The Object-Oriented Database System Manifesto

Malcolm Atkinson
University of Glasgow

François Bancilhon
Altaïr

David DeWitt
University of Wisconsin

Klaus Dittrich
University of Zurich

David Maier
Oregon Graduate Center

Stanley Zdonik
Brown University

Abstract

This paper attempts to define an *object-oriented database system*. It describes the main features and characteristics that a system must have to qualify as an object-oriented database system.

We have separated these characteristics into three groups:

- *Mandatory*, the ones the system must satisfy in order to be termed an object-oriented database system. These are complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility, computational completeness, persistence, secondary storage management, concurrency, recovery and an *ad hoc* query facility.
- *Optional*, the ones that can be added to make the system better, but which are not mandatory. These are multiple inheritance, type checking and inferencing, distribution, design transactions and versions.
- *Open*, the points where the designer can make a number of choices. These are the programming paradigm, the representation system, the type system, and uniformity.

We have taken a position, not so much expecting it to be the final word as to erect a provisional landmark to orient further debate.

1 Introduction

Currently, object-oriented database systems (OODBS) are receiving a lot of attention from both experimental and theoretical standpoints, and there has been considerable debate about the definition of such systems.

Three points characterize the field at this stage: (i) the lack of a common data model, (ii) the lack of formal foundations and (iii) strong experimental activity.

Whereas Codd's original paper [Codd 70] gave a clear specification of a relational database system (data model and query language), no such specification exists for object-oriented database systems [Maier 89]. We are not claiming here that no complete object-oriented data model exists, indeed many proposals can be found in the literature (see [Albano *et al.* 1986], [Lécluse and Richard 89], [Carey *et al.* 88] as examples), but rather that there is no consensus on a single one. Opinion is slowly converging on the gross characteristics of a family of object-oriented systems, but, at present, there is no clear consensus on what an object-oriented system is, let alone an object-oriented database system.

The second characteristic of the field is the lack of a strong theoretical framework. To compare object-oriented programming to logic programming, there is no equivalent of [Van Emdem and Kowalski 76]. The need for a solid underlying theory is obvious: the semantics of concepts such as types or programs are often ill defined. The absence of a solid theoretical framework, makes consensus on the data model almost impossible to achieve.

Finally, a lot of experimental work is underway: people are actually building systems. Some of these systems are just prototypes [Bancilhon *et al.* 88], [Nixon, *et al.* 87], [Banerjee *et al.* 87], [Skarra *et al.* 86], [Fishman *et al.* 87], [Carey *et al.* 86], but some are commercial products, [Atwood 85], [Maier, *et al.* 84], [Caruso and Sciore 87], [G-Base 88]. The interest in object-oriented databases seems to be driven by the needs of design support systems (e.g., CAD, CASE, Office Information Systems). These applications require databases that can handle very complex data, that can evolve gracefully, and that can provide the high-performance dictated by interactive systems.

The implementation situation is analogous to relational database systems in the mid-seventies (though there are more start-ups in the object-oriented case). For relational systems, even though there were some disagreements on a few specific points, such as the form of the query language, or whether relations should be sets or bags, these distinctions were in most cases superficial and there was a common underlying model. People were mainly developing implementation technology. Today, we are simultaneously choosing the specification of the system and producing the technology to support its implementation.

Thus, with respect to the specification of the system, we are taking a Darwinian approach: we hope that, out of the set of experimental prototypes being built, a fit model will emerge. We also hope that viable implementation technology for that model will evolve simultaneously.

Unfortunately, with the flurry of experimentation, we risk a system emerging as *the* system, not because it is the fittest, but because it is the first one to provide a significant subset of the functionality demanded by the market. It is a classical, and unfortunate, pattern of the computer field that an early product becomes the *de facto* standard and never disappears. This pattern is true at least for languages and operating systems (Fortran, Lisp, Cobol and SQL are good examples of such situations). Note however, that our goal here is not to standardize languages, but to refine terminology.

It is important to agree now on a definition of an object-oriented database systems. As a first step towards this goal, this paper suggests characteristics that such systems should possess. We expect that the paper will be used as a straw man, and that others will either

invalidate or confirm the points mentioned here. Note that this paper is not a survey of the state of the art on OODBS technology and do not pretend to assess the current status of the technology, it merely proposes a set of definitions.

We have separated the characteristics of object-oriented database systems into three categories: *mandatory* (the ones that the system must satisfy to deserve the label), *optional* (the ones that can be added to make the system better but which are not mandatory) and *open* (the places where the designer can select from a number of equally acceptable solutions). In addition, there is some leeway how to best formulate each characteristic (mandatory as well as optional).

The rest of this paper is organized as follows. Section 2 describes the mandatory features of an OODBS. Section 3 describes its optional features and Section 4 presents the degrees of freedom left to the system designers.

2 Mandatory features: the Golden Rules

An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first criterion translates into five features: persistence, secondary storage management, concurrency, recovery and an *ad hoc* query facility. The second one translates into eight features: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility and computational completeness.

2.1 Complex objects

Thou shalt support complex objects

Complex objects are built from simpler ones by applying constructors to them. The simplest objects are objects such as integers, characters, byte strings of any length, booleans and floats (one might add other atomic types). There are various complex object constructors: tuples, sets, bags, lists, and arrays are examples. The minimal set of constructors that the system should have are set, list and tuple. *Sets* are critical because they are a natural way of representing collections from the real world. *Tuples* are critical because they are a natural way of representing properties of an entity. Of course, both sets and tuples are important because they gained wide acceptance as object constructors through the relational model. *Lists* or *arrays* are important because they capture order, which occurs in the real world, and they also arise in many scientific applications, where people need matrices or time series data.

The object constructors must be orthogonal: any constructor should apply to any object. The constructors of the relational model are not orthogonal, because the set construct can only be applied to tuples and the tuple constructor can only be applied to atomic values. Other examples are non-first normal form relational models in which the top level construct must always be a relation.

Note that supporting complex objects also requires that appropriate operators must be provided for dealing with such objects (whatever their composition). That is, operations on a complex object must propagate transitively to all its components. Examples include the retrieval or deletion of an entire complex object or the production of a "deep" copy (in contrast to a "shallow" copy where components are not replicated, but are instead referenced by the copy of the object root only). Additional operations on complex objects may be defined, of course, by users of the system (see the extensibility rule below). However, this capability requires some system provided provisions such as two distinguishable types of references ("is-part-of" and "general").

2.2 Object identity

Thou shalt support object identity

Object identity has long existed in programming languages. The concept is more recent in databases, e.g., [Hall *et al.* 76], [Maier and Price 84], [Khoshafian and Copeland 86]. The idea is the following: in a model with object identity, an object has an existence which is independent of its value. Thus two notions of object equivalence exist: two objects can be identical (they are the same object) or they can be equal (they have the same value). This has two implications: one is object sharing and the other one is object updates.

Object sharing: in an identity-based model, two objects can share a component. Thus, the pictorial representation of a complex object is a graph, while it is limited to be a tree in a system without object identity. Consider the following example: a Person has a name, an age and a set of children. Assume Peter and Susan both have a 15-year-old child named John. In real life, two situations may arise: Susan and Peter are parent of the same child or there are two children involved. In a system without identity, Peter is represented by:

```
(peter, 40, {(john, 15, {})})
```

and Susan is represented by:

```
(susan, 41, {(john, 15, {})}).
```

Thus, there is no way of expressing whether Peter and Susan are the parents of the same child. In an identity-based model, these two structures can share the common part (john, 15, {}) or not, thus capturing either situations.

Object updates: assume that Peter and Susan are indeed parents of a child named John. In this case, all updates to Susan's son will be applied to the object John and, consequently, also to Peter's son. In a value-based system, both sub-objects must be updated separately. Object identity is also a powerful data manipulation primitive that can be the basis of set, tuple and recursive complex object manipulation, [Abiteboul and Kanellakis 89].

Supporting object identity implies offering operations such as object assignment, object copy (both deep and shallow copy) and tests for object identity and object equality (both deep and shallow equality).

Of course, one can simulate object identity in a value-based system by introducing explicit object identifiers. However, this approach places the burden on the user to insure the

uniqueness of object identifiers and to maintain referential integrity (and this burden can be significant for operations such as garbage collection).

Note that identity-based models are the norm in imperative programming languages: each object manipulated in a program has an identity and can be updated. This identity either comes from the name of a variable or from a physical location in memory. But the concept is quite new in pure relational systems, where relations are value-based.

2.3 Encapsulation

Thou shalt encapsulate thine objects

The idea of encapsulation comes from (i) the need to cleanly distinguish between the specification and the implementation of an operation and (ii) the need for modularity. Modularity is necessary to structure complex applications designed and implemented by a team of programmers. It is also necessary as a tool for protection and authorization.

There are two views of encapsulation: the programming language view (which is the original view since the concept originated there) and the database adaptation of that view.

The idea of encapsulation in programming languages comes from abstract data types. In this view, an object has an interface part and an implementation part. The interface part is the specification of the set of operations that can be performed on the object. It is the only visible part of the object. The implementation part has a data part and a procedural part. The data part is the representation or state of the object and the procedure part describes, in some programming language, the implementation of each operation.

The database translation of the principle is that an object encapsulates both program and data. In the database world, it is not clear whether the structural part of the type is or is not part of the interface (this depends on the system), while in the programming language world, the data structure is clearly part of the implementation and not of the interface.

Consider, for instance, an Employee. In a relational system, an employee is represented by some tuple. It is queried using a relational language and, later, an application programmer writes programs to update this record such as to raise an Employee's salary or to fire an Employee. These are generally either written in a imperative programming language with embedded DML statements or in a fourth generation language and are stored in a traditional file system and not in the database. Thus, in this approach, there is a sharp distinction between program and data, and between the query language (for *ad hoc* queries) and the programming language (for application programs).

In an object-oriented system, we define the Employee as an object that has a data part (probably very similar to the record that was defined for the relational system) and an operation part, which consists of the *raise* and *fire* operations and other operations to access the Employee data. When storing a set of Employees, both the data and the operations are stored in the database.

Thus, there is a single model for data and operations, and information can be hidden. No operations, outside those specified in the interface, can be performed. This restriction holds for both update and retrieval operations.

Encapsulation provides a form of "logical data independence": we can change the im-

plementation of a type without changing any of the programs using that type. Thus, the application programs are protected from implementation changes in the lower layers of the system.

We believe that proper encapsulation is obtained when only the operations are visible and the data and the implementation of the operations are hidden in the objects.

However, there are cases where encapsulation is not needed, and the use of the system can be significantly simplified if the system allows encapsulation to be be violated under certain conditions. For example, with ad-hoc queries the need for encapsulation is reduced since issues such as maintainability are not important. Thus, an encapsulation mechanism must be provided by an OODBS, but there appear to be cases where its enforcement is not appropriate.

2.4 Types and Classes

Thou shalt support types or classes

This issue is touchy: there are two main categories of object-oriented systems, those supporting the notion of class and those supporting the notion of type. In the first category, are systems such as Smalltalk [Goldberg and Robson 83], Gemstone [Maier, *et al.* 84], Vision [Caruso and Sciore 87], and more generally all the systems of the Smalltalk family, Orion [Banerjee *et al.* 87], Flavors [Bobrow and Steifik 81], G-Base [G-Base 88], Lore [Caseau 89] and more generally all the systems derived from Lisp. In the second category, we find systems such as C++ [Stroustrup 86], Simula [Simula 67], Trellis/Owl [Schaffert, *et al.* 86], Vbase [Atwood 85] and O_2 [Bancilhon *et al.* 88].

A *type*, in an object-oriented system, summarizes the common features of a set of objects with the same characteristics. It corresponds to the notion of an abstract data type. It has two parts: the interface and the implementation (or implementations). Only the interface part is visible to the users of the type, the implementation of the object is seen only by the type designer. The interface consists of a list of operations together with their signatures (i.e., the type of the input parameters and the type of the result).

The type implementation consists of a data part and an operation part. In the data part, one describes the internal structure of the object's data. Depending on the power of the system, the structure of this data part can be more or less complex. The operation part consists of procedures which implement the operations of the interface part.

In programming languages, types are tools to increase programmer productivity, by insuring program correctness. By forcing the user to declare the types of the variables and expressions he/she manipulates, the system reasons about the correctness of programs based on this typing information. If the type system is designed carefully, the system can do the type checking at compile-time, otherwise some of it might have to be deferred at compile time. Thus types are mainly used *at compile time* to check the correctness of the programs. In general, in type-based systems, a type is not a first class citizen and has a special status and cannot be modified at run-time.

The notion of *class* is different from that of type. Its specification is the same as that of a type, but it is more of a run-time notion. It contains two aspects: an object factory and

an object warehouse. The object factory can be used to create new objects, by performing the operation *new* on the class, or by cloning some prototype object representative of the class. The object warehouse means that attached to the class is its extension, i.e., the set of objects that are instances of the class. The user can manipulate the warehouse by applying operations on all elements of the class. Classes are not used for checking the correctness of a program but rather to create and manipulate objects. In most systems that employ the class mechanism, classes are first class citizens and, as such, can be manipulated at run-time, i.e., updated or passed as parameters. In most cases, while providing the system with increased flexibility and uniformity, this renders compile-time type checking impossible.

Of course, there are strong similarities between classes and types, the names have been used with both meanings and the differences can be subtle in some systems.

We do not feel that we should choose one of these two approaches and we consider the choice between the two should be left to the designer of the system (see Section 4.3). We require, however, that the system should offer some form of data structuring mechanism, be it classes or types. Thus the classical notion of database schema will be replaced by that of a set of classes or a set of types.

We do not, however, feel that is necessary for the system to automatically maintain the extent of a type (i.e., the set of objects of a given type in the database) or, if the extent of a type is maintained, for the system to make it accessible to the user. Consider, for example, the *rectangle* type, which can be used in many databases by multiple users. It does not make sense to talk about the set of all rectangles maintained by the system or to perform operations on them. We think it is more realistic to ask each user to maintain and manipulate its own set of rectangles. On the other hand, in the case of a type such as *employee*, it might be nice for the system to automatically maintain the employee extent.

2.5 Class or Type Hierarchies

Thine classes or types shalt inherit from their ancestors

Inheritance has two advantages: it is a powerful modeling tool, because it gives a concise and precise description of the world and it helps in factoring out shared specifications and implementations in applications.

An example will help illustrate the interest in having the system provide an inheritance mechanism. Assume that we have Employees and Students. Each Employee has a name, an age above 18 and a salary, he or she can die, get married and be paid (how dull is the life of the Employee!). Each Student has an age, a name and a set of grades. He or she can die, get married and have his or her GPA computed.

In a relational system, the data base designer defines a relation for Employee, a relation for Student, writes the code for the *die, marry* and *pay* operations on the Employee relation, and writes the code for the *die, marry* and *GPA computation* for the Student relation. Thus, the application programmer writes six programs.

In an object-oriented system, using the inheritance property, we recognize that Employees and Students are Persons; thus, they have something in common (the fact of being a Person), and they also have something specific. We introduce a type Person, which has attributes

name and *age* and we write the operations *die* and *marry* for this type. Then, we declare that Employees are special types of Persons, who inherit attributes and operations, and have a special attribute *salary* and a special operation *pay*. Similarly, we declare that a Student is a special kind of Person, with a specific *set-of-grades* attribute and a special operation *GPA computation*. In this case, we have a better structured and more concise description of the schema (we factored out specification) and we have only written four programs (we factored out implementation). Inheritance helps code reusability, because every program is at the level at which the largest number of objects can share it.

There are at least four types of inheritance: *substitution* inheritance, *inclusion* inheritance, *constraint* inheritance and *specialization* inheritance.

In substitution inheritance, we say that a type t inherits from a type t', if we can perform more operations on objects of type t than on object of type t'. Thus, any place where we can have an object of type t', we can substitute for it an object of type t. This kind of inheritance is based on behavior and not on values.

Inclusion inheritance corresponds to the notion of classification. It states that t is subtype of t', if every object of type t is also an object of type t'. This type of inheritance is based on structure and not on operations. An example is a *square* type with methods *get, set(size)* and *filled-square*, with methods *get, set(size)*, and *fill(color)*.

Constraint inheritance is a subcase of inclusion inheritance. A type t is a subtype of a type t', if it consists of all objects of type t which satisfy a given constraint. An example of such a inheritance is that *teenager* is a subclass of *person*: teenagers don't have any more fields or operations than persons but they obey more specific constraints (their age is restricted to be between 13 and 19).

With specialization inheritance, a type t is a subtype of a type t', if objects of type t are objects of type t which contains more specific information. Examples of such are persons and employees where the information on employees is that of persons together with some extra fields.

Various degrees of these four types of inheritance are provided by existing systems and prototypes, and we do not prescribe a specific style of inheritance.

2.6 Overriding, overloading and late binding

Thou shalt not bind prematurely

In contrast to the previous example, there are cases where one wants to have the same name used for different operations. Consider, for example, the *display* operation: it takes an object as input and displays it on the screen. Depending on the type of the object, we want to use different display mechanisms. If the object is a picture, we want it to appear on the screen. If the object is a person, we want some form of a tuple printed. Finally, if the object is a graph, we will want its graphical representation. Consider now the problem of displaying a set, the type of whose members is unknown at compile time.

In an application using a conventional system, we have three operations: *display-person, display-bitmap* and *display-graph*. The programmer will test the type of each object in the set and use the corresponding display operation. This forces the programmer, to be aware

of all the possible types of the objects in the set, to be aware of the associated display operation, and to use it accordingly.

```
for x in X do
  begin
    case of type(x)
      person: display(x);
      bitmap: display-bitmap(x);
      graph: display-graph(x);
    end
  end
```

In an object-oriented system, we define the display operation at the object type level (the most general type in the system). Thus, display has a single name and can be used indifferently on graphs, persons and pictures. However, we *redefine* the implementation of the operation for each of the types according to the type (this redefinition is called *overriding*). This results in a single name (display) denoting three different programs (this is called *overloading*). To display the set of elements, we simply apply the display operations to each one of them, and let the system pick the appropriate implementation at run-time.

```
for x in X do  display(x)
```

Here, we gain a different advantage: the type implementors still write the same number of programs. But the application programmer does not have to worry about three different programs. In addition, the code is simpler as there is no case statement on types. Finally, the code is more maintainable as when a new type is introduced as new instance of the type are added, the display program will continue to work without modification. (provided that we override the display method for that new type).

In order to provide this new functionality, the system cannot bind operation names to programs at compile time. Therefore, operation names must be resolved (translated into program addresses) at run-time. This delayed translation is called is called *late binding*.

Note that, even though late binding makes type checking more difficult (and in some cases impossible), it does not preclude it completely.

2.7 Computational completeness

Thou shalt be computationally complete

From a programming language point of view, this property is obvious: it simply means that one can express any computable function, using the DML of the database system. From a database point of view this is a novelty, since SQL for instance is not complete.

We are not advocating here that designers of object-oriented database systems design new programming languages: computational completeness can be introduced through a reasonable connection to existing programming languages. Most systems indeed use an existing programming language [Banerjee *et al.* 87], [Fishman *et al.* 87], [Atwood 85], [Bancilhon *et al.* 88]; see [Bancilhon and Maier 88] for a discussion of this problem.

Note that this is different from being "resource complete", i.e., being able to access all resources of the system (e.g. screen and remote communication) from within the language. Therefore, the system, even though computationally complete might not be able to express a complete application. It is, however, more powerful than a database system which only stores and retrieves data and performs simple computations on atomic values.

2.8 Extensibility

Thou shalt be extensible

The database system comes with a set of predefined types. These types can be used at will by programmers to write their applications. This set of type must be extensible in the following sense: there is a means to define new types and there is *no distinction in usage between system defined and user defined types.* Of course, there might be a strong difference in the way system and user defined types are *supported* by the system, but this should be invisible to the application and to the application programmer. Recall that this type definition includes the definition of operations on the types. Note that the encapsulation requirement implies that there will be a mechanism for defining new types. This requirement strengthens that capability by saying that newly created types must have the same status as existing ones.

However, we do not require that the collection of type constructors (tuples, sets, lists, etc.) be extensible.

2.9 Persistence

Thou shalt remember thy data

This requirement is evident from a database point of view, but a novelty from a programming language point of view, [Atkinson *et al.* 83]. Persistence is the ability of the programmer to have her/his data survive the execution of a process, in order to eventually reuse it in another process. Persistence should be *orthogonal*, i.e., each object, independent of its type, is allowed to become persistent as such (i.e., without explicit translation). It should also be implicit: the user should not have to explicitly move or copy data to make it persistent.

2.10 Secondary storage management

Thou shalt manage very large databases

Secondary storage management is a classical feature of database management systems. It is usually supported through a set of mechanisms. These include index management, data clustering, data buffering, access path selection and query optimization.

None of these is visible to the user: they are simply performance features. However, they are so critical in terms of performance that their absence will keep the system from performing some tasks (simply because they take too much time). The important point

is that they be invisible. The application programmer should not have to write code to maintain indices, to allocate disk storage, or to move data between disk and main memory. Thus, there should be a clear independence between the logical and the physical level of the system.

2.11 Concurrency

Thou shalt accept concurrent users

With respect to the management of multiple users concurrently interacting with the system, the system should offer the same level of service as current database systems provide. It should therefore insure harmonious coexistence among users working simultaneously on the database. The system should therefore support the standard notion of atomicity of a sequence of operations and of controlled sharing. Serializability of operations should at least be offered, although less strict alternatives may be offerered.

2.12 Recovery

Thou shalt recover from hardware and software failures

Here again, the system should provide the same level of service as current database systems. Therefore, in case of hardware or software failures, the system should recover, i.e., bring itself back to some coherent state of the data. Hardware failures include both processor and disk failures.

2.13 Ad Hoc Query Facility

Thou shalt have a simple way of querying data

The main problem here is to *provide* the functionality of an *ad hoc* query language. We do not require that it be done in the form of a query language but just that the service be provided. For instance, a graphical browser could be sufficient to fulfill this functionality. The service consists of allowing the user to ask simple queries to the database simply. The obvious yardstick is of course relational systems, thus the test is to take a number of representative relational queries and to check whether they can be stated with the same amount of work. Note that this facility could be supported by the data manipulation language or a subset of it.

We believe that a query facility should satisfy the following three criteria: (i) It should be *high level*, i.e., one should be able to express (in a few words or in a few mouse clicks) non-trivial queries concisely. This implies that it should be reasonably declarative, i.e., it should emphasize the *what* and not the *how*. (ii) It should be efficient. That is, the formulation of the queries should lend itself to some form of query optimization. (iii) It should be application independent, i.e., it should work on any possible database. This last requirements eliminates specific querying facilities which are application dependent, or require writing additional operations on each user-defined type.

2.14 Summary

This concludes the list of mandatory features and the distinction between traditional and object-oriented database systems should be clear. Relational database systems do not satisfy rules 1 through 8. CODASYL database systems partially satisfy rules 1 and 2. Some people have argued that object-oriented database systems are nothing more than CODASYL systems. It should be noted that (i) CODASYL systems do not completely satisfy these two rules (the object constructors are not orthogonal and object identity is not treated uniformly since relationships are restricted to be 1:n), and (ii) they do not satisfy rules 3, 5, 6, 8 and 13.

There is a collection of features for which the authors have not reached consensus on whether they should be required or optional. These features are:

- view definition and derived data;
- database administration utilities;
- integrity constraints;
- schema evolution facility.

3 Optional features: the goodies

We put under this heading things which clearly improve the system, but which are not mandatory to make it an object-oriented database system.

Some of these features are of an object oriented nature (e.g. multiple inheritance). They are included in this category because, even though they make the system *more* object-oriented, they do not belong in the core requirements.

Other features are simply database features (e.g. design transaction management). These characteristics usually improve the functionality of a data base system, but they are not in the core requirement of database systems and they are unrelated to the object oriented aspect. In fact most of them are targeted at serving "new" applications (CAD/CAM, CASE, Office automation, etc.) and are more application oriented than technology oriented. Because many object-oriented database systems are currently aiming at these new applications, there has been some confusion between these features and the object-oriented nature of the system.

3.1 Multiple inheritance

Whether the system provides multiple inheritance or not is an option. Since agreement on multiple inheritance in the object-oriented community has not yet been reached, we consider providing it to be optional. Note that once one decides to support multiple inheritance, there are many possible solutions for dealing with the problem of conflict resolution.

3.2 Type checking and type inferencing

The degree of type checking that the system will perform at compile time is left open but the more the better. The optimal situation is the one where a program which was accepted by the compiler cannot produce any run-time type errors. The amount of type inferencing is also left open to the system designer: the more the better, the ideal situation is the one in which only the base types have to be declared and the system infers the temporary types.

3.3 Distribution

It should be clear that this characteristic is orthogonal to the object-oriented nature of the system. Thus, the database system can be distributed or not.

3.4 Design transactions

In most new applications, the transaction model of classical business oriented database system is not satisfactory: transactions tend to be very long and the usual serializability criterion is not adequate. Thus, many OODBSs support design transactions (long transactions or nested transactions).

3.5 Versions

Most of the new applications (CAD/CAM and CASE) involve a design activity and require some form of versioning. Thus, many OODBSs support versions. Once again, providing a versioning mechanism this is not part of the core requirements for the system.

4 Open choices

Every system which satisfies rules 1 through 13 deserves the OODBS label. When designing such a system, there are still a lot of design choices to be made. These are the degrees of freedom for the OODBS implementors. These characteristics differ from the mandatory ones in the sense that no consensus has yet been reached by the scientific community concerning them. They also differ from the optional features in that we do not know which of the alternatives are more or less object-oriented.

4.1 Programming paradigm

We see no reason why we should impose one programming paradigm more than another: the logic programming style [Bancilhon 86], [Zaniolo 86], the functional programming style [Albano *et al.* 1986], [Banerjee *et al.* 87], or the imperative programming style [Stroustrup 86], [Eiffel 87], [Atwood 85] could all be chosen as programming paradigms. Another solution is that the system be independent of the programming style and support multiple programming paradigms [Skarra *et al.* 86], [Bancilhon *et al.* 88].

Of course, the choice of the syntax is also free and people will argue forever whether one should write "john hire" or "john.hire" or "hire john" or "hire(john)".

4.2 Representation system

The representation system is defined by the set of atomic types and the set of constructors. Even though we gave a minimal set of atomic types and constructors (elementary types from programming languages, and set, tuple and list constructors) available for describing the representation of objects, can be extended in many different ways.

4.3 Type system

There is also freedom with respect to the type formers. The only type formation facility we require is encapsulation. There can be other type formers such as generic types or type generator (such as set[T], where T can be an arbitrary type), restriction, union and arrow (functions).

Another option is whether the type system is second order. Finally, the type system for variables might be richer than the type system for objects.

4.4 Uniformity

There is a heated debate on the degree of uniformity one should expect of such systems: is a type an object? is a method an object? or should these three notions be treated differently? We can view this problem at three different levels: the implementation level, the programming language level and the interface level.

At the implementation level, one must decide whether type information should be stored as objects, or whether an ad hoc system must be implemented. This is the same issue relational database systems designers have to face when they must decide whether to store the schema as a table or in some *ad hoc* fashion. The decision should be made based on performance and ease of implementation grounds. Whatever, decision is made is, however, independent from the one taken at the next level up.

At the programming language level, the question is the following: are types first class entities in the semantics of the language. Most of the debate is concentrated on this question. There are probably different styles of uniformity (syntactical or semantical). Full uniformity at this level is also inconsistent with static type checking.

Finally, at the interface level, another independent decision must be made. One might want to present the user with a uniform view of types, objects, and methods, even if in the semantics of the programming language, these are notions of a different nature. Conversely, one could present them as different entities, even though the programming language views them as the same thing. That decision must be made based on human factor criteria.

5 Conclusions

Several other authors, [Kim 88] and [Dittrich 1986] argue that an OODBS is a DBMS with an underlying object-oriented data model. If one takes the notion of a data model in a broad sense that especially includes the additional aspects going beyond record-orientation, this view is certainly in accordance with our approach. [Dittrich 1986] and [Dittrich 1988]

introduce a classification of object-oriented data models (and, consequently, of OODBS): if it supports complex objects, a model is called structurally object-oriented; if extensibility is provided, is it called behaviorally object-oriented ; a fully object-oriented model has to offer both features. This definition also requires persistence, disk management, concurrency, and recovery; it at least implicitly assumes most of the other features (where applicable, according to the various classes); in total, it is thus slightly more liberal than our approach. However, as most current systems and prototypes do not fulfill all requirements mandated by our definition anyway, this classification provides a useful framework to compare both existing and ongoing work.

We have proposed a collection of defining characteristics for an object-oriented database system. To the best of our knowledge, the golden rules presented in this paper are currently the most detailed definition of an object-oriented database system. The choice of the characteristics and our interpretation of them devolves from out experience in specifying and implementing the current round of systems. Further experience with the design, implementation, and formalization of object-oriented databases will undoubtedly modify and refine our stance (in other words, don't be surprised if you hear one of the authors lambasting the current definition in the future). Our goal is only to put forth a concrete proposal to be debated, critiqued and analyzed by the scientific community. Thus, our last rule is:

Thou shalt question the golden rules

6 Acknowledgements

We wish to thank Philippe Bridon, Gilbert Harrus, Paris Kanellakis, Philippe Richard, and Fernando Velez for suggestions and comments on earlier drafts of the paper. David Maier's work was partially supported by NSF award IST 83-51730, co-sponsored by Tektronix Foundation, Intel, Digital Equipment, Servio Logic, Mentor Graphics and Xerox.

References

[Abiteboul and Kanellakis 89] S. Abiteboul and P. Kanellakis, "Object identity as a query language primitive", *Proceedings of the 1989 ACM SIGMOD*, Portland, Oregon, June 89

[Albano *et al.* 1986] A. Albano, G. Gheli, G. Occhiuto and R. Orsini, "Galileo: a strongly typed interactive conceptual language", *ACM TODS, Vol 10, No. 2*, June 1985.

[Atkinson *et al.* 83] M. Atkinson, P.J. Bayley, K. Chilsom, W. Cockshott and R. Morrison, "An approach to persistent programming", *Computer Journal, 26(4)*, 1983, pp 360-365.

[Atwood 85] T. Atwood, "An object-oriented DBMS for design support applications", *Ontologic Inc. Report.*

[Bancilhon 86] F. Bancilhon, "A logic programming object oriented cocktail", *ACM SIGMOD Record, 15:3,* pp. 11-21, 1986.

[Bancilhon and Maier 88] F. Bancilhon and D. Maier, "Multilanguage object-oriented systems: new answer to old database problems", in *Future Generation Computer II,* K. Fuchi and L. Kott editors, North-Holland, 1988.

[Bancilhon *et al.* 88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard et F. Velez, "The design and implementation of O_2, an object-oriented database system", *Proceedings of the ooDBS II Workshop*, Bad Munster, FRG, September 1988.

[Banerjee *et al.* 87] J. Banerjee, H.T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou and H.J. Kim, "Data model issues for object-oriented applications", *ACM TOIS*, January 1987.

[G-Base 88] "G-Base version 3, Introductory guide", *Graphael*, 1988.

[Bobrow and Steifik 81] D. Bobrow and M. Steifik, " The Loops Manual", *Technical Report LB-VLSI-81-13*, Knowledge Systems Area, Xerox Palo Alto Research Center, 1981.

[Carey *et al.* 86] M. Carey, D. DeWitt, J.E. Richardson and E.J. Shekita, "Object and file management in the EXODUS Extensible Database System", *Proceedings of the 12th VLDB,* pp 91-10, August 1986.

[Carey *et al.* 88] M. Carey, D. DeWitt and S. Vandenberg, "A Data Model and Query Language for EXODUS", *Proceedings of the 1988 ACM SIGMOD Conference,* Chicago, June 1988.

[Caruso and Sciore 87] "The VISION Object-Oriented Database Management System", *Proceedings of the Workshop on Database Programming Languages,* Roscoff, France, September 1987

[Caseau 89] "A model for a reflective object-oriented language", *Sigplan Notices*, Special issue on Concurrent Object-Oriented Programming, March 1989.

[Codd 70] E. F. Codd, "A relational model for large shared data banks", *Communication of the ACM,* Volume 13, Number 6, (June 1970), pp 377-387.

[Dittrich 1986] K.R. Dittrich, "Object-Oriented Database System : The Notions and the issues", in : *Dittrich, K.R. and Dayal, U. (eds): Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, IEEE Computer Science Press

[Dittrich 1988] K. R. Dittrich, "Preface", In : Dittrich, K.R. (ed): *Advances in Object-Oriented Database Systems,* Lecture Notes in Computer Science, Vol, 334, Springer-Verlag, 1988

[Eiffel 87] "Eiffel user's manual", Interactive Software Engineering Inc., TR-EI-5/UM, 1987.

[Fishman *et al.* 87] D. Fishman et al, "Iris: an object-oriented database management system", *ACM TOIS 5:1, pp 48-69,* January 86.

[Hall *et al.* 76] P. Hall, J. Owlett, S. Todd, "Relations and Entities", in *"Modeling in Data Base Management Systems", G.M. Nijssen (ed.),* pp 201-220, North-Holland, 1976.

[Goldberg and Robson 83] A. Goldberg and D. Robson, "Smalltalk-80: the language and its implementation", *Addison-Wesley*, 1983.

[Kim 88] W. Kim, "A foundation for object-oriented databases", *MCC Technical Report*, 1988.

[Khoshafian and Copeland 86] S. Khoshafian and G. Copeland, "Object identity", *Proceedings of the 1st ACM OOPSLA conference,* Portland, Oregon, September 1986

[Lécluse and Richard 89] C. Lécluse and P. Richard, "The O_2 Database Programming Languages", *Proceedings of the 15th VLDB Conference*, Amsterdam, August 1989.

[Maier and Price 84] D. Maier and D. Price, "Data model requirements for engineering applications", *Proceedings of the First International Workshop on Expert Database Systems,* IEEE, 1984, pp 759-765

[Maier 89] D. Maier, "Why isn't there an object-oriented data model?" *Proceedings IFIP 11th World Computer Conference,* San Francisco, CA, August-September 1989.

[Maier, *et al.* 84] D. Maier, J. Stein, A. Otis, A. Purdy, "Development of an object-oriented DBMS" *Report CS/E-86-005*, Oregon Graduate Center, April 86

[Nixon, *et al.* 87] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulos and M. Stanley, "Design of a compiler for a semantic data model", *Technical note CSRI-44,* University of Toronto, May 1987.

[Simula 67] "Simula 67 Reference Manual"

[Schaffert, *et al.* 86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian and C. Wilpolt, "An introduction to Trellis/Owl", *Proceedings of the 1st OOPSLA Conference,* Portland, Oregon, September 1986

[Skarra *et al.* 86] A. Skarra, S. Zdonik and S. Reiss, "An object server for an object oriented database system," *Proceedings of the 1986 International Workshop on Object Oriented Database System,* Computer Society Press, IEEE, pp. 196-204, 1986

[Stroustrup 86] B. Stroustrup, "The C++ programming language", *Addison-Wesley*, 1986.

[Van Emdem and Kowalski 76] M. Van Emdem and R. Kowalski, "The semantics of predicate logic as a programming language", *JACM, Vol 23, No. 4*, pp. 733-742, October 1976,

[Zaniolo 86] C. Zaniolo, "Object-oriented programming in Prolog ", *Proceedings of the first workshop on Expert Database Systems*, 1985.