Separating Structural and Semantic Elements in Object Oriented Knowledge Bases

Erich J. Neuhold+ Yehoshua Perl*
James Geller* Volker Turau+

+GMD-IPSI Integrated Publication and Information Systems Institute
Dolivostr. 15, D-6100 Darmstadt, FRG
*Institute for Integrated Systems, Department of Computer Science
New Jersey Institute of Technology, Newark, New Jersey 07102

Abstract

In this paper we present an object-oriented data-model with types. We separate in the definition of the object class the structural parts from those parts describing the semantics of the object class in the real world. In order to express that all instances of a class have the same structure and behavior we consider them to be of the same abstract data type. This type containing the structural parts of the object class is called the object type of that object class. Object types are organized in a hierarchy enabling inheritance. Different object classes may have the same object type. By separating these two parts of the specification a better abstracting mechanism is achieved. The semantics of the objects in the context of an application is described in the definition of the object classes by determining their semantic relations with other object classes, regardless of their object types.

1 Introduction

In order to store and retrieve the information for an application of a certain enterprise from a database, a suitable representation of the application's environment is required. Such a representation is called a conceptual model. It is an abstract representation that contains the properties of the environment relevant to the application. In the past years numerous models with the aim of representing semantic structures beyond the capabilities of the first-normal-form relational model were presented [PM88]. The representation mechanisms are all based on a collection of abstraction techniques. The most significant technique is to collect the objects of the domain of interest into classes. The class concept has been introduced in Simula [DN66] and reimplemented by Smalltalk [GR83].

A class can be regarded as a container for objects which are similar in their structure and their semantics in the enterprise. Furthermore, the objects have a similar behavior from the user's point of view. Therefore, a class can be regarded as a description of the structure and behavior of objects.

Following [W87], the characteristics of object-oriented languages are the notions of objects, classes, and inheritance. The description of an object class contains both structural parts and parts that describe the semantics the classes have in the application. In many systems e.g. [CM89, KNS88, F87, GR83, SR86], the structural and semantic parts are mingled together. This causes some difficulties when we have to distinguish whether an object part is structural or contains semantic information.

To provide reusability, classes are organized in a class hierarchy. Subclasses inherit the properties of their super classes. In many systems as O2 [LR88] and Vbase [AM87] the subclass hierarchy is used for two purposes at the same time:

a) to factorize common structure and behavior of classes and

b) to express additional semantic relationships between classes.

This leads to a situation that two classes modeling semantically related objects could only be dealt with, if the objects in question are structurally related as well. The use of a single hierarchy for two conceptually distinct connections among specifications resulted in inadequate conceptual models. Therefore, we decided to separate these two parts of the specification.

In our model we separate in the definition of the object class the structural parts from those parts describing the semantics of the class in the real world. In order to express that all instances of a class have a common structure and behavior we consider them to be of the same abstract data type. The type is called the *object type* of that class.

Hence, we associate with each object class an object type. Regardless of their corresponding object type, object classes can be related according to their relations in the application. These are called semantic relations and can be formulated independently of the object types.

For example it may be desirable to represent the same object in different contexts of the application, or to deal with an object in different levels of detail. All the semantic constraints are now expressible in the model, regardless of the structural relations between the relevant classes.

Some object-oriented programming languages use the concept of *abstract classes* [GR83,MSOP86], which are classes which cannot be instantiated. Their only purpose is to define useful attributes, relationships, and methods that can be inherited by concrete subclasses. We achieve the same result by defining object types which include these attributes, relationships, and methods and then utilize type inheritance. In this way the anomalous concept of abstract classes is avoided.

Thus, our specification allows on one side a structural hierarchy (an acyclic directed graph) of object types, and on the other a network of classes. In making this distinction we hope to achieve a better abstraction mechanism, giving a more accurate representation of the application.

Section 2 formally introduces the notion of object type which provides the structural description of an object class. Section 3 deals with the semantic description of object classes. Our conclusions are summarized in Section 4. For an extended version of this paper see [NPGT89].

2 The Structural Description of Object Classes

2.1 Types and Classes

In order to structure the set of objects in the domain of our interest we collect objects into classes. An object class can be regarded as a container for objects which are similar in their structure and their semantics in the real world. They have similar behavior from the user's point of view.

2.3.2 Relationships

A relationship property is represented by a name and an object type. The name is again a selector for the property. The object type referenced must be defined elsewhere. In a relationship we allow not only a single object type but also a composite structure of object types using the same set, tuple, and disjoint union constructors that we used in defining composite data types. (However, note that we do not allow a mixture of data types and object types in such a composite structure).

Let R be a relationship from an object type A to an object type B. Let $D(A)$ and $D(B)$ be the domains of the object types A and B respectively. (The definition of the domain of an object type is complex and will be presented in Section 2.3.4). The relationship R is a relation from the set $D(A)$ to the set $D(B)$, referring here to the mathematical notation of relation as a subset of the Cartesian product

$D(A)$ x $D(B) = \{$ (a , b) / $a \in D(A), b \in D(B)\}$.

Hence, the relationship is a set of ordered pairs of objects. If the relationship has a composite object type B, and its domain is the composition of the individual domains of the object types, then the objects will form a complex structure, e.g. a set of objects of the same object type, or a tuple of objects of different object types, or elements from a disjoint union type, or a further nesting of such types.

2.3.3 Methods

In the following we use the Smalltalk [GR83] terminology for methods but have redefined some of its meaning. A method is a program segment with one required parameter of some object type, and any number of optional parameters. We will assume that every method also returns a value of an object type or data type. The method name together with these optional parameters is called the "message" which is sent to the object identified by the required parameter.

If a program segment is needed that takes values of a data type as arguments then it must be defined as an operation of this data type rather than a method, and it will also return a value of a data type.

The signature of a method or of an operation defines the types of its input parameters and the type of the return value. The definition of an object type contains a signature for every method that is defined for that object type as its required parameter.

In the following we will formalize methods that do not perform any side effects in persistent memory, i.e. methods that perform only local computations and that influence their environment only by their return value. We will give a recursive definition of such a method.

A *computational method* is a program segment with one required parameter of some object type and some optional parameters that makes use of the functionality of the underlying programming language (e.g. C++) but does not modify any stored values outside of its own local memory and returns a value of an object type.

A *primitive method* is either a relationship or a computational method.

A *method chain* is either a primitive method or a primitive method composed with a method chain.

A *transformer* is a program segment that takes as a required argument a value of an object type and returns a value of a data type. Other than that it behaves like a computational method.

A *primitive transformer* is either a transformer or an attribute.

An *operation chain* is either an operation or an operation composed with an *operation chain*. (Operations have been defined previously for data types).

A *transformer chain* is either a primitive transformer or a primitive transformer composed with an operation chain.

With all these terms in place we can now define a method formally. A *method* is either a method chain or a transformer chain or a composition of the two, namely a method chain composed with a transformer chain.

According to our definition relationships and attributes are just special cases of methods. Nevertheless, they are conceptually important special cases which warrant our three way distinction between attributes, relationships, and methods.

We will refer to the set of all possible argument values of one parameter as the *domain* of this parameter. The *domain of a method* is the cross product of the domains of all its parameters. We define the *range of a method* as the set of all results that this method can generate given all possible values of its domain as inputs.

With the above assumptions one can view a method with no side effects as a relation. This relation will be from the domain of the method to its range.

R: Domain → Range

In case that the domain is a cross product this relation takes the following form. Let Dom_i be the domain of the i-th argument of the method.

R: Domain(Object type) x Dom_1 x Dom_2 .... x Dom_n → Range

This basic idea can be carried over easily to all elements that have been used in the formal definition of a method. For an operation we get

R: Data type x Dom_1 x Dom_2 .... x Dom_n → Data type.

For a transformer we get

R: Domain(Object type) x Dom_1 x Dom_2 .... x Dom_n → Data type

and for a computational method

R: Domain(Object type) x Dom_1 x Dom_2 .... x Dom_n → Range = Domain(Object type).

Our definition of a method permits the chaining of the above defined entities in the following pairs. (1) For method chains: computational method - computational method, relationship - relationship, relationship - computational method, computational method - relationship. (2) For transformer chains: operation - operation, attribute - operation, transformer - operation. (3) Chaining method chains with transformer chains creates the following additional possible pairs: computational method - operation, computational method - transformer, computational method - attribute, relationship - operation, relationship - transformer, and relationship - attribute.

It is obvious that with our restrictions all the mentioned compositional pairs can be represented as the compositions of relations.

E. g. If R_1: A → B is a relation from A to B and R_2: B → C is a relation from B to C then we can define the composite relation R_3: A →C as

$R_3 = R_1 \circ R_2$

such that "o" defines relation composition in the usual sense:

$R_1 o R_2 = \{(x, y) |\ xR_1z\ \&\ zR_2y\}$

Clearly our recursive definition covers composition chains of any length, e.g. if a method R is defined by chaining relationships R_1, R_2, R_3 to an attribute A_1, then R = $(((R_1 \circ R_2) \circ R_3) \circ A_1)$. Therefore, a method chain, an operation chain, a transformer chain, and thus a method can each be represented as a composition of relations.

2.3.4 The Domain of an Object Type

We need to define the domain of an object type. This domain should reflect the attributes, relationships, and methods of the object type. To facilitate the definition we use the *selector Cartesian product* which is defined in Section 2.2.

In an object type the properties are identified by their selector, rather than by a serial number. Thus we define the domain of an object type as a mapping M from the tuple of the properties of the object type to the Cartesian product of n sets as follows:

$M : (PROPERTY1, PROPERTY2, \ldots, PROPERTYn) \rightarrow A_{PROPERTY1} \times A_{PROPERTY2} \times \ldots \times A_{PROPERTYn}$.

An element of the selector Cartesian product is a mapping $(PROPERTY1, PROPERTY2, \ldots, PROPERTYn) \rightarrow (a_{PROPERTY1}, a_{PROPERTY2}, \ldots, a_{PROPERTYn})$, where $a_{PROPERTYi} \in A_{PROPERTYi}$ for $1 \leq i \leq n$.

We still have to define the $A_{PROPERTYi}$. If $PROPERTYi$ is an attribute then $A_{PROPERTYi}$ is the type defined for this attribute in the definition of the object type. If $PROPERTYi$ is a relationship to another objecttype OT, then we define $A_{PROPERTYi}$ to be the set of all possible classes whose object type is OT. If $PROPERTYi$ is a method M then we define $A_{PROPERTYi}$ to be the method M itself. Thus the domain of the object type STUDENT in Section 3.2 is

So far we have assumed that any two object classes model different objects of the real world. But if we want to model the same real world object in two different contexts, in which the objects have different realizations, we must introduce two different object classes. Each object class has a different object type. To capture the semantic connection between the two classes we use the roleof concept [SN88]. It is used to express the fact, that two (or more) object classes model the same real world object in different contexts (or equivalently the objects are represented in different roles). This concept has consequences regarding message passing, because if a method is sent to an object and the method is not contained in the interface of that object, then it may be forwarded to one of its roles. More details are given in Section 3.3.

Note that the roleof concept is not symmetric i.e. the fact that A is roleof B does not imply the converse relation. Therefore, an object can only exist in the context described by the object class A if it already exists in the context described by the object class B. The object types of the classes A and B do not have to be sub- or supertype of each other. The roleof concept reflects purely a semantic constraint, which is not reflected by their types.

The second modelling device for relating object classes is called categoryof. In some way it is dual to the roleof construct. It is used to model the same real world object with additional knowledge, but still in the same context. So it is a refinement of the description with respect to one aspect of its former description. Whereas in the case of roleof the additional information was about the object in a different context. So if more specialized information about the instances of an object class is available, we can categorize the instances into different object classes and relate these classes via the categoryof concept with the original class.

Note that if the object class A is categoryof object class B, then the instances of the class A model the same real world objects as the class B. This has to be seen in contrast to two object classes not related by categoryof in which the corresponding object types are in a sub-/supertype relation. Here the two object classes model objects which are structurally closely related, but they may correspond to different real world objects from dissimilar areas.

The set of object classes forms with respect to the roleof and categoryof specifications a network. This network is defined independently from the hierarchy of the object types. This has the advantage, that the domains of two object classes modeling the same real world objects need no longer be in a sub-supertype relation. Such a relation always implied a structural similarity of the instances. As we shall see this is the case in the situation of category specialization, but certainly not in the case of role specialization. There, the same real world object can have totally different structures in different roles. By separating the semantic specification from the definition of the object types, our model is closer to the real world.

The network of object classes is used for message passing. The corresponding concepts are described in Section 3.3. In the next subsection we shall give an example illustrating the above described concepts.

3.2 Example

In this subsection an example from a "university-database" is presented (in Figure 1), illustrating the use of object type and object class definitions, the structural subtype hierarchy and the semantic *roleof* and *categoryof* relations. The database contains information about students, former students, instructors, teaching assistants, research assistants and sections of courses. For all students, instructors and assistants we want to store at least personal data details. To avoid defining the personal data details several times, we introduce an object type PERSON with the attributes PersData and Address. The PERSON object type is the supertype of the object types STUDENT, INSTRUCTOR, and ASSISTANT. Thus, these object types inherit the attribute PersData. The object types INSTRUCTOR and ASSISTANT inherit also the attribute Address. But for STUDENT an attribute Address is defined (to contain his local address which may be different from his home town address) and is overriding the attribute Address in PERSON. The object types INSTRUCTOR and ASSISTANT have an attribute DepartAddr which contains their work address. Thus a student which is a teaching assistant may have three different addresses: a home town address in PERSON which is inherited to TEACHING_ASSISTANT through ASSISTANT. A local address in STUDENT and a work address inherited from DepartAddr in ASSISTANT. Since we want to distinguish between teaching assistants and research assistant, we define corresponding subtypes of the object type ASSISTANT. Note that these object types inherit the attributes of PERSON and ASSISTANT.

Some of the students may already be assistants. So there is information about persons in the context of being a student and of being an assistant. To capture this semantics a general object class person is introduced. The classes student and assistant will be defined as roleof the class person. Furthermore, some of the assistants are teaching assistants, some of them are research assistants and some are both, where the attribute PosPercent specifies the appropriate percentage of the position. This implies that we need two object classes research-assistant and teaching-assistant, which both are categories of the object class assistant.

The names of the object classes are spelled with small letters, the names of the object types are printed with capital letters, names for attributes and relationships have only the first letter capitalized and keywords are in bold face. Some composite data types are used. Their exact definitions are omitted for lack of space and appear in [NPGT89]. However, their names give a vague idea of their contents. The two columns describe object types (left) and classes (right).

We omit relationships and methods from the class descriptions whenever there exists only one class for a given object type, even though it is in principle necessary to specify them. Thus we write the classes only for object types which have multiple classes. In our example these are the classes STUDENT, STUDENTS, and ORGANIZATION. Thus the relationship Sections and the method NumStudents are omitted in the class description. On the other hand the method MyStudents appears in the class description of instructor since its access path contains STUDENTS for which we need to specify students rather than formerstudents. Similarly the relationship "Membership" is not omitted in student and formerstudent as there are two classes, union and alumniorganization, for the object type ORGANIZATION.

"MyStudents" is a method chain. If it is sent to INSTRUCTOR the message handler will first follow the TeachingSections relationship to retrieve a set SECTIONS of all sections taught by this instructor. By using the structural relationship "setof" we obtain the set of SECTION as {SECTION}. Following the Students relationship defined for SECTION a set of STUDENTS will be derived. NumStudents is a similar method, composed of a method chain and a transformer chain, counting the number of students of an instructor. It uses an operation Sum, which computes the sum of a set of NATURALs.

The objecttype TEACHING_ASSISTANT is a subtype of two objecttypes, INSTRUCTOR and ASSISTANT. This is an example of multiple inheritance and implies that attributes and relationships for TEACHING_ASSISTANT are inherited via the subtype hierarchy from both object types.

The field name of the attribute PersData of the class person is marked essential. Therefore the value of the field name has for every element of the extension of the class person a value different from nil. The system will check the values of the essential attributes before inserting an instance.

3.3 Inheritance via roleof or categoryof

So far in our model we have always made a sharp distinction between object types and object classes. We shall retain this dichotomy in our discussion about inheritance. In section 2 we introduced object types and the mechanism of subtyping. By subtyping new types are derived from existing types. The subtypes inherit the properties of their supertypes, as discussed in Section 2.4. On the other hand object classes which are connected via roleof or categoryof have an inheritance mechanism which influences the message passing concepts used for methods. So there are principally two different kinds of inheritance in our model. The first one is via the subtype hierarchy and the second is based on the network of object classes.

The message handling system identifies the implementation of a method according to the precedence list. In the case the method is not contained in the object's interface and the object class is not related via roleof or categoryof to other object classes, then the method can not be executed and the object signals an error. If there are roleof or categoryof specializations of the class in question, a message passing mechanism will be applied. This mechanism is explained in the following.

The inheritance mechanism between two classes A and B depends on whether the corresponding object types are connected in the subtype hierarchy or not. Therefore, we shall distinguish these two cases in the following. For convenience we call the object type of class A the object type A.

Case 1: The object type B is a subtype of the object type A.

Case 1.1: There is no semantic connection between the object classes A and B.

In this case the inheritance mechanism consists solely of that given by the connection of the types. This means that all methods defined for object type A can also be applied to instances of the object class B and that every attribute and every relationship defined for object type A is also defined for object type B; but there is no connection between the instances of class A and class B, i.e. there is no transfer of the values of attributes or relationships from class A to class B.

Case 1.2: There is a semantic connection between the object classes A and B.

For the following the nature of the semantic relationship, i.e. roleof or categoryof is not relevant. In this case their inheritance behavior coincides. First of all one part of the inheritance is via the subtype hierarchy, i.e. methods of A can be applied to B and attributes and relationships of A are defined in B. On top of that there is an inheritance of values of attributes and relationships. For every instance of the class B there exists a corresponding instance of the class A. Now let P be an attribute or a relationship defined in the object class A. Then the value of the property P for an instance b of the class B is determined as follows: if P has a value defined in b then this is the actual value of P, but if P has no value defined in b then the value is taken from the corresponding instance of the object class A. This way there is a transfer of values between the object class A and the object class B.

Case 2: The object type B is not a subtype of the object type A.

Case 2.1: There is no semantic connection between the two classes. Then there is clearly no inheritance.

Case 2.2: There is a semantic relation between the two classes in question.

As we shall explain now this must be a roleof connection. To see this, note that by the definition of categoryof the object class B represents the same real world objects represented by the object class A in the same context but with some extra information describing the refinement. However, in such a case the object type B must contain all the properties of the object type A and some additional ones to reflect the more refined information known in the object class B. But this is precisely the case if the object type B is a subtype of the object type A. Hence, if the object class B is a categoryof an object class A, then the corresponding objecttype B must be a subtype of the object type A.

In summary, we only have to consider the case that the object class B is a roleof the object class A. The attributes and relationships that can get assigned a value in the object class B are only the attributes and relationships that are defined in object class B. Moreover, if an attribute or relationship defined for object class B has not been assigned a value, then this attribute or relationship is undefined.

Now, suppose that a method is defined by the object class A and is not defined by the object class B, then it is considered defined by

```
objecttype RESEARCH_ASSISTANT
  subtypeof:      ASSISTANT
  attributes:
    Research:     STRING
    PosPercent:   PERCENTTYPE

objecttype TEACHING_ASSISTANT
  subtypeof:      INSTRUCTOR,
                  ASSISTANT
  attributes:
    PosPercent:   PERCENTTYPE


objecttype SECTION
  memberof SECTIONS
  attributes:
    Department:   STRING
    SectionNo:    NATURAL
    NoOfStudents: NATURAL
    Room:         ROOMTYPE
    Time:         TIMETYPE
  relationships:
    Instructor:   INSTRUCTOR
    Students:     STUDENTS

objecttype SECTIONS
  setof:          SECTION
  attributes:
    NoOfSections: NATURAL
```

```
class research_assistant
  objecttype: RESEARCH_ASSISTANT
  categoryof: assistant


class teaching_assistant
  objecttype: TEACHING_ASSISTANT
  categoryof: instructor, assistant


class section
  objecttype: SECTION
  essential: SectionNo
 relationships:
  Students: students


class sections
  objecttype: SECTIONS
```

Figure 1: An Example, Second Part