Fitting Round Objects Into Square Databases[1]

Dennis Tsichritzis
Oscar Nierstrasz

Centre Universitaire d'Informatique
12 rue du Lac, CH-1207 Geneva, Switzerland
E-mail: {oscar,dt}@cui.unige.ch, oscar@cgeuge51.bitnet
Tel: +41 (22) 787.65.80, Fax: +41 (22) 735.39.05

Abstract

Object-oriented systems could use much of the functionality of database systems to manage their objects. Persistence, object identity, storage management, distribution and concurrency control are some of the things that database systems traditionally handle well. Unfortunately there is a fundamental difference in philosophy between the object-oriented and database approaches, namely that of *object independence* versus *data independence*. We discuss the ways in which this difference in outlook manifests itself, and we consider the possibilities for resolving the two views, including the current work on object-oriented databases. We conclude by proposing an approach to co-existence that blurs the boundary between the object-oriented execution environment and the database.

1. Introduction

Consider an environment for running object-oriented applications. There are a number of *object management* issues that should ideally be directly supported by the environment: persistence, creation and destruction of objects, concurrency control, management of object identifiers, binding between object instances and classes, etc. We will call the part of the environment responsible for these issues the *object manager*. Some of these issues are operating system issues (e.g., scheduling of activities), but many are inherently database issues. It is therefore quite natural to propose that the object manager use a database system for dealing with database issues. An integration of object-oriented and database systems can also be attractive for several other reasons. First, it may provide a way for interfacing to existing databases. Next, we may get some additional database functionality available to our objects, namely querying and transaction support. Finally, it may be that certain object-oriented ideas will be useful for organizing databases and making databases easier to use.

In fact, these two worlds are not so easily merged. We present the problem graphically in figure 1. Objects as they exist in object-oriented systems are depicted as round. Database instances, on the other hand, are square. We use two different shapes to represent the fact that very different properties of "objects" are emphasized by the two approaches. Furthermore, the object-oriented approach emphasizes what we call *object independence*, represented as an encapsulation barrier between objects, whereas the database approach emphasizes *data independence*, namely a barrier between the database and the applications. The object manager needs to provide

1. In *Proceedings of the European Conference on Object-oriented Programming*, ed. S. Gjessing and K. Nygaard, Lecture Notes in Computer Science 322, pp. 283-299, Springer Verlag, 1988.

persistence, and possibly other database functionality, for its round objects. The database system provides persistence, among other facilities, for square objects. The issue is simple. To what extent can the object manager use square database facilities to manage its round objects?

The paper consists of two parts. First, we shall discuss the main difference in outlook between object-oriented systems and database systems in terms of object independence versus data independence, and we shall see a number of specific ways in which this difference manifests itself. Second, we will illustrate the different approaches that can be taken for enabling object-oriented and database systems to co-exist. We will conclude with a proposal for merging the two by factoring out the common functionality, and permitting the two systems to each manage their own objects according to the appropriate paradigm.

2. Object Independence vs. Data Independence

Before we examine the differences between object-oriented and database approaches in detail, it is instructive to contrast their basic principles.

Object-oriented systems emphasize *object independence* by encapsulation of individual objects. Objects' contents and the implementation of their operations (i.e., their methods) are hidden from other objects. Interaction with objects is through a well-defined interface, illustrated by the paradigm of communication via message-passing. Object independence can be viewed as fundamental to *all* object-oriented concepts, including, for example, all forms of inheritance [Nierstrasz 1988]. We depict object independence as a boundary around objects, as in figure 1.

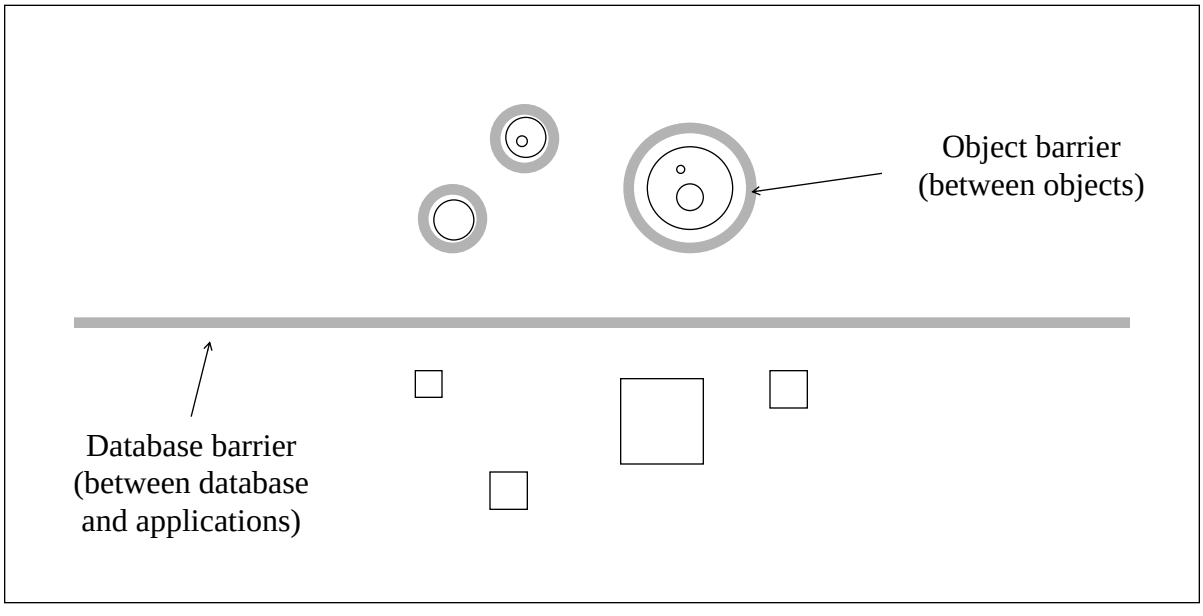

Figure 1 Object Independence vs. Data Independence

Database systems, on the other hand, emphasize *data independence* by separating the world into two independent parts, namely the data and the applications operating on them. The independence boundary is between the database and the rest of the world. This separation serves many purposes, but the most important effect is that the responsibility of the database system is well-defined, and consequently the database interface can be relatively simple.

These two principles constitute a major commitment in each field, and lead to a major cultural difference. (This has also been pointed out in [Bloom and Zdonik 1987].) Object independence is fundamental to object-oriented systems, just as data independence is fundamental to database systems. If database techniques are to be relevant to object management, we must ask whether these two principles can co-exist, and, if so, how?

Consider, for example, the design of a large application. If we believe in data independence then we must split our application into persistent, uninterpreted data, and programs that manipulate and share them. An object-oriented approach, however, would lead to a design composed entirely of objects, some of which are persistent. Manipulation of persistent objects is via their interfaces, and shareability is implicit in the message-passing paradigm. It seems clear that the two approaches lead to very different designs.

Can the two paradigms be reconciled? Can we support both data independence and object independence for the same set of objects, or must we always put our objects into the one world or the other? Maybe some objects, and therefore applications, can be handled more effectively in one way or the other. Maybe objects at some level (round objects) must be handled in one way and other objects (square objects) at a different level must be handled in a different way. Maybe the same objects can be either round or square depending on the context or the operations performed on them.

In order to develop some intuition to answer these questions, we shall try to evaluate precisely how deep this cultural difference runs. Briefly, we shall look at the following issues:

1. Are classes objects?
2. What relationships may exist between classes?
3. Should navigation be supported?
4. What operations exist on objects?
5. How are objects identified?
6. How are objects selected?
7. What is the role of classification?
8. Can object classes evolve?
9. Should the network be visible in a distributed system?
10. How are active/passive objects handled?
11. Is the object world closed (complete) or open?

2.1 Instance/class separation

Traditionally databases make a very strong distinction between instances and classes. Instances are in the database, whereas class information, i.e., the schema, is stored in the data dictionary. Many database systems have two different languages to deal with instances and classes. The Data Manipulation Language (DML) deals with operations on instances. The Data Definition Language (DDL) deals with operations, mainly creation, on classes.

It is obvious that object-oriented systems need to manage both instances and classes. In some object-oriented languages, notably Smalltalk-80 [Goldberg and Robson 1983], classes are themselves genuine objects, and can be manipulated as such. With the emergence of relational systems the DML-DDL separation was blurred, at least conceptually. After all, data dictionary tables could be viewed as relations and they could be manipulated with relational operations. However, most database systems retain a strong separation between schema and database.

The evolution from databases to knowledge bases forced a reconsideration of the instance/class separation. In conceptual models for knowledge bases, classes and instances are dealt with together and operations on classes are allowed [Mylopoulos and Levesque 1983; Brachman 1988].

The database research community has already accepted that classes can be manipulated, and that they can be structured with PART-OF and IS-A relationships, etc. Existing commercial database systems do not provide such facilities. They do provide, however, extensive facilities for class definitions in the database dictionary. It is conceivable that these facilities can be made available, and integrated as database operations. However, in doing so database systems will lose some of the simple user interfaces. The great advantage of relational systems is based on the relative few, very basic and very clear operations.

2.2 Relationships between classes

The relationships supported between database classes, whether they be relations or record types, etc., are quite restricted. They may be statically defined between classes, as in entity-relationship schemas. Relational systems, on the other hand, allow many relationships, but they are completely syntactic, based on contents and operations like joins.

In object-oriented systems we need the ability to deal with many relationships, some defined only at the instance level. Objects which know each other, or communicate with each other, are somehow related. It is not easy to model such relationships in databases. In many cases the relationships are explicit and not implied by the object's contents. Explicit relationships of this sort were forbidden in relational systems to emphasize relationships in terms of contents. Contents, however, are generally hidden in objects due to encapsulation.

Research in object-oriented databases tries to deal with that problem. Unfortunately relational systems, on the other hand, which are by now well-established, do not seem to offer the appropriate capabilities. Mapping object relationships into relational tuples or joins is not an easy task. On the other hand, going back to "information-bearing" relationships between database instances is considered a step backward.

2.3 Navigation

After many years of debate, database systems have de-emphasized point-to-point navigation between instances in the database. Such a facility was present in some older systems (notably, network and hierarchical databases) but it is now considered at worst harmful and at best old-fashioned. Many database specialists believe that reintroducing navigation in databases would be a step backward.

It is not easy, however, to see how else one should access objects with many instance-to-instance relationships when they are stored in a database. Relational systems give very powerful set-oriented operations but they are not appropriate for navigating from one object instance to another.

Fortunately people working in object-oriented databases are aware of the problem and they will probably come up with a solution. They will probably either have to utilize the relational interface in some innovative way (!) or they will have to adopt a different functional data manipulation language (e.g., Daplex).

2.4 Operations on objects

Database systems traditionally provide very few generalized types (i.e., record types, relations, etc.). As a result they can provide a small number of very general operations for queries and updates on the database objects. The operations are the same regardless of the semantics of the object involved. Queries and updates on employees, cars, accounts etc. utilize the same operations. In addition, the operations are simple. More sophisticated operations are encapsulated in transactions, which are treated separately from the database objects.

Object-oriented systems require that all objects provide their own set of operations, with some sharing through object classes and inheritance mechanisms. In addition, the methods can be logically complex. Most of the work in object-oriented databases deals with extending database operations to accommodate particular object types [Bancilhon 1988]. The extensions take two forms. First, complex objects can be defined, thus dealing with structural complexity within objects. Second, operations specific to object classes can be defined. Multiple inheritance can be used to define new classes that share operations and attributes (i.e., visible instance variables) with existing classes.

2.5 Object identifiers

Database systems utilize object identifiers internally for implementation purposes. These identifiers used to be visible and available for manipulation by the user in older database systems. The identifiers had a connotation of physical location, and they were used to physically place the database object in the system files. For this reason, they were considered harmful, and they were therefore removed from the database interface. In the relational model, and in some relational systems, tuples do not have a visible identifier. They are identified by their contents, via primary or secondary keys.

In object-oriented systems object identifiers are very important for two reasons. First, identifiers provide a permanent handle for objects that may evolve or move, in much the same way that file names hide the fact that a file’s contents and physical location may change. Second, if an object’s contents are properly encapsulated, they cannot be expected to provide a means for identification.

We need, therefore, to reintroduce identifiers into databases. These identifiers should be purely for identification purposes and should not, of course, be related to the physical location of objects in the database. In addition, allowable operations on identifiers should be strictly limited, since they cannot be treated in the same way as other attributes of objects. Although there

is some reluctance in the database area for introducing identifiers, it should not be very difficult, at least conceptually. For particular systems, identifiers which were always present should become available and visible in some form through the database interface.

2.6 Content addressability

Databases traditionally provide operations based on selection by contents. This is especially true in relational systems, where all relationships between entities are represented by contents, and all operations are based on contents.

In object-oriented systems object contents are typically encapsulated, i.e., hidden. We are not supposed to know the values of an object's variables. Even when objects advertise visible attributes, we may not know whether they are "real" attributes, or virtual attributes computed by the object upon request. This situation presents a double dilemma. First, how can we take advantage of existing indexing mechanisms and content-oriented selection in database systems for object selection? Second, what mechanisms are appropriate for object selection in an object-oriented system?

In the first case we should try to represent some of the behaviour and the characteristics of objects in terms of attribute values visible to the other objects and to the object manager. Such external attributes play the same role as keywords for text retrieval. They are supposed to be representative for retrieval purposes. The problem is that these attributes may not ideally capture the information we need to select the objects. The work on databases for multimedia documents points out some of the problems and solutions.

In the second case we need other mechanisms to select objects which are not based on contents. Since objects encapsulate behaviour, they should also be selectable in terms of their behavioural aspects. Unfortunately databases offer very poor facilities for such selection. Very few database systems offer even simple facilities such as, "Get me the last updated record." In object-oriented systems we need to select objects in terms of where they have been, what operations they have launched on other objects, and what operations they have performed for others. What other objects a particular object knows, or has previously communicated with, may also be relevant. There is a need for behavioural selection methods and their implementation on traditional database content selection.

We should also mention that, since the selectivity for an object oriented system is not particularly high, we have to accept that browsing facilities become very important. Ideas from multimedia document browsing can be used and extended to browse through objects in an object-oriented system. This implies that we need to represent both the objects and their relationships in ways that reflect the user's model of what these objects do.

2.7 Classification

Databases traditionally have very few classes, with large numbers of instances per class. Classification in databases serves mainly to provide a means to efficiently manage and present large amounts of highly regular data. The differentiation between entities is represented by attribute contents and not by subdividing or creating extra classes. Classification in object-oriented sys-

tems serves a very different function, namely to support instantiation, encapsulation and class inheritance.

It is debatable whether object-oriented systems, or, for that matter, the real world, can be classified to such an extent. Databases were able to exploit rampant classification because they left the interpretation of classes to the manipulation programs and transactions. For certain object-oriented systems it may be difficult to force such extensive classification. It is not uncommon to find object-oriented applications with many objects that are the sole instance of their object class.

On the other hand, if instances are only one or two orders of magnitude more numerous than classes then databases have a difficulty dealing with them. There will probably be a need to treat object classes as instances as far as the database is concerned. In any case we will need a mapping between object classes and instances to database classes and instances.

2.8 Schema evolution

Traditional databases allow very little flexibility for evolution of their classes. Schema evolution is very restricted. Relational systems are better than other systems in that they sometimes permit adding attributes. However, dropping attributes or moving them to other relations is seldom permitted.

In object-oriented systems object classes should be able to change to accommodate software evolution [Skarra and Zdonik 1987]. If object classes correspond to database classes the different approaches will certainly create problems. We are again tempted to propose that object classes should be treated as database instances. This differentiation should not be a surprise to database people. After all, logical database instances are not always in correspondence with physical database instances. If object instances and classes are mapped to database instances, this will facilitate object instances changing their class. Since both the object instance and the two object classes are all treated as database instances, there should be fewer problems in representation. The problem of maintaining database consistency in the face of schema evolution has already been addressed in the object-oriented database field [Banerjee et al. 1987b].

2.9 Distribution

Traditional databases deal with distribution, if at all, by hiding it. A distributed database is a logically integrated, physically distributed database. The network is not visible, and we seldom have a notion of context, either as a geographic location, i.e., a workstation, or as a logical context.

The physical notion of context was present before in databases as the notion of an area (i.e., physical volume area where data was stored) but it was taken out. The logical context does not exist except as a logical view, which implies a partition, and perhaps transformation, of the data.

Object-oriented systems need a strong definition of context. First, we believe that objects should be aware of where they are. Physical location in the network may affect their behaviour. Second, objects, or collection of objects, may encapsulate beliefs, and we therefore need a context to define a boundary. (See, for example, [Tsichritzis et al. 1987].) Third, objects' behaviour

may be affected by their context. Sometimes they should even directly inherit methods from their context. A simple example is a *text* object that inherits formatting characteristics from its surrounding scope. Finally, it seems extremely difficult to implement an environment of globally coordinated object managers where objects are managed by local object managers but in a completely integrated and transparent manner.

We are therefore faced with an interesting dilemma. On one hand distributed databases strive to provide a uniform globally integrated database. On the other hand object-oriented systems seem to require a strong notion of context. To what extent the two can co-exist depends a lot on how objects are mapped into databases.

2.10 Passive/active objects

Databases have always fundamentally viewed data objects as being passive. Operations performed on the database are issued from outside, and cause the database to be modified from one consistent state to the next. There is no real notion of the contents of the database being potentially active in the same way that processes managed by an operating system are active.

In certain cases databases have been extended with automatic triggers, but in an ad hoc manner. Triggers are low level alarm facilities for handling exceptions or for chaining operations together. Transactions encapsulate activities, but they are treated separately. Most data models and many systems completely separate the data and the transactions on them.

One of the most interesting aspects of objects is that they can be viewed as active agents [Agha 1986; Nierstrasz 1987]. They not only respond to requests from other objects, but they can trigger themselves. We do not separate objects into active and passive except in degree of activity.

Extensions of databases into object-oriented databases do not handle active objects. Other extensions in terms of automatic procedures and office tasks offer better support for activities [Zloof 1981]. We need, however, a general model of active and passive objects. Databases, as they are today, can only treat objects as passive entities.

2.11 Closed world assumption

Traditional database systems implicitly make the assumption that all information not in the database is not true. This assumption has been attacked for quite some time now by researchers who have extended databases with an inference engine. There is a solid basis for combining logic and databases, and for introducing support for recursive queries and inference.

This issue may seem irrelevant for object-oriented systems but it is important. Objects may incorporate rules. They should be able to augment the knowledge they have using inference. Since objects provide a clear context for inference, the knowledge they manipulate can represent a belief local to the object. We therefore have two effects. First, inference augments the information present in the database or in the objects. Second, objects can give us a context for partitioning knowledge into independent and potentially inconsistent beliefs.

2.12 Overloading, message passing, etc.

There are a host of other issues that seem, at first glance, to be treated differently by databases and object-oriented systems. After some reflection, however, we see that these issues have never been handled by databases and probably do not need to be.

As an example, operator overloading was not needed since databases did not deal with abstract data types, and their operations were very simple. As another example, message passing is a very appropriate paradigm for communication between objects, yet it is absent in databases. Since databases assume a passive view of data, a shared memory model is far more appropriate than a message-passing model. On the other hand, the database itself can be viewed as a large object with which one communicates through an abstract interface (i.e., via “message passing”).

Finally, object-oriented systems emphasize reusability. So do databases, in an extreme manner. Passive objects are shared, and transaction definitions can be reused. Whatever notions the databases capture can be reused. Unfortunately they do not capture very many.

3. Co-existence Approaches

Now that we have examined in detail many of the apparent incompatibilities between object-oriented systems and database systems, we shall consider some of the ways in which the two viewpoints can be reconciled in order to provide some database support for objects, and possibly to provide some object-oriented functionality for databases. The possibilities we shall consider are summarized as follows:

1. Provide an interface between independent object-oriented and database systems.
2. Transparently store dormant objects in a database.
3. Add object-oriented functionality to a database.
4. Add attributes and database functionality to an object-oriented system.
5. Integrate the common functionality required by both systems, and provide additional support for querying, transactions etc., for “database objects.”

3.1 Separate co-existence

The simplest approach is depicted in figure 2. The application is divided into two clear parts: programs and data. Programs are encapsulated as objects using the object-oriented system. Data are managed by the database. Objects can issue direct database commands to the database. The object manager can also use the database for its own needs. There is a very clear distinction between objects and database instances.

Persistence for objects is provided directly by the object manager through, for example, a large, persistent virtual memory. Support for object creation, destruction, concurrency, etc., is dealt with by the object manager independently of the database system. This is analogous to the traditional distinction between operating systems and database systems.

This approach does not require major changes to either database systems or object-oriented systems. In addition, it allows existing applications implemented in a traditional manner to co-

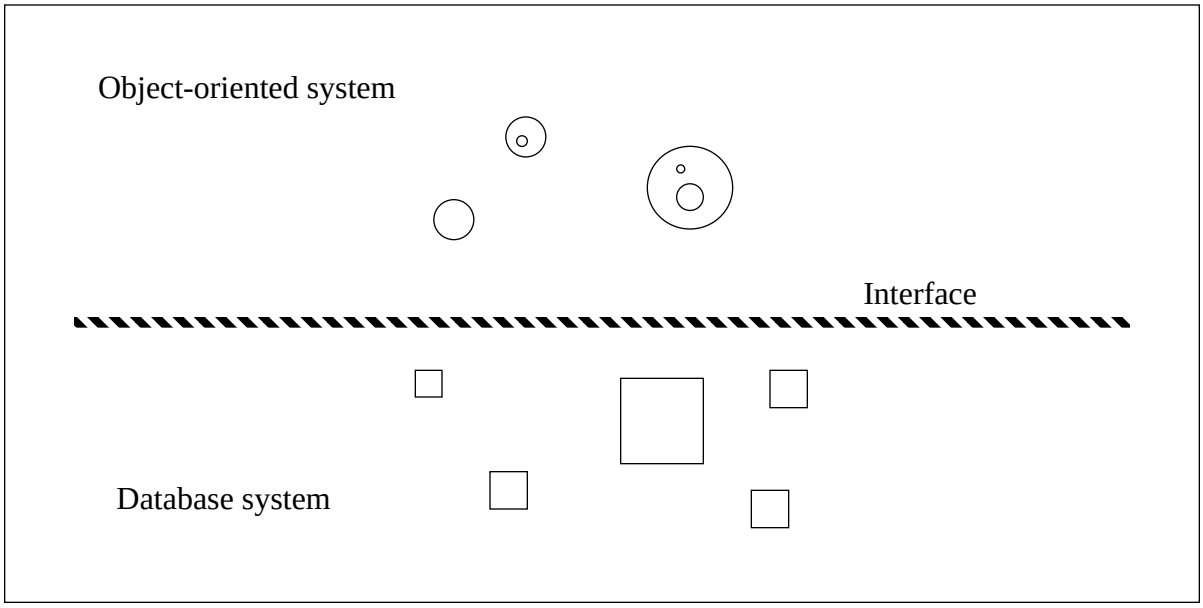

Figure 2 Separate Object-Oriented and Database Systems

exist with object-oriented systems. Furthermore, the principles of object independence and data independence are generally reconciled. There are, however, two ways for objects to communicate with one another, either directly by message-passing, or indirectly through the database. If there is a need for objects to encode part of their contents in the database, then object independence can be compromised through sharing of database instances.

There are two other drawbacks to this approach. First, there is a lack of uniformity. Some application entities become objects, others become database instances. Second, there will be some duplication of effort. The object manager will require some mechanisms similar to those used by the database system.

3.2 Active and dormant objects

A second approach is depicted in figure 3. Objects can be active or dormant. Active objects exist in a large, persistent workspace managed by the object manager. Objects become “dormant” either by putting themselves asleep, or by being retired by other privileged objects. Various policies can be used for deciding when to retire objects. The problem is similar to that of managing a virtual memory. When objects are dormant they are labeled using their identifier, and perhaps some other attributes, and they are stored in the database. As far as objects are concerned, there is no such thing as a database.

Dormant objects can be re-awakened by the object manager, for example, when a message is sent to them. Re-activated objects have precisely the same state as they did when they were put to sleep. Creation of a new object implies initialization of variables and a new identifier. Reactivation implies the old identifier and the same variable contents as at the time the object died.

This approach has the advantage that it is completely object-oriented. The database does not exist except as a facility for storing dormant objects. This allows objects to maintain the illusion of a purely object-oriented environment where object independence is emphasized. On the other hand, it does not address the problem of access to existing databases, nor does it pro-

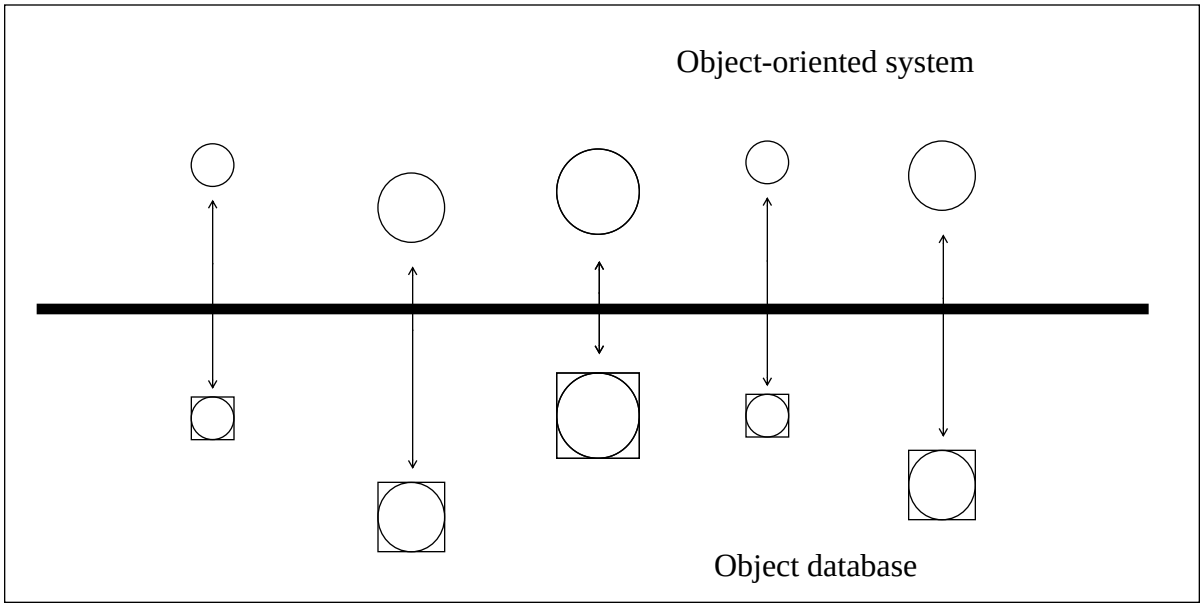

Figure 3 Active and Dormant Objects

vide a means to exploit other database functionality from within the object-oriented system. The database is used like a file system. Querying and transaction management are unavailable except to the object manager.

3.3 Object-oriented databases

Object-oriented databases are depicted in figure 4. This term is *not* used to mean "databases for supporting object-oriented systems." It refers, rather, to databases that have been extended by incorporating various object-oriented concepts, i.e., abstract data types, complex objects, multiple inheritance, etc. [Bancilhon 1988; Maier and Stein 1987; Banerjee et al. 1987a; Fishman et al. 1987].

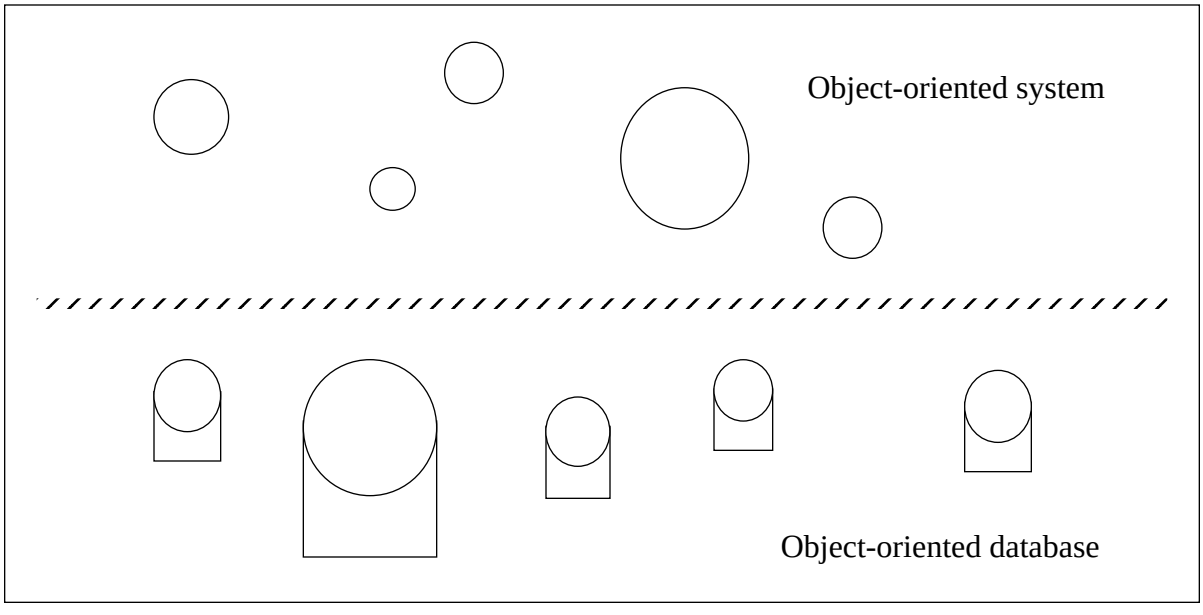

Figure 4 Object-Oriented Databases

Reconciling the independence principles poses no problem. Database "objects" are clearly separated from the rest of the objects, thus providing data independence. The other objects emphasize object independence between themselves.

Due to the inclusion of object-oriented features, the interface to object-oriented databases resembles that of object-oriented systems. Interaction with database objects is similar to communication with other objects. The object manager is relieved of the responsibility of managing the vast collection of relatively docile and well-structured objects. It should be able to do a better job of catering to the needs of the other objects. Furthermore, by extending database systems with object-oriented concepts, we greatly enhance the functionality and usability of databases (though object-oriented databases will not have the simple set of generic operations that current databases have).

There is, nevertheless, a clear distinction between database instances as objects and the other objects. First of all, it is not clear that the particular object model adopted by the object-oriented system will match that of the object-oriented database, unless they have been designed with that purpose in mind. Next, object-oriented databases have been developed to support data-intensive applications, like CAD/CAM, that have to deal with abstract "objects," not to support object-oriented applications in general. As a consequence, there is an emphasis on large numbers of similar, well-structured objects with external attributes. It is not clear that object-oriented databases have much to offer for objects that do not satisfy these assumptions. Finally, it is not clear how to use object-oriented databases to store active objects. The execution state of an object involved in a long-term activity is not something that databases are normally expected to deal with.

3.4 Objects with external attributes

This approach is depicted in figure 5. In a large, object-oriented system, there can be a severe selection problem. (This is analogous to the problem of posing an ad hoc query on a file system.) One approach for dealing with this problem is to abstract an object's behaviour and properties in terms of a number of external attributes. These attributes, possibly including text descriptions, represent the salient properties of the object to the outside world. An object can be selected by posing a query in terms of the values of these attributes. The attribute values may be set by the object itself, or, in some cases, by the object manager (e.g., the last update time). We can view this approach as providing another (square) database-oriented interface to objects.

If we follow this approach, it would be reasonable to use the same attributes to store and retrieve the objects. All communication is addressed via the object identifier or the values of the external attributes. The object manager can therefore use these attributes to store objects in a database, and to re-activate them when required.

This approach deals well with both the addressing and the persistence problems. Existing database instances can be treated as special objects that have only external attributes. The disadvantage is that objects cannot be purely encapsulated. We introduce a notion of visible attributes. Objects are accessed not only through their identifiers but through the values of the attributes. Furthermore, it does not really solve the problem of object selection, since static attributes cannot fully capture all of the interesting dynamic properties of objects. How do you decide what

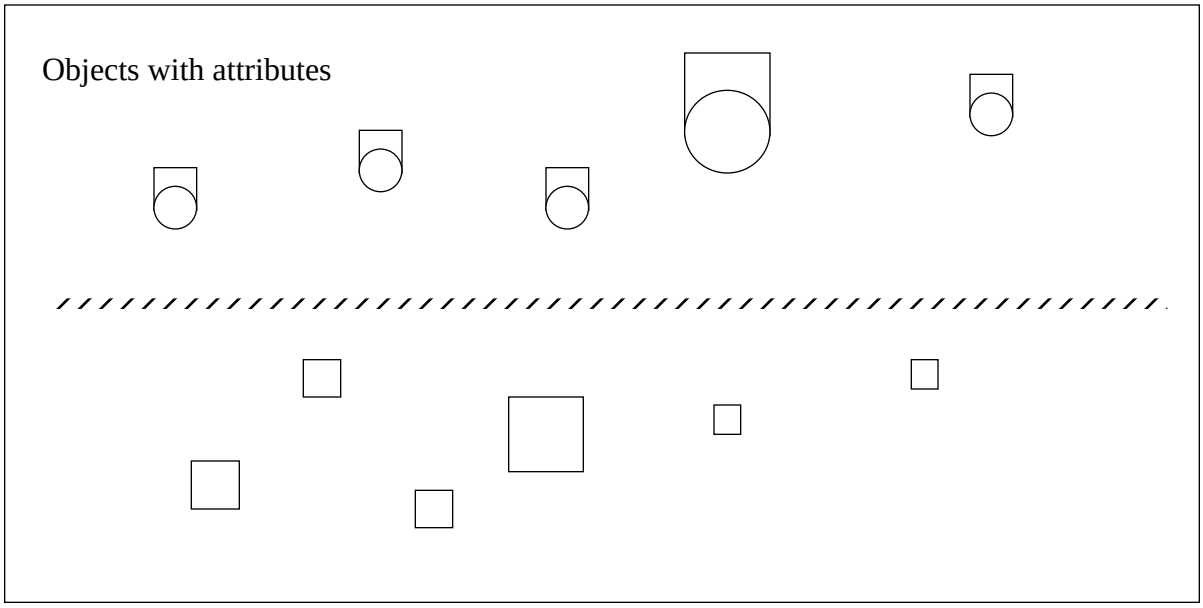

Figure 5 External attributes

visible attributes your objects should have? Nevertheless, this is a nice compromise between the object-oriented and database paradigms.

3.5 “Plastic” objects

What we would really like is to merge the object-oriented and database approaches. “Plastic” objects would have it all (figure 6). They would behave like objects, obeying object independence, but they could also look like database objects, if necessary. They would be round and square at the same time!

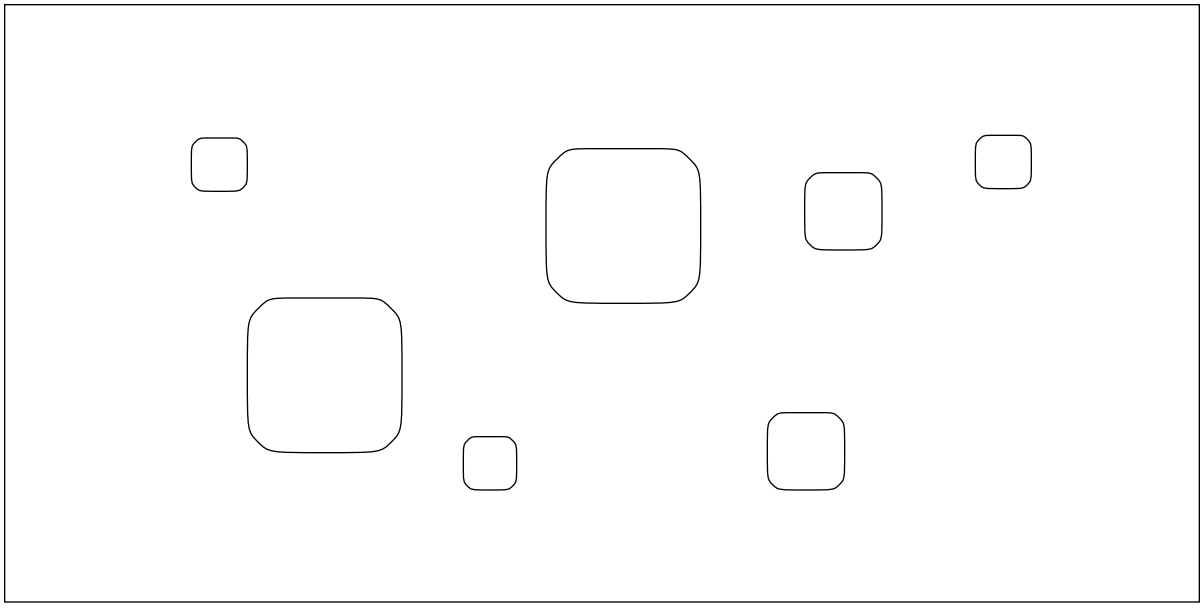

Figure 6 “Plastic” objects

The problem with all of the approaches we have described up till now is that they do not successfully break through the database barrier. The object manager can use some database sup-

port for nearly all objects, namely for persistence and for maintaining object identifiers. Other issues, dealing with querying, indexing, efficient storage and transaction management, may be relevant only for a large number of well-structured objects belonging to few object classes. Highly active objects can be handled in more efficient ways than by keeping track of all their changes in a database.

What we propose as a solution is to factor out the common database and object management support into a shared subsystem that is not seen by objects. Highly active objects can be stored and managed using the low-level object management subsystem. More passive objects with visible attributes exploit the same object management subsystem, but are provided with additional support for querying, etc. Given support for schema evolution, it is even possible for objects to migrate between the two parts of the object environment, either becoming more like database objects, or becoming more active. For example, redefining an object class to provide visible attributes for its instances will cause those objects to migrate to the database part of the object environment.

This approach would also accommodate existing databases or object-oriented databases in the same way as we have described above. Since external databases constitute independent object worlds, all that is required is an interface for communicating with such objects.

The advantage of this approach is that it is completely object-oriented, retaining all the claimed advantages of object-orientation. The obvious disadvantage is that the object manager is asked to solve many difficult problems. However, the real difficulty is that the application designer will be offered a choice within the same system. Applications can be designed in such a way as to emphasize a separation of activities and data, or, alternatively, they can be designed in a completely integrated manner. Sometimes the choice will be obvious. In some cases it may be very difficult.

4. Conclusions

We have discussed the difficult issues arising from a co-existence of object-oriented and database systems. Some of these issues are a natural follow-up to differentiation of approaches between programming languages and database systems. Programming languages emphasize operations while databases emphasize information representation.

There are many possible attitudes we can take.

1. The object-oriented and database fields can ignore each other. This will not be realistic and fortunately is not happening.
2. The object-oriented and database fields can develop, but each using the facilities of the other in a decoupled way.
3. The object-oriented and database fields can borrow concepts from each other, each trying to duplicate the other’s efforts.

4. The object-oriented and database fields can try to merge their capabilities to arrive at systems which can smoothly integrate the facilities for both, without prohibiting either a purely database-oriented approach or a purely object-oriented approach to the problem.

We obviously prefer the fourth solution, and we believe that it is a promising direction to pursue. This presupposes, however, an open approach, accepting ideas from a different area, and respecting the limitations and constraints that each area poses. The limitations and constraints of object-oriented systems and database systems did not arise through chance or oversight. They arose because other principles and other ideas were heavily emphasized.

Finally, we should note that a discussion on whether databases should be extended, or another field should redevelop its capabilities is not new to databases. For instance, the same difficulties arose with knowledge bases. Should knowledge bases be implemented on top of databases? Should databases be extended to incorporate knowledge base ideas? Or should knowledge bases provide their own storage and retrieval facilities? This controversy has not been settled completely and we are now embarking on similar discussions concerning object-oriented systems. In the end, it is not important whether object-oriented or database ideas prevail. Rather, it is important what facilities the final system offers.

References

[Agha 1986] G.A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

[Bancilhon 1988] F. Bancilhon, "Object-Oriented Database Systems," Proceedings 7th ACM SIGART/SIGMOD/SIGACT Symposium on Principles of Database Systems, Austin, Texas, March 1988.

[Banerjee, et al. 1987a] J. Banerjee, H. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou and H. Kim, "Data Model Issues for Object-Oriented Applications," ACM TOOIS, vol. 5, no. 1, pp. 3-26, Jan 1987.

[Banerjee, et al. 1987b] J. Banerjee, W. Kim, H-J Kim and H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," Proceedings ACM SIGMOD '87, vol. 16, no. 3, pp. 311-322, Dec 1987.

[Bloom and Zdonik 1987] T. Bloom and S.B. Zdonik, "Issues in the Design of Object-Oriented Database Programming Languages," ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 441-451, Dec 1987.

[Brachman 1988] R.J. Brachman, "The Basics of Knowledge Representation and Reasoning," AT\&T Technical Journal, vol. 67, no. 1, pp. 7-24, Jan/Feb 1988.

[Fishman, et al. 1987] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan and M.C. Shan, "Iris: An Object-Oriented Database Management System," ACM TOOIS, vol. 5, no. 1, pp. 48-69, Jan 1987.

[Goldberg and Robson 1983] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*, Addison-Wesley, May 1983.

[Maier and Stein 1987] D. Maier and J. Stein, "Development and Implementation of an Object-Oriented DBMS," in *Research Directions in Object-Oriented Programming*, ed. B. Shriver, P. Wegner, pp. 355-392, The MIT Press, Cambridge, Massachusetts, 1987.

[Mylopoulos and Levesque 1983] J. Mylopoulos and H. Levesque, "An Overview of Knowledge Representation," in *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages}, ed. M. Brodie, J. Mylopoulos, pp. 3-17, Springer-Verlag, New York, 1983.*

[Nierstrasz 1987] O.M. Nierstrasz, "Active Objects in Hybrid," ACM SIGPLAN Notices Proceedings OOPSLA '87, vol. 22, no. 12, pp. 243-253, Dec 1987.

[Nierstrasz 1988] O.M. Nierstrasz, “A Survey of Object-Oriented Concepts,” in *Object-Oriented Concepts, Applications and Databases*, ed. W. Kim and F. Lochovsky, Addison-Wesley, 1988, (to appear).

[Skarra and Zdonik 1987] A.H. Skarra and S.B. Zdonik, “The Management of Changing Types in an Object-Oriented Database,” in *Research Directions in Object-Oriented Programming*, ed. B. Shriver, P. Wegner, pp. 393-415, The MIT Press, Cambridge, Massachusetts, 1987.

[Tsichritzis, et al. 1987] D.C. Tsichritzis, E. Fiume, S. Gibbs and O.M. Nierstrasz, “KNOs: KNowledge Acquisition, Dissemination and Manipulation Objects,” ACM TOOIS, vol. 5, no. 1, pp. 96-112, Jan 1987.

[Zloof 1981] M.M. Zloof, “QBE/OBE: A Language for Office and Business Automation,” IEEE Computer 14, pp. 13-22, May 1981.