Introducing Fuzziness in Object Models and Database Interfaces through Aspects

Miguel-Angel Sicilia, Elena García-Barriocanal and León González-Sotos
Computer Science Department, University of Alcalá
Ctra. Barcelona km. 33.6 — 28871 Alcalá de Henares, Madrid, Spain
{msicilia, elena.garciab, leon.gonzalez}@uah.es

Abstract

Imperfection in information can be considered a cross–cutting concern that manifests itself in diverse kinds of imprecision, uncertainty or inconsistency in the data models of a software system. The extension of existing programming and querying interfaces for the different aspects of information imperfection requires a proper modularization of the different concerns of numerical imprecision handling, so that the extensions do not interfere with existing programming practices and do not obscure the original design. Aspect–oriented design (AOD) enables such form of non–intrusive extensions to be added to existing software libraries, clearly separating fuzziness or other imperfections in data as a differentiated concern, that can be considered from the early phases of development. In this paper, a general framework for aspect–based extension of data models and fuzzy databases is described, and some design and implementation issues of such AOD–based extensions on `OJB` database libraries are described as a case study.
Keywords Aspect–orientation, fuzzy databases, orthogonal persistence.

1 Introduction

A growing body of research regarding fuzzy databases and fuzzy querying has emerged in the last years, encompassing diverse extensions to the relational and object data models, e.g. [3, 4, 7]. As a result, several research–oriented implementations of fuzzy queries on top of commercial database systems or standard interfaces have been developed as those described in [18, 30, 8, 31]. In addition, a tendency to develop object–oriented programming and querying interfaces independent from the actual data sources —

which was the focus of the persistence services of CORBA[1] and also of several vendor–specific products — has resulted in standardized programming and querying interfaces like the Jdo specification [15]. One of the key characteristics of Jdo is the fact that it is deliberately independent on the underlying storage and architecture of the data sources, so that some form of *data mapping* is processed by a Jdo software layer, informed by a set of metadata definitions. In consequence, interfaces for orthogonal persistence [1] of objects are provided to the programmers, irrespective of the actual management of data and query resolution, which can be relational or object–oriented.

The extension for fuzziness of such kind of interfaces can be accomplished by adding elements to the query syntax — as is done in [5]— and also by augmenting programming interfaces to deal with the desired fuzzy modelling capabilities — e.g. as in [18]. In any case, several reasons point to some implementation characteristics that are required for a seamless integration with existing interfaces. On the one hand, extensions should be *strictly additive*, i.e. they should not interfere with the non–fuzzy capabilities of the programming and querying interfaces, both for the sake of backward–compatibility and of ease of learning [21]. And on the other hand, the extensions should be properly *modularized*, not obscuring the original design and architecture of the extended database libraries. The first requirement can be met through a careful design, using polymorphism and proxy objects as described in [18], and using reflective capabilities when required as described in [2]. Nonetheless, the second requirement calls for specialized design and implementation capabilities that allow the extension of software with cross–cutting concerns (as fuzziness can be considered with regards to data representation). Aspect–oriented design (AOD) [27] provides the required modularization capabilities for the latter issue, since *aspects*, *advices* and *introductions* as programming constructs are able of separating the details of the handling of imperfection in data.

Recent research has sketched how aspect–oriented design can be used to extend existing database interfaces for fuzziness [23]. This paper provides a broader, more comprehensive framework for such technical issues, and highlights the principal design elements that must be considered when developing such kind of extensions. The concept of "early aspects" [22] provides a conceptual, analysis–level model for introducing the different facets of information imperfection [24].

The implementation issues of fuzzy extensions to object–database interfaces are approached from the perspective of AOD, addressing both general

[1] http://www.omg.org/corba/

implementation issues and a concrete case study using `aspectj` to extend the Java–based interfaces of the `ObjectRelationalBridge (OJB)`[2] open–source libraries. Related work includes studies of fuzziness related to prototype theory of categorization [26] to classify types of software units, and advances on extending programming language semantics through reflection [2], but no previous research apart from [23] exists on the introduction of fuzziness in software units through aspect–oriented techniques.

The approach to introducing fuzziness described in this paper starts with requirement analysis, considering imperfection in data as an analysis concern. The identified requirements can then be mapped to concrete extensions at design time, and aspect–oriented programming provides the mechanism to implement such concerns in a modular way. In addition to the traceability and early consideration of imperfection, this approach provides the benefits of being non–intrusive, in the sense that the existing applications can be augmented for imperfection, and new ones can benefit from an isolation of the details of the numerical algorithms that deal with fuzziness, uncertainty or inconsistency.

The rest of this paper is structured as follows. Section 2 provides a general framework to introduce diverse information imperfection issues in conceptual modeling. In Section 3, a concrete extension case is described. Concretely, the main general issues of extending `OJB` interfaces with fuzziness are described. Then, a simple case of fuzzy extension is sketched in Section 4 for illustration purposes. Finally, conclusions and future research issues are provided in Section 5.

2 Addressing Information Imperfection through Aspect-orientation

2.1 Information imperfection as a concern

The *separation of concerns* principle has recently been applied to early stages of development like requirements engineering [25, 14, 11] and software architecture [28]. The early separation of cross–cutting concerns results in improved localization, and eventually in improved development and maintenance activities. In the research literature, a number of examples of cross–cutting concerns are often used for illustration purposes, or they are described when reporting case studies. Recurring examples include, for example, security, usability, persistence or performance.

[2] http://db.apache.org/ojb/

The focus of this paper is that of *imperfection* in information as a cross–cutting system concern that is currently overlooked in many application models. Imperfection is a multifaceted concept including imprecision, uncertainty and inconsistency, being classical probability a model for a specific type of imperfection among many others. Currently, general and mature mathematical frameworks for the management of imperfection are available [13], although their widespread use in mainstream development technologies and industrial systems is still to come. To adhere to an unambiguous interpretation of the terms used for the various sub–aspects of information imperfection, we'll use them in the sense given in Smet's taxonomy described in [24].

Imperfection in information should be addressed early in the lifecycle due to the specifics of uncertainty and imprecision in conceptual modeling [6], and its impact on architectural and implementation decisions, most notably in persistence and querying [18]. Information imperfection is a logical "matter of interest", according to COSMOS [25] terminology.

2.2 Classifying concerns for information imperfection

An straightforward *classification* would include one class for each of the principal aspects of imperfection:

- `Imprecision-Related`
- `Uncertainty-Related`
- `Inconsistency-Related`
- `Hybrid`. This category accounts for any combination of the above.

According to Smets, "imprecision and inconsistency are properties related to the content of the statement: either more than one world or no world is compatible with the available information, respectively. Uncertainty is a property that results from a lack of information about the world for deciding if the statement is true or false. Imprecision and inconsistency are essentially properties of the information itself whereas uncertainty is a property of the relation between the information and our knowledge about the world".

Classes inside those categories may refer to more specific types of imperfection according to Smet's taxonomy, e.g. `FuzzyElement` refers to "imprecision without error" without decidability as in "age is close to 30", while `PossibleElement` refers to "happen–ability" as a kind of uncertainty.

In addition, imperfection manifestations can be classified in `Domain-Imperfection`, `UserImperfection` and `System-Imperfection` according to the source from which the imperfection originates. For example, inferences internal to the system may generate imperfect information from perfect inputs. Imperfections coming from the user might be used to reflect somewhat uncertainty in the modeling itself. This concept is similar to that of the Fuzzy-EER at level L_1 for entities, relationships and attributes, so that, for example, the set of entities in a model can be given a membership grade [6], that can be interpreted, for example, as "it's not completely sure the role element E plays in the context of the model".

According to the kind of element in the conceptual model that is subject to imperfection, we can have additional classifications, namely `ImperfectElement` and `ImperfectRelationship`, the former containing classes `ImperfectClass`, `ImperfectAttribute`, `ImperfectFunction` and `ImperfectResult` which roughly correspond to classes, attributes, method (or more generally, functionality), and method results in conceptual models, and the latter containing classes for each type of relationship (association, generalization, etc.). Imperfect conceptual model elements can be expressed in the domain model through extensions to the `UML` like the one sketched in [17]. Figure 1 depicts some of the just described elements and some of example relationships between them, representing classifications as `UML` packages.

Properties as defined in COSMOS are "concerns that characterize other logical concerns". Concerns that arise in the context of imperfection include `Granulation-level` and `Interpretability`. The former refers to the degree of "summarization" of an element, e.g. "large pages" may be interpreted to subsume "very–large" and "moderately large", thus summarizing information. The latter refers to the ease of interpretation of the information by humans, and is often considered as a quality criteria in rule bases. They can be used as requirements constraining the design of other concerns, e.g. `High-Interpretability` may involve selecting a fuzzy rule simplification algorithm at later stages for the computation of a given conceptual model element.

2.3 Information imperfection concerns in the Software Engineering process

It should be noted that the imperfection–related concepts introduced so far do not require understanding neither about the mathematical frameworks for uncertainty handling not about their software representation, so that they can be considered a concern in domain and requirements engineering. This

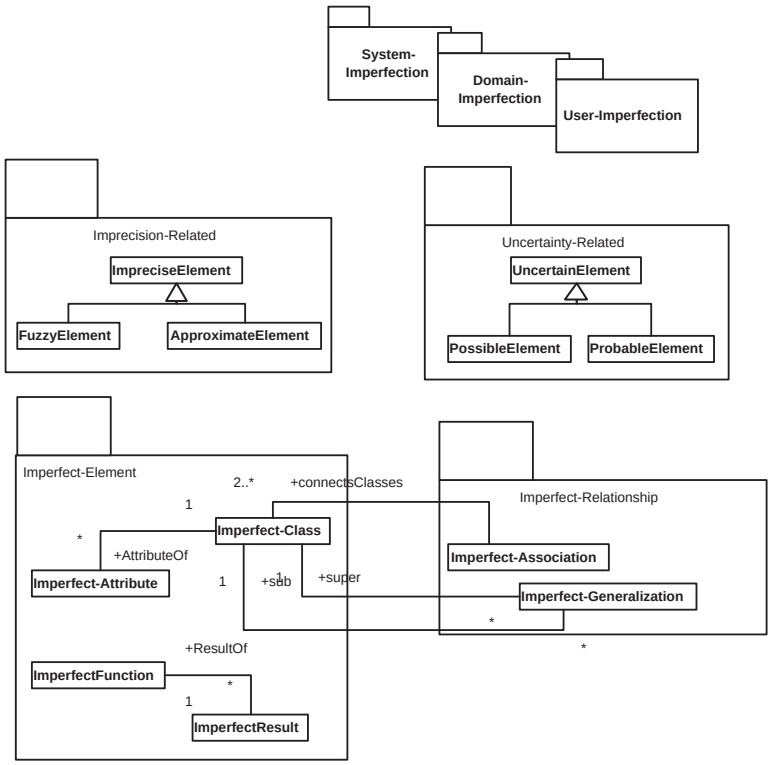

Figure 1: General concern dimensions related to information imperfection

makes possible that they can be used at the stage of requirement analysis, which would result in a first specific practice.

Specific practice 1 *Identify and classify the information imperfection associated to each conceptual modeling data element, if any.*

This entails an analysis of data elements (e.g. clases, attributes) and the adoption of a classification framework as the one described above.

Such concern–analysis enables the early identification of imperfection–related requirements, that would be mapped in subsequent phases of the software engineering life cycle to concrete architectural elements and ultimately to libraries or class frameworks dealing with fuzziness or other imprecision and uncertainty handling aspects. In fact, classical error theory that manifests in required degrees of precision in numerical computations is just a model to deal with a concrete facet of imperfection related to numerical representation and error in measurement instruments.

The second practice entails design, since a computational framework must be selected for each analysis–level concern. This leads to a second specific practice.

Specific practice 2 *For each information imperfection concern identified in the analysis, choose the appropriate mathematical formalism for its representation.*

This design step entails the explicit documentation of the decision made to choose a concrete mathematical framework [13]. A consideration of computational complexity, and the availability of libraries could be accounted for in these design activities.

Aspect–oriented design (AOD) can be used as the candidate detailed design technique whenever concerns cross–cut software modules, and it becomes the required option if existing libraries are required to be extended for fuzziness without changing the syntax and semantics of their interfaces, since *aspects* like those of `aspectj`[10] allow the proper modularization and encapsulation of concerns, avoiding tangling existing source code.

Specific practice 3 *If some existing persistence libraries or interfaces are used, use AOD and AOP to achieve non–intrusive extensions. Else, carefully design the interfaces of the tailored code to allow for aspect development of the code that deals with the specifics of information imperfection handling.*

Further, AOD enhances maintainability, since the added concerns for fuzziness are separated from "crisp" functionality, so that defects can be easily located. In addition, AOD for the implementation of fuzziness eases the production of "crisp" and "fuzzy" versions of the same software, since the features of *advice* and *introduction* contained in aspects are dynamic, and thus they are "disabled" by simply excluding the aspects from the concrete build of the system. This enables the key consequence that "backward compatibility" is achieved, i.e. the introduction of fuzziness can be simply ignored by the original applications that did not dealt with fuzziness.

2.4 An example: Market segmentation

Market segmentation systems provide support for the classification of customers with the purpose of targeting marketing strategies [29]. Market segmentation criteria are in many cases uncertain, and the resulting subsets can be considered to have no sharp boundaries, specially in the new relationship marketing paradigm [20].

A basic model for market segmentation using the relationship value model in [20] will result in the classes and instances showed in Table 1.

Segments can be considered imperfect classes of users inferred by the system from a net value relationship model that estimates the value of each customer relationship from rough estimates of increments in purchases and expected relationship duration. Those estimates are rough pessimistic and optimistic values obtained from experts. Clustering algorithms are often used for the delineation of the boundaries of such segments.

Customer similarity is derived from the segments and perhaps from the analogies in purchasing behavior of a pair of customers, as often interpreted in recommender systems [16].

This example serves an illustrative purpose. It should be noted that the decisions about the classification of the different elements is contingent to the actual knowledge of the specificities of the problem domain, and these in turn determine the numerical handling of imperfection that will come at design time.

3 Extending `OJB` with Aspects for Fuzziness

As mentioned above, the techniques of aspect-oriented design (AOD) provide improved modularity to software systems by focusing on separation of concerns [27]. Fuzziness as a concrete form of information imperfection can be considered a cross–cutting concern for existing database processing libraries, so that it can be added to existing crisp software without altering the programming interfaces that are being used.

As a proof of concept for introducing fuzziness in existing libraries, the `OJB` framework has been extended by using `AspectJ`. The `AspectJ` framework [10] is an AOD extension to the Java programming language based on the concept of dynamic *pointcut* (the intersection of a number of well–defined execution points) and *advice* (code attached to specific pointcuts). Using aspects in `OJB` requires a previous recompilation of its source code version, which can be accomplished by adding an `Ajc` task to the `build.xml` ANT file of `OJB`.

OJB supports multiple persistence application programming interfaces (APIs):

- An ODMG 3.0[3] compliant API.

[3] http://www.odmg.org/

- A JDO 1.0[4] compliant API.
- An Object Transaction Manager (OTM) layer that contains all features that JDO and ODMG have in common.
- A low-level PersistenceBroker API which serves as the OJB persistence kernel.

The OTM-, ODMG- and JDO-implementations are build on top of the persistence kernel, so that it makes sense to concentrate first on it, and later address higher–level interfaces. In addition, a principle of "minimum difference" with existing interfaces and programming idioms is followed, in an attempt to maximize the *usability* of the extensions in the sense described in [21]. In the rest of this section, the main design issues surrounding the extension of `OJB` are briefly described.

3.1 Extending metadata handling interfaces

Metadata handling in `OJB` is centralized in the `MetadataManager` class, implementing a singleton pattern [9] that can be used to obtain a reference to the `DescriptorRepository` instance containing object mapping and manipulation information for persistent objects. The class needs to be extended to deal with the required metadata describing fuzzy constructs.

The better way to do it is by merging standard metadata descriptions with fuzzy ones. This can be accomplished by invoking the `mergeDescriptorRepository` method of `MetadataManager` after a (successful) call to the `MetadataManager.init` method at construction time. Since `init` is a private method in `MetadataManager`, the pointcut could be changed to any constructor of the class. This will prevent that an evolution or refactoring of the internals of the libraries requires a change in the aspect code. It should be noted also that the documentation of the libraries state that "all metadata is read at startup of OJB, when the first call to `PersistenceBrokerFactory` [...] or `MetadataManager` class was done". Consequently, the calls to the factory may also be included to cover all the possible initialization paths.

The merging can be done by capturing its *pointcut* through an around *advice*. The following code fragment sketches an aspect encapsulating such processing:

```
public aspect FuzzyMetadataManagement {
    private DescriptorRepository globalRepository;
```

[4] http://java.sun.com/products/jdo/

```
    void around (MetadataManager m):
         target(m) && call(* MetadataManager.init(..)){
        try{
                proceed(m);
        }catch(MetadataException e){throw e;}

        // create a mapping repository:
        globalRepository = loadFuzzyDescriptorRepository();
        // merge with standard OJB mappings:
        m.mergeDescriptorRepository(globalRepository);
    }

    private DescriptorRepository loadFuzzyDescriptorRepository(){
        DescriptorRepository dr = new DescriptorRepository();
        //  load descriptor repository...
        return dr;
    }
    //...
}
```

The `loadFuzzyDescriptorRepository` method simply carries out the processing of the fuzzy schema description residing in a XML file, similar to that described in [18]. For convenience, a `globalRepository` reference is maintained in the aspect instance, facilitating processing of issues only related with fuzzy data descriptions by invoking other methods in the aspect. It should be noted that neither inheritance nor reflection would have been enough to make this change without modifying existing libraries, since at least the `getInstance` method of `MetadataManager` would had to be changed to instantiate a new subclass or build it dynamically.

3.2 Describing `imperfect-data` elements

The current interface of `DescriptorRepository` makes use of `ClassDescriptor` for the object–relational mapping descriptions, which in turn uses `FieldDescriptor` to specify the storage of class attributes. Both elements could be extended to their fuzzy counterparts through subclassing, and other conceptual data elements like associations [19] can be derived from the common super–class `DescriptorBase` which is an open–ended hook for them, that only provides a basic descriptive functionality. The last version of OJB has declared `ClassDescriptor` as final, which then forces to have a workaround of this

design. The alternative is that of using directly `DescriptorBase`.

`Imperfect-Class` and `Imperfect-Attribute` concerns can be implemented by extending the `ClassDescriptor` and `FieldDescriptor` classes respectively with methods to query for the data mapping of the membership degrees, required to implement querying and storage functionality. Figure 2 depicts some of the details of such extension as a `UML` diagram.

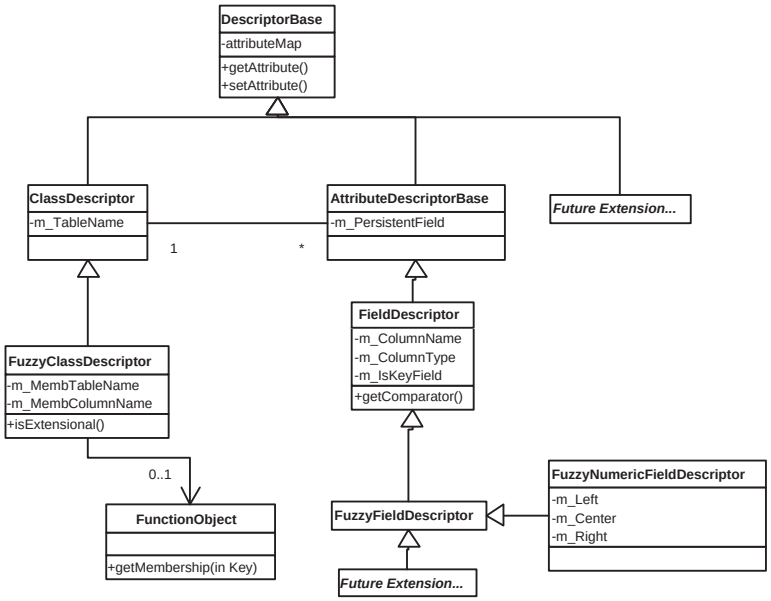

Figure 2: Extension of `OJB` metadata description classes

As showed in Figure 2, the `AttributeDescriptorBase` is used as an intermediate extension point for persistence mappings that are not necessarily relational, while `FieldDescriptor` provides the specific details of the relational mapping. `FuzzyFieldDescriptor` is provided as the base class for any relational–mapping of imperfect attributes, and `FuzzyNumericFieldDescriptor` is one of its subclasses representing the mapping for (triangular) fuzzy numbers. `FuzzyClassDescriptor` encapsulates the details for the relational mapping of fuzzy classes and sub–classes, with the possible variants described in [18]. *Extensional* mappings store explicitly the membership degrees for each object in the database, while *intensional* mappings provide a `FunctionObject` instance encapsulating arbitrarily complex computations of membership degrees for the class. In this latter case, Java's reflection capabilities enable the specification of a `FuzzyObject` subclass in the configuration file that is instantiated dynamically at run–time.

3.3 Basic fuzzy storage

Object storage both in `ODMG` and `JDO` mappings proceeds in cascade, storing the graph of references starting from the object being stored. For example, JDO provides a method `makePersistent` in the `PersistentManager` interface to make concrete instances persistent, and it also provides persistence by "reachability", so that any instance linked to a persistent one (transitively) is also made persistent. The underlying core `OJB` API ultimately uses the `store` methods in the `PersistenceBroker` interface to resolve those calls, which explicitly handles also the storage of `Collection` implementing classes.

The storage of fuzzy classes only requires code modifications for *extensional* fuzzy classes, in which membership degrees are explicitly stored. To do so, variants of the `store` methods with an additional parameter are required. `Aspectj` *introductions* can be used to accomplish such extension, avoiding subclassing such a complex class like `PersistenceBrokerImpl`. The aspect can be targeted to the interface `PersistenceBroker` to guarantee that future implementation classes also are provided with fuzzy storage methods. The following code fragment sketches such extension for the simpler of the definitions of `store`:

```
package org.apache.ojb.broker.core.fuzzy;

import org.apache.ojb.broker.core.*; import
org.apache.ojb.broker.metadata.*;

public aspect FuzzyStorageHandling{
    public void PersistenceBroker.store(java.lang.Object obj, Double m,
                                        String fuzzyClass)
                  throws PersistenceBrokerException;

    // introduces a new implementation:
    public void PersistenceBrokerImpl.store(java.lang.Object obj, Double m,
    String fuzzyClass) throws PersistenceBrokerException {
        store(obj);
        storeMembership(obj, m, fuzzyClass);
    }

    public void PersistenceBrokerImpl.storeMembership(
             java.lang.Object obj, Double m, String fuzzyClass)
```

```
                throws PersistenceBrokerException {
      // Obtain descriptor:
      FuzzyClassDescriptor cld = (FuzzyClassDescriptor)
            descriptorRepository.getDescriptorFor(fuzzyClass);

      if (cld.isExtensional()){
        // call access layer to store m...
      }
    }
}
```

The `fuzzyClass` attribute is required since a single object may belong to more than one fuzzy class with different degrees, i.e. multiple classification outside of the capabilities of the programming language is assumed.

The storage of fuzzy attributes is delegated in `PersistenceBrokerImpl` to a `JdbcAccess` interface which acts as a layer encapsulating the construction of SQL sentences from metadata descriptors and actual objects to be stored. Its existing implementation `JdbcAccessImpl` uses sequences of calls to `getXStatement` and `bindXStatement` (where "X" stand for insert, delete, etc.) on the instance of `StatementManagerIF` provided by `PersistenceBroker.serviceStatementManager()`. The delegation chain can be followed through the `StatementManager` — responsible for the value `bind`ing process — class to the `StatementsForClass` interface and implementation – which only provides caching for statement templates — and to the `SqlGenerator` interface and implementation, which actually build the SQL queries from class descriptions by delegating to a number of classes, one for each type of SQL sentence. In consequence, it is in the `SqlUpdateStatement` where the actual sentence creation logic resides, where it can be seen that no modification is required to store fuzzy classes, since it relies in the (extended through polymorphism) behavior of `ClassDescription`.

In contrast, the storage of fuzzy fields as those described by `FuzzyNumericFieldDescriptor` require dynamic extension of the methods `appendListOfColumns` and `appendListOfValues` since they assume a single table column for each field. This can be accomplished by an aspect design as the one sketched in what follows:

```
package org.apache.ojb.broker.accesslayer.sql.fuzzy; import
org.apache.ojb.broker.core.*;
//...
public aspect FuzzySqlFieldHandling{
```

```
    List around (SqlInsertStatement i):
        target(i) && args(cld) && args(buf)
        && call(List SqlInsertStatement.appendListOfColumns(
                  ClassDescriptor cld, StringBuffer buf)){
     List aux=null;
       try{
              aux =  proceed(m, cld, buf);
        }catch(Exception e){throw e;}
       if (cld.getClass().equals(
           new FuzzyNumericFieldDescription().getClass())){
           // add columns to SQL INSERT in buf...
           return l;
       }
    }
    // ...
}
```

The *around* advice is able to dynamically extend the behavior of the SQL–forming methods by manipulating parameters and return values. In common fuzzy extensions, this is basic a matter of providing additional definitions for holding the membership values.

3.4 Fuzzy querying through aspects

The abstract `SqlQueryStatement` class and its concrete subclass `SqlSelectStatement` together implement the generation of SQL SELECT clauses from queries. Queries for fuzzy classes and fuzzy attributes can be formed by extending `getStatement` invocations by using an *around* advice and an implementation technique similar to the one described above for insertions.

In addition, it is required that the membership degrees of the result collections are differentiated from standard attributes of the objects. To do so, query results returning from methods like `getCollectionByQuery` in `PersistenceBroker` need to be wrapped into objects representing pairs $(o, \mu_q(o))$. This must be done at the level of the implementation of the method `getCollectionByQuery` in `PersistenceBrokerImpl` since memberships come from the access layer as conventional retrieved database columns.

```
public aspect FuzzyObjectWrapping{
    Collection around (PersistenceBroker p):
        target(p) && args(cld) && args(buf)
        && call(Collection PersistenceBroker.getCollectionByQuery(Query query))
```

```
            {
        Collection aux=null;
          try{
               aux =  proceed(p, query);
         }catch(PersistenceBrokerException e){throw e;}
         return this.wrapResultCollection(aux);
    }
    // ...
}
```

The drawback of this approach is that it wraps every call to `getCollectionByQuery`, but it can be easily avoided by introducing a method in `PersistenceBroker` that allows for switching on/off the wrapping, which could be consulted in the code of the aspect (concretely, through parameter `p`). Moreover, fuzzy criteria instead of crisp ones can be introduced by extending the `Criteria` class and associated code. This can be accomplished alternatively by introducing additional methods to `Criteria` or by subclassing it.

Fuzzy query results can be processed by casting to a class representing the pairs, resulting in a programming idiom like that described in [21]:

```
it = e.fuzzyIterator(); while (it.hasNext()){
   FuzzyObject aux = (FuzzyObject) it.next();
   X anX = (X) aux.getObject();
   double mu = aux.getMembership();
   // do processing...
 }
```

4 Example: Adding Fuzzy Classes and Fuzzy Numbers to OJB

The general design issues provided in the previous section can be used to implement a wide range of fuzzy extensions. In this section, we focus on two simple extensions for the sake of illustration, providing some important implementation details. Concretely, we will extend `OJB` with fuzzy classes and fuzzy numbers as attributes of classes, so that simple flexible queries can be issued through standard means. The domain used as a case study is that of market segmentation under fuzziness, taken from [20]. A conceptual model for the basic definitions in the case study is provided in Figure 3.

In Figure 3 customers are instances of `CustomerBase`, and the estimated value of their relationship with the company is described from the marketing

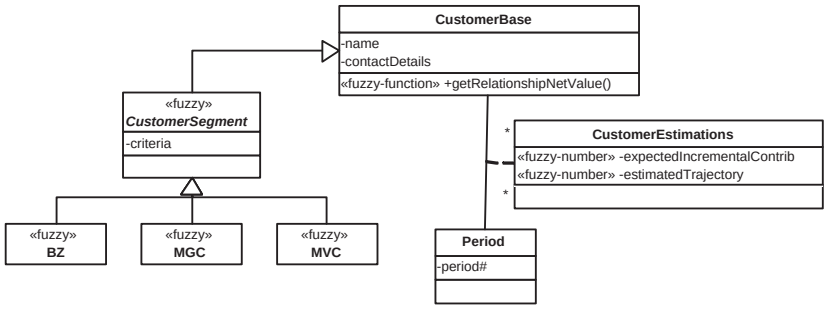

Figure 3: Main conceptual model elements in the market segmentation case study

perspective in terms of estimations about duration (loyalty) and estimated increase in purchase volume for each period in the medium–term forecast. Both estimations are imprecise fuzzy numbers (triangular for the sake of simplicity) as marked by the stereotype `<<fuzzy-number>>`. The net value of their relationship is computed from those estimations by an algorithm producing also imprecise results (`<<fuzzy-function>>`), and this value is used by a process of fuzzy clustering not covered here that produce fuzzy classes BZ (below zero), MGC (most growable customers) and MVC (most valuable customers).

The system then uses the values for a sequences of periods to compute the membership degree of each customer for each of the classes (as in [20]), storing it in a explicit, extensional way, since marketing experts are able to change them due to other factors that may affect the relationship trajectory.

The storage of the example is used by specifying an XML schema like the following:

```
<class-descriptor  class="CustomerEstimations"
    table="CUSTOMER_EST" >
    <field-descriptor
       name="expectedIncContrib"
       column="EXP" left="EXP0" right="EXP1" type="FuzzyNumeric"
       att-left="left" att-center="center" att-right="right"
       primarykey="false" />
 ...
</class-descriptor>
<class-descriptor class   class="MGC"
    table="MGC" type="fuzzy" extensional="true"
```

```
    membershipTable="CUSTOMER_BASE" membershipField="MGC">
  ...
 </class-descriptor>
```

Basically, schema definitions are OJB–like schemas with extended data mapping attributes and elements intended to be processed by `loadFuzzyDescriptorRepository`. Fuzzy attributes (numbers) are mapped in the simplest way, by explicitly declaring the properties of the class that hold the left, center and right points describing the fuzzy number, so that the appropriate `getX()` methods could be invoked through reflection in the access layer. More complex approaches can be devised to include classes representing by themselves fuzzy data types.

The storage of explicit membership degrees for imprecise segments is done through sentences like the following:

```
PersistenceBroker.store(cust,
computeMVC(cust.getRelationshipNetValue()), "MVC");
```

Where `computeWC` represents the classification criteria mentioned above. The fuzziness of attributes is stored automatically by the extended access layer as described above.

Queries can be issued to the persistence layer by standard means provided in the core interfaces. For example, the following query returns a fuzzy subset of "BZ" (a subset of the crisp class `CustomerBase`) that has values (approximately) greater than 4.3.

```
broker.wrapFuzzySubsets();
Criteria criteria = new Criteria();
criteria.addGreaterOrEqualThan("expectedIncContrib", new Double(4.3));
QueryByCriteria query = new QueryByCriteria(CustomerBase.class, criteria, "BZ");
Collection results = broker.getCollectionByQuery(query);
```

It should be noted that higher–level interfaces like `Jdo` can be used alternatively to carry out fuzzy queries. For example, the following `Jdo` query returns the fuzzy subset of MVC filtered (in a crisp way) by state and area.

```
String filter =
   "contactDetails.state == state && " +
   "contactDetails.area > area";
Extent extent = pm.getExtent(CustomerBase.class, true,
"asc;fuzzySubset=MVC"); Query query = pm.newFuzzyQuery(extent,
```

```
filter); ((FuzzyQuery)query).interpretAllFuzzy();
query.declareParameters(
   "String state, String area ");
Collection result = (Collection)query.execute(
   "Georgia", "200");
```

I the above example, the "MVC" string encoded in the parameters to `getExtent` is used to specify the class descriptor associated to the actual Java class `CustomerBase`, and the `interpretAllFuzzy()` method explicitly forces the wrapping of query results for membership processing, as described in the previous section.

5 Conclusions and Future Work

Fuzziness can be considered as a separate cross–cutting concern in existing software, and in consequence, AOD techniques provide a convenient framework to implement fuzzy extensions to existing libraries. The consideration of generic information imperfection concerns can be introduced at requirement analysis time, so that such early specification is later transformed to detailed design decisions.

As a proof of concept for such approach, the AOD extension of the `OJB` libraries using `aspectj` has been described, along with a concrete case study regarding implementations of class and attribute concerns for fuzziness. The resulting design combines inheritance and aspects to come up with an extension that entails no modifications to existing `OJB` source code. Concretely, metadata processing is weaved at initialization and querying times, and polymorphism is used to provide alternate metadata and query result processing idioms that handle fuzziness, achieving full backwards compatibility with existing client code. The approach taken are dependant on the non–public parts of the libraries, which entails that structural change of those parts will force a change in the aspects developed. However, since *refactoring* code is currently becoming standard practice in Software Engineering, the changes could be expected to be properly documented, serving as a blueprint for changing the extensions thereafter.

Future work is needed to implement with the same techniques the range of existing mathematical models and frameworks for imperfection handling. In addition, it may be desirable to seamlessly integrate fuzzy database mapping with conceptual modelling tools based on variants of the `UML`[17], which would result in improved traceability of the concerns of imperfection throughout the software life cycle.

From a broader perspective, further work is required in the layering and more convenient factoring of persistence interfaces for the many existing persistence interfaces. In any case, research on engineering information imperfection — and more concretely fuzziness — as aspects provides a promising direction in the application of fuzzy technologies to practical problems that require complex software.

References

[1] Atkinson, M. P., Daynes, L., Jordan, M.J., Printezis, T., Spence, S.: An Orthogonally Persistent Java. ACM Sigmod Record, 25, 4 (1996).

[2] Berzal, F., Cubero, J.C., Marín, N. and Pons, O.: Enabling Fuzzy Object Comparison in Modern Programming Platforms through Reflection. In: T. Bilgiç, B. de Baets and O. Kaynak (eds.): "Fuzzy Sets and Systems", Lecture Notes in Artificial Intelligence, LNAI 2715, pp. 660–667 Springer Verlag, 2003

[3] Bosc, P., Pivert, O.: Fuzzy Querying in Conventional Databases, In: Zadeh, L., Kacprzyk, J. (eds.): Fuzzy Logic for the Management of Uncertainty. John Wiley, New York (1992) 645–671

[4] Buckles, B.P., Petry, F.E: A Fuzzy Representation of Data for Relational Databases. Fuzzy Sets and Systems **7** (1982) 213–226

[5] Callens, B. de Tré, G., Verstraete, J., Hallez, A.: A Flexible Querying Framework (FQF): Some Implementation Issues. Lecture Notes on Computer Science 2869: Proceedings of International Symposium of Computer and Information Sciences (ISCIS) 2003, 260–267.

[6] Chen, G. Fuzzy logic in data modeling : semantics, constraints, and database design, Kluwer Academic Publishers, 1998.

[7] De Caluwe, R. (ed.): Fuzzy and Uncertain Object-Oriented Databases, Concepts and Models. World Scientific, Singapore (1997)

[8] Galindo, J., Medina, J.M., Pons, O., Cubero, J.C.: A Server for Fuzzy SQL Queries. In: Andreasen, T., Christiansen, H., Larsen, H.L. (eds.): Lecture Notes in Artificial Intelligence (LNAI), Vol. 1495, Springer-Verlag, Berlin Heildelberg New York (1998) 164–174

[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object Oriented Design. Addison Wesley (1995)

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: "An Overview of AspectJ". In: Proc. of the European Conference on Object-Oriented Programming (ECOOP) (2001)

[11] Grundy, J. Aspect-Oriented Requirements Engineering for Component-based Software Systems. In Proceedings of the 4th IEEE International Symposium on Requirements Engineering. IEEE Computer Society Press (1999): 84–91.

[12] Kaufmann, A., and M. M. Gupta. Introduction to fuzzy arithmetic: theory and applications. New York: Van Nostrand Reinhold, 1985.

[13] Klir, G., Wierman, M.: Uncertainty-Based Information: Elements of Generalized Information Theory. Springer-Verlag (1998).

[14] Rashid, A., Sawyer, P., Moreira, A. and Araujo, J. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In Proceedings of the IEEE Joint International Conference on Requirements Engineering. IEEE Computer Society Press. (2002): 199–202.

[15] Russell, C. et al (2001). Java Data Objects (JDO) Version 1.0 proposed final draft, Java Specification Request JSR000012 (2001).

[16] Sarwar, B. M., Karypis, G., Konstan, J. A., and Riedl, J. "Analysis of Recommender Algorithms for E-Commerce". In proceedings of the ACM E-Commerce 2000 Conference. Oct. 17-20, 2000, pp. 158-167.

[17] Sicilia, M. A., García-Barriocnal, E., Gutiérrez, J. A. (2002). Integrating fuzziness in object oriented modelling languages: towards a fuzzy-UML. In: Proceedings of the International Conference on Fuzzy Sets Theory and its Applications (FSTA 2002), 66-67.

[18] Sicilia, M.A., García-Barriocanal, E., Díaz, P. and Aedo, I.: Extending Relational Data Access Programming Libraries for Fuzziness: The fJDBC Framework. In: Proceedings of the 5th International Conference on Flexible Query Answering Systems. Lecture Notes in Computer Science 2522, Springer (2002):314–328

[19] Sicilia, M.A., Gutiérrez, J.A., García-Barriocanal, E. (2002). Designing Fuzzy Relations in Orthogonal Persistence Object-Oriented Database Engines. Advances in Artificial Intelligence - IBERAMIA 2002, Lecture Notes in Computer Science 2527 Springer, 243-253

[20] Sicilia, M.A., García-Barriocanal, E. On Fuzziness in Relationship Value Segmentation: Applications to Personalized e-Commerce. ACM SIGECOM Newsletter, 4(2):1–10.

[21] Sicilia, M.A., García-Barriocanal, E., Gutiérrez, J.A. (2004). Introducing Fuzziness in Existing Orthogonal Persistence Interfaces and Systems. In: Advances in Fuzzy Object-Oriented Databases: Modeling and Applications, IDEA Group Publishing (to appear).

[22] Sicilia, M.A., García-Barriocanal, E. (2004) On imperfection in information as an "early" crosscutting concern and its mapping to aspect-oriented design. Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop. Vancouver

[23] Sicilia, M.A., García-Barriocanal, E. (2006) Extending object database interfaces with fuzziness through aspect–oriented design. *ACM SIGMOD record*, June 2006.

[24] Smets, P.: Imperfect information: Imprecision-Uncertainty. Uncertainty Management in Information Systems: From Needs to Solutions. Kluwer Academic Publishers (1997), 225-254.

[25] Sutton Jr., S.M. and Rouvellou, I. Modeling Software Concerns in Cosmos. In *Proceedings of the First International Conference on Aspect–Oriented Software Development* (AOSD 2002), Enschede, The Netherlands, Apr. 2002, ACM Press, 127-133.

[26] Sutton Jr, S.M. and Rouvellou, I.: Applicability of Categorization Theory to Multidimensional Separation of Concerns. In: Proceedings of the Workshop on Advanced Separation of Concerns, OOPSLA 2001, October 14, 2001, Tampa, Florida.

[27] Sutton Jr., S. M. and Tarr, P.: "Aspect-Oriented Design Needs Concern Modeling". In: Proc. of the Aspect Oriented Design Workshop on Identifying, Separating and Verifying Concerns in the Design (2002), Enschede, The Netherlands.

[28] Tekinerdogan, B. ASAAM: Aspectual Software Architecture Analysis Method. In Proceedings of the Aspect–Oriented Requirements Engineering and Architecture Design Workshop, Boston, US, 2002.

[29] Wedel, M., Kamakura, W.A. 1999. Market Segmentation: Conceptual and Methodological Foundations, Kluwer Academic Publishers, 2nd edition.

[30] Yazici, A., George, R., Aksoy, D. (1998). Design and Implementation Issues in the Fuzzy Object-Oriented Data Model. Information Sciences, 108, 1-4, 241–260

[31] Zadrozny, S., Kacprzyk, J: FQUERY for Access: Towards Human Consistent Querying User Interfaces. In: Proceedings of the 1996 ACM Symposium on Applied Computing (SAC'96) (1996) 532–536

Table 1: Example imperfection–related concerns in a basic market segmentation setting.

Instances	Classes/classifications	Description
Expected-Incremental-Purchases	SubjectiveUncertainElement, Domain-Imperfection, ImperfectAttribute	Expectations reflect uncertainty regarding the future, and they are estimated in this case in a purely subjective way. The uncertainty can be considered to be inherent to the domain of forecasting, and they are represented as uncertain values, thus modelled as attributes.
Estimated-Relationship-Duration	SubjectiveUncertainElement, Domain-Imperfection, ImperfectAttribute	The same as the above.
Relationship-Value	ImpreciseElement, System-Imperfection, ImperfectResult	The value of a relationship is the result of a computation involving the above elements. In this case, the value is used as certain but with a degree of imprecision that comes from the uncertainty in its components, and the subjective adjustment of the parameters of the computation algorithm. It is the system that computes the values which produces the degree of imprecision.
Customer-Segment	FuzzyElement, System-Imperfection, ImperfectClass	Segments are fuzzy sets of customers computed from available data.
Customer-Similarity	FuzzyElement, System-Imperfection, ImperfectAssociation	Similarity is represented as a relation between pairs of customers, with a given level of "strength" imprecisely computed from the segments and other available data.
Net-Relationship-Value-Model	FuzzyElement, System-Imperfection, ImperfectFunction	This represents an algorithm that computes Relationship-Values during time with estimated values for discount and inflation.