GIP ALTAIR

F. Bancilhon

Rapporteur: A. Petrie

Object-Oriented Database Systems

François Bancilhon
Altaïr
BP 105, 78153 Le Chesnay Cedex
France

Abstract

This paper describes my vision of the current state of object-oriented database research. I first briefly define this field by its objectives, and relate it to other database subfields. I describe what I consider to be the main characteristics of an object oriented system, i.e. those which are important to integrate in a database system: encapsulation, object identity, classes or types, inheritance, overriding and late binding. I point out the differences between an object oriented system and an object oriented database system. I also point out the advantages and drawbacks of an object oriented database system with respect to a relational system. Finally, I list some research issues.

1 Introduction

The database field is concerned with the management of *large amounts of persistent, reliable and shared data.* "Large" means too big to fit in a conventional main memory. "Persistent" means that data persists from one session to another. "Reliable" means recoverable in case of hardware or software failures. "Sharable" means that several users should be able to access the data in an orderly manner. These four adjectives characterize *the* database problem and they define the specificity of the field.

Invited lecture
7th ACM SIGART-SIGMOD-SIGACT
Symposium on Principles of Data-Base Systems.
Austin, Texas, March 1988.

While this statement is (I hope) trivial, I think it is good to remember the exact problems we are solving. I did not say that the database field was concerned with cutting relations in smaller pieces or finding optimal join algorithms or designing a new distributed version of 2PL. These are not problems, but solutions we have devised to solve *the* problem.

It is therefore possible to find new solutions (for instance outside of the world of relational systems) to the basic problem of shared access to large amounts of reliable persistent data.

However, to many outsiders, the field might look (or might have looked) concerned with the design and implementation of relational systems for business type applications. The emphasis on business is obviously historical : the importance of the database problem was first felt by implementors of business applications; hence a bias towards the specific requirements of this community. The emphasis on relational systems comes from the fact that they provided a good answer for these applications.

Three new phenomena are appearing :

1. New non business-type users are feeling the need for large amounts of reliable, sharable and persistent data (CAD, CASE, office automation and AI). These new customers of database technology bring in new requirements.

2. The main memory cost and the memory-to-disk cost ratio are changing, thus modifying the assumptions behind current database system designs

3. We have discovered the impedance mismatch. Relational systems are well suited for the *ad hoc* query mode but not for application development. Application development requires the communication between a relational query language and a programming language. These two types of languages do no mix well : they have different types, they have different computational models, relational systems are set-at-a-time while programming languages are record-at-a-time. Solving the impedance mismatch requires integrating database and programming language technology.

Up to now, the database community has strongly felt the first phenomenon: we put a lot of work into understanding new applications and building systems for them. The second was felt to a lesser extent: some work was recorded on main memory databases. The search for a solution to the impedance mismatch has been fairly active recently. We have tried to merge logic programming and databases, functional programming and databases and more recently object-oriented programming and databases.

2 Nature of the field

I think three main factors contributed to the interest for object-oriented database systems.

(i) People building object-oriented systems needed database functionality. Adding some of this functionality to a system turned it into a database system.

(ii) The database community perceived the impedance mismatch : they started looking for technologies to solve this mismatch. Object-orientation looked promising because it provided a framework to represent and manage both program and data.

(iii) An interest of database people for some of the object-oriented technology. Other aspects of this technology (essentially the modeling power of the approach and the semantic data model aspect) tempted some database system designers and they included some in new prototypes of systems. The emphasis was on solving the database design problem by providing more powerful tools to model the real world.

The overall objective of the field is to integrate database technology and the object-oriented approach in a single system. It is therefore a system driven effort which will be judged by the quality of the systems it generates.

The approach taken by most researchers is the following:

1. Specify an object-oriented database system: this means define an object-oriented data model, its connection to a programming language (or the integration of a programming language in the model) and the dynamic aspects of the system (transaction management for instance).

2. Develop the necessary technology support : algorithms to implement the system and theories to gain better understanding of the model and the system.

3. Build the system and make it run fast.

3 Status of the object-oriented database field

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

Whereas, Codd's original paper set the goal by giving the specification of a relational system (data model and query language), no such specification exists for object-oriented systems. There is no clear consensus on what an object-oriented system is, let alone an object-oriented database system. There is still some argument on the basic principles and on the main characteristics of an object-oriented database system. People still argue about definitions of concepts. The favorite game of the opponents of the approach is to ask a crowd of object-oriented zealots to define object-orientation : the result is in general a civil war among the crowd. Even if one agrees on the definition, there is a lot of freedom in choosing seemingly minor aspects of the system: such questions as "is a type an object?" or "should we support type extensions?" are still subject to debate, and these "minor features" have a strong impact on the system design and implementation.

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 the [Van Emdem and Kowalski 76] paper. The need for a solid underlying theory is obvious: the semantics of types, of programs, of object identity is very often poorly defined. This absence of a good theoretical framework is, in my mind, the cause of the lack of consensus on the data model.

Finally, a lot of experimental work is going on at this moment: people are actually building systems. Some of them are just prototypes [Lecluse et al 87], [Nixon et al 87], [Banerjee et al 87], some are commercial products. The situation is analogous to that of relational database systems in the mid seventies (even though there seems to be more start-ups [Maier *et al* 84], [Atwood 85]) in the object-oriented case). For relational systems, the specifications of the system were common to everybody and people were mainly developing support technology.

Today, we are choosing at the same time the specification of the system and the technology to support its implementation. Thus, concerning the specifications of the system, we are taking a Darwinian approach: it is hoped that, out of the set of experimental prototypes being built, the best model will emerge. We hope that at the same time the support technology will be developed.

Unfortunately, with this flurry of experimentation, we run the risk of seeing a system emerging as *the* system, not because it is the best (or just good) but because it is the first one. It is a classical and unfortunate pattern of the computer area that the first product to appear becomes the *de facto* standard and never disappears. This is at least true for languages and systems (Fortran, Lisp, Cobol and SQL are good examples of such situations).

4 Object-orientation

I now describe what I consider to be the main characteristics of an object-oriented system. These are the features one should put in a DBMS to make it object-oriented. Of course, my choice is somewhat subjective. I chose these features because I consider they represent the more original ideas and those who will have the most impact on programmer productivity.

4.1 Encapsulation

Encapsulation is the principle that one should model at the same time data and operations. Thus, an object has an interface part and an implementation part. The interface part is the specification of the set of operations which can be performed on the object. It is the only visible part of the object. The implementation part has a data part and an operation part. The data part is the memory of the object and the operation part describes, in some programming language, the implementation of each operation.

Consider, for instance, the Employee object. In a relational system, it is represented by some tuple. It is queried using a relational language and, later, an application programmer writes programs to update this record. Programs are written to raise the Employee's salary and to fire the Employee. These are written, either in some imperative programming language with embedded SQL statements, or in some fourth generation language. These programs are stored in a traditional file system, separately from 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 which has a value part (probably very similar to the record which was defined for the relational system) and an operation part which consists of the *raise* and *fire* operations and maybe some extra operations to consult the Employee data. When storing a set of Employees in the database, we store at the same time the data and the application programs.

Thus, there is a single model for data and operations, and information can be hidden. Once the interface to the object is defined, no operation, other that the ones specified in this interface, can be performed. This is true both for update and consultation.

4.2 Object identity

This issue has a tendency to be obscured by some religious and philosophical concerns. Object identity has been is programming languages for quite a time now (Lisp and Smalltalk). The discussion is more recent in databases. I don't know where it was discussed first, maybe in [Maier and Price 84]. A complete and more recent paper on the topic can be found in [Koshafian and Copeland 86]. It is a good idea, but there is nothing very profound about it and I am worried that we might soon be getting panels or conferences on the topic.

The idea is the following: in a model with object identity, an object has an existence which is independent of its value. Thus two objects can either 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 graphical representation of a complex object is a DAG, while it 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 have the same child or they don't. 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 have the same child. In an identity based model, these two structures can share the common part

(john, 15, {}) or not, thus modeling both situations.

Object updates : assume now that Peter and Susan do indeed have the same child John, then when we update Susan's son John, this updates the object John and Peter's son is updated. In a value based system, we have to take care of updating both sub-objects.

Of course, one can simulate object identity over a value based system by introducing object identifiers all over the place, but this puts the burden on the user (and this burden can be quite heavy for operations such as garbage collection).

Note that identity based models have always been known 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 the relational world where relations are value based.

4.3 Types and classes

A type, in an object-oriented system, describes a set of objects with the same characteristics. It corresponds to the notion of an abstract data type. It has two parts: the interface to the object, and the implementation of the object. 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 data. Depending on the power of the system, the structure of this data part can be more or less complex. The operation part gives a program which implements each of the operations in the interface part.

In programming languages, types are tools to increase programmer productivity, by insuring program correctness. By forcing the user to declare the structure of the objects he manipulates, the system checks that the user does not perform wrong assignments or manipulations on objects. Thus types are used *at compile time* to check the correctness of the programs.

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 container. The object factory means that the class can be used to instanciate new objects, by performing the operation *new* on the class. The object container means that attached to the class is its extension, i.e. the set of objects of the system which belong to the class at this time. The user can manipulate the container by applying operations on all elements of the class. Classes are not used for checking correctness of programs but to create and manipulate objects.

Of course, there are strong similarities between classes and types, and the differences can be subtle in some systems.

4.4 Inheritance

This is probably the most powerful concept in object-oriented programming : it allows objects of different structures to share operations related to their common part.

Assume that we have Employees and Students. Each Employee has a name, an age 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 the Student as a special kind of Person, with a specific *set-of-grades* attribute and a special operation *GPA computation*. In this case, we have only written four programs.

This has two advantages: it is a powerful modeling tool, because it gives a concise and precise description of the world. It helps code reusability, because every program is at the level at which the largest number of objects can share it.

4.5 Overriding and late binding

There are cases where, on the contrary, one wants to have the same name used for different operations. Consider for instance the *display* operation : it takes an object as input and displays it on the screen. Depending on the object, we want to use different kinds of display : 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 being printed, and if the object is a graph, we will want its graphical representation. Consider now the problem of displaying a set of objects, whose type is unknown at compile time.

In a standard system, we have three operations: *display-person, display-bitmap* and *display-graph*. The programmer will test for the type of each object and use the corresponding display operation. This forces the programmer, when he displays an object, to be aware of the type of the object (extra-knowledge at compile time) and to be aware of the associated display operation and to use it accordingly (more information to remember).

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 body of the operation for each of the types according to the type specificity (this 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 method to each one of them.

In this case, we have a different gain: we still write the same number of programs. But the programmer does not have to worry about three different programs. The code written is simpler : there is no case statement on types. Finally the code is also re-usable: if we introduce a new type in the system and in the set of objects to be displayed, the same display program works (provided we override the display method for that new type).

To offer this new functionality, the system cannot bind operation names to programs at compile time. Therefore operation names are resolved (translated into program addresses) at run time (this is called *late binding*).

4.6 Degrees of freedom

I listed above what I consider to be the major characteristics of an object-oriented system, i.e. the minimal features it should have to deserve the object-oriented label. These features do not completely specify a system and degrees of freedom are left to the designer.

The major one is the computational model which is independent of the approach. A specific computational model has to be chosen, but its choice is left open. Many object-oriented systems are based on functional languages, [Bobrow and Steifik 81], [Cardelli 84], Suggestions for mixing the logic programming approach can be found in the literature [Bancilhon 86], [Zaniolo 86]. Many others use more classical imperative languages [Lecluse et al 87], [Stroustrup 86], [Eiffel 87].

The object constructors which the model uses are open and most models differ on these. Some systems are typed [Cardelli 84], while others are not and only use classes, [Goldberg and Robson 83].

The degree of uniformity can vary from one system to another. Some systems view things uniformly (every thing is an object: objects, types and methods; type and method manipulation is done by message passing), some system view things non uniformly (types and methods are not objects and are manipulated by special commands).

This non exhaustive list explains why, even if there were an agreement on the minimal features of an object-oriented system, there would still be room for a lot of different systems.

5 Related fields

A number of new database subfields are related to the object-oriented field. These subfields are : semantic data models, nested relations, extensible database systems, database programming languages and persistent programming languages.

Semantic data models appear as an integration of AI and database concepts. Semantic data models have in common with object-oriented data models, the notion of complex objects and the hierarchy of types (called the *isa* relationship). These models however ignore encapsulation and late binding. They are in general more concerned with data modeling issues and queries than with running application programs. The associated query languages are not computationally complete.

Of course, the distinction I introduce here is arbitrary: some researchers in semantic data model have been concerned with those problems [Nixon et al 87]. But the point I want to make is the following: one of the main issue in object oriented systems is how to find good support for encapsulation, overriding and late binding. This is where the fusion between databases and programming language is interesting and hard to do. Part of the semantic data model people are more interested in the conceptual modeling issues than in those system aspects.

The area of **nested relations** (which seems a much better name than NF2) is a fairly active field, devoted to the extension of relations to nested relations. The concern is more the extension of relational concepts (query languages, query evaluation, schema design and relational theory) from flat to nested relations. Some prototypes have been or are being designed and implemented [Verso 86], [Dadam et al 86]. The major differences are the lack of encapsulation, object identity and computational completeness.

The appearance of **extensible database systems** follows a natural trend of computer science : that of moving from compilers to compiler-compilers, from syntax editor to syntax editor generators etc. An extensible database system, as examplified by [Carey *et al*] is in essence a database generator : the user of the system specifies the database system he needs (concurrency control mechanisms, access paths, data model etc.) and the generator produces for him the corresponding system.

I doubt database generators, at their current stage of development, could be used to write an object-oriented database system (as I define it). They could (and should?) however be used to write parts of it.

Database programming languages are the result of the effort of database people to extend the functionality of a database system to that of a programming language, while **persistent programming languages** are the result of the effort of programming language people to extend the functionality of programming language systems to that of a database system. Eventually these two subfields should merge into a single one [Atkinson 86], [Bancilhon and Buneman 87]. Object-oriented databases, as I have defined them, lay in the intersection of these two approaches.

6 Object-oriented systems vs database systems

It is important to have a clear notion of what is the advantage brought by the the introduction of database functionality in the object-oriented system. In other words, what is missing in an object-oriented system to make a database system.

6.1 Set programming

The history of sets in computer systems is interesting:

Traditional programming languages do not have in general the concept of sets: they use other structures to implement sets: arrays, lists or files. This is why they have no specific operations to manipulate sets.

In logic programming, sets exist only at the upper level of the hierarchy: the database consists of a collection of named sets (the predicates). Below this level, all we have are lists (or terms). If one wants to manipulate sets at a lower level, one uses the system hack *set-of* which turns a set into a list of elements. Recent efforts to change this state come from the database community, [Tsur and Zaniolo 86], [Kuper].

Functional programming people have always been very fond of lists and do not feel the need for a set construct.

This is also true of most object-oriented systems: the set construct and the class structure are in fact distinct. In Taxis [Nixon et al 87] for instance, there is no set constructs and the only way to manipulate sets is by creating types and manipulating their extension.

The main thing that database people have brought is considering sets as first class citizens. The relational model introduced an algebra based on sets with the associated operations. By defining selection and join as the major operations, the relational model gave the description of the operations to optimize. The limitation of sets in relational systems is due to the fact that only sets of tuples and tuples of atomic values are considered.

Nested relations are an attempt to raise this restriction: one can build sets of tuples of sets and so on. In most nested relation models, there are some restrictions (set and tuple constructors have to alternate, objects in a set must be of the same type etc.) and, in almost all of them, the database is a collection of named sets (relations).

I think that treating sets uniformly is one of the major challenge facing designers of object-oriented database systems.

6.2 Persistence and reliability

Most object-oriented systems do not offer persistence. Thus, the only way to keep data from one session to another is to use a file system and save the necessary data. This requires an explicit storage operation from the programmer. It also requires translation of the object from the application format in the file format. This puts extra burden on the programmer, complexifies the code and degrades performance.

The system provides no protection against hardware or software failures, it provides no mechanisms for roll back or to insure transaction atomicity.

6.3 Sharing

These systems are single user and do not provide any control over the concurrent access to the same data.

6.4 Managing large amounts of data

Most system run in main or virtual memory, application programs are therefore limited, in the size of the data they manipulate, to the virtual address space. Furthermore, the disk being managed only through the virtual memory mechanism, the performance of the manipulation of large amounts of data is in general not good.

These systems make no use of indices, smart buffer management schemes, clustering of objects on disk, clever strategies for selection and joins or intelligent query optimization.

7 Object-Oriented vs Relational

Current relational systems together with their connection to a general purpose programming language can do almost every thing: they insure persistency, reliability, data sharing, they can model any data and can perform any possible computation. But, even though every application can be written on top of such a relational system, it might be extremely hard to do or it might be incredibly slow. Thus, to compare a new system to a relational system, the criterion for improvement is not computing power, it is computing speed or ease of programming.

7.1 A step forward

In many regards object oriented systems are superior to relational systems.

7.1.1 Dealing with complex objects

The ability to store and manipulate complex objects is a feature of many object-oriented systems, while relational systems are restricted to store and manipulate only flat tuples. This gives more modeling power: complex structures can be represented directly, and do not have to be mapped onto lower level relational structure.

7.1.2 Object identity

The notion of object identity, as introduced in object-oriented system, is clearly a plus in modeling power. It allows the user to model directly some situations (object sharing and cyclic objects) without adding an extra layer (of surrogates for instance) on top of the system. It provides operations such as equal and identical. It finally provides a simple and natural semantics for updates.

7.1.3 Extensibility

This is a major advantage of object-orientation: by adding new types (or classes) in the system, one can extend it capabilities. This is especially important to adapt the system to new types of applications.

7.1.4 Storing programs and data

While, in a relational system, data is stored in the database system, application programs have to be stored in some other system. This means that none of the features of the DBMS can be used to manage the programs. In an object-oriented database system, programs and data are stored under the same formalism in the same system.

7.1.5 Typing and inheritance, overriding and late binding

Relational systems do not have any notion of typing : each relation has its own type and it is impossible for instance to declare a type and to assert that several relations are of that type. Inheritance, overriding and late binding are tools which make the programmer's life much easier.

7.2 A step backward

While object-oriented database systems are superior in many ways to relational systems, some of the benefits are lost by switching to this new approach. Sometimes we loose because the technology is not there yet, and sometimes we loose because the paradigm makes things inherently more difficult.

7.2.1 Simplicity

One obvious advantage of the relational model is its simplicity: there are very few basic concepts and the whole story can be told in a few transparencies (which is a good test for a data model). Clearly, some of this simplicity is lost when we move to the object world: there are more concepts, some of them overlap, and once again the lack of a clear unifying formalism does not help. This point is also emphasized by the lack of a solid formal framework for the object-oriented data model.

7.2.2 *Ad hoc* query languages

Relational systems favor the *ad hoc* query mode for using databases, and they provide nice query languages and interfaces. Object-oriented databases do not have, at this point, good query languages. The reasons for this are two-fold :

(i) Because of the more complex structure of their data and the lack of a good formal underlying model, object-oriented systems do not have, like relational systems, a simple and powerful query language. Some languages have been suggested, either derived from functional or semantic data models, but none has emerged. Those attempts have also shown that designing simple query languages for such a model is not such an easy task: there is a clear tradeoff between simplicity of the query language and the power of expression of the data model.

(ii) The second problem is that query languages and encapsulation are somewhat antinomic : the encapsulation principle states that data should be hidden from users, while the query needs to see the internal structure of the objects.

7.2.3 Declarative queries

Because object-oriented systems are more functional in nature, data manipulation tends to be more navigational. Thus the user is loosing some of the declarativeness of relational systems. Naturally, if we are able to incorporate set constructors in our model, we might regain some of this declarativeness.

7.2.4 Relational interface

The absence of relational interfaces on object-oriented databases is a drawback. It might seem strange to mention this as a problem. However from an industrial point of view it is certainly an issue. In the years to come, relational systems will keep increasing their share of the market. SQL is becoming a *de facto* standard. It will certainly become a standard for exchange of data between heterogeneous systems. So, even if objects take over the world, we will keep exchanging data between systems in relational form.

7.2.5 Speed

Of course, the performance issue will be used as an argument against these systems, just as it was used, not so long ago, against relational ones. This is a clear challenge for system developers. Can we build a system which will perform selections on sets of objects, as fast as a relational system currently select tuples in a relation? Can we run the same amount of debit/credit transactions per second on a object system than on a relational system?

8 Conclusion and research issues

I believe that object oriented database systems have a reasonable chance to take over relational systems. My prediction is based on the following facts:

- Object-orientation will win as a programming paradigm: it has some obvious qualities, it is very appealing, it is a fashionable, it mixes with a lots of different styles of programming, and finally it has shown to be very successful in areas such as AI and user interface development.

- Object-oriented databases blend the most successful programming paradigm with database technology, thus solving the impedance mismatch.

- Their extensibility will allow them to evolve and accommodate new types of data types and new functionalities.

Of course, this will work only if we are able to solve the major research issues. Some of these are :

- Define a formal framework in which one can define an acceptable object-oriented data model; this data model should include a programming language. Give formal semantics to types and programs in this framework.

- Define *Ad hoc* query languages and build a relational interface to those systems.

- Solve the performance problem, i.e. design algorithms for memory management, buffering clustering and garbage collection.

- Find a good formalism for sets in an object oriented database system, find a way of doing associative access to sets, to use indices and to re-use relational query optimization techniques.

9 Acknowledgements

I wish to thank Serge Abiteboul, Didier Plateau, Marc Shapiro, Eric Simon, and Fernando Velez for their comments, corrections and suggestions on an earlier draft of this paper.

References

[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 86] Proceedings of the Workshop Persistence and Data types, Appin, September 86.

[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 Buneman 87] F. Bancilhon and P. Buneman (Ed), Proceedings of the Workshop on Database and Languages, Roscoff, September 1987.

[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.

[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.

[Cardelli 84] L. Cardelli, "Amber", AT&T Bell Labs Technical Memorandum 11271-840924-10TM, 1984.

[Carey *et al*] M. Carey and D. DeWitt, "The architecture of the EXODUS extensible DBMS", Proceedings of the International Workshop on object-oriented database systems, Pacific Grove, September 1986.

[Dadam et al 86] P. Dadam et al, "A DBMS prototype to support extended NF2 relations: an integrated view on flat tables and hierarchies", Proceedings ACM Sigmod, Washington 1986.

[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, January 86, pp 48-69.

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

[King and McLeod 85] R. King and D. McLeod, "Semantic Database Models", in S.B. Yao Ed., Database design, Springer Verlag, N.Y. 1985.

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

[Kuper] G. Kuper, "Logic programming with sets", Proceedings 6th PODS, San Diego, March 1987.

[Lecluse et al 87] C. Lecluse, P. Richard and F. Velez, "O2, an object-oriented data model", Proceedings of the Workshop on Database Programming Languages, Roscoff, France, September 1987

[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 *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, Univesity of Toronto, May 1987.

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

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

[Tsur and Zaniolo 86] S. Tsur and C. Zaniolo, "LDL: a logic-based data-language", Proceeding of the 85 Conference on VLDB, September 1985

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

[Verso 86] J. Verso, "Verso: a database machine based on non 1NF relations", INRIA Rapport de Recherche No. 523, May 1986.

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

DISCUSSION

When, during his talk, Dr. Banchilon expressed doubts about whether the Darwinian principle would apply to the various experiments that were being made, **Professor Randell** remarked that it was survival of the fittest and not survival of the best.

In the discussion following the lecture, **Professor Nygaard** said that **Dr. Banchilon**'s set of requirements were very similar to the set of requirements used by the people working on the object store at Parc or the people working on VBase, and consequently were fairly standard. However a very important thing has been omitted. When doing object oriented programming one is going to want to put into the database objects that are active and in an unfinished state of execution. For example one might have something that is managing a complex sequence of actions taking place in an office, which one will want to put it in the database, take out and make active again and then store its next state. **Professor Nygaard** said that this was a new requirement but one which he found that database people did not think about. Nevertheless he felt that it was very important one for object oriented applications.

A second point was that **Dr. Banchilon** had emphasised speed and performance. This was right but that for people who were concerned with designing and using large scale object oriented applications or environments it might be less in order to make demands on speed because of the need to hand over things to the database and get them back again. Extreme demands on speed do not apply to everyone and one of the problems is to achieve progress in spite of standardisation. This is perhaps a case of the object oriented community being willing to live with flexibility in spite of less performance.

Dr. Banchilon suggested that perhaps **Ms. Goldberg** might like to answer the question. **Ms. Goldberg** said there wasn't an object oriented programming system or programmer who would tolerate a slow storage manager or garbage collector. She said that you might think that you would have to give up flexibility to get performance, but the goal should be flexibility shouldn't give up on performance.

Professor Bayer said that storing objects in a semi-finished or intermediate state was very closely related to generalising the transaction concept and database people have been thinking about this for a long time. They are very aware of the problem but they haven't come up with any generally accepted answers. There are many solutions but no really good ones or generally accepted ones.

He asked whether, with respect to languages and computational completeness, **Dr. Banchilon** was thinking of generalising SQL or extending something like C++. One view of programming is that changing languages is not acceptable so you will have to make a decision which way you want to go.

Dr. Banchilon said that each database system would have SQL and that no one would sell a database system in the near future without it. If the implementation of SQL is adequate for querying a database then everything is fine. If not, then each one of these database systems will also have its own custom made query language apart from SQL.

Professor Bayer said that one was then stuck with several languages.

Dr. Banchilon said that they were for different purposes. You need SQL because you want to talk to other people. You need a specialised query language to allow the user to interact easily with the database. This query language might be purely graphical. It might just be a browser.

Professor Tannenbaum said **Dr. Banchilon** had wanted to roll the clock back fifteen years with regard to data independence and so on, and had used an example which had programs for "Move Employee" and "Fire Employee" in the database. **Professor Tannenbaum** said that he would envision that, in a complex database, the number of possible operations that you would want to perform would be so immense that the only way that you could possibly put the operations in the database would be to have, for a tuple with n fields, basically 2n operations, one for reading and for writing each of the n fields. In which case, technically you have made the database object oriented but it is basically pointless. **Professor Tannenbaum** said that he didn't see how you put operations in the database without having, basically, an operation to read and write each individual field.

Dr. Banchilon said that that was a very interesting comment and that the following was what happens. There are internal methods and what is needed in the interface is exported used the export function. If he looks at the examples written by himself and colleagues he finds that at the end of every type there is written "export all". He could take the same programs and put back most of the program in the objects. It is a question of whether or not you want to follow the philosophy of tying a large complex program to a given type. **Dr. Banchilon** said that usually when he looked at these programs he could rewrite them reasonably well and say "this is what you want to tie to this type" and "this doesn't have to go out there". The date of birth of an employee is not a method. There is no point in having a specific operation to deal with it. It is inside there for programs which need it.