ГОДИШНИК НА СОФИЙСКИЯ УНИВЕРСИТЕТ „СВ. КЛИМЕНТ ОХРИДСКИ"

ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА
Книга 1 — Математика
Том 83, 1989

ANNUAIRE DE L'UNIVERSITE DE SOFIA „ST. KLIMENT OHRIDSKI"

FACULTE DE MATHEMATIQUES ET INFORMATIQUE
Livre 1 — Mathématiques
Tome 83, 1989

AN INTELLIGENT OBJECT-ORIENTED DATA MODEL

Pavel Azalov, Fanny Zlatarova

Павел Азалов, Фани Златарова. ОДНА ИНТЕЛЛИГЕНТНАЯ ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ МОДЕЛЬ ДАННЫХ

Настоящая работа является коротким введением в одну объектно-ориентированную модель данных. Объекты классифицированы в классы, которые формируют иерархию. Внимание отделено тоже и на описание т. наз. системных классов. Их тип может быть выбран из следующих типов: примитивный, форматный и неформатный. Рассматриваемая модель данных характеризуется включением некоторых знаний в определение объектного класса. Рассматриваются два основных типа знаний: пассивные и активные. Первый тип знаний относится к генерированию некоторых объектов, а второй тип предназначен для автоматического активирования действий, когда удовлетворяются определенные условия, которые заранее специфицированы. Операция Rel_Ship используется для установления некоторых связей между объектами, принадлежащими к разным классам. Правила типа триггеров используются для автоматической регистрации связей, полученных таким способом.

Pavel Azalov, Fanny Zlatarova. AN INTELLIGENT OBJECT-ORIENTED DATA MODEL

This paper is a brief introduction to an object-oriented data model. The objects are classified in classes being organized in a class hierarchy. Some attention has been paid to describe the so called system classes. Their type may be chosen among the following ones: primitive, formatted and unformatted. The data model under consideration is characterized by knowledge being incorporated in the object class definition. Two basic types of knowledge are considered: passive and active. The first type of knowledge refers to generating some objects while the second one (the triggers) aims at activating automatically actions when some conditions predetermined in advance are satisfied. The Rel_Ship operation is applied to establish some links among objects from different classes. Triggers are used to registrate automatically the links thus obtained.

1. INTRODUCTION

1.1. Motivation. Nowadays there exists a number of different data models (DM). Each DM has distinct evolutionary applications. However, only few of them, for example, the relational DM, are used quite often in practice and research. The

joint processing of text, numbers, graphical images and combinations of them, i.e. the so called unformatted data result in new problems which usually lead to the necessity of extending and sophisticating the existing DM, on one hand, and to elaborating some new ones letting us to process complex informational objects, on the other hand. Such DM have to meet the following basic requirements:

A. The data structures maintained by the DM have to correspond to concrete user's needs. The user should be able to define new data types which should share the same syntax and semantics together with the existing ones. He should be able to define a rich set of operations, permitting the specification of the objects behaviour with respect to the modeled subject area as well [6].

B. To maintain the processing of both traditional formatted data (integers, fixed-length strings, etc.) and unformatted data such as text or graphics [5,8].

These features cannot be discovered in the traditional DM and in particular in the relational DM [3]. The object–oriented DM presented in this paper attempts to meet the above mentioned requirements.

1.2. Object–oriented concepts. Recently the object–oriented programming has become very popular and it has been used to design and elaborate complex application systems in the field of office automation, artificial intelligence, etc. The most important advantage of the object–oriented approach in this case is its letting us to model the entities, typical of the application domain under consideration by using the object as a main concept [9].

The term *object-oriented* can be interpreted in a different way. For example, the *Iris* objects represent entities and concepts from the application domain being modeled [3,4]. They are unique entities in the database with their own identity and existence.

An *OZ+* object is a discrete entity encapsulating both data and an allowable set of operations on these data [10].

In the *GemStone* DM the object is considered as a chunk of private memory with a public interface. *GemStone* merges object-oriented language concepts with those of database systems, and provides an object-oriented database language called *OPAL* which is used for data definition, data manipulation and general computation [6,7].

In general objects represent entities belonging to the modeled subject area [1]. Every object is considered as an instance of an object class, and every object class is a specification of an abstract data type. Objects having the same properties are grouped in one and the same object class. An object class consists of some private memory for a collection of instance variables. The values of these variables determine the state of a particular object. The value of each instance variable is an object too, i.e. this value will have its own private memory for the states it might have with the exception of the simplest objects (i.e. the so called primitive ones, such as a string or an integer). The objects have a well determined behaviour, which is expressed through the set of operations applied to them.

The operations or the so called *methods* are elements of the class defirition. Each operation is composed of two parts: the interface and the implementation. The first one describes their external characteristics, such as: name, arguments together with their types and result type. The operation implementation indicates how to perform it. Methods as well as instance variables are invisible out of the object. For that reason the objects are treated as incapsulated ones. The operations

are performed through the so called *communications*, permitting the communication among objects. Every object can be either an operation argument or a result of such operations. The instances differ from each other by their content and unique identification. The object name identifies the object uniquely at each moment of its existence. Hence, each class is named *collection of objects*. For example, object classes are: ***People***, ***Books***, ***Cars***, etc. Some of their properties might be as follows: ***Name***, ***Birth_date***, ***Editor***, ***Car_make***, ***Height***, etc.

Among the classes defined in an object-oriented system *is_a* relationships can be established. This is achieved by using a concept subclass. For example, if the ***Person*** class is defined, then it is naturally the ***Employee*** class to be its subclass.

That means, every object of the ***Employee*** class is an object of the ***Person*** class, and an object of the ***Employee*** class can be used wherever an object of the ***Person*** class can be used too. Thus a class hierarchy might be created. It is a hierarchy of the classes, where a child node is a specialization of the parent node and the parent node is a generalization of the child node [16]. Each subclass might *inherit* various characteristics of the superclass. Thus some of the inherited characteristics may be overridden.

2. OVERVIEW OF THE MODEL

The basic concept of our model named intelligent object oriented DM is the *object*. The object is treated as an instance of an *object class*, which is the structural specification of a class of objects.

2.1. Objects. Every object is an instance of a class of objects. When a user wants to specify and use the system, he pays attention to the nature of the objects under consideration. For that purpose he defines his own classes and methods and then creates instances of these classes. For each of the classes the common structure of all the instances of the class are defined. We will consider the following example:

```
Class Person;
   Attributes:
      Name:string[24];
      Address:string[60];
      Phone:string[12];
   Methods:
      First_name:Name ⟶ string[12];
      Last_name:Name ⟶ string[12];
      City:Address ⟶ string[16];
   Knowledge:
      R2.Ident:Unique(Name,Address);
      R1.Tel_Val:Multivalue(Phone);
      R1.Tel_Exist:Optional(Phone);
   Initialization:
      Phone:="(0073)895370"
End.
```

The *Person* class is characterizad by three parts: **Attributes**, **Methods** and **Knowledge**. Properties, characterizing the objects of the given class are defined in the first part. Such attributes are: ***Name***, ***Address*** and ***Phone***. Each of them is specified by its type. It can be either of a primitive type (string, integer, real, boolean) or of a type which has already been defined for another existing object. The objects behaviour typical of all the classes is described in the other parts. The first one includes the operations characterizing the respective objects. For example, in this case they are as follows: ***First_name***, ***Last_name*** and ***City***. Thus by using the ***City*** operation we may obtain the name of the city from the ***Address*** address. The **Knowledge** part comprises the knowledge needed to describe the objects of that class. They are invariant in respect to the particular instances. The presentation, the utilization and the knowledge types used for the processing of the objects in the classes under consideration are described in Part 3.

The class definition may include the **Initialization** section as well, which indicates the initial values of some of the attributes, if necessary.

2.2. Inheritance and hierarchy. Classes are organized in a hierarchical structure that supports generalization and specialization. A class may be declared as a subclass of another class. The objects of each subclass provide the behaviour of the objects of the superclass plus something extra. In other words, the properties of the superclass objects are one and the same for the objects of its subclasses as well. That is why we say that the properties are *inherited* by the subclass. For example, let us consider the definition of the ***Employee*** class.

```
Class Employee is_subclass_of Person;
   Attributes:
      Department:string[24];
      Salary:integer
End.
```

In this case the ***Employee*** class is a subclass of the ***Person*** class. It inherits all the instance variables and methods from its ***Person*** superclass and declares additional ones such as ***Department*** and ***Salary***. All algorithms working on the ***Person*** objects automatically work also on the ***Employee*** objects too.

3. KNOWLEDGE INCORPORATION IN OBJECTS

The knowledge part of the object class definition comprises a set of rules. It can contain one or more than one rules, or it can be the empty set. Two types of rules are considered: *passive* and *active*. The first one is used in order to present knowledge, needed to generate some rules typical of different classes. Rules of the second type, called *triggers*, are used for the knowledge representation applicable to activating automatically actions when some given conditions take place [5,9,11]. Each rule is named and characterized by two parts: rule type and its specific name which identifies that rule in the set comprising all the rules. Next we will describe briefly each of the rules discussed.

Existential Rules **(E)**

```
E.<name>:Mandatory(<Attr_list>);
E.<name>:Optional(<Attr_list>);
E.<name>:Multivalue(<Attr_list>);
```

Constraints on attribute values independent of the values of other attributes are specified by using the existential rules.

Uniqueness Rules (U)

U.<*name*>:Unique(<*Attr_list*>);

This rule specifies that the value of one or more than one attributes identifies a given object among the other objects of the same class uniquely.

Content Derivation Rules (C)

C.<*name*>:*Attr_name*:=*Expression*;

This rule is used to specify how a certain attribute is given a value. The attribute value is calculated whenever a query to it appears.

Access Control Rules (A)

A.<*name*>:Access(<*User_list*>);

The user's access rights to the objects of a given class are determined by using the access control rules. Thus the access to all superclasses of this class is defined.

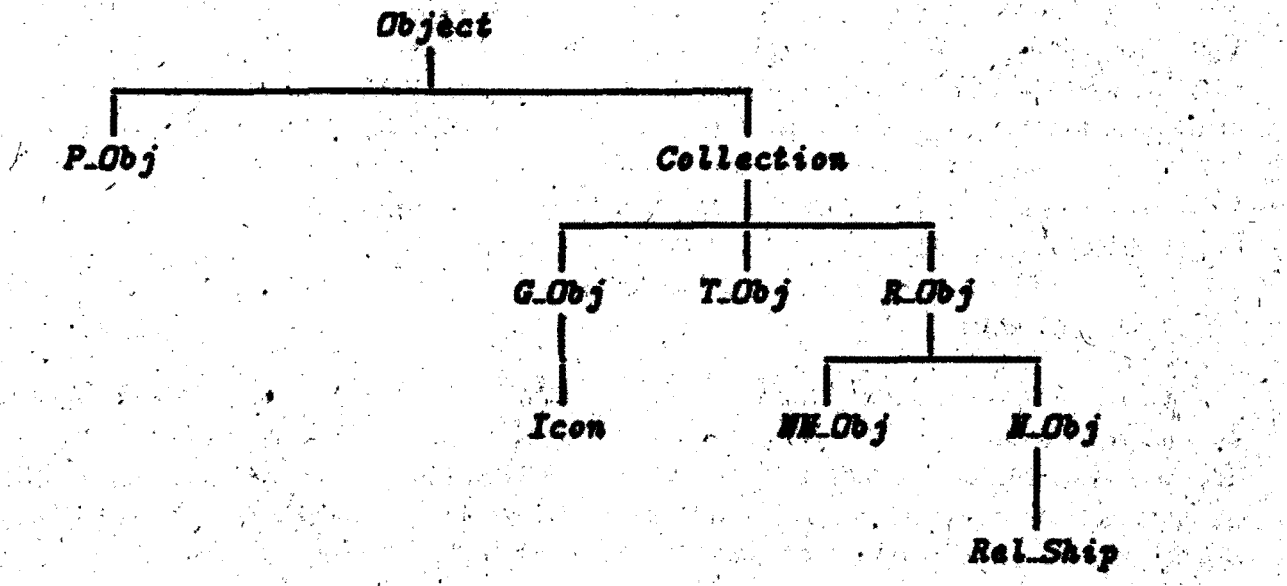
Triggering Rules (T)

T.<*name*>:if *Cond* then *Action* [error <*Message*>]

Triggering rules are used to activate automatically the performance of the *Action* action when a given event specified by the *Cond* expression occurs. The error part (which is not obligatory) is used to handle the exceptions.

4. SYSTEM CLASSES

Similarly as in other object-oriented systems we also define a class called the *Object* class which is assumed to be the root of the class hierarchy. The figure given below indicates all the system-defined subclasses of the *Object* class.



The *Object* Class

The *Object* class is a superclass of all the classes and it is used to provide a consistent basic functionality and default behaviour. Many methods in this class are overridden in subclasses. The comparison of two objects and the determination of the class when given an object can illustrate methods of this class being of great significance. The *Object* class has two subclasses: *P_Obj* and *Collection*.

The *P_Obj* Class

This is the class of the primitive objects. The standard defined classes **Integer, Real, String, Boolean** and **Point** are subclasses of it. Here they are used in a trivial way and that is why for that reason we are not going to consider them.

The *Collection* Class

The *Collection* class consists of objects being collections of other objects. The class properties can be described by using two attributes: the maximum number of objects and the maximum object size. The methods typical of this class are as follows:

— determining the number of elements included in a collection;
— sorting the collection elements;
— existence control of a given element belonging to a collection, etc.

The elements of a given collection can be related to different classes.

4.1. Formatted classes. The *R_Obj* class consists of objects which are relations, i.e. sets of elements being of the same type. Each element is an n-tuple. Then two types of relations will be permitted:

— normalized relations (the *N_Rel* class);
— unnormalized relations (the *NN_Rel* class).

The properties of the objects of the *R_Obj* class are described by using the number of attributes in this relation and the maximum number of elements included in a relation. The basic methods typical of this class are as follows: adding rows, deleting rows, updating the existing attribute values in a row, etc.

The *N_Rel* class includes normalized relations. Next we can mention some operations of the relational algebra such as: union, difference, project, restriction and decart product which are among the methods being of great significance. Each user's subclass of the *N_Rel* class defines a particular relational scheme and it may determine some methods typical of the user's applications. Thus we may apply the methods of the classical relational approach to the case under discussion.

The *Rel_Ship* class consists of binary relations having a special function. Links among the objects of the different classes are established by using these relations. But we have to point out that in this case these links are maintained automatically. It is necessary only to define some criteria for the respective classes. This problem is considered in section 5.

The *NN_Rel* class consists of hierarchical relations which sometimes might appear to be of some use.

4.2. Unformatted classes. Many users' problems lead to studying some of the informational objects which cannot be structured and processed by applying the well known and standard methods. Mostly these are the text and graphical informational objects. For instance, a document may consists of: *P_objects* (date, sender); *T_objects* (letter contents); *Icon_objects* (Company Logo).

The *T_Obj* class is the class of the text objects. For example, an abstract of a paper or book; short biographical remarks, etc. They are represented as variable length character strings. The operations typical of such a class are as follows: text comparison; insertion and deletion of a substring in a given text; searche of a substring; exchange of substrings; concatenation of two texts, conversion of a digital text in a number; conversion of a text in a graphical image, etc. The text length is used as a class attribute.

The *G_Obj* class includes the graphical objects such as a graph, a pie chart, a histogram, an icon or an arbitrary picture which can include a text too. All of them are visualized in a rectangular field, whose maximum sizes are presented as attributes in a certain class. The methods typical of this class are, as follows: edition of a graphical image; comparison of graphical images; conversion of graphical images in a text string; graphical image visualization, etc. These operations can be considered as a special subclass of the *G_Obj* class, and the icon processing is typical of them. As an example we can consider the *Student* class:

```
Class Student is_a_subclass_of Person;
   Attributes:
      Photo:Icon[60,80];
      Language:integer;
      Mathematics:integer;
      Autobiography:T_Obj;
   Methods:
      Average:(Language,Mathematics) ⟶ real
End.
```

5. OBJECT ENVIRONMENT

5.1. Object creation and manipulation. Having defined the classes we can begin with generating the instances (objects) from them [2]. The command used in this case is the **new** command. For example, we will consider the instance generation in the *Student* class.

```
new Student (Name,Languages,Mathematics);
      Objects: J_S('John Smith',6.5);
               P_K('Peter Koch',6,6);
```

Using this command two objects *J_S* and *P_K* are created explicitly. Only some of the objects attributes may obtain values. Therefore the names of these attributes have to be indicated. Next the other attributes can be given values by using the **update** command:

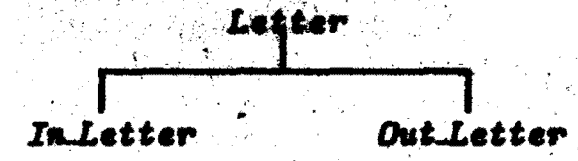
```
update J_S.Phone:='251-167';
```

In this example the *Phone* attribute of the *J_S* object either acquires a new value or it is updated.

The **destroy** command removes an object from a given class; all the objects from the class or the whole class. These functions are illustrated by the following examples:

```
destroy J_S;
destroy Class Student;
```

5.2. A world of linked objects. The class instances may be independent of each other within a certain class and they may be independent of the instances of other classes as well. However there exist some cases, where the objects included in different classes and the objects within a certain class should correspond to each other. For example let us consider the *Letter* class and its subclasses *In_Letter* (input mail)



and *Out_Letter* (output mail). Each of the letters received can result in zero, one or more than one letters to be sent. Then, we may say that between the objects of the classes *In_Letter* and *Out_Letter* there exists an explicit link. Thus the object searche will be facilitated to a great extent by creating and mantaining such links. In our model this link is realized through defining a particular instance which belongs to the *Rel_Ship* class. This link itself is a binary relation. Thus we have:

```
Class Link is_a_subclass_of Rel_Ship;
   Attributes:
      InL:string[12];
      OutL:string[12]
End.
Class Letter;
   Attributes:
      Answer:string[12];
      Sender:Person;
      Receiver:Person;
      Date_Send:Date;
      Body:T_Obj;
   Knowledge:
      E.Ans:Multivalue(Answer)
End.
Class Out_Letter is_a_subclass_of Letter;
   Knowledge:
      T.LO:if Answer<>"
              then update Link.InL:=Answer,
                   update Link.OutL:=?
End.
Class In_Letter is_a_subclass_of Letter;
   Attributes:
      Date_Receive:Date;
   Knowledge:
      T.LI:if Answer<>"
              then update Link.InL:=?,
                   update Link.OutL:=Answer
End.
```

Let us give some explanations. The *Answer* attribute comprises the names (numbers) of the letters received (if any) as its values.

If an output letter is created in reply to some existing letters, then the *LO* trigger will registrates its links with them. The role of the *LI* trigger is similar to that of the *LO* trigger. Thus we accomplish an automatic link between the *In_Letter* class and the *Out_Letter* class.

REFERENCES

1. Banerjee, J., H. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, H. Kim. Data Model Issues for Object-Oriented Applications. *ACM TOIS*, vol. 5, № 1, 1989, 3–26.
2. Beech, D. A Foundation for Evolution from Relational to Object Databases. *In*: EDBT'88, Advances in DB Technology, Lecture Notes in Computer Science, vol. 303, 1988, 251–270.
3. Derrett, N., W. Kent, P. Lyngbaek. Some Aspects of Operations in an Object-Oriented Database. *Database Engineering*, vol. 8, № 4, 1985, 66–74.
4. Fishman, D., D. Beech, H. Cate, E. Chow, T. Connors, J. Davis, N. Derret, C. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. Neimat, T. Ryan, M. Shan. IRIS: An Object-Oriented Database Management System. *ACM Trans. on Office Inf. Systems*, vol. 5, № 1, January 1987, 48–68.
5. Gibbs, S., D. Tsichritzis. A Data Modeling Approach for Office Information Systems. *ACM TOIS*, vol. 1, № 4, 1983, 299–319.
6. Mayer, D. Object-Oriented Database Development at Servio Logic. *Database Engineering*, vol. 8, № 4, 1985, 58–65.
7. Mayer, D., J. Stein. Development and Implementation of an Object-Oriented DBMS. *In*: Research Directions in Object-Oriented Programming, ed. B. Shriver and P. Wegner, 1987, 355–392.
8. Shek, H., P. Pistor. Data Structures for an Integrated Data Base Management and Information Retrieval System. *In*: Proceedings of the 8th Int. Conf. on VLDB, 1982, 197–207.
9. Tsichritzis, D. Object Species. *Database Engineering*, vol. 8, № 4, 1985, 2–7.
10. Weiser, S., F. Lochovsky. OZ+: An Object-Oriented Database System. *In*: Office and Data Base Systems Research'87, ed. F. Lochovsky, Technical Report CSRI-195, CSRI, University of Toronto, 1987, 174–193.
11. Yang, J. A Proposal for Incorporating Rules in ODA-Documents. *In*: Office Information Systems, ed. B. Pernici and A. Verrijn-Stuart, 1988, 131–147.

Received 31.05.1990