INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

Persistence in the Object-Oriented Database Programming Language VML

Wolfgang Klas *, Volker Turau †

TR-92-045

July 1992

Abstract

In this paper the principles of handling persistent objects in the object-oriented database programming language VML is presented. The main design criteria of VML with respect to persistence were: persistence independent programming, data type completeness and operations manipulating the extension of a class. After defining the above mentioned concepts an example is used to compare the modelling and computational power of VML with the database programming languages Adaplex, PS-algol, and Galileo. The distinction of types and classes is the basis for defining persistence in VML. Instances of classes are always persistent and those of data types are always transient. All instances are referenced by object identifiers, values of datatypes are referenced independently of the fact whether they are attached to persistent objects (and are therefore persistent itself) or whether they are "stand alone".

*On leave from GMD-IPSI, Dolivostr. 15, D-6100 Darmstadt, Germany; e-mail: klas@darmstadt.gmd.de
†GMD-IPSI, Dolivostr. 15, D-6100 Darmstadt, Germany; e-mail: turau@darmstadt.gmd.de

Persistence in the Object-Oriented Database Programming Language VML

Wolfgang Klas †
International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, CA 94704, USA
e-mail: klas@ICSI.Berkeley.EDU

Volker Turau
GMD-IPSI
Integrated Publication and
Information Systems Institute
Dolivostr. 15, D–6100 Darmstadt, FRG
e-mail: turau@darmstadt.gmd.de

Abstract

In this paper the principles of handling persistent objects in the object-oriented database programming language VML is presented. The main design criteria of VML with respect to persistence were: persistence independent programming, data type completeness and operations manipulating the extension of a class. After defining the above mentioned concepts an example is used to compare the modelling and computational power of VML with the database programming languages Adaplex, PS-algol, and Galileo. The distinction of types and classes is the basis for defining persistence in VML. Instances of classes are always persistent and those of data types are always transient. All instances are referenced by object identifiers, values of datatypes are referenced independently of the fact whether they are attached to persistent objects (and are therefore persistent itself) or whether they are ”stand alone”.

---

† On leave from GMD-IPSI, Dolivostr. 15, D–6100 Darmstadt, Germany; e-mail: klas@darmstadt.gmd.de

1 Introduction.

There are two main roots for object-oriented database systems: the development of semantic data models and the development of abstraction-based programming languages. This is reflected in the two different approaches to build an object-oriented database. The bottom-up approach tries to extent conventional database technology to become object-oriented by introducing richer data models including facilities for describing the operational semantics of the entities. The top-down approach tries to enrich object-oriented programming languages by database features such as persistence and object sharing.

The aim of the VODAK project at GMD–IPSI is to develop an object-oriented database system. The database programming language of VODAK is called VML (Vodak Modelling Language). VML is an object-oriented language based on a system of meta classes. In a programming language offering or manipulating persistent objects, there are many design issues which must be resolved. Over the time several requirements with regards to persistent programming languages evolved.

The first one is ”persistence independent programming”. A language should be defined that a procedure or a method may be written without knowing whether it will be supplied with persistent or transient data as the actual values of its parameters (i.e. code used to manipulate a value should not depend on its persistence). This implies that the programmer is freed from the burden to include explicit statements to initiate or organize transfer of data objects in the code. The required transfers between stores should be inferred from the operations on the data.

The second one is ”data type completeness”. All data types must enjoy equal status within the language and the rules for using the data types must be complete. In existing persistent programming languages some data types have been allowed to have only persistent instances, others have been allowed only transient instances. Data type completeness implies that type checking rules can be applied and programs carry with them enough information that they can be understood without recourse to other texts. Data type completeness is best explained with the help of type constructors, i.e. types that are parameterized by other types (e.g. set, array, ..). If the language allows a *set*(*t*) construct, then it should allow any type for t (e.g. *set*(*set*(*array-of integer*))). Whether the set of type constructors should be fixed in the language or whether the user should be allowed to define new constructors is still open. Maybe the ”right” set of type constructors is sufficient for database work.

The third requirement concerns operations on objects. Programming languages are predominantly restricted to operations only on their basic data types (integers, reals, ..) and tend to rely on procedural abstraction to provide operations on the composite data items. In database systems, bulk operations are considered very useful (e.g. select, join, ..). Thus, operations manipulating the extension of a class (i.e. the set of existing instances at a certain time) as a whole are needed. For example iterators over the extension of a class are indispensable.

The purpose of this paper is to describe the design of VML with regards to the above requirements. In order to illustrate these issues we have chosen an example. The example is taken from [1] and covers the relevant aspects. This example is used in [1] to compare the modelling and computa-

tional power of the database programming languages Adaplex, PS-algol, and Galileo. This gives a good framework for comparing VML with existing work.

The example database represents the inventory of a manufacturing company. In particular it represents the way certain parts are manufactured out of other parts: the subparts that are involved in the manufacture of a part, the cost of manufacturing a part from its subparts, the mass increment or decrement that occurs when the subparts are assembled. Manufactured parts may themselves be subparts in a further manufacturing process. Hence, the subpart relation forms an acyclic graph. In addition, certain information must be held on the parts themselves: their name, identifying number and, if they are imported, (i.e. manufactured externally) the supplier and purchase cost. As in [1] we present a solution for the following four tasks which we have slightly extended:

1. Describe the database.
2. Print the names, cost and mass of all imported parts that cost more than $100. Furthermore, increment the costs of each of these parts by 10%.
3. Print the total mass and total cost of a composite part.
4. Record in the database a new manufacturing step, i.e. how a new composite part is manufactured from subparts.

This example covers certainly only very few aspects of what would actually be involved in a manufacturing database, but it covers all those aspects needed to illustrate the features of VML with respect to the above listed requirements.

The paper is organized as follows: Section 2 briefly describes those features of VML which are relevant to understand the example and to illustrate the concept of persistency. Section 3 focuses on the concept of persistent objects, and section 4 discusses transient values and their relationships to persistent objects. Section 5 discusses the notion of persistency in the context of a class extension, and section 6 concludes the paper. The appendix shows the complete implementation of some classes used in the sample schema.

2 The VODAK Model Language (VML).

The VODAK Model Language (VML) [6] is an open object-oriented data model. It provides concepts for the definition of database schemas, object classes, properties, and methods. The concept of metaclasses [7] allows to define specific semantic modelling constructs, such as specialization, aggregation, component-of, which can be plugged in to the kernel model. This openness of VML allows to tailor the model to specific application needs ([8], [9], [10]). In this section we only describe the relevant concepts of VML with respect to persistence. In the following the concepts are illustrated by the database schema of our sample database.

Database Schemas

A VML *database schema* contains the definition of classes, object and data types, constants, and the implementations of the methods defined for the instances of classes. Furthermore, a database schema can specify some definitions to be imported from another database schema. Our sample

database schema *Manufacturer* contains the definitions for four classes: *Part*, *Base*Part, *CompositePart*, and *Supplier* amongst the definition of a constant and a data type. The description of this database, which provides a solution for task 1, is shown in Figure 1.

Objects, Classes, and Types

Classes collect objects of the same type. Every class has associated an object type as its instance-type which specifies the structure of the class's instances and the methods defined to operate on these instances. For example, the class *Part* has associated the instance-type *Part_InstType* which specifies a few general methods defined for parts, e.g., *name()* retrieving the name of a part, *id()* retrieving the unique part identifier, and *cost()* retrieving the total cost of a part. Optionally, a class may have associated an object type as its own-type which specifies properties and methods of the class itself. Usually, own-types specify specific methods to create and initialize new instances.

In general, object types associated with classes specify methods and properties. In Figure 1 only the interface specification is shown. The properties and the implementations of the methods are given later. The interface specifications consist of the signatures of the methods, i.e., the method name, a (possibly empty) list of formal parameters and their domains, and, optionally, the domain of the result returned by the method.

Every class is a first class object and is an instance of another class, called its metaclass. Treating classes as regular objects allows to apply the same mechanisms defined for instances also to classes. In our sample database schema the classes are instances of either the metaclass specified with the METACLASS-clause or a predefined default metaclass if no such clause is specified as in the definition of the class *Supplier*. Further details on metaclasses are given in the following.

Semantic Modeling Primitives and Metaclasses

VML provides a specific mechanism which allows to tailor the data model to meet specific requirements. Specific modelling primitives can be introduced into the data model by defining appropriate metaclasses. Every class has associated such a metaclass. A metaclass defines specific methods for the class and its instances it is associated to. The set of methods defined for a class *C* consists of the methods provided with the associated metaclass and the own-type of *C*. The set of methods defined for the instances of the class *C* is determined by the methods provided by the metaclass and the instance-type of *C*. Hence, the class and its instances are able to behave according to the semantics provided by the metaclass.

In our example we have to model parts which can be simple base parts bought from a supplier or composite parts manufactured from other parts. To reflect this situation we define a class *Part* which is categorized into two disjoint sets with respect to the composition of parts. These sets are represented by the classes *BasePart* and *CompositePart*. More precisely, we define the classes *BasePart* and *CompositePart* to be *category specializations* of the class *Part*. That is, every part is either a base part (i.e., instance of class *BasePart*) or a composite part (i.e., instance of class *CompositePart*). As an instance of class *Part* a part is represented just as a general part abstracting from its categorization into base part or composite part. The semantics, i.e., the behavior, associated with the semantic modelling primitive *category specialization* is provided by the metaclasses *GenCatSpecClass* and *CatSpecClass*. In our sample database schema these two classes are im-

```
SCHEMA Manufacturer
  IMPORT GenCatClass, CatSpecClass FROM ObjectSpecializationMetaclasses
  DEFINE MAXQUANTITY 200            // maximum number a part can be used as a subpart
  DATATYPE Subpart = [ comp: Part; quantity : 1 .. MAXQUANTITY ];

  CLASS Part METACLASS GenCatClass
    OWNTYPE     Part_OwnType
        METHODS     findPart(name: STRING) : Part; // returns the part with that name
    INSTTYPE    Part_InstType
        METHODS     name() : STRING;                // returns the name of the part
                    setName(aValue: STRING);        // assigns a name to the part
                    id(): STRING;                   // returns the identifier of the part
                    setId(aValue: STRING);          // assigns an identifier to the part
                    cost(): REAL;                   // returns the total cost of the part
                    mass() : REAL;                  // returns the mass of the part
  END

  CLASS BasePart METACLASS CatSpecClass
    OWNTYPE     BasePart_OwnType
        METHODS     create(id: STRING, name: STRING, supplier: Supplier,
                           purchaseCost : REAL, mass : REAL) : BasePart;
                    findPart(name: STRING) : Part; // returns the part with that name
    INSTTYPE    BasePart_InstType
        METHODS     mass() : REAL;                  // returns the mass of the part
                    setMass(aValue: REAL);          // assigns the mass
                    supplier() : Supplier;          // returns the supplier of a base part
                    setSupplier(aValue: Supplier);  // assigns the supplier to a b. part
                    cost() : REAL;                  // returns the cost of the base part
                    setCost(aValue: REAL);          // assigns the cost of a base part
    INIT    defCategoryOf(Part)
  END

  CLASS CompositePart METACLASS CatSpecClass
    OWNTYPE     CompPart_OwnType
        METHODS     // creates a new composite part
                    create(id: STRING, name: STRING, initialSubparts : { Subpart },
                           assCost: REAL, totalMass: REAL) : CompositePart;
                    findPart(name: STRING) : Part; // returns the part with that name
    INSTTYPE    CompPart_InstType
        METHODS     mass() : REAL;              // returns the mass of a composite part
                    setMass(aValue: REAL);      // assign the total mass of a comp. part
                    deltaMass() : REAL;         // returns the (de-) or increment of mass
                    setDeltaMass(aValue : REAL); // assigns the de/increment of mass
                    cost() : REAL               // returns the total cost of a composite part
    INIT    defCategoryOf(Part)
  END

  CLASS Supplier
    INSTTYPE    Supplier_InstType   .... // not further specified here
  END
```

Figure 1: VML schema for the example database.

ported from the VML schema *ObjectSpecializationMetaclasses*. The class *Part* has associated the metaclass *GenCatSpecClass* which provides the semantics and behavior needed to categorize the class into other classes. The classes *BasePart* and *CompositePart* have associated the metaclass *CatSpecClass* which provides the semantics and behavior needed to define both classes as a categorization of the class *Part*. Details about the specific methods provided for classes and their instances by the metaclasses *GenCatSpecClass* and *CatSpecClass* are given later.

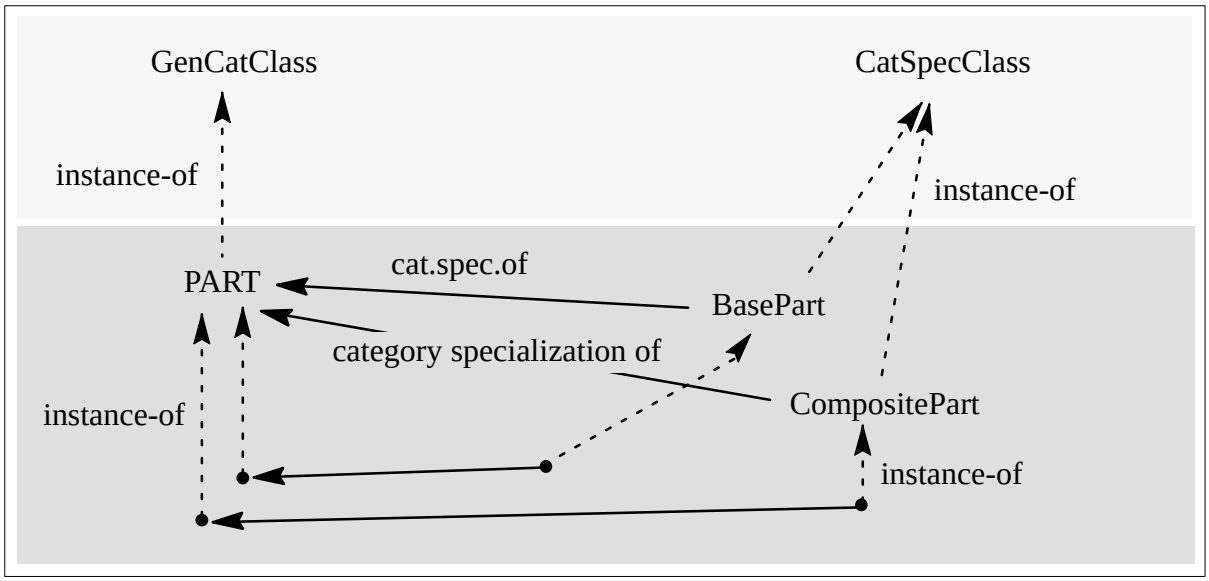

Figure 2: Category specialized classes *BasePart* and *CompositePart* are disjoint partitions of class *Part* with respect of the composition of a part. That is, an instance of class *PART* is categorized as an instance of either class *BasePart* or clas *CompositePart*

Message Passing and Method Execution

The properties of an object can be accessed (read or manipulated) only through the execution of methods defined for it. The execution of a method *m* is invoked by sending a message *rcvr–>m*(*arguments*) to the object *rcvr.*

The semantics of sending a message *rcvr–>m*(*arguments*) to an object are as follows:

- If the method *m* is defined for the object *rcvr,* the code specified for *m* is executed.
- If the method *m* is not defined for the object *rcvr,* the message *rcvr.inheritanceBehavior*(m, *arguments*) is executed, where the method *m* and its *arguments* are passed as arguments to the user specifiable method *inheritanceBehavior*. The implementation of this method determines the future execution of the method *m* within the scope of other objects existing in the database that may even be members of other object classes.

The delegation of messages to other objects via the method *inheritanceBehavior* allows, as we shall see in the next chapter, the specification of a particular inheritance behavior for semantic relationships such as category specialization between objects. The method *inheritanceBehavior* is implemented by a metaclass for the instances of a class. In particular, this ability has proven useful, when adding specialized modelling primitives for hypermedia and argumentative networks [8] and for database integration [9] to VML. Table 1 shows all the predefined methods defined for classes and their instances.

Method	Receiver	Semantics	Defined by
class()	objects, classes	returns the object identifier of the receiver's class	*system predefined*
new()	classes	creates a new object as an instance of the class and returns the object identifier of the newly created object	*system predefined*
allInstances()	classes	returns the set of object identifiers of all instances of the class	*system predefined*
delete(instance)	classes	removes and destroys that instance of the class which is identified by the parameter.	*system predefined*
isInstance(instance)	classes	tests whether the object identified by the parameter is an instance of the class or not.	*system predefined*

Table 1: Methods defined for all objects, i.e., classes and their instances unless elsewhere redefined.

Specific Metaclasses for Category Specialization

VML constitutes a kernel model which can be tailored to specific application needs. It provides a few predefined metaclasses for specific semantic modelling primitives such as object specialization, object generalization and aggregation. As previously mentioned, the metaclasses *GenCatClass* and *CatSpecClass* provide the necessary behavior for classes and their instances which are categorized into several disjoint sets by appropriate method definitions. A detailed description of these two metaclasses including their definition in VML is given in [6]. In this paper we only describe the relevant part of the interface provided by the metaclasses in order to explain the various tasks defined in section 1. Table 2 shows the specific methods provided for classes and instances which are categorized into several disjoint sets as far as needed for our examples.

With the INIT-clause of a class definition one can specify some methods to initialize the class. In our example (Figure 1) the classes *BasePart* and *CompositePart* are initialized by the method *defCategoryOf* to be a category specialization of the class *Part.* The method *defCategryOf* is provided by the metaclass *CatSepcClass* and is defined for classes[1].

1. In the actual implementation of the metaclass *CatSpecClass* (as shown in the appendix) the method *defCategoryOf* takes a second argument which is used to distinguish between several categorization criteria, e.g., parts can be categorized into base and composite parts with respect to their composition, but they also can be categorized into licensed and non-licensed parts with respect to the licence status of a part. But as we do not need this feature in this paper we do not describe it in more detail and ignore it in the description herein.

Method	Receiver	Semantics	Defined by
new()	instances of metaclass *CatSpecClass*	creates a new instance of the receiver class as a category-specialized object of a general instance which is automatically created too. Returns the object identifier of the newly created category specialized instance.	*CatSpecClass* (instance-type)
defCategoryOf(inst)	– ” –	defines the receiver class to be a category-specialization class of the general class identified by the first parameter with respect to the aspect identified by the second parameter.	– ” –
hasCategory()	instances of a class with metaclass *CatSpecClass*	returns the identifier of that instance which represents the receiver object in the appropriate category with respect to the aspect identified by the parameter.	*CatSpecClass* (instance-instance-type)
categorySpecializationOf()	– ” –	returns the identifier of that instance of which the receiver object is a category specialization.	*CatSpecClass* (instance-instance-type)

Table 2: Specific methods defined for the semantic relationship category specialization as far as they are needed in the examples. There are additional methods available with that semantic relationship but they are not discussed here.

3 Persistence in VML.

The term persistence is used to describe that property of data that determines how long it should be kept. It is an orthogonal property of data, that any data item may exist for an arbitrarily long time (e.g. longer than the duration of one program execution). Some form of orthogonal persistence exists for some time in other systems [3]. It consists of a mechanism to save and restore the current workspace (so-called all–or–nothing persistence). Procedures can be saved in files and read in again in some subsequent session. Problems of scale, lack of a transaction mechanism, and lack of adequate mechanism for the independent development of program and data are the chief problems.

The starting point for defining persistence in VML is the distinction of types and classes. Roughly speaking instances of classes are always persistent and those of data types are always transient. Thus, the distinction between types and classes serves as a platform for defining sharing of objects as well as persistence of objects. This decision was motivated by the general modelling philosophy that every interesting entity of the domain of interest should become a class. As a consequence all interesting entities are persistent and can be shared. This way VML does not fully support data type completeness but it supports *class completeness* which is the relevant requirement in an object-oriented framework: the instances of all classes are persistent regardless of the corresponding object type. This approach towards persistence differs from other systems where persistence is

defined via reachability (e.g. CODL), that is the collection of objects form a directed graph where the edges are established through references and the root is the database. All objects which are reachable from the root object are automatically persistent.

Our approach differs in certain aspects from this graph based approach. In our model there are several directed graphs whose entry points are the databases. From the roots the classes can be reached and from there the instances and from those via the properties other instances of classes, which are also referenced by their classes. Hence, everything which is reachable from a root is also persistent. But this is a consequence of our definition and not the definition itself. Another difference is that in our model there is no need for garbage collection, this is because an object disappears only if it was deleted explicitly. The deletion of an object will never have the consequence that other object are no longer reachable, as it is the case in these graph based models where garbage collection is necessary.

A consequence of this decision is that VML does not need an explicit operator for making objects persistent. In case a language doesn't provide a mechanism for automatically making objects persistent, there must exist procedures for doing this (e.g. PS–Algol [5]). If the language does not provide overloading this will be cumbersome, because there must be such a procedure for every type used in the schema. In VML the creation of a new instance of a class automatically implies that this instance will be inserted into the database. This is achieved by a *new* operator (see Table 1 andTable 2). This is similar to the approach taken in Adaplex [4].

In any system that supports some form of persistence, a mechanism is needed to enable a user of that system to name, and subsequently access persistent objects. Hence there must exist a binding between symbols in the program and objects in the persistent store. In a language without persistence, the long term store has traditionally been implemented by files. The persistent name space consists of the file names maintained by the operating system. To access persistent objects in this case, a run time call to a system routine establishes the binding between the program and the file and read/write calls perform the actual access. In VML the persistent resources are the instances of the classes. Hence, mechanisms are needed to allow the user of a persistent object to specify the object in such a way that the system can locate the object and load it. The persistent name space consists of names of databases and of names of classes. As shown in figure 1 the names of classes are declared within a schema definition.

Database names are defined when a database is created for a specific schema by the CREATE DATABASE statement (see figure 3). Existing databases can be used in a VML program by associating it to a database variable. The declaration of a database variable specifies the schema name of a database which can be bound to the database variable by a subsequent OPEN statement. This mechanism allows to open several databases within one program.

A database variable provides the root to access the objects of a database. The only objects which can be accessed directly from a database variable are those objects which represent a class. Access is performed by using the class name qualified with the name of the database variable, e.g., in our example program *PartDB.Part* or *PartDB.BasePart*. The qualification of a class name can be omitted, if the class name is unambigous, i.e. if there is at most one database variable of each schema and if the class names of different schemata are disjoint. All other objects, i.e. instances in a database which are not classes, can be accessed only starting from a class object using the methods defined in the interface.

```
PROGRAM   CreateManufactureDatabase;
  CREATE DATABASE ”ManufacturingDatabase” SCHEMA Manufacturer;
END;

PROGRAM   BuildUpManufactureDatabase;
  DATABASE  PartDB  SCHEMA  Manufacture;

  OPEN  PartDB  DATABASE ”ManufacturingDatabase”
  // Statements which create base parts and composite parts. An example is given
  //  later in chapter 5.
  FORALL p in { PartDB.Part–>allInstances() } DO
    PRINT <<  p–>name();  <<  NEWLINE;
  END
  CLOSE  PartDB;
END;
```

Figure 3: The creation, opening and closing of a database in VML.

In addition to this user level naming scheme, the system is based on a uniform method for naming persistent objects. This relies on the use of unique object identifiers (oid’s) that are guaranteed to be unique throughout the system. An oid is never used twice in the lifetime of a database. Furthermore, oid’s are immutable i.e. they cannot be changed once they have been created and bound to an object, neither by the user nor by the system. They are the handles to persistent objects and are invisible in an application program. Therefore, oid’s allow persistence independent programming in VML. This is because all references to instances of classes in the code of a method are made implicitly through them. This aspect of VML is discussed in detail in the next section. Object identifiers allow the VODAK object manager to locate the objects in the persistent store.

So far we have described persistent objects, but there are situations where transient values are needed (for example in the body of a method). In VML values of datatypes are not persistent. Hence, in a body of a method it is possible to define structured datatypes and to use them freely. The scope of their existence is then the enclosing method body. Of course, when a value of such a datatype is assigned to a property of a persistent object, these values will be persistent.

4 Variables and Assignments.

In VML a variable is introduced with a variable definition clause of the form:

VAR *identifier : datatype;*

where **VAR** is a keyword. This clause introduces a variable with name *identifier.* The variable is constrained to assume values from the domain of the specified type. There are basically three possibilities for datatypes: primitive types, structured types and class identifiers. The structured types are build with the following type constructors: record, variant, set and array. These type constructors can be nested to any depth and their basic ingredients are the primitive types (Boolean, Integer, etc.) and the class identifiers. The domains of these types are the usual for primitive types and for class identifiers they consist of the set of possible instances of the corresponding classes. The domains of the structured types are defined on the basis of the above defined domains in the usual manner for each constructor individually. All the domains contain an element NIL. For example in the implementation of the method *cost()* for the class *CompositePart* the two variables *aSubpart* and *totalCost* are declared (see figure 6). The first one is of the type *Subpart* which is a record type (note the type of the first field is a class, see figure 1) and the type of the second is the primitive type *REAL*.

In VML variables can be regarded as locations for values of the corresponding datatype. These locations reside in a time varying store and introduce modifiability in the language. For example, the variable declaration "**VAR** totalCost : REAL" denotes a location for a value of the domain of REAL. If the datatype of the variable is a classidentifier then the location for a variable can hold a handle to an instance of the corresponding class. Since the handles to instances of classes are the object identifiers, this location will eventually contain an oid. But from a users' point of view it will be the instance itself, because the set of methods applicable to that variable are exactly those defined for the instances of the corresponding class. Furthermore, any update of a property of that instance is also an update of the persistent object. This concept is illustrated in the implementation of task 2, which is given in figure 4. This concept implies that the sizes of the locations for variables with datatype a class identifier are always the same and are known at compile-time.

The implementation of task 2 defined in section 1, is given in figure 4. The names, cost, and mass of all imported parts that cost more than $100 are printed within a loop over the set of the appropriate objects. Within this loop the costs of the parts are increased by 10%. This increment is achieved with the method setCost( ). In case a method is sent to an object and this method changes the state of the object, this is automatically propagated to the persistent store. Thus, sending the method setCost( ) to a variable of type BasePart updates automatically the property purchaseCost of the instance hold by that variable in the database (the implementation of the method setCost( ) is omitted in figure 6). Hence, the user is freed from the burden of taking care that the updates of properties are correctly accomplished in the database.

```
VAR impPart, aPart: BasePart;
FORALL impPart IN { aPart IN BasePart–>allInstances() | aPart–>cost() > 100 } DO
   PRINT << "Name: " << impPart–>name() << NEWLINE;
   PRINT << "Cost: " << impPart–>cost() << NEWLINE;
   PRINT << "Mass: " << impPart–>mass() << NEWLINE;
   impPart–>setCost(impPart–>cost() * 1.1);
END
```

Figure 4: VML implementation of task 2, i.e., print the name, cost, and mass of all imported parts that cost more than $100.

The variable declaration **"VAR** compPart : CompositePart" denotes a location for an instance of the class *CompositePart* (which must be defined previously) with the name *comPart* (see figure 5). Initially the location contains the value NIL. For another example consider the following record type: **DATATYPE** Subpart = [comp: Part; quantity : 1 .. MAXQUANTITY] (see figure 1). A variable declaration of the form **"VAR** aSubpart : Subpart" sets aside space for a handle to an instance of the class Part and for a value of the domain of the integer subrange type 1 .. MAXQUANTITY (see figure 6).

The assignment operator := is provided to change the value of a location in the store. An assignment to a variable changes the value of the corresponding location but has no effect on the object being assigned or on the previous value of that location. For example an assignment of the form compPart := CompositePart–>findPart("Ship"); retrieves an instance of the class CompositePart with name "ship" and assigns this object to the location named compPart (see figure 5).

The statement compPart := CompositePart–>new() generates a new persistent object. The assignment of an instance to a location has no effect on the instance itself (i.e. the value of its attributes are unchanged etc.). Thus, if the variable compPart gets a new value assigned, the former object is not changed (see for example the variable *impPart* in the implementation of task 2; figure 4). If the data type of a variable is a class identifier, the declaration has the effect of supplying space for the persistent handle for an instance of the class Part. In contrast to this the operator new generates a new instance of a class, makes this instance persistent and returns this new object. Hence, the new operator and the declaration of a variable have totally different semantics.

The equality operator which is defined for variables of the same data type compares the contents of the corresponding locations. As a consequence in case the datatype of a variable is a class identifier the object identifiers are compared. This is called identity in the literature about object-oriented programming languages [2]. To achieve the semantics of shallow or deep equality, a user has to write appropriate methods using the system provided equality operators in the implementation.

Example Task 3

The implementation of task 3 defined in section 1, is given in Figure 5 . First, a part called "Ship" is selected and its representation as a composite part is then assigned to the variable compPart. Then the methods cost() and mass() are sent to the object referred to by the variable to compute the total

```
VAR compPart : CompositePart;
compPart := CompositePart–>findPart(”Ship”);
IF (compPart != NULL) THEN
   PRINT <<  ” Cost: ”  <<  compPart–>cost()  <<  NEWLINE;
   PRINT <<  ” Mass: ”  <<  compPart–>mass()  <<  NEWLINE;
END
```

Figure 5: VML implementation of task 3, i.e., print the total cost and total mass of a composite part.

cost and mass of the composite part and, subsequently, to print the results. The implementation of the methods cost() and mass() are given with the object type *CompPart_InstType* associated with class *CompPart* (see Figure 6).

The method cost() computes the total cost by adding up the assembly cost and the cost of the subparts times the number of subparts. The cost of each component is computed by the method cost() which is sent to every subpart referred to by the property *subparts* which is defined for composite parts with type *CompPart_InstType*. Every subpart is an instance of class *Part* (compare the data type definition *SubPart*). Hence, at this point, the method cost() implemented for parts, i.e., the method implemented with type *Part_InstType* is executed. This method in turn calls the method cost() which is either the cost method implemented for base parts, i.e., implemented with type *BasePart_InstType*, or the cost method implemented for composite parts.

The total mass of a composite part is computed in a similar manner to the total cost by the method mass() defined for composite parts and it is not further explained.

```
OBJECTTYPE CompPart_InstType
  IMPLEMENTATION
    PROPERTIES    subparts :      { Subpart };
                  deltaMass :     REAL;
                  assemblyCost:   REAL;
    MEtHODS
      compositeMass() : REAL;              // returns the mass of a composite part
        { VAR aSubpart : Subpart;
                totalMass: REAL;
          totalMass := 0;
          FORALL aSubpart IN subparts DO
            totalMass = totalMass + aSubpart.comp–>mass() * aSubPart.quantity;
          END
          return totalMass }
      mass() : REAL;          { return self–>compositeMass() + deltaMass; }
      cost() : REAL;                       // returns the total cost of a composite part
        { VAR aSubpart : Subpart;
                totalCost: REAL;
          totalCost := assemblyCost;
          FORALL aSubPart IN subparts DO
            totalCost = totalCost + aSubPart.comp–>cost() * aSubPart.quantity;
          END
          return totalCost }
END

OBJECTTYPE Part_InstType
  IMPLEMENTATION
    PROPERTIES    name:  STRING;
                  id:    STRING;
    METHODS
      cost() : REAL;          { return self–>hasCategory()–>cost(); }
      mass() : REAL;          { return self–>hasCategory()–>mass(); }
END

OBJECTTYPE BasePart_InstType
  IMPLEMENTATION
    PROPERTIES    supplier:       Supplier;
                  purchaseCost:   REAL;
                  mass:           REAL;
    METHODS
      mass() : REAL;          { return mass; }
      cost() : REAL;          { return purchaseCost; }
END
```

Figure 6: VML implementation of the methods cost() and mass() used in task 3. Only the relevant fragments of the implementation of the object types are shown.

5 Manipulation of the Extension of a Class.

As already stated above the extension of a class (i.e. all instances of a class) is persistent. This extension will change during the lifetime of a database. The main operations to manipulate the extension of a class are creating new instances, deleting existing instances and updating properties of existing instances. The first operation is accomplished by the new operator. This operator creates a new object as an instance of a class.Consider the following program segment:

```
VAR obj : Part;
obj := Part–>new();
```

The variable obj gets assigned an instance of the class Part. This instance is newly created with the new operator. After the execution of the program segment the extension of the class Part contains one additional element. All the properties of this new instance have the value NIL.

The deletion of existing objects has far more consequences than creation of an object. The main issues are referential integrity and complex objects. Is it possible to delete an instance which is referenced by other instances and if so, what happens to the references? Furthermore, is it possible to delete a complex object as whole (i.e. an instance with all the objects it references)?

One approach is to define persistence with the help of a ”reachability relation”. Each object which is reachable from the database root is persistent. If an object is referenced by another object, then it is possible to remove the former object from the extension of its class. But the object is still in the database, because it is still referenced by an object reachable from the database root. Hence, the instance is still present but not as a member of the extension of its class. To avoid this anomalous situation a different approach is used in VML.

VML provides two methods for deleting instances of classes, which are in the interface of every instance. The first one is called *delete*. If this method is sent to an instance of a class, the instance is deleted from the extension of its class if this object is not reference directly by another object. Otherwise nothing is done. In the former case, the instance is removed from all sets it is contained in. Hence, this method guarantees referential integrity. The second method to delete objects is called *deleteall*. If this method is sent to an instance of a class, the instance is removed from the extension of its class and all references to it are set to NIL. Furthermore, the instance is removed from all sets it is contained in. So this method also guarantees referential integrity. The implementation of *deleteall* is based on a technique called ”lazy deletion”.

On the basis of these two methods, it is possible to define methods with a special delete semantic (for example to delete an object and all the objects it references, etc.)

In VML we have adopted the following strategy which is called lazy deletion. The *deleteall* method marks the object identifier such that a successive attempt to access this object realizes that the object is deleted. The object itself is not immediately deleted. The marked objects are deleted at the end of a transaction. Any attempt to access a deleted object through a reference from an existing object will be recognized, because every object access method checks the object whether it is marked or not.

As already mentioned bulk operations are needed in database applications. Therefore, VML provides a method allInstances in the interface of every class. The method returns the set of all in-

stances of that class, i.e. Part–>allInstances() is the set of all instances of the class Part. The datatype of the return value is {Part}. This value can then be processed with the operations available for sets (e.g. intersection, iterators, etc.). Another category of bulk operations is given by the query language (see [6]).

Example Task 4

The implementation of task 4 defined in section 1, is given in Figure 7. First, three base parts are created: a button, a case, and a cable. The variables *part1*, *part2*, and *part3* represent the quantity of the usage of these base parts in a composite part ”Mouse”, which is manufactured from the three base parts. The implementation of the methods *create* defined for the classes *BasePart* and *CompPart* are given in Figure 8. These methods create both a general representation of the base parts and the composite part as instances of class *part* and the base part or composite part representation through instances of *BasePart* and *CompositePart* employing the method *new* provided with the metaclass *CatSpecClass*. The *create* methods return the newly created instance of either *BasePart* or *CompositePart*. The *categorySpecializationOf* method returns the corresponding instance of class *Part* the receiver object is a categorization of.

```
// DATATYPE Subpart = [ comp: Part; quantity : 1 .. MAXQUANTITY ]
VAR    part1, part2, part3 : Subpart;
// Create a few base parts

part1.comp := BasePart–>create(”BTN 90–020/C”, ”Button”,                    // $20, 10 g
                               ”SunMiircoSystems”, 20, 10)–>categorySpecializationOf();
part1.quantity := 3;
part2.comp := BasePart–>create(”CC 90–020/D”, ”Case”,                       // $80, 200 g
                               ”SunMiircoSystems”, 80, 200)–>categorySpecializationOf();
part2.quantity := 1;
part3.comp := BasePart–>create(”CC 90–020/D”, ”Cable”,                      // $2, 15 g
                               ”SunMiircoSystems”, 2, 15)–>categorySpecializationOf();
part3.quantity := 1;
CompPart–>create( ”MSC 90–001/A”, ”Mouse”, {part1, part2, part3}, 30, 250); // $30, 250g
```

Figure 7: VML implementation of task 4, i.e., record in the database a new manufacturing step, i.e., how a new composite part is manufactured from subparts.

```
OBJECTTYPE BasePart_OwnType
  IMPLEMENTATION
    METHODS // creates a new base part
        create(id: STRING, name: STRING, supplier: Supplier,
                purchaseCost : REAL, mass : REAL) : BasePart;
          { VAR newPartObj : Part; newBasePartObj : BasePart;
            // first create both a base part object and a part object
            newBasePartObj := self–>new();    // see CatSpec_InstType, appendix
            newBasePartObj–>setSupplier(supplier);
            newBasePartObj–>setCost(purchaseCost);
            newBasePartObj–>setMass(mass);
            // get the representation as Part; see CatSpec_InstInstType, appendix
            newPartObj := newBasePartObj–>categorySpecializationOf();
            newPartObj–>setName(name);
            newPartObj–>setId(id);
            return newBasePartObj; }
END

OBJECTTYPE CompPart_OwnType
  IMPLEMENTATION
    METHODS // creates a new composite part
        create(id: STRING, name: STRING, initialSubparts : { Subpart },
                assCost: REAL, totalMass: REAL) : CompositePart;
          { VAR newPartObj : Part; newCompPartObj : CompositePart;
            // first create both a composite part object and a part object
            newCompPartObj := self–>new();  // see  CatSpec_InstType, appendix
            newCompPartObj–>setAssemblyCost(assCost);
            newCompPartObj–>setSubparts(initialSubparts);
            newCompPartObj–>setMass(totalMass);
            // get the representation as Part; see CatSpec_InstInstType, appendix
            newPartObj := newCompPartObj–>categorySpecializationOf();
            newPartObj–>setName(name);
            newPartObj–>setId(id);
            return newCompPartObj; }
  END
```

Figure 8: Implementation of the object creation methods needed for a manufacturing step of a composite object.

6 Conclusions.

This paper has presented the principles of handling persistent objects in the object-oriented database programming language VML. The first design criteria was that programming code should be free from statements to initiate or organize the transfer of objects from/to the persistent store. Thus, the code is written in a form, which can work with persistent or transient data. This persistence independent programming is achieved by taking the distinction of types and classes as the basis for defining persistence in VML. Instances of classes are always persistent and those of data types are always transient. Thus, the distinction between types and classes serves as a platform for

defining the sharing of objects as well as the persistence of objects. All instances are referenced by oid's, values of datatypes are referenced independently of the fact whether they are attached to persistent objects (and are therefore persistent themself) or whether they are "stand alone". This way VML supports *class completeness* which is the relevant requirement in an object-oriented framework: the instances of all classes are persistent regardless of the corresponding object type.

VML provides operators manipulating the extension of a class (i.e. the set of existing instances at a certain time). For example iterators can be defined for every set, including the extension of a class, this way the expressiveness of a full query language is achieved. Furthermore, VML provides two methods for deleting instances of classes, which are in the interface of every instance. These are *delete* and *deleteall*. The first one does not guarantee referential integrity while the second does. On the basis of these two methods, it is possible to define methods with a special delete semantics.

The implementation of task 3 is very simple, because the methods for mass and cost are already included in the schema. In general one can say that application programs will be much shorter and easier to write. This can also be seen from the implementation of task 4. Here, the method *create* is used to record a new composite part. The implementation of the *create* method is based on the method *new*, which creates simultaneously a base part and a part. This way the constraint that each base part is also a part is satisfied and the application program is freed from the burden to check semantic constraints.

7 References

[1] M.P. Atkinson and O.P. Buneman, Types and Persistence in Database Programming Languages, ACM Comput. Surv., 19(2):105–190, June 1987.

[2] S. Khoshafian and G. Copeland, Object identity, Proc. first ACM OOPSLA conference, Portland, Oregon, September 1987.

[3] A. Goldberg and D. Robson, Smalltalk 80: The language and its implementation, Addison-Wesley, Reading, MA, 1983.

[4] J.M. Smith, S. Fox and T. Landers, ADAPLEX: Rationale and Reference Manual, 2nd edition, Computer Corporation of America, Four Cambridge Center, Cambridge, Massachusetts 02142, 1983.

[5] The PS–Algol reference manual – 2nd edition, Department of Computing Science Persistent Programming Research Group, University of Glasgow, Technical Report PPR–12–85, Glasgow G12 8QQ, Scotland, 1985.

[6] Klas W., E.J.Neuhold, R.Bahlke, P.Fankhauser, M.Kaul, P.Muth, T.Rakow, V.Turau: VML – The VODAK Model Language. Technical Report. September 1991.

[7] Klas W.: A Metaclass System for Open Object-Oriented Data Models, Doctoral Thesis, Technical University of Vienna, January 1990.

[8] Wolfgang Klas, Erich J. Neuhold: Designing Hypertext Systems using an Open Object-Oriented Database Model, Technical Report No. 489, Arbeitspapiere der GMD, Birlinghoven, 1990.

[9] Wolfgang Klas, Erich J. Neuhold, Michael Schrefl: Metaclasses in VODAK and their Application in Database Integration, Technical Report No. 462, Arbeitspapiere der GMD, Birlinghoven, 1990.

[10] Wolfgang Klas, Erich J. Neuhold, Michael Schrefl: Using an Object-Oriented Approach to Model Multimedia Data, Computer Communications, Special Issue on Multimedia Systems, Vol. 13, No. 4, pp. 204–216, May 1990.

8 Appendix: Definition of Metaclasses

Figures 9, 10, 11, and 12 show the definitions of the classes, object types, and data types for the metaclasses which implement the category-specialization modeling primitive as used in the example of this paper. The implementation has been realized using the VODAK prototype version 1.0 at ICSI.

```
SCHEMA ObjectSpecializationMetaclasses
   DEFINE ASPECTMIN 1
   DEFINE ASPECTMAX 5
   DATATYPE AspectId = INT;

CLASS GenCatClass METACLASS Metaclass
INSTTYPE GenCatSpec_InstType
INSTINSTTYPE GenCatSpec_InstInstType
END;

CLASS CatSpecClass METACLASS Metaclass
INSTTYPE CatSpec_InstType
INSTINSTTYPE CatSpec_InstInstType
END;
```

Figure 9: Definition of the metaclasses *GenCatClass* and *CatSpecClass* providing the category-specialization modelling primitive.

```
OBJECTTYPE GenCatSpec_InstType SUBTYPEOF Metaclass_InstType;
  INTERFACE
  METHODS    defHasCategory(catClass: OID, aspect: AspectId)   ;
             hasCategories(aspect: AspectId) : {OID}  ;
IMPLEMENTATION
  EXTERN prints(s: STRING); printo(i: OID); printi(i: INT); endline();
  PROPERTIES  hasCategoryCls: ARRAY [SUBRANGE ASPECTMIN .. ASPECTMAX]
                                        OF {OID};
  METHODS
  defHasCategory(catClass: OID, aspect: AspectId)  ;
    { INSERT catClass INTO hasCategoryCls[aspect] END; };
  hasCategories(aspect: AspectId) : { OID } ;
    {  VAR emptySet : { OID} ;
    emptySet := {};
   IF (aspect < ASPECTMIN) | (aspect > ASPECTMAX)
    THEN   {prints('ERROR in <hasCategories>, invalid aspect id.'); endline();
            RETURN emptySet; }
    ELSE    RETURN hasCategoryCls[aspect];
   END; };
END;
OBJECTTYPE GenCatSpec_InstInstType SUBTYPEOF Metaclass_InstInstType;
  INTERFACE
  METHODS    checkHasCategory(aspect: AspectId) : BOOL ;
             initHasCategory(catObj: OID, aspect:AspectId) ;
             hasCategory(aspect:AspectId) : OID ;
             as(catCls:OID) : OID ;
IMPLEMENTATION
  EXTERN prints(s: STRING); printo(i: OID); printi(i: INT); endline();
  PROPERTIES   hasCategoryObj: ARRAY [SUBRANGE ASPECTMIN ..
                                                ASPECTMAX] OF OID;
  METHODS
  checkHasCategory(aspect: AspectId) : BOOL ;
    { RETURN hasCategoryObj[aspect] != NULL; };
  initHasCategory(catObj: OID, aspect:AspectId) ;
    { hasCategoryObj[aspect] := catObj; };
  hasCategory(aspect:AspectId) : OID ;
    { RETURN hasCategoryObj[aspect]; };
  as(catCls:OID) : OID ;
    { VAR cls : OID; result : OID;
    result := NULL;
    FOR I := ASPECTMIN BY 1 TO ASPECTMAX DO {
      IF hasCategoryObj[I] != NULL THEN  { cls := hasCategoryObj[I]–>(OID)class();
                                           IF  cls == catCls
                                               THEN { result := hasCategoryObj[I]; }
                                           END; }
      END; } END;
    RETURN result; };
END;
```

Figure 10: Definition of instance-type and instance-instance-type of the metaclass *GenCatClass*.

```
OBJECTTYPE CatSpec_InstType SUBTYPEOF Metaclass_InstType;
  INTERFACE
  METHODS   defCategoryOf(genClass: OID, asp: AspectId) ;
            createCategoryOf(generalInst : OID) : OID ;
            checkIsCategoryOf(generalClass: OID): BOOL ;
            categoryOf() : OID ;
            new() : OID;
IMPLEMENTATION
  EXTERN prints(s:STRING); printo(o:OID); endline();
  PROPERTIES  categoryOfCls: OID;
              aspect: AspectId;
  METHODS
  defCategoryOf(genClass: OID, asp: AspectId) ;
    { categoryOfCls := genClass;
    aspect := asp;  genClass–>defHasCategory(SELF, asp); };
  createCategoryOf(generalInst : OID) : OID ;
    { VAR categoryObj : OID;
    IF  NOT SELF–>checkIsCategoryOf(generalInst–>(OID)class())
      THEN { prints('Such a category object cannot be created for this object.');
           endline(); RETURN NULL; }
    END;
    IF generalInst–>(BOOL)checkHasCategory(aspect)
      THEN { prints('Creation failed: category for this object already exists.'); endline();
           RETURN NULL; }
    END;
    categoryObj := SELF–>new();
     categoryObj–>initCategoryOf(generalInst);
    generalInst–>initHasCategory(categoryObj, aspect);
    RETURN categoryObj; };
  checkIsCategoryOf(generalClass: OID): BOOL ;
    { RETURN categoryOfCls == generalClass; };
  categoryOf() : OID ;
    {RETURN categoryOfCls; };
  new() : OID;
    { VAR generalInst : OID;
      generalInst := SELF–>(OID)categoryOf()–>(OID) new();
      RETURN SELF–>(OID)createCategoryOf(generalInst); };
END;
```

Figure 11: Definition of the instance-type of the metaclass *CatSpecClass*.

```
OBJECTTYPE CatSpec_InstInstType SUBTYPEOF Metaclass_InstInstType;
  INTERFACE
  METHODS    initCategoryOf(generalInst: OID) ;
             categorySpecializationOf(): OID ;
IMPLEMENTATION
  EXTERN prints(s:STRING); endline();
  PROPERTIES   categoryOfObj: OID;
  METHODS
  initCategoryOf(generalInst: OID) ;
    { categoryOfObj := generalInst; };
  categorySpecializationOf(): OID ;
    { RETURN categoryOfObj; };
  NOMETHOD
    { RETURN  categoryOfObj–>currentMeth(arguments); }
END;
END_SCHEMA;
```

Figure 12: Definition of the instance-instance-type of the metaclass *CatSpecClass*.