and query are expressed as propositions in a natural language. Then, propositions are processed by CW-based methods. L.A. Zadeh lists the following four principal rationales for the use of CW [34]:

1. *do not know rationale:* the precision with which values of variables and parameters are known is not sufficient to justify the use of conventional methods of numerical computing;
2. *do not need rationale:* tractability, robustness and low solution costs can be achieved by exploiting a tolerance for imprecision;
3. *cannot solve rationale:* the problem cannot be solved by using conventional methods;
4. *cannot define rationale*: a concept is too complex to be defined in terms of numerical criteria.

A careful analysis of the problem of software development automation shows that all these rationales can be highlighted, thus motivating the use of CW in this framework. We would like to point out that also classical logical systems based on two-valued logic provide means to manage propositions expressed in a natural language. However, CW supplies a much more expressive language for knowledge representation and much more versatile machinery for reasoning and computation [34]. In the following, we introduce some basic concepts of CW. As the methodology of CW is quite wide, we restrict ourselves to the aspects which are relevant to our purposes.

## Computing with Words

Computing with words is a means to express and elaborate human perceptions automatically [33]. In CW, the initial data set is assumed to consist of a set of propositions expressed in a natural language. The meaning of a proposition is determined by a generalized constraint, *X is R*, where *X* is the constrained variable and *R* is a constraining relation. Relation *R* identifies a *granule*, which is a fuzzy set of elements on *X* drawn together by similarity. First, CW translates the initial propositions into constraints, called *antecedent* constraints. Then, appropriate rules of constraint propagation based on fuzzy logic theory allow deriving new constraints, denoted *consequent* constraints, from the antecedent constraints. Finally, the consequent constraints are retranslated into a natural language. In the following, to explain how CW works, we introduce some basic concepts of fuzzy set and fuzzy logic theories. We adopt a terminology closer to the earlier papers which have rooted CW than to recent papers: this choice is motivated by the wish of using terms which are commonly accepted in the literature.

### *Fuzzy Sets and Fuzzy Relations*

A fuzzy set *S* of a universe of discourse *U* is characterized by a membership function which associates with each element *u* of *U* a number $\mu_S(u)$ in the interval [0,1] which represents the grade of membership of *u* to *S* [32]. A number of operations are defined on fuzzy sets. Given two fuzzy sets *A* and *B* in a universe of discourse *U*, some basic operations are the following:

- Complement $\neg A = \int_U (1-\mu_A(u))/u$
- Intersection $A \cap B = \int_U T(\mu_A(u),\mu_B(u))/u$
- Union $A \cup B = \int_U T^*(\mu_A(u),\mu_B(u))/u$

where the integral sign $\int_U \mu(u)/u$ stands for the union of the points $u$ at which $\mu(u)$ is positive, and $T$ and $T^*$ identify a triangular norm and the corresponding conorm, respectively. The definitions of some popular triangular norms and corresponding conorms are given in Table 1.

**Table 1.** Some popular triangular norms and corresponding conorms

| T | T(a,b) | $T^*$(a,b) |
|---|---|---|
| Minimum | min(a,b) | max(a,b) |
| Product | ab | a+b-ab |
| Bounded product | max(0,a+b-1) | min(1,a+b) |
| Drastic product | $\begin{cases}\min(a,b) & \text{if } \max(a,b)=1\\ 0, & \text{otherwise}\end{cases}$ | $\begin{cases}\max(a,b) & \text{if } \min(a,b)=0\\ 1, & \text{otherwise}\end{cases}$ |

A fuzzy relation $R(x_1, ..., x_N)$ is a fuzzy subset of the Cartesian product $U_1 \times \ldots \times U_N$, where $U_1 \times \ldots \times U_N$ is the collection of ordered tuples $u_1,\ldots,u_N$, with $u_1 \in U_1$, ..., $u_N \in U_N$. $R$ is characterized by a multivariate membership function $\mu_R(u_1,\ldots,u_N)$ and is expressed as

$$\int_{U_1\times\ldots\times U_N} \mu_R(u_1,\ldots,u_N)/u_1\ldots u_N \,. \tag{1}$$

Relations can be composed. Let $R$ and $S$ be fuzzy relations in $U \times V$ and $V \times W$, respectively. The *composition* of $R$ and $S$ is a fuzzy relation denoted by $R \circ S = \int_{U\times W} sup_{v\in V}\, T(\mu_R(u,v),\mu_S(v,w))/(u,w)$. $R$ or $S$ can be unary relations. The *sup-T* operator is called *composition operator*.

### Linguistic Variables

The basic idea in CW is that words define fuzzy constraints on the values of variables. Let us consider the variable Temperature. The propositions *Temperature is low*, *Temperature is medium* and *Temperature is high* identify sets, probably imprecise and vague, of possible values of temperature. The words *low*, *medium* and *high* express fuzzy constraints on these possible values. CW represents these constraints by fuzzy sets defined on the definition domain of Temperature. Let Temperature be defined on the scale of degrees centigrade between 0 and 40. The words *low*, *medium* and *high* can be defined as labels of the fuzzy sets shown in Fig. 2. Each degree of the scale belongs to the fuzzy set associated with a linguistic value at a different grade. For instance, the temperature 29 °C belongs to *medium* and *high* with membership value 0.5 and to *low* with membership value 0. Variable Temperature is known as linguistic variable in the literature. More precisely, a *linguistic variable* defined on universe *U* is a variable whose values, called *linguistic values*, are words of a natural or artificial language [16, 32]. The meaning of a linguistic value *v* is the fuzzy set *M(v)* of universe *U* for which *v* serves as label. The three fuzzy sets shown in Fig. 2 express the meaning of *low*, *medium* and *high* temperature.

Syntactically, a linguistic value is a composition of the following atomic terms:

1. *primary terms*, which are labels of specified fuzzy sets in the universe of discourse. For instance, *low*, *medium* and *high* for the linguistic variable Temperature;
2. *modifiers,* such as *more or less*, *very*, *minus*, *plus*, which modify the meaning of the atomic term which they are applied to;
3. negation *not* and connectives *and* and *or*;
4. *markers* such as *parentheses*.

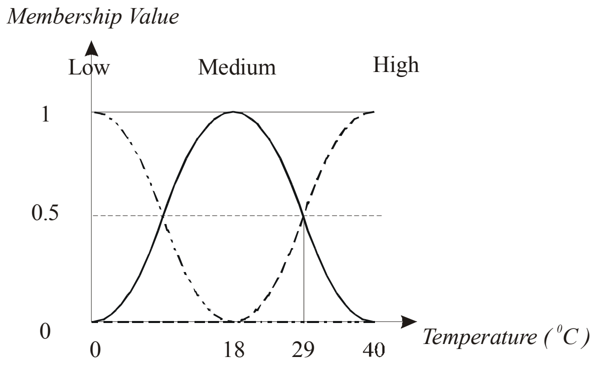

**Fig. 2.** Linguistic variable Temperature.

All possible values of a linguistic variable can be generated by a context-free grammar *G=(T, N, P)*, where T and N are the terminal and non-terminal symbols, respectively, and P is the production system. The terminal symbols are the atomic terms. The meaning associated with each possible linguistic value is determined by a semantic rule *R*, which maps each linguistic expression into an operation on fuzzy sets. For instance, modifiers correspond to unary operations on the unit interval [0,1], negation *not* complements the corresponding fuzzy set, connectives *and* and *or* are defined as the intersection and the union between fuzzy sets, respectively [16]. The markers change the normal sequence of the operations.

It follows that a linguistic variable is characterized by a quintuple *(x, TN(x), U, G, R)* where *x* is the name of the variable, *TN(x)* is the term set of *x*, that is, the union of terminal and non terminal symbols of *x* with each value being a fuzzy set defined on universe *U*, *G* is the context free grammar for generating the symbols of *x*, and *R* is the semantic rule [16]. The definition of *G* and *R* is generally shared among all the linguistic variables except for the primary terms and their meanings. In general, therefore, a linguistic variable is completely characterized by defining the universe, the primary terms and their associated meanings.

## *Fuzzy Logic*

A proposition in a natural language is viewed as a network of fuzzy constraints. Constraints can have a variety of forms. In particular, a constraint may be conditional, that is, expressed by a fuzzy rule in the form **IF** $X_1$ is $A_1$ **AND** .... **AND** $X_N$ is $A_N$ **THEN** Y is B, or for short $I(A_1\wedge \ldots \wedge A_N, B)$, where $X_i$, with $i$=1..*N*, and *Y* are linguistic variables defined on the universes $U_i$ and *V*, respectively, $A_i$ and *B* are linguistic values of $X_i$ and *V*, respectively, and *I* is a fuzzy implication operator. In fuzzy logic, each logical operator is defined in terms of fuzzy set operations. Thus, the connective **AND** and the fuzzy implication are implemented as fuzzy relations. For the connective **AND**, $A_1\wedge \ldots \wedge A_N = \int_{U_1\times\ldots\times U_N} T(\mu_{A_1},\ldots,\mu_{A_N})/u_1\ldots u_N$, with *T* a triangular norm and $\mu_{A_i}$ the membership function associated with the primary term $A_i$. A *fuzzy implication* is defined for all $x \in X$ and $v \in V$ by $I(A,B) = \int_{X\times V} F(\mu_A(x), \mu_B(v))/(x,v)$, where *F* may be each function from $[0,1]\times[0,1]$ to $[0,1]$ that satisfies the boundary conditions *F*(0,0)=*F*(0,1)=*F*(1,1)=1 and *F*(1,0)=0. Several families of fuzzy implication operators have been proposed in the literature. Comparative studies can be found in [16].

Given a fuzzy rule **IF** $X_1$ is $A_1$ **AND** .... **AND** $X_N$ is $A_N$ **THEN** Y is B and a fact $X_1$ is $\hat{A}_1$ **AND** .... **AND** $X_N$ is $\hat{A}_N$, the inference mechanism used to infer a conclusion *B'* is normally implemented by a generalization of the modus ponens, called generalized modus ponens or compositional rule of inference. Conclusion *B'* is computed as $B' = (A'_1 \wedge \ldots \wedge A'_N) \circ I(A_1 \wedge \ldots \wedge A_N, B)$, where $\circ$ denotes the composition operator and *I* a fuzzy implication operator [16]. The conclusion *B'* is therefore obtained by first computing the fuzzy sets corresponding to the fact and to the rules, and then composing these fuzzy sets by the composition operator. It follows that in fuzzy logic a reasoning tool like the generalized modus ponens is implemented as a sequence of operations on fuzzy sets. Notice that the generalized modus ponens allows inferring conclusions also if the fact corresponds only approximately to that expected in the antecedent of the rule.

Typically, the causal relationships between the variables are expressed by a set of rules: the conclusion inferred from all the rules will be obtained by aggregating the conclusions inferred by the single rules. Aggregation is generally implemented as a fuzzy intersection or union. The choice of the type of aggregation operation depends on the type of fuzzy implication and composition operators. In general, the criterion adopted in the choice is the fundamental requirement for fuzzy reasoning, i.e., given a fact that matches the antecedent of a rule, the conclusion has to match the consequent of that rule. A detailed analysis on the relationships among types of aggregation, implication and composition operators, and partitions of the input and output spaces for satisfying the fundamental requirement for fuzzy reasoning can be found in [19, 30].

A conclusion inferred by the generalised modus ponens is in general a fuzzy set, which has to be approximated to one of the possible linguistic values defined for the linguistic variable in the consequents of the rules. Alternatively, if we are interested in a crisp value, we can defuzzify the conclusion by a defuzzification strategy [16]. A defuzzification strategy is aimed at producing the crisp value that best represents the linguistic value. At present, the most commonly used strategies are the *mean of maxima* and the *centre 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 characterising the conclusion with maximum grade. The centre of area strategy produces the centre of gravity of the fuzzy set characterising the conclusion.

## Fuzzy artifacts

The methodology of computing with words allows reproducing more faithfully than two-valued logic the method engineers' intuition and software engineers' perception. Method engineers can define properties of artifact types as linguistic variables and investigate a set of linguistic values which are meaningful and can be easily used by software engineers. The increased expressive power allows method engineers to capture more closely the relation between values of properties of pre-artifact types and membership to the goal-artifact type of a rule. Thus, for instance, the method engineer can express that a partial relevance and a partial autonomy of an entity originate an artifact which only partially is an instance of artifact type Tentative Class. On the other hand, software engineers find a complete correspondence between their perception and the inputs requested from the methodological rules. Partial membership to an artifact type is realized by implementing artifact types, denoted *fuzzy artifact types*, as fuzzy sets. Thus, artifacts can be instances of an artifact type at different grades.

A fuzzy artifact type is represented as [T, (Membership, $D_M$), ($P_1$, $D_1$), ($P_2$, $D_2$),..., ($P_n$, $D_n$)], where *T* is the artifact type name, property *Membership* represents the membership grade, $D_M$ is the definition domain of *Membership*, $P_i$ is a property of *T* and $D_i$ is the definition domain of $P_i$. With respect to the definition proposed in case of classical artifact types, two basic differences can be highlighted. First, the presence of property *Membership*. This property is typically defined on the real interval [0,1], where the extremes 0 and 1 mean, respectively, that the artifact is not and is an instance of the artifact type. In the case of current methods, membership can assume only the two values 0 and 1. If an instance of an artifact type is created, then this instance has an implicit membership grade equal to 1; otherwise, the instance is not created. In the case of fuzzy artifacts, the membership to a fuzzy artifact type can be between 0 and 1, and therefore each instance has to be characterised by a value of membership.

Second, the definition of $D_i$ is more complex than that used in classical artifact types. Here, method engineers have to define all the possible linguistic values which can be used to assign a value to the property. Further, they have to associate a unique meaning with each of these linguistic values. The meaning has to be evident to the software engineers in such a way that they can select the closest linguistic value to their perception. To this aim, the method engineer identifies a universe of discourse for each property and defines a set of primary terms on this universe. As we suppose that the context free grammar *G* and the semantic rule *R* are equal for all the linguistic variables, the definition of the primary terms determines all the possible linguistic values and their associated meanings.

To identify the primary terms for each property used in the example method shown in the next section, we adopted the following procedure: we selected a pool of software engineers and asked them to define the linguistic variables. Then, we stimulated a revision process within the pool aimed at reaching an agreement. In general, we observed that software engineers tend to partition uniformly the universe of discourse and to use smooth membership functions to describe fuzzy sets. Concerning Relevance and Autonomy of an entity, for instance, the pool concluded that property Relevance can be expressed as *weakly*, *slightly*, *fairly*, *substantially* and *strongly relevant*, and property Autonomy as *dependent*, *partially dependent* and *autonomous*. Figures 3 and 4 show the meaning associated with the primary terms of Relevance and Autonomy. Here, standard piecewise quadratic functions are used to define membership functions. We defined Relevance and Autonomy on the real interval [0,1]. Actually, this choice was made only for convenience. The aim of the definition is to show the relation between the linguistic values and this relation is independent of the universe of discourse. The definition of the universe of discourse for a linguistic variable is important when the software engineer can input numerical values for that property. In the case of Relevance and Autonomy, a numerical value would be very questionable and probably devoid of reliability and meaning. Thus, the only important aspect in the definition of these linguistic variables is to show the reciprocal relations between primary terms.

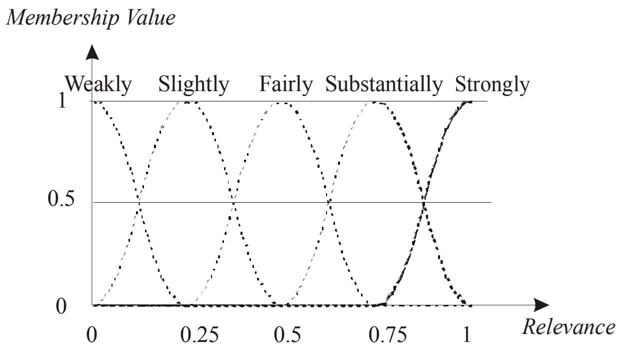

**Fig. 3.** Definition of linguistic variable Relevance.

A fuzzy artifact is modelled as Name ← [T, (Membership: $V_M$), ($P_1$: $V_1$), ($P_2$: $V_2$),..., ($P_n$: $V_n$)]. The membership value $V_M$ depends on the truth-values of antecedents of rules, which have *T* as goal-artifact type. For instance, the value of membership of *Nam*e to artifact type Tentative Class depends on the truth-value of the antecedent of rule *Tentative Class Identification*. This truth-value is affected by the values of Relevance and Autonomy in its turn.

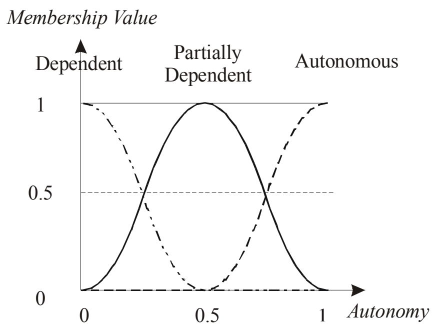

**Fig. 4.** Definition of linguistic variable Autonomy.

## Fuzzy Example Method

The enriched expressive power achieved by implementing properties of artifacts as linguistic variables allows method engineers to formulate the methodological rules emulating their linguistic ability. The relation between the properties of pre-artifacts and the membership to the set of instances of the goal-artifact type can now be expressed in detail using the linguistic values in the definition domains of the properties. Consider, for example, rule *Tentative Class Identification*. This rule should express the following method engineers' intuition: the more an entity is autonomous and relevant, the more the entity is an instance of artifact type tentative class. To represent rules conveniently, property Membership is represented as a linguistic variable too. We realized that meaningful primary terms for Membership are: *weakly*, *slightly*, *fairly*, *substantially* and *strongly*. The definition of these linguistic values is the same as in Fig. 3. The schema of fuzzy rule *Tentative Class Identification* is:

**F(1)** *Tentative Class Identification:*

**IF** AN ENTITY IN A REQUIREMENT SPECIFICATION IS <u>RELEVANCE VALUE</u> RELEVANT **AND** CAN EXIST <u>AUTONOMY VALUE</u> AUTONOMOUS IN THE APPLICATION DOMAIN **THEN** IT IS <u>MEMBERSHIP VALUE</u> A TENTATIVE CLASS**.**

Here, <u>relevance value</u>, <u>autonomy value</u> and <u>membership value</u> indicate the domains of properties Relevance, Autonomy and Membership, respectively. Each combination of relevance and autonomy values of an entity has to be mapped into one of the five membership values to artifact type Tentative Class. The resulting 15 *sub-rules* are shown in Table 2. Each element of the table, shown in italics, represents the consequent part of the sub-rule. For example, if the relevance and autonomy values are respectively *strongly* and *autonomous*, then membership value to Tentative Class is *strongly*. Adopting the same notation used to codify methodological rules expressed in two-valued logic, this sub-rule can also be represented as:

> *P* $\leftarrow$ ***[Entity**, (**Membership**: 1) (**Relevance**: strongly),(**Autonomy**: strongly)]* $\Rightarrow^f$
> *P* $\leftarrow$ ***[Tentative Class**, (Membership: strongly)]*

Here, P indicates a variable, which is instantiated to the artifact being reasoned on, and $\Rightarrow^f$ represents a fuzzy implication operator. The value of Membership to Entity is 1 because Entity is considered as the starting artifact type.

The choice of the values shown in Table 2 derives from the intuition which the rule is based on. We can suppose that the membership value to Tentative Class can be computed as product of the relevance and autonomy values. The sub-rules shown in Table 2 have been generated based on this observation and on the meaning associated with each linguistic value. For instance, when an entity is weakly relevant and dependent on another entity, the entity has weakly the characteristics to be identified as a tentative class. With the increase of relevance and autonomy, the entity is more and more characterized as a tentative class.

**Table 2.** Sub-rules of fuzzy rule *Tentative Class Identification*.

| P ← [Tentative Class, (Membership: | P ← [Entity, (Relevance: | | | | |
|---|---|---|---|---|---|
| | weakly | slightly | fairly | substantially | strongly |
| Dependent | *weakly* | *weakly* | *weakly* | *weakly* | *slightly* |
| Partially Dependent | *weakly* | *slightly* | *slightly* | *fairly* | *fairly* |
| Autonomous | *weakly* | *slightly* | *fairly* | *substantially* | *strongly* |
| P ← [Entity, (Autonomy: | | | | | |

Similarly, the fuzzy version of rule *Tentative Attribute Identification* is as follows*:*

**F(2)** *Tentative Class Identification:*

> **IF** AN ENTITY IN A REQUIREMENT SPECIFICATION IS <u>RELEVANCE VALUE</u> RELEVANT **AND** CAN EXIST <u>AUTONOMY VALUE</u> AUTONOMOUS IN THE APPLICATION DOMAIN, **THEN** IT IS <u>MEMBERSHIP VALUE</u> A TENTATIVE ATTRIBUTE.

The sub-rules corresponding to the rule are shown in Table 3 and are derived from the following intuition of artifact type Tentative Attribute: the more an entity is relevant and its existence is dependent on another entity, the more the entity is a tentative attribute.

As discussed in the brief review of fuzzy logic, if an appropriate combination of fuzzy implication, composition and aggregation operators, and suitable partitions of input and output spaces are chosen, the conclusion inferred from a set of fuzzy rules corresponds to the consequent of the rule whose antecedent matches the fact. We would like to point out that the partitions shown in Figs. 3 and 4 are suitable partitions. If a software engineer inputs primary terms, the conclusion inferred from the rules is a primary term in its turn. For instance, if the software engineer decides that an entity is strongly relevant and autonomous, the entity is strongly a tentative class. This primary term may fire a chained rule (for instance, one of the conversion rules) and produce primary terms in its turn.

**Table 3.** Sub-rules of fuzzy rule *Tentative Attribute Identification*.

| P ← [Tentative Attribute, (Membership: | P ← [Entity, (Relevance: | | | | |
|---|---|---|---|---|---|
| | weakly | slightly | fairly | substantially | strongly |
| Dependent | *weakly* | *slightly* | *fairly* | *substantially* | *strongly* |
| Partially Dependent | *weakly* | *slightly* | *slightly* | *fairly* | *fairly* |
| Autonomous | *weakly* | *weakly* | *weakly* | *weakly* | *slightly* |
| P ← [Entity, (Autonomy: | | | | | |

Let us now consider rules *Class to Attribute Conversion* and *Attribute to Class Conversion*. These rules reason on the property of an artifact of being responsible for a set of operations. Both rules implement only a part of the intuition derivable from this property. Let us consider rule *Class to Attribute Conversion*. This rule captures the following intuition: The less a class is responsible for a set of operations, the more it is an attribute. There exists, however, another part of intuition concerning the property: the more an artifact is a tentative attribute and the less is responsible for operations, the more it is an attribute. In current methods, this part of intuition is not implemented because it is not relevant: if an artifact is already a tentative attribute, being responsible for no operation can only confirm that the artifact is an attribute. On the contrary, in fuzzy methods, both intuitions are relevant and can be used to revalue/devalue the membership to attribute. Thus, the fuzzy version of rule *Class to Attribute Conversion* has the membership values to both tentative class and tentative attribute as inputs.

The fuzzy version of rule *Class to Attribute Conversion* is as follows:

**F(3)** *Class to Attribute Conversion:*

**IF** *P* IS MEMBERSHIP VALUE A TENTATIVE ATTRIBUTE **AND**
*P* IS MEMBERSHIP VALUE A TENTATIVE CLASS **AND**
OPERATIONS BELONG TO *P* COHESION VALUE **THEN**
*P* IS MEMBERSHIP VALUE AN ATTRIBUTE

Property Cohesion has the same primary terms as property Membership. The antecedent of rule *Class to Attribute Conversion* has three input linguistic variables. To represent the sub-rules of this rule, we still adopt a tabular representation, but each row of the table corresponds to one of the possible combinations of the primary terms of two input linguistic variables, and each column to one of the primary terms of the remaining linguistic variable. To reduce the number of rows and simplify the representation, we suppose that a software engineer can input only primary terms for rules *Tentative Attribute Identification* and *Tentative Class Identification*. This implies that the conclusions inferred from these rules are primary terms of Membership to artifact types Tentative Attribute and Tentative Class. From the definitions given in Tables 2 and 3, it can be deduced that only 11 combinations of membership values to Tentative Class and Tentative Attribute are possible. Table 4 defines the sub-rules of rule *Class to Attribute Conversion* under this assumption. Here, the rows indicate the pairs of possible membership values to tentative attribute and tentative class, and the columns the values of property Cohesion. In the first column, for space problems, we denoted *weakly*, *slightly*, *fairly*, *substantially* and *strongly* as *w*, *sl*, *f*, *sub*, and *st*, respectively. Note that Cohesion is a property of both Tentative Class and Tentative Attribute. Actually, property Cohesion is one of those properties that determine the inconsistency between Class and Attribute in current methods. Table 4 defines the sub-rules of rule *Class to Attribute Conversion*. The definition of the sub-rules takes the following aspects into account.

- The less a set of operations belongs to P, the more P is an attribute.
- The less P is a tentative class and a tentative attribute, the less P is an attribute independently of how much a set of operations belongs to P.

**Table 4.** Sub-rules of rule *Class to Attribute Conversion*.

| P ← [Attribute, (Membership: | P ← [Tentative Class, (Cohesion: P ← [Tentative Attribute, (Cohesion: | | | | |
|---|---|---|---|---|---|
| | weakly | slightly | fairly | substantially | strongly |
| {w, w} | *weakly* | *weakly* | *weakly* | *weakly* | *weakly* |
| {w, sl} | *slightly* | *slightly* | *weakly* | *weakly* | *weakly* |
| {w, f} | *fairly* | *slightly* | *weakly* | *weakly* | *weakly* |
| {w, sub} | *substantially* | *fairly* | *slightly* | *weakly* | *weakly* |
| {sl, w} | *slightly* | *slightly* | *slightly* | *weakly* | *weakly* |
| {sl, sl} | *slightly* | *slightly* | *slightly* | *weakly* | *weakly* |
| {sl, st} | *substantially* | *substantially* | *fairly* | *weakly* | *weakly* |
| {f, w} | *fairly* | *fairly* | *slightly* | *slightly* | *weakly* |
| {f, f} | *substantially* | *fairly* | *fairly* | *slightly* | *weakly* |
| {sub, w} | *substantially* | *substantially* | *fairly* | *slightly* | *weakly* |
| {st, sl} | *strongly* | *substantially* | *fairly* | *slightly* | *weakly* |
| P ← [Tentative Attribute, (Membership: P ← [Tentative Class, (Membership: | | | | | |

Let us consider rule *Attribute to Class Conversion*. Here, the intuition represented by the rule is: The more an attribute is responsible for a set of operations, the more it is a class. There exists, however, another part of intuition concerning the property: the more an artifact is a tentative class and the more is responsible for operations, the more it is a class. The fuzzy version of rule *Attribute to Class Conversion* is as follows:

**F(4)** *Attribute to Class Conversion:*

**IF** *P* IS MEMBERSHIP VALUE A TENTATIVE ATTRIBUTE **AND**

*P* IS MEMBERSHIP VALUE A TENTATIVE CLASS **AND**
OPERATIONS BELONG TO *P* COHESION VALUE **THEN**
*P* IS MEMBERSHIP VALUE A CLASS

Table 5 defines the sub-rules of rule *Attribute to Class Conversion*. The definition of the sub-rules takes the following intuitive aspects into account.

1. The more a set of operations belongs to P, the more P is a class independently of the membership value of P to tentative attribute.
2. The more P is a tentative class, the more P is a class.

**Table 5.** Sub-rules of rule *Attribute to Class Conversion*.

| P ← [Class, (Membership: | P ← [Tentative Class, (Cohesion: P ← [Tentative Attribute, (Cohesion: | | | | |
|---|---|---|---|---|---|
| | weakly | slightly | fairly | substantially | strongly |
| {w, w} | *weakly* | *slightly* | *fairly* | *substantially* | *substantially* |
| {w, sl} | *weakly* | *slightly* | *fairly* | *substantially* | *substantially* |
| {w, f} | *weakly* | *slightly* | *fairly* | *substantially* | *strongly* |
| {w, sub} | *weakly* | *slightly* | *fairly* | *substantially* | *strongly* |
| {sl, w} | *weakly* | *weakly* | *slightly* | *fairly* | *substantially* |
| {sl, sl} | *weakly* | *slightly* | *fairly* | *fairly* | *substantially* |
| {sl, st} | *weakly* | *slightly* | *fairly* | *substantially* | *strongly* |
| {f, w} | *weakly* | *weakly* | *slightly* | *fairly* | *substantially* |
| {f, f} | *weakly* | *slightly* | *fairly* | *substantially* | *strongly* |
| {sub, w} | *weakly* | *slightly* | *fairly* | *substantially* | *substantially* |
| {st, sl} | *weakly* | *slightly* | *fairly* | *fairly* | *substantially* |
| P ← [Tentative Attribute, (Membership: P ← [Tentative Class, (Membership: | | | | | |

After identifying classes, rules *Aggregation Identification* and *Inheritance Identification* determine whether and which relation exists between classes. Obviously, the value of membership of a relation to the set of aggregation or inheritance relations depends on the values of membership of the artifacts being reasoned on to class.

The fuzzy version of rule *Aggregation Identification* is as follows:

**F(5)** *Aggregation Identification:*

**IF** $P_1$ IS MEMBERSHIP VALUE A CLASS **AND** $P_2$ IS MEMBERSHIP VALUE A CLASS **AND**
$P_1$ CONTAINMENT VALUE CONTAINS $P_2$ **THEN**
RELATION BETWEEN $P_1$ AND $P_2$ IS MEMBERSHIP VALUE AN AGGREGATION.

Property Containment has the same primary terms as Membership. Table 6 shows the sub-rules of rule *Aggregation Identification*. Here, the rows indicate the pairs of possible membership values of $P_1$ and $P_2$ to Class, and the columns the values of property Containment. Sub-rules are defined based on the following intuition: the more $P_1$ and $P_2$ are classes and $P_1$ contains $P_2$, the more the relation between $P_1$ and $P_2$ is an aggregation.

Similar to rule *Aggregation Identification*, the fuzzy version of rule *Inheritance Identification* is as follows:

**F(6)** *Inheritance Identification:*

**IF** $P_1$ IS MEMBERSHIP VALUE A CLASS **AND** $P_2$ IS MEMBERSHIP VALUE A CLASS **AND**
$P_2$ IS-A-KIND-OF VALUE IS-A-KIND-OF $P_1$ **THEN**
INHERITANCE BETWEEN $P_1$ AND $P_2$ IS RELEVANCE VALUE RELEVANT.

Property Is-a-kind-of has the same primary terms as Membership. The definition of the sub-rules of rule *Inheritance Identification* can be obtained replacing the property Containment with the property Is-a-kind-of in Table 6.

**Table 6.** Sub-rules of rule *Aggregation Identification*

| $P_3$ ← [Aggregation, | $P_1$ ← [Class, (Containment: | | | | |
|---|---|---|---|---|---|
| (Membership: | weakly | slightly | fairly | substantially | strongly |
| {w, w} | *weakly* | *weakly* | *weakly* | *weakly* | *weakly* |
| {w, sl} | *weakly* | *weakly* | *weakly* | *weakly* | *weakly* |
| {w, f} | *weakly* | *weakly* | *weakly* | *weakly* | *weakly* |
| {w, sub} | *weakly* | *weakly* | *weakly* | *weakly* | *weakly* |
| {w, st} | *weakly* | *weakly* | *weakly* | *weakly* | *weakly* |
| {sl, w} | *weakly* | *weakly* | *weakly* | *weakly* | *weakly* |
| {sl, sl} | *weakly* | *slightly* | *slightly* | *slightly* | *slightly* |
| {sl, f} | *weakly* | *slightly* | *slightly* | *slightly* | *slightly* |
| {sl, sub} | *weakly* | *slightly* | *slightly* | *slightly* | *slightly* |
| {sl, st} | *weakly* | *slightly* | *slightly* | *slightly* | *slightly* |
| {f, w} | *weakly* | *weakly* | *weakly* | *weakly* | *weakly* |
| {f, sl} | *weakly* | *slightly* | *slightly* | *slightly* | *slightly* |
| {f, f} | *weakly* | *slightly* | *fairly* | *fairly* | *fairly* |
| {f, sub} | *weakly* | *slightly* | *fairly* | *fairly* | *fairly* |
| {f, st} | *weakly* | *slightly* | *fairly* | *fairly* | *fairly* |
| {sub, w} | *weakly* | *weakly* | *weakly* | *weakly* | *weakly* |
| {sub, sl} | *weakly* | *slightly* | *slightly* | *slightly* | *slightly* |
| {sub, f} | *weakly* | *slightly* | *fairly* | *fairly* | *fairly* |
| {sub, sub} | *weakly* | *slightly* | *fairly* | *substantially* | *substantially* |
| {sub, st} | *weakly* | *slightly* | *fairly* | *substantially* | *substantially* |
| {st, w} | *weakly* | *weakly* | *weakly* | *weakly* | *weakly* |
| {st, sl} | *weakly* | *slightly* | *slightly* | *slightly* | *slightly* |
| {st, f} | *weakly* | *slightly* | *fairly* | *fairly* | *fairly* |
| {st, sub} | *weakly* | *slightly* | *fairly* | *substantially* | *substantially* |
| {st, st} | *weakly* | *slightly* | *fairly* | *substantially* | *strongly* |
| $P_1$ ← [Class, (Membership: | | | | | |
| $P_2$ ← [Class, (Membership: | | | | | |

Fig. 5 shows the casual dependencies between the rules of the fuzzy example method. By comparing Fig. 5 with Fig. 1, some differences between the classical example method and the fuzzy example method are evident. First, rules *Tentative Class Identification* and *Tentative Attribute Identification* are applied concurrently to the same entities, which can become instances (at different grades) both of Tentative Class and of Tentative Attribute. The two artifact types Tentative Class and Tentative Attribute are not conflicting in the fuzzy method: this allows to reason concurrently on different design alternatives.

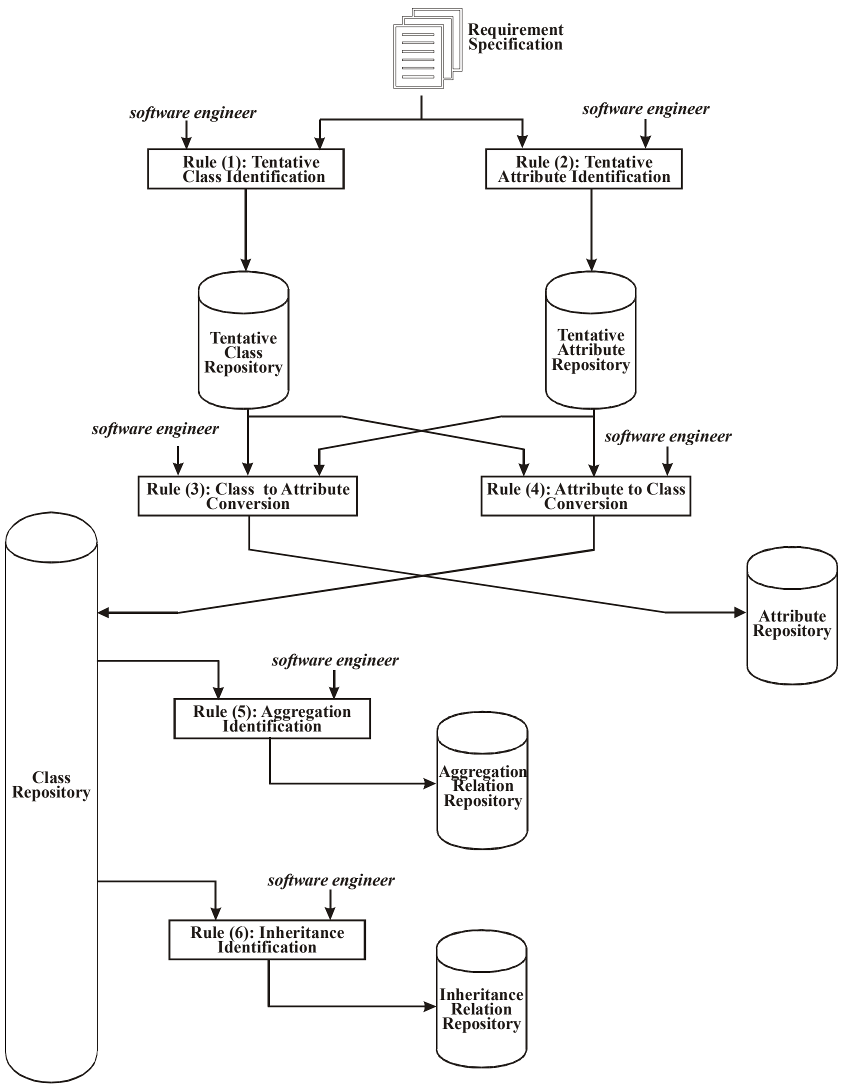
Requirement Specification
software engineer
software engineer
Rule (1): Tentative Class Identification
Rule (2): Tentative Attribute Identification
Tentative Class Repository
Tentative Attribute Repository
software engineer
software engineer
Rule (3): Class to Attribute Conversion
Rule (4): Attribute to Class Conversion
Attribute Repository
Class Repository
software engineer
Rule (5): Aggregation Identification
Aggregation Relation Repository
software engineer
Rule (6): Inheritance Identification
Inheritance Relation Repository

**Fig. 5.** The casual dependencies between the rules of the fuzzy example method.

Rules *Class to Attribute Conversion* and *Attribute to Class Conversion* are the proof of this concurrent evaluation. Indeed, these rules are applied to the same artifact taking into consideration both the membership to Tentative Class and the membership to Tentative Attribute. The two values of membership, together property Cohesion, contribute simultaneously to define the membership degrees to Attribute and Class. Though we maintained the same names as in the classical example method, in the fuzzy method the fuzzy rules *Class to Attribute Conversion* and *Attribute to Class Conversion* perform no conversion, but rather a revaluation (devaluation) of the artifact as an attribute and a class, respectively.

Thus, if fuzzy logic-based methodological rules are applied, no alternative design solutions are theoretically eliminated. Software engineers are not forced to take abrupt decisions, but rather they are encouraged to express their perception of artifacts in their natural language. This reduces the loss of information and increases the quality of the overall development process. The complexity of this concurrent analysis of multiple alternative solutions is managed by using an appropriately designed CASE environment, which will be described in the next section.

The concurrent analysis of design alternative solutions can continue until a product has to be released. Then, only one of a set of possible conflicting solutions has to be chosen. For instance, we should decide whether an entity should be implemented as a class or as an attribute. The membership values of the artifact to the conflicting artifact types can be exploited to perform this choice. Intuitively, if an entity is substantially a class and slightly an attribute, it is natural to select the entity as a class rather than as an attribute.

To take a decision in case of conflicting design solutions, we need to trace the evolution of each element taken into consideration during the software development process. For each element, our CASE maintains the membership values of the element to the artifact types which have been visited during the development process. Each element knows which artifact types are conflicting. For instance, an entity knows that it cannot be a class and an attribute at the same time. When a product has to be released, the element is triggered to activate a conflicting resolution policy. Policies depend on the inconsistency type and may be affected by contextual factors such as application type, sensibility and experience of the software engineer and desired level of quality. Typically, the defuzzified membership values to the conflicting artifact types are compared and the artifact type with the highest value is selected. Only if the highest value is above an "existence" threshold (typically fixed to 0.5), the artifact is included into the final product; otherwise it is eliminated. The existence threshold allows eliminating for example classes or attributes which derive from weakly or slightly relevant entities, and are not revalued by the application of the conversion rules. Optionally, before the selection or the elimination is made, the software engineer may be consulted. In particular, our CASE gets an opinion from the software engineer if the defuzzified membership values to conflicting artifact types are close to each other (for instance, the difference is less than 0.1), and if the value of the winning artifact type is close to the existence threshold. Further details on the selection of one among a group of conflicting design solutions are given in [24]. An example of application of a fuzzy method can be found in [2].

Concluding, the methodology of computing with words seems to be a very natural basis for automating the software development process. Software engineers should not have a lot of trouble to migrate to this new technology. This hope is supported by the rapid success that fuzzy logic has reached in fields which can be considered analogous to software engineering. For example, in control engineering, control engineers judge very familiar to codify their experience using rules expressed in natural language [20]. Since methodological rules are conceptually quite close to control rules, we believe that the approach described in this chapter can easily be accepted by software engineers. Further, fuzzy logic has been already successfully applied to software engineering. Fuzzy techniques have been used to handle the uncertainty arisen from the classification of components according to software behavioral properties [13], to cope with unreliable data in a business process modeling method [5], to specify and analyze imprecise requirements [21], and to monitor software processes based on the detection of deviations between the actual enacting process and the process enactment plan [9-9]. Similarly to our approach, the use of fuzzy logic is justified by its ability to naturally represent uncertain and imprecise information, and to constitute a good framework for approximate reasoning.

## CASE environment

Our CASE environment is implemented as a fuzzy rule-based expert system. We recall that an expert system is basically composed of three modules: a knowledge base, an inference engine and a user interface. In our case, the knowledge base contains the linguistic variables and the methodological rules. The inference engine implements the generalised modus ponens. The user interface has a twofold role: on the one hand it helps method engineers codify properties of artifact types as linguistic variables and define fuzzy rules, and on the other hand guides software engineers during the development process to insert the values necessary to the inference process.

The method engineer's user interface is composed of two editors: the linguistic variable editor and the rule editor. The former allows method engineers to characterize completely a linguistic variable by defining its name, the universe of discourse, the set of primary terms and the fuzzy sets associated with these primary terms. Rectangular, trapezoidal, triangular and standard piecewise quadratic membership functions are the default alternatives supported by the editor. After the properties involved in the definition of a rule have been defined, the method engineer can use the rule editor to codify the methodological rules. The rule editor helps method engineers fill the tables that describe the sub-rules. For each combination of the primary terms of the properties involved in the antecedent of a rule, the method engineer can choose one of the primary terms of the property involved in the consequent. Further, a set of parameters has to be initialised to determine the implication, composition and aggregation operators that the method engineer desires to use in the inference process. This choice has been supported to allow method engineers to experiment with different implementations of fuzzy reasoning and find the most suitable to their aims.

During the execution of the rules, software engineers interact with a user-friendly interface, which guides them to insert the input values necessary to the inference process. For each property involved in the rule being investigated, software engineers can input both a linguistic value among those allowed by the grammar and a numerical value. To make the software engineer conscious of the meaning associated with each primary term by the method engineer, the user interface shows the membership functions, which define the primary terms. Each new input provided by the software engineer infers a number of sub-rules: for instance, the value of Relevance of an entity in the requirement specification infers both the sub-rules defined for instantiating the entity as a tentative class and those for instantiating the entity as a tentative attribute. In this way, each path of the development process is investigated concurrently and different design solutions can be analysed at the same time.

For each artifact type, the fuzzy expert system maintains a repository which contains all the instances of the artifact type. To reduce the complexity of the development process, the CASE allows software engineers to decide that only instances in the repository with a membership degree higher than a threshold can activate rules which have the artifact type as pre-artifact. Let $C_T(x)$ be the set of fuzzy conditions defined by the antecedents of rules which have artifact type $T$ as goal-artifact type in the consequent. Then, the set of instances $x$ of artifact type $T$ which can fire a forward chaining are $\{x\text{: } \mathrm{C}_\mathrm{T}(x) > \sigma\}$, where $0<\sigma<1$ is fixed by the software engineer. The value of the threshold can also be a linguistic value. For example, a software engineer can decide that instances belonging less than slightly to an artifact type are discarded in the development process. Thus, if an entity is selected as weakly an attribute, no rule, which reasons on artifact type Attribute, will be inferred for that entity. In addition, the tool allows a software engineer to fix priorities in investigating conflicting alternatives. For instance, the software engineer can establish to be guided to work first on the alternative with the highest measure and then on the others. These options allow software engineers to reach a compromise between complexity and accuracy.

To implement the fuzzy expert system, we used the object-oriented fuzzy inference framework described in [4]. This framework was developed by using the approach proposed in [3]. The top-level knowledge graph of this framework is shown in Fig. 6. Node *Fuzzy Inference Element* implements the inference mechanism. This element contains *Rule*, *Fact*, *G.M.P.* and *Conclusion*. During the initialization phase, *Rule* and *Fact* communicate with the node *Linguistic Variable* to create a representation of themselves in terms of fuzzy sets. For each proposition involved in *Rule* and *Fact*, the corresponding fuzzy set is created. Obviously, while the representation of *Rule* can be produced when the *Fuzzy Inference Element* is created, the representation of *Fact* can be computed only when the necessary input values are provided by the software engineer. The output values of *Rule* and *Fact*, again expressed in terms of fuzzy sets, are provided to the node *Generalized Modus Ponens* (*G.M.P.*). This node carries out the inference process and generates a result. The node *Conclusion* combines all the outputs of the related generalized modus ponens nodes using an aggregation operator. The result of this combination is also expressed in terms of fuzzy sets. The node *Linguistic Variable* is used either to associate a linguistic value to the fuzzy set produced by *Conclusion* or to defuzzify the fuzzy set. The forward chaining is implemented as follows: the conclusion produced by node *Conclusion* is propagated to node *Linguistic Variable*, which in its turn propagates it to each *Fact* that refers to the linguistic variable in one of its composing propositions.

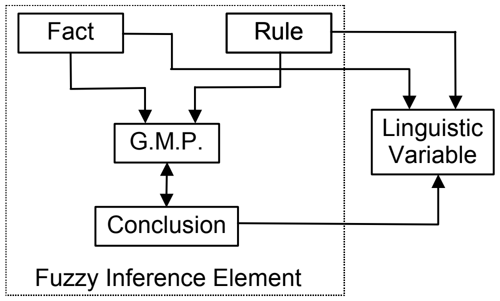

**Fig. 6.** The top-level knowledge graph for the fuzzy-logic reasoning framework.

All fuzzy inference elements used to create instances of an artifact to different artifact types are stored so as to save the story of the artifact. As each node of the top-level knowledge graph is implemented as an object, this artifact evolution is maintained using the same paradigm with which the software product is implemented. Each software product can therefore preserve its story in an executable format so as to guarantee an increase in maintainability and reusability.

## Conclusions

The growing complexity of the today's software systems requires CASE environments that do not provide only tools to model structure and behaviour of artifacts, but rather guide software engineers in the overall development process. To this aim, methods have to be available in a computational format. Since designing is a human-intensive activity, the formalism must adopt designer's language as well. In addition, it must be able to represent and cope with multiple design alternatives and uncertainty.

In this chapter, we have highlighted that classical set theory and two-valued logic, which are typically used to express methodological rules in current methods, are able to capture completely neither method developers' intuition in defining an artifact type nor software engineers' perception in identifying an artifact. To overcome this problem, we have proposed to model the software development process using the methodology of computing with words (CW). With respect to two-valued logic, CW supplies a much more expressive language for knowledge representation and much more versatile machinery for reasoning and computation. It has been shown that CW 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 the loss of information during the development process. Further, CW leads the software engineer to investigate each relevant property before eliminating an artifact. Finally, CW allows managing multiple design alternatives concurrently and provides a means to software engineers for evaluating and selecting one among the alternatives, if needed.

## References

1. Advanced Package Design (2003), Cadence Design Systems, Inc., url: http://www.cadence.com/.
2. Aksit M, Marcelloni F (2001) Deferring elimination of design alternatives in object-oriented methods. Concurrency and Computation – Practice and Experience, John Wiley & Sons, Inc., 13 (14): 1247-1279.
3. Aksit M, Marcelloni F, Tekinerdogan B (2000) Developing OO Frameworks Using Domain Models. ACM Computing Surveys 32 (1es).
4. Aksit M, Tekinerdogan B, Marcelloni F, Bergmans L (1999) Deriving Object-Oriented Frameworks From Domain Knowledge. In: Fayad M and Schmidt D, eds., Object-Oriented Frameworks, John Wiley & Sons, Inc., New York, pp. 169-198.
5. Benedicenti L, Succi G, Vernazza T, Valerio A (1998) Object Oriented Process Modeling with Fuzzy Logic. In: Proceedings of the 1998 ACM symposium on Applied Computing, pp. 267 – 271.
6. BridgePoint (2003), Project Technology, Inc., url: http://www.projtech.com/.
7. Booch G (1993) Object-oriented Analysis and Design with Applications. Second edition. Benjamin Cummings, Redwood City.
8. Christie AM (1995) Software Process Automation: The Technology and its Adoption. Springer-Verlag.
9. Cîmpan S, Oquendo F (1998) Fuzzy indicators for monitoring software processes. In: Proceedings of the 6th European Workshop on Software Process Technology, Springer-Verlag, Berlin, Germany, pp. 43-59.
10. Cîmpan S, Oquendo F (1999) On the application of fuzzy set theory to the monitoring of software-intensive processes. In: Proceedings of the Eighth International Fuzzy Systems Association World Congress. Nat. Central Univ, Chungli, Taiwan, pp. 1071-1076.
11. Cîimpan S, Oquendo F (2000) Dealing with software process deviations using fuzzy logic based monitoring. Applied Computing Review 8 (2): 3-13.
12. Coleman D, Arnold P, Bodoff S, Dollin C, Gilchrist H, Hayes F, Jeremaes P (1994) Object-Oriented Development: The Fusion Method. Prentice Hall.
13. Damiani E, Fugini M G (1996) Fuzzy techniques for software reuse. In: Proceedings of the 1996 ACM symposium on Applied Computing, pp. 552 – 557.
14. Graham I (1994) Object-Oriented Methods. Second Edition. Addison-Wesley.
15. IC Design (2003). Mentor Graphics Corp, Wilsonville, OR USA. http://www.mentor.com/
16. Klir GJ, Yuan B (1995) Fuzzy Sets and Fuzzy Logic - Theory and Applications. Prentice Hall, Upper Saddle River, NJ 07458.
17. Jacobson I, Christerson M, Jonsson P, Overgaard G (1992) Object-Oriented Software Engineering - A Use Case Driven Approach. Addison-Wesley/ACM Press.

18. Jackson P (1999) Introduction to Expert Systems. Third Edition. Addison Wesley, Longman, Harlow, England.
19. Lazzerini B, Marcelloni F (2000) Some Considerations on Input and Output Partitions to Produce Meaningful Conclusions in Fuzzy Inference. Fuzzy Sets and Systems 113 (2): 221-235.
20. Lee CC (1990) Fuzzy Logic in Control Systems: Fuzzy Logic Controller, Part II. IEEE Transactions on Systems, Man, and Cybernetics, 20 (2): 419-435.
21. Liu XF, Yen J (1996) An analytic framework for specifying and analyzing imprecise requirements. In: Proceedings of the 18th International Conference on Software Engineering, pp. 60 – 69.
22. Marcelloni F, Aksit M (1999) Reducing Quantization Error and Contextual Bias Problems in Software Development Processes by Applying Fuzzy Logic. In: Proceedings of NAFIPS'99, New York, IEEE Press, pp. 268-272.
23. Marcelloni F, Aksit M (2000) Improving object-oriented methods by using fuzzy logic. ACM Applied Computing Review 8 (2): 14-23.
24. Marcelloni F, Aksit M (2001) Leaving Inconsistency using Fuzzy Logic, Information and Software Technology 43: 725-741.
25. Paradigm Plus (2003), Computer Associates International, Inc., url: http://www.cai.com/products/alm/paradigm_plus.htm.
26. Parsaye K, Chignell M (1988) Expert Systems for Experts. John Wiley & Sons, Inc.
27. Rational Rose, Rational, url: http://www.rational.com/products/rose/.
28. Riel A (1996) Object oriented design heuristics. Addison-Wesley.
29. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W (1991) Object-Oriented Modeling and Design. Prentice-Hall.
30. Turksen IB, Tian Y (1993) Combination of Rules or Their Consequences in Fuzzy Expert Systems. Fuzzy Sets and Systems 58: 3-40.
31. Yourdon E (1989) Modern Structured Analysis. Englewood Cliffs, New Jersey: Yourdon Press.
32. Zadeh L A (1973) Outline of a New Approach to the Analysis of Complex Systems and Decision Processes. IEEE Transactions on Systems, Man, and Cybernetics, SMC-3 (1): 28-44.
33. Zadeh LA (1996) Fuzzy Logic = Computing with Words. IEEE Transactions on Fuzzy Systems 4 (2): 103-111.
34. Zadeh LA (1999) From Computing with Numbers to Computing with Words-From Manipulation of Measurements to Manipulation of Perceptions. IEEE Transactions on Circuits and Systems-I: Fundamental, Theory and Applications 45 (1): 105-119.