Encapsulating Classification in an OODBMS for Data Mining Applications

Lina Al-Jadir
American University of Beirut
Department of Mathematics and Computer Science
P.O. Box 11-0236, Beirut, Lebanon
lina.al-jadir@aub.edu.lb

Abstract

Classification is an important task in data mining. Encapsulating classification in an object-oriented database system requires additional features: we propose multiobjects and schema evolution. Our approach allows us to store classification functions, and to store instances of each group in order to retrieve them later. Since the database is operational, it allows us also to perform dynamic classification, i.e. add/remove instances to/from groups over time. Moreover, it allows us to update classification functions (if we choose another population sample or apply another classifier) and have the instances of groups consequently reclassified. We illustrate our approach with a target mailing application.

1. Introduction

Data mining [9] consists in finding interesting trends in large databases in order to guide decisions about future activities. *Classification* is one of the tasks of data mining and is defined in [1] as follows. We are given a large population database. The population is known to comprise of m groups, but the population instances are not labeled with the group identification. Also given is a population sample in which the group labels of the instances are known. A *classifier* generates a classification function for each group that can be used to retrieve all instances of a specified group from the population database. Classification has numerous applications including target mailing, credit approval, and medical diagnosis.

In a target mailing application, a history of responses to various promotions is maintained. Based on this response history, classification functions are developed for identifying new candidates for future promotions. For example, a car company stores information about all its customers (*population database*): name, address, profession, gender, birthdate, civil status, number of children (*attributes*). This year, new car models are available on market with a catalog for each. Each model belongs to one of the following categories: sport cars, family cars, and city cars. The company sends a questionnaire to its customers who bought a car last year (*population sample*, a subset of the population database) asking them about their expectations and the car categories they are or may be interested in (additional attribute interest which is multivalued). Based on the customer responses, the company identifies the profile for the customers of each car category (*classification function* for each *group*). For instance, the customer profile for sport cars is found to be: gender is male, and age is between 25 and 35, and address is in Champel (a luxurious area). Instead of sending all the catalogs to all its customers, the company sends a catalog only to the potential buyers of the corresponding category.

Several classifiers have been proposed in the literature based on neural nets or decision trees [1] [13] [19] [11] [21]. The authors in [1] argue that "classification should be encapsulated as part of future database systems". This issue is not addressed in the mentioned classification papers which present classifier algorithms. In this paper we address this issue. We assume that we already have a classifier and are interested in the following tasks:

- We would like to store the classification functions (generated by the classifier) in the database, as well as the instances of each group in order to be able to retrieve them fast later. How and where to store the classification functions ? How and where to store all instances of each group ?
- Since the database is operational and is being used, we would like to classify new customers according to the stored classification functions. We would like also to reclassify existing customers because they may become/ cease to be candidates of a car category. How to perform dynamic classification ? How to handle the fact that the set of instances of a group may have new members or lose members over time ?

0-7695-0996-7/01 $10.00 © 2001 IEEE

- We would like to be able to take another classifier (with the same population sample) or another population sample (with the same classifier), get new classification functions and apply them to the database. How to take into account the new classification functions ? How will this affect the database ?

Encapsulating classification in an object-oriented database system (OODBMS) requires additional features of that system: we propose multiobjects and schema evolution.

The remainder of the paper is organized as follows. We start by illustrating our approach with an example in Section 2. Then we present our approach in sections 3 and 4. Section 3 describes multiobjects in the F2 object-oriented database system and shows its use for classification. Section 4 describes schema evolution in F2 and shows its use for classification. Section 5 concludes the paper. F2 is a general purpose database system developed at C.U.I. (University of Geneva) and used to experiment several features such as: updatable views, information system design methods, knowledge databases, database integration, schema evolution. It is written in Ada and runs under SunOS, DEC/ALPHA, MacOS and Windows 95/98.

2. Illustrative example

Our approach to encapsulate classification in an OODBMS is based on multiobjects and schema evolution. We illustrate it with the example introduced in section 1. The initial database schema of the car company contains the classes *Category*, *Model*, *Car*, *Customer* (see fig. 1). *Car* has the following attributes: *car_nb*, *model* (of domain *Model*), *color*, *options*, *manufacture_date*, *sale_date*. *Customer* has the following attributes: *customer_nb*, *name*, *address*, *profession*, *gender*, *birthdate*, *civil_status*, *number_children*, *car_bought* (multivalued attribute of domain *Car*). It contains objects representing all customers. For example, the customer number 2030 (named Laurent Bonjour) bought the car number 111981 which is a Peugeot 205 on November 10, 1995. The model Peugeot 205 belongs to the city cars (category).

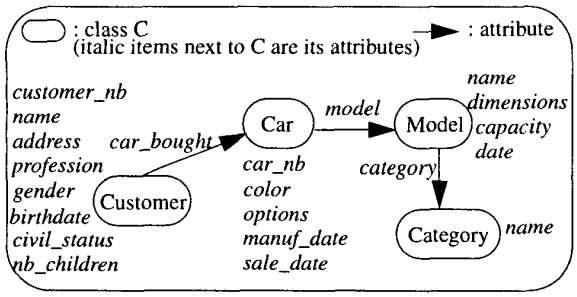

Figure 1. Initial schema.

Situation 1: In order to have a population sample, the database (DB) administrator adds to the schema the class *Buyer_1999* as a subclass of *Customer* with a specialization constraint on it: *car_bought containsAnObjectOf {c in Car s.t. year(c.sale_date) = 1999}*. This subclass will be automatically populated by objects of *Customer* satisfying its specialization constraint. Consequently, a customer who bought a car in 1999 will be implemented by a multiobject (set of objects) composed of two objects, $b_{Customer}$ in *Customer* and b_{Buyer_1999} in *Buyer_1999*, sharing the same instance identifier. In order to store the interests of those buyers according to the questionnaire responses, the administrator adds an attribute *interest* to the class *Buyer_1999*, which is multivalued and whose domain is *Category*. The database with the updated schema is shown in figure 2.

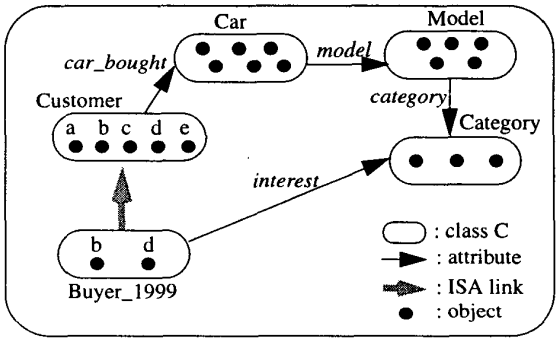

Figure 2. Database with updated schema.

As we see, the administrator needs a DBMS which supports schema evolution, i.e. allows to update the schema of a populated database, and offers among its schema changes: add a subclass with specialization constraints, and add an attribute to a class. Moreover the DBMS should support multiobjects, and object classification when adding a subclass.

Situation 2: The DB administrator applies the classifier (which may be implemented as a method) on the class *Buyer_1999* and gets for each group a classification function. For each group, he/she adds to the schema a subclass of *Customer* (*Candidate_sport*, *Candidate_family*, and *Candidate_city*) with specialization constraints corresponding to its classification function. Each of these subclasses will be automatically populated by objects of *Customer* satisfying its specialization constraints. For example, a customer who is a candidate for family cars and city cars will be implemented by a multiobject composed of three objects, $c_{Customer}$ in *Customer*, $c_{Candidate\_family}$ in *Candidate_family* and $c_{Candidate\_city}$ in *Candidate_city*, sharing the same instance identifier. The database after classification is shown in figure 3.

Here again, the administrator needs a DBMS supporting schema evolution and multiobjects.

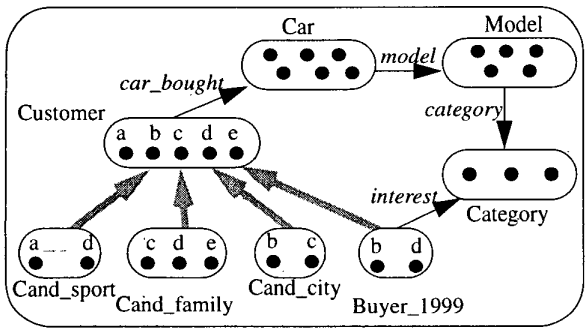

Figure 3. Database after classification.

Situation 3: In January 2000, the company sends catalogs to the candidates of the corresponding categories. In June 2000, the company offers a promotion and wants to send the catalogs again. During the last six months, the company may have attracted new customers and they should have been classified as candidates in order to send them catalogs. During that period, the database continued to be used, queried and manipulated. When the administrator added a multiobject to *Customer*, it was automatically classified in *Candidate_sport*, *Candidate_family*, *Candidate_city*, according to its values and to the specialization constraints of these subclasses.

As we see, the administrator needs a DBMS which supports automatic classification when creating a multiobject.

Situation 4: Let the customer profile for sport cars be: gender is male, and age is between 25 and 35, and address is in Champel. Marc Dupont used to live in Champel, and was found a candidate when the classification was applied in December 1999. Two months later, Dupont moves to Saconnex (another area) and sends the car company his new address. He ceases to be a candidate for sport cars for the next promotion. The database administrator updates the address of the multiobject representing Dupont. Consequently, it is automatically removed from *Candidate_sport* since it does not satisfy anymore the specialization constraints of this subclass.

The DB administrator needs a DBMS which supports object migration when updating a multiobject's attribute value (since it may satisfy or cease to satisfy some specialization constraints).

Situation 5: In December 2000, the car company sends questionnaires to customers who bought a car in 2000 in order to identify the candidates in 2001. The DB administrator takes another population sample (buyers in 2000), gets the questionnaire responses, and applies the classifier again. He/she may get three other classification functions. For example, in 2000 older customers tended to buy sport cars (age is between 27 and 37 instead of being between 25 and 35). The administrator replaces the specialization constraints of *Candidate_sport*, *Candidate_family* and *Candidate_city* by new ones corresponding to the newly obtained classification functions. The *Customer* objects are automatically reclassified.

The administrator needs a DBMS which supports among its schema changes: add and delete a specialization constraint of a subclass with object reclassification.

Situation 6: Two months later, the car company stops selling city cars. The DB administrator deletes the *Candidate_city* subclass without losing the customers who were city cars candidates.

The administrator needs a DBMS which supports deleting a subclass while retaining its objects in its superclass.

3. Multiobjects

We describe in this section the multiobject mechanism (introduced in [4]) which is implemented in the F2 database system, and show how it allows on-line classification in our car company application.

3.1. Defining a multiobject

In the F2 model [6] an *object* o_C is an instance of *class* C and has the oid $<id_C, id_o>$ where id_C is the class identifier and id_o the instance identifier within C. Objects structure is defined by class attributes. Objects behaviour is defined by primitive methods and triggered methods. A class, called *subclass*, can be declared as a specialization of another class called *superclass*. The class hierarchy is a forest, i.e. a set of *specialization trees*: a subclass has only one superclass (single inheritance), and there is not a root system-defined class. On a subclass may be defined *specialization constraints*. An object belongs to a subclass if and only if it satisfies the specialization constraints of the subclass. The *ancestors* of a class are its direct and indirect superclasses. The *descendants* of a class are its direct and indirect subclasses.

We assume that the reality consists of *entities*. Entities have several facets. For example, a human being may be seen as a person, an employee, a tennis player, a student, etc. An entity is implemented in the multiobject mechanism by a set of objects in distinct classes of a specialization tree, $M_o = \{o_{C1}, o_{C2}, ..., o_{Cn}\}$, called *multiobject*. Each object o_{Ci} denotes a facet of the entity and carries data specific to its corresponding class C_i. All the objects of M_o have the same instance identifier. A multiobject M_o satisfies the following constraint: if o_{Ci}, $1 \le i \le n$, belongs to M_o and C_i is a subclass of C_j then there must be an object o_{Cj}, $1 \le j \le n$ and $j \ne i$, which belongs to M_o. In other words, if an entity

possesses an object in class C, then the entity must also possess objects for all the ancestors of C. For example (see fig. 4), the class *Candidate_sport* is a subclass of *Customer*. A candidate for sport cars is implemented by a multiobject containing two objects $o_{Candidate\_sport}$ in *Candidate_sport* and $o_{Customer}$ in *Customer*.

Subclasses can be inclusive, i.e. a multiobject may contain two objects o_{Ci} and o_{Cj} where C_i and C_j are sibling classes. For example (see fig. 4), the class *Customer* has another subclass *Candidate_family*. A customer who is a candidate for sport cars *and* family cars is implemented by a multiobject containing three objects: $o_{Customer}$ in *Customer*, $o_{Candidate\_sport}$ in *Candidate_sport* and $o_{Candidate\_family}$ in *Candidate_family*.

In the multiobject mechanism, attributes are not inherited in the classical sense of inheritance; they are *reached* by navigating in a specialization tree. For the objects of a subclass C, only the values on attributes locally defined at C are stored. The values on attributes defined at the superclass S of C are not stored with C objects but with their related S objects (i.e. of the same multiobject). For example (see fig. 4), *name* is a local attribute of class *Customer*. If the name of $o_{Customer}$ is "Marc Dupont" and $o_{Candidate\_sport}$ in *Candidate_sport* is related to $o_{Customer}$ then $o_{Candidate\_sport}$ is named Marc Dupont. The *name* values are stored with *Customer* objects.

In the multiobject mechanism, specialization constraints are evaluated when: (i) a multiobject is created or updated (§3.2.), (ii) a specialization constraint is added to or removed from a subclass (§4.2.).

3.2. Manipulating a multiobject

A multiobject can be created, deleted, and updated. The detailed algorithms of these primitive methods can be found in [4].

Creating a multiobject. The *create(C, [a_1:v_1, a_2:v_2, ..., a_p:v_p])* primitive method creates a multiobject including an object in class C. The create algorithm searches the classes of the multiobject in the specialization tree of C (beginning from the root), $SC = \{C_1, C_2, ..., C_n\}$, according to the attribute values $[a_1:v_1, a_2:v_2, ..., a_p:v_p]$ and to the classes' specialization constraints. If C does not belong to SC or if the origin class of one of the attributes a_j does not belong to SC, an error is returned. Otherwise, an object o_{Ci} is added to each class C_i of SC and all these objects carry the same instance identifier. Each attribute value is stored with the object o_{Ci} which belongs to the origin class of the attribute.

For example, in situation 3 of §2., the class *Customer* has the subclasses *Candidate_sport*, *Candidate_family*, and *Candidate_city* with specialization constraints on each (see fig. 4.a). The DB administrator adds a new customer Marc Dupont to the database. The following expression (in F2-DML) creates a multiobject containing three objects: $o_{Customer}$ in *Customer* (root class of the specialization tree), $o_{Candidate\_sport}$ in *Candidate_sport* (the constraints *gender = "m" and age > 25 and age < 35 and area = "Champel"* are satisfied[1]), $o_{Candidate\_family}$ in *Candidate_family* (the constraints *civil_status = "married" and age > 30 and age < 55* are satisfied) (see fig. 4.b).

$o_{Customer}$:= **create** Customer' [customer_nb: 2222, name: "Marc Dupont", address: "12 rue Miremont, Champel", profession: "liberal", gender: "m", birthdate: "01-JAN-1968", civil_status: "married", nb_children: 1];

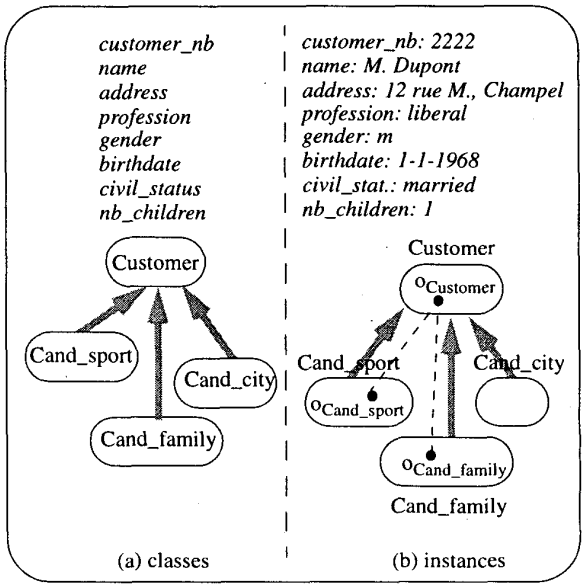

Figure 4. Creating a multiobject.

Deleting a multiobject. The *delete(o_C)* primitive method deletes the multiobject containing the object o_C, i.e. it removes o_C and *all* its related objects. The delete algorithm searches the classes of the multiobject in the specialization tree of C (beginning from the root), $SC = \{C_1, C_2, ..., C_n\}$, and removes its object o_{Ci} from each class C_i of SC.

Updating a multiobject. The *update(o_C, [att:val])* primitive method sets the value of the multiobject containing the object o_C on the attribute *att* to *val*. Since the attribute *att* could be used in specialization constraints on the descendants *SD* of its origin class *Orig*, the multiobject may gain new objects or/and lose existing objects in *SD* be-

[1] We assume that there is a method computing the age of a customer according to his/her birthdate, and a method returning the area according to the customer's address.

cause it may now (with the new value *val*) validate or invalidate those specialization constraints. This is referred to as object migration. The update algorithm searches in the specialization tree of *C*, beginning from *Orig*: (i) the set of gained classes and adds an object (carrying the same instance identifier as o_C) to each of them; (ii) the set of lost classes and removes the related object to o_C from each of them.

For example, in situation 4 of §2., Dupont moves from Champel to Saconnex. The following expression (in F2-DML) expresses that he has a new address. As a result (see fig. 5), (i) $o_{Candidate\_sport}$ is automatically removed from *Candidate_sport*, (ii) $o_{Candidate\_city}$ is automatically added to *Candidate_city* (the constraint $area \neq$ *"Champel"* is satisfied). The multiobject contains now the objects $\{o_{Customer}, o_{Candidate\_family}, o_{Candidate\_city}\}$.

update $o_{Customer}$ address: "24 ch. Coudriers, Saconnex";

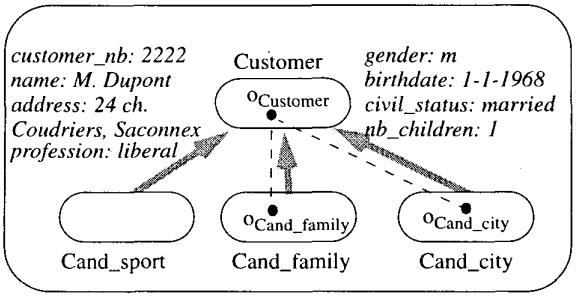

Figure 5. Updating a multiobject (address attribute).

4. Schema evolution

We give in this section the framework of schema evolution in the F2 OODBMS and then describe four schema changes that are used in our car company application.

4.1. Framework of schema evolution in F2

Set of schema changes in F2. An important feature of the F2 DBMS is the uniformity of its objects described in [6] [5]. We consider objects of three levels: database objects, schema objects and meta-schema objects. Uniformity of objects in F2 includes:

– uniformity of representation. The same structures are used in F2 to represent database objects, schema objects and meta-schema objects;

– uniformity of access and manipulation. The same primitive methods are used in F2 to access and manipulate database objects, schema objects and meta-schema objects.

Thanks to the uniformity of the F2 DBMS, we built the set of schema changes in F2 as follows [6] [5]: for *each* class of the F2 meta-schema we apply the primitive methods *create*, *delete* and *update* on its objects (see fig. 6).

(1) Create a new class
  (1.1) Create an atomic class
  (1.2) Create a tuple class
  (1.3) Create a tuple subclass
(2) Delete an existing class
(3) Update an existing class
  (3.1) Change its name
  (3.2) Change its interval if atomic class
  (3.3) Change its maximal length if atomic string class
  (3.4) Change its superclass
  (3.5) Make it a subclass, i.e. attach it to a spec. tree
  (3.6) Make it a non-subclass, i.e. detach it from a spec. tree
(4) Create a new attribute of a class
(5) Delete an existing attribute
(6) Update an existing attribute
  (6.1) Change its name
  (6.2) Change its maximal cardinality
  (6.3) Change its minimal cardinality
  (6.4) Change its domain class
  (6.5) Change its origin class
(7) Create a new key of a class
(8) Delete an existing key
(9) Update an existing key
  (9.1) Change the class on which it is defined
  (9.2) Change its attributes
  (9.3) Enable / disable it
(10) Create a new specialization constraint
(11) Delete an existing spec. constraint
(12) Update an existing spec. constraint
  (12.1) Change its name
  (12.2) Change the list of subclasses on which it is defined
(13) Create a new trigger
(14) Delete an existing trigger
(15) Update an existing trigger
  (15.1) Change the event for which it is defined
  (15.2) Change the list of methods it triggers
(16) Create a new event
(17) Delete an existing event
(18) Update an existing event
  (18.1) Change the class on which it is defined
  (18.2) Change its kind
  (18.3) Change its attribute
(19) Create a new triggered method
(20) Delete an existing triggered method
(21) Update an existing triggered method
  (21.1) Change its name

Figure 6. F2 schema changes.

Semantics of schema changes. We defined the semantics of each schema change in F2 with pre-conditions and

post-actions [6] [5] such that the F2 model invariants are preserved. Pre-conditions must be satisfied to allow a schema change to occur; otherwise it is rejected. Post-actions are repercussions to be executed on schema objects and database objects in order to keep the database structurally consistent. We implemented pre-conditions and post-actions by triggered methods [6] [5].

Propagation of schema changes. In F2 schema changes are propagated immediately [5] [3], i.e. the repercussions of a schema change are executed as soon as the schema change is performed.

4.2. Examples of schema changes

Create a subclass. In situation 2 of §2., the schema contains the class *Customer* which has several attributes including *address*. *Customer* has five objects $\{a_{Customer}, b_{Customer}, c_{Customer}, d_{Customer}, e_{Customer}\}$ (see fig. 7.a), which take the value Champel, Meyrin, Servette, Champel, Champel, respectively on the *address* attribute. After applying the classifier on the population sample, the DB administrator creates the class *Candidate_city* as a subclass of *Customer* and adds the specialization constraint *area* ≠ *"Champel"* (according to the obtained classification function) on it. As a result of this schema change, the objects $b_{Candidate\_city}$ and $c_{Candidate\_city}$ are automatically added to the *Candidate_city* subclass because they satisfy its specialization constraint (see fig. 7.b). Each of the multiobjects *b* and *c* contain now two objects. The address values (and other *Customer* attribute values) remain stored within the objects $b_{Customer}$ and $c_{Customer}$; they are not copied but reached by the objects $b_{Candidate\_city}$ and $c_{Candidate\_city}$.

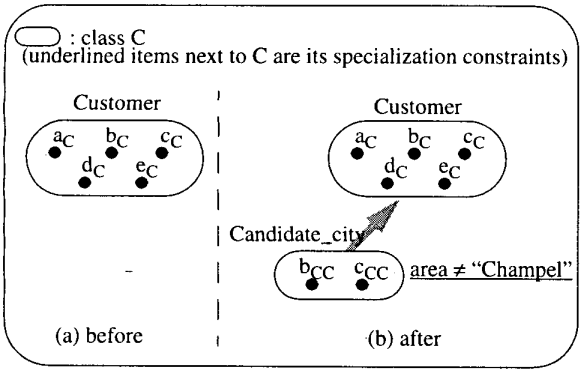

Figure 7. Adding the Candidate_city subclass and a specialization constraint on it.

Create an attribute. Situation 1 of §2. is similar to situation 2. The DB administrator creates the class *Buyer_1999* as a subclass of *Customer* and adds a specialization constraint on it, in order to have a population sample. Consequently, this subclass is populated by objects representing customers who bought a car in 1999. In addition, the administrator adds the attribute *interest* to this subclass. As a consequence, all objects of *Buyer_1999* take the null value on it. Then the administrator updates their value on *interest* according to the questionnaire responses.

Delete a class. In situation 6 of §2., the class *Candidate_city* is a subclass of *Customer*. It has two objects $b_{Candidate\_city}$ and $c_{Candidate\_city}$ which are related to $b_{Customer}$ and $c_{Customer}$ respectively in *Customer* (see fig. 7.b). The DB administrator deletes this class because the car company stops selling city cars. As a result of this schema change, the objects $\{b_{Candidate\_city}, c_{Candidate\_city}\}$ are removed while their related objects $\{b_{Customer}, c_{Customer}\}$ remain in *Customer* (see fig. 7.a). This means that the company does not lose its customers who were city cars candidates.

Create/delete a specialization constraint. In situation 5 of §2., the DB administrator applies the classifier on another population sample and gets other classification functions. He/she replaces (delete followed by create) the specialization constraints of the *Candidate_sport* subclass by *gender = "m" and age > 27 and age < 37 and area = "Champel"*. As a result of this schema change, the object $e_{Candidate\_sport}$ (age: 36, male, lives in Champel) is automatically added to the class *Candidate_sport* while the object $a_{Candidate\_sport}$ (age: 26, male, lives in Champel) is removed from it (see fig. 8.b). The object $d_{Candidate\_sport}$ (age: 31, male, lives in Champel) remains in *Candidate_sport*.

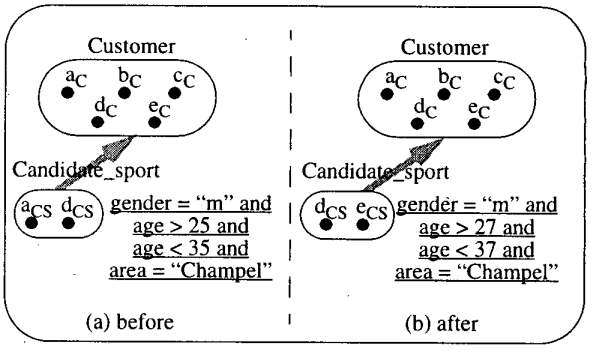

Figure 8. Replacing the specialization constraints of the Candidate_sport subclass.

5. Conclusion

Classification is an important task in data mining. Applying a classifier on a database is not enough. We need to store the result of the classification process and to continue to use the classified database. We proposed to use schema evolution and multiobjects in order to encapsulate classification in an OODBMS. We showed that our approach allows us to store classification functions as specialization constraints. It allows us to store all instances of a group as objects of a subclass. It allows us to add and remove instances of a group over time by creating or updating multiobjects. It allows us to modify the stored classification functions by creating/deleting specialization constraints and have the group instances accordingly reclassified. We illustrated our approach with a target mailing application.

Multiobjects and schema evolution are implemented in the F2 database system. F2 supports a non-classical transposed storage [3]. We ran the OO7 benchmark on F2 as well as a benchmark for schema evolution and obtained interesting results in [3].

Future work consists of keeping a history of classifications, i.e. keeping track of which classifier was used, when it was used, on which population sample it was used, and its resulting classification functions.

References

1. Agrawal R., Ghosh S., Imielinski T., Iyer B., Swami A., An Interval Classifier for Database Mining Applications, *Proc. Int. Conf. on Very Large Data Bases, VLDB*, Vancouver 1992.
2. Albano A., Bergamini R., Ghelli G., Orsini R., An Object Data Model with Roles, *Proc. Int. Conf. on Very Large Data Bases, VLDB*, Dublin 1993.
3. Al-Jadir L., Léonard M., Transposed Storage of an Object Database to Reduce the Cost of Schema Changes, *Proc. ER'99 Int. Workshop on Evolution and Change in Data Management, ECDM*, Paris 1999.
4. Al-Jadir L., Léonard M., Multiobjects to Ease Schema Evolution in an OODBMS, *Proc. Int. Conf. on Conceptual Modeling, ER*, Singapore 1998.
5. Al-Jadir L., *Evolution-Oriented Database Systems*, Ph.D. thesis, Faculty of Sciences, University of Geneva, 1997.
6. Al-Jadir L., Estier T., Falquet G., Léonard M., Evolution Features of the F2 OODBMS, *Proc. Int. Conf. on Database Systems for Advanced Applications, DASFAA*, Singapore 1995.
7. Andany J., Léonard M., Palisser C., Management of Evolution in Databases, *Proc. Int. Conf. on Very Large Data Bases, VLDB*, Barcelona 1991.
8. Banerjee J., Kim W., Kim H-J., Korth H.F., Semantics and Implementation of Schema Evolution in Object-Oriented Databases, *Proc. Int. Conf. on Management Of Data, ACM SIGMOD*, San Francisco 1987.
9. Fayyad U.M., Piatetsky-Shapiro G., Smyth P., Uthurusamy R. (eds), *Advances in Knowledge Discovery and Data Mining*, AAAI Press/MIT Press, 1996.
10. Ferrandina F., Meyer T., Zicari R., Ferran G., Madec J., Schema and Database Evolution in the O2 Object Database System, *Proc. Int. Conf. on Very Large Data Bases, VLDB*, Zürich 1995.
11. Kamber M., Winstone L., Gong W., Cheng S., Han J., Generalization and Decision Tree Induction: Efficient Classification in Data Mining, *Proc. Int. Workshop on Research Issues on Data Engineering, RIDE*, Birmingham 1997.
12. Kuno H.A., Ra Y-G., Rundensteiner E.A., The Object-Slicing Technique: A Flexible Object Representation and Its Evaluation, *Technical Report, CSE-TR-241-95*, University of Michigan, 1995.
13. Mehta M., Agrawal R., Rissanen J., SLIQ: A fast scalable classifier for data mining, *Proc. Int. Conf. on Extending Database Technology, EDBT*, Avignon 1996.
14. Morsi M.M.A., Navathe S.B., Kim H-J., A Schema Management and Prototyping Interface for an Object-Oriented Database Environment, in: *Object Oriented Approach in I.S.*, F. Van Assche & B. Moulin & C. Rolland (eds), IFIP, North-Holland, 1991.
15. Odberg E., Category Classes: Flexible Classification and Evolution in Object-Oriented Databases, *Proc. Int. Conf. on Advanced Information Systems Engineering, CAISE*, Utrecht 1994.
16. Penney D.J., Stein J., Class Modification in the GemStone Object-Oriented DBMS, *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA*, Orlando 1987.
17. Peters R.J., Özsu M.T., An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems, *ACM Transactions on Database Systems*, vol. 22, no 1, march 1997.
18. Sciore E., Object Specialization, *ACM Transactions on Information Systems*, vol. 7, no 2, april 1989.
19. Shafer J., Agrawal R., Mehta M., SPRINT: a scalable parallel classifier for data mining, *Proc. Int. Conf. on Very Large Data Bases, VLDB*, Bombay 1996.
20. Skarra A.H., Zdonik S.B., Type Evolution in an Object-Oriented Database, in: *Research Directions in OO Programming*, B. Shriver & P. Wegner (eds), MIT Press, 1987.
21. Zaki M., Ho C-T., Agrawal R., Parallel Classification for Data Mining on Shared-Memory Multiprocessors, *Proc. Int. Conf. on Data Engineering, ICDE*, Sydney 1999.