Mixing Classes and Prototypes: An Object Oriented Approach to Semantic Image modelling

Youssef Lahlou

GMD, German National Research Center for Information Technology

E-Mail: lahlou@gmd.de

Classical object oriented database management systems fail in semantic modelling of images, because such objects are not easily definable in terms of conceptual structures (classes, database schemes, ...).

Besides, object-oriented languages have evolved in various directions, including but not limited to class-based languages. Prototype-based languages represent a serious alternative to class-based ones and "organizing programs without classes" (dixit the Self language designers) is a radically different approach to object-oriented programming.

In this paper, we show that prototype-like languages can be of a great importance in semantic image modelling if the class paradigm is maintained for the sake of retrieving large collections of objects.

Semantic image modelling requires describing image structure and content within a suitable model which should provide structures and constructs capable of managing the rich variety of possible image features.

The object oriented paradigm has shown itself very suitable in most new database applications. However, adapting it to semantic image modelling is not straightforward since images are a very special case of complex objects because they can not be easily modelled through only simple use of hierarchical structures, or abstraction into class structures. Common applications more often require object content indexing rather than grouping into well-structured classes. Furthermore, unstructured objects may also refer to structured objects and vice versa, especially in multimedia environments.

Object-oriented databases make use of the class-based view of object orientation. This means that objects in such databases are structured into classes. Each object belongs to a class which abstracts its structure into a "conceptual structure" (an intensional definition of the object structure). Thus, before creating objects, a conceptual schema has to be designed to capture object structures. Concrete objects are then created with respect to the conceptual schema (they are instances of some classes).

This particular view of object-orientation is well suited for managing large collections of objects, since the conceptual schema is an invaluable source of information for querying the database, specifying and checking integrity constraints, query processing, indexing, and so on. But, in certain cases this schema is hard to design, when object structures can not be predicted before creation. This is true for image databases since semantic image content varies a lot from one image to another.

On the other hand, the prototype-based approach to object-orientation has shown itself very useful in object-oriented languages. There is no notion of class in this approach. Objects are created either *ex-nihilo* or by “cloning” an existing object. Each object can then evolve independantly from the one it has been cloned from, especially by adding new features (attributes, methods) to its structure.

The main feature of prototype-based languages is the absence of classes. Object structures are not abstracted into some higher level conceptual entities. This feature frees the user from having to predict object structures before actually creating them.

However, when retrieving large collections of objects, the absence of classes is a major drawback, since nothing is known *a priori* about object structures and they have to be fully checked to respond to a query.

We propose a hybrid model that is able to cope with the representation and the manipulation of both unstructured and structured objects (by structured objects we mean objects for which abstract structure can be predicted within a conceptual schema). This model is particularily suited for image modelling since such objects have both common and individual features.

The model draws its inspiration from both class-based and prototype-based systems, in the sense that objects are tied to classes which only abstract a minimal structure implemented by the related objects. Each object can in turn implement an extra individual structure independantly from its class.

We thus relax the traditional instantiation link between an object and its class, and rebaptize it the *realization link*. A class is no longer a set of objects having the same structure; it is only a minimal structure that have to be implemented by each object tied to it.

<u>Examples:</u>
$Image : [author : Photographer,\ date : Date,\ size : 2D\_size]$ is a class.
A specific class $Monument\_image$ can be defined by augmenting the $Image$ class with the attribute $monument : Monument$.

Realization is a mechanism that links an object to a class. In other words, an object can never exist without a class to which it is tied. The mechanism of realization can be seen as the operation of giving a value to the attributes of the class by assigning an object to each of them. **The possibility is then left for objects to have in turn other additional objects in their structure, besides those of the class.** This enables objects to have individual structures, whence the improved flexibility and extensiveness of the model.

<u>Examples:</u>
$o_1 : [author : p_1,\ date : d_1,\ size : s_1,\ \mathbf{none} : o]$ is an object that might be a realization of class $Image$, if p_1, d_1 and s_1 are respectively realizations of $Photographer$, $Date$ and $2D\_size$ classes. An additional object named o is composing o_1 with no particular role (whence a *none* attribute).
$o_2 : [author : p_2,\ date : d_2,\ size : s_2,\ monument : m_2]$ is an object that might be a realization of class $Monument\_image$, if m_2 is a realization of class $Monument$.

The main problems arising from this model appear when it comes to querying the database. In classical structured models (relational, semantic, object-oriented), query formulation is based on class structures, assuming that objects only instanciate those structures with other objects or values.

So, when querying a database for 50 years old employees, living in Bonn, the user is aware *a priori* of the existence of a class named *Employee*, representing employees and having an attribute giving their age (say *age*) and an other giving the town they live in (say *address*). The query can then be expressed by the set:

$$\{o \in Employee, o.age = 50 \wedge o.address.town = "Bonn"\}$$

As is the case in structured data models, this kind of query can also be specified in our model, since each object realizing the *Employee* class would have an *age* attribute, and an *address* attribute, and each object realizing the class *Address* would have a *town* attribute. But, in order to fully make use of the power of the realization link, queries have to deal also with the additional part of object structures that is not in their class structures. The problem is that this structure is not known at the conceptual level; for instance, the object o_1, realizing the *Image* class uses, in its (additional individual) structure, the object o, realizing, say, the *Employee* class. This is not a mandatory (predictable) reference in the *Image* class.

Thus, the main question is how to make the user be able to use individual object components in queries so that he could specify such queries as images of 50 years old employees, living in Bonn?

In our model, a query is a quadruple $q = (c, Cl, Q, p)$ where:

- c is a class, called *target* of q,
- $Cl = \{cl_1, ..., cl_n\}$ is a set of clauses, called *criteria* of q; a clause is a logical expression based on litterals and valid paths for c (e.g. $address.town = “Bonn''$).
- $Q = \{q_1, ..., q_m\}$, is a set of queries, called *sub-queries* of q,
- p is a valid path for c, called *projection* of q.

Criteria, sub-queries and/or projection may be empty.

The semantics of query q is that it looks for objects realizing class c, satisfying all clause members of Cl and referencing, for each query member of Q, at least one object resulting from this query. The result of q is the set of path p destinations for all objects satisfying those three conditions.

Examples:
Query $q_1 = (c_1, Cl_1, Q_1, p_1)$ looks for 50 years old employees, living in Bonn.
$c_1 = Employee$
$Cl_1 = \{age = 50, address.town = “Bonn''\}$
$Q_1 = \emptyset$
p_1 is empty.

Query $q_2 = (c_2, Cl_2, Q_2, p_2)$ looks for author names of images where such employees appear.
$c_2 = Image$
$Cl_2 = \emptyset$
$Q_2 = \{q_1\}$
$p_2 = author.name.$

The realization link is taken into account in our query specification language, within the set Q, which specifies criteria on all referenced objects of the main object (those predicted in the class and those proper to the object). Conceptual knowledge on objects, coming from their class structure is used in the set Cl.

To validate our approach, we designed a prototype system within the Smalltalk-80 object oriented programming environment on a SPARC station.