

---

---

# Carnegie Mellon

**45-872**

**Information Resources Management**

---

## **The Evolution of Object-Relational Databases**

Authors:	Chris Lord and Sandhya Gupta
Edit Number:	64
Last Revised:	3/4/2002 3:57 PM
Date Printed:	5/7/2002 11:17 PM
File Location:	arsenic.ini.cmu.edu/45-872/Project

---

---

---

## Table of Contents

<b>Introduction .....</b>	<b>1</b>
<b>Object-Oriented Database Systems.....</b>	<b>1</b>
Features and Strengths .....	2
Technical Characteristics .....	3
Example of Objects.....	4
Weaknesses.....	4
<b>Extended Relational Database Systems .....</b>	<b>5</b>
SQL as Fundamental.....	5
The Relational Model as Fundamental .....	6
Combined Extensions .....	6
<b>Object-Relational Database Systems .....</b>	<b>6</b>
Standardization .....	6
An Example of Object Definitions.....	8
<b>Conclusion .....</b>	<b>8</b>

## Related Web Sites

Description	Site	Comment
ANSI	web.ansi.org	American National Standards Institute
ISO	www.iso.ch	International Organization for Standardization
NCITS	www.ncits.org	National Committee for Information Technology Standards
IBM DB2	www.ibm.com/software/data/db2/	Object-relational database management system
eXcelon Corp	www.exceloncorp.com/products/index.shtml	Object-oriented database system and relational middleware
Oracle	www.oracle.com/ip/dep/otn/database/oracle9i/	Object-relational database management system
Microsoft	www.microsoft.com/sql/default.asp	Object-relational database management system

## References

- [1] ANSI/ISO/IEC, *Information Systems—Database Language—SQL—Part 1: Framework (SQL/Framework)*, 9075-1:1999, September 1999.
- [2] Atkinson, M., et al, “The Object-Oriented Database System Manifesto,” *Proceedings of the 1<sup>st</sup> DOOD Conference*, 1989. (Hardcopy Provided)
- [3] Carey, M. and DeWitt, D., “Of Objects and Databases: A Decade of Turmoil,” *Proceedings of the 22<sup>nd</sup> VLDB Conference*, 1996. (Hardcopy Provided)
- [4] Chandler, J., “SE.OOSD1: Object Oriented Software Development Lecture Notes,” <http://www.dis.port.ac.uk/~chandler/OOLectures/OOSD.html>, University of Portsmouth, England.
- [5] Chaudri, A., “Object Database Management Systems: An Overview,” *BCS OOPS Newsletter*, No. 18, pp. 6-15, Summer 1993. (Hardcopy Provided)
- [6] Codd, E. F., “A Relational Model of Data for Large Shared Data Banks,” *Communications of the ACM*, Vol. 13, No. 6, June 1970.
- [7] Darwen, H., Date, C. J., “The Third Manifesto,” *SIGMOD Record 24(1)*, March 1995. (Hardcopy Provided)
- [8] Devarakonda, R., “Object-Relational Database Systems: The Road Ahead,” *ACM Crossroads*, Vol. 7, No. 3, 2001.
- [9] Dumas, M., “Object-Oriented Systems,” Lecture Notes, <http://sky.fit.qut.edu.au/~milliner/OO/>, 2001.
- [10] Manola, F., “An Evaluation of Object-Oriented DBMS Developments,” GTE Laboratories Incorporated, TR-0263-08-94-164, August 31, 1994. (Hardcopy Provided)
- [11] Oracle Corporation, “Simple Strategies for Complex Data: Oracle9i Object-Relational Technology,” Oracle Technical White Paper, July 2001.
- [12] Oracle Corporation, *Oracle9i Application Developer's Guide Release 1 (9.0.1)*, Part Number A88878-01, 2001
- [13] Stonebraker, M., et al, “Third-Generation Database System Manifesto,” *SIGMOD Record 19(3)*, July 1990. (Hardcopy Provided)

## Introduction

The strength of the relational model comes first from its mathematical consistency through first-order predicate calculus. But its real power lies in two fundamental improvements over prior database systems: abstraction and isolation. In his seminal 1970 paper, Codd describes these key characteristics of the relational model:

“It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation on the other.” [6]

In the technology of the time, this was a radical leap forward. The relational model allowed data independence from hardware and storage implementations—the bane of earlier data management systems—through abstraction in the form of relations represented as tables. It also formalized the boundary between how data was stored and how data was used through a high-level query language.

Object-oriented database systems represent a similar leap forward in database technology. While the relational model, in the words of Chandler, “hammer the world flat” [4] through a process of normalization, object-oriented database systems allow objects to be represented in forms that closely match their real-world correspondents through the use of complex user-defined types and flexible relationships. By incorporating the relationships of inheritance and polymorphism, an object in a database can accurately reflect the relationships between objects in the real world.

In a sense, the expressive power of the object-oriented approach is analogous to that of the conceptual schemas in information systems architecture, but applied to the data itself. An object-oriented approach separates the content of a datum from its form through a well-defined interface just as a conceptual schema isolates the semantics of data from its physical organization through the interface of SQL.

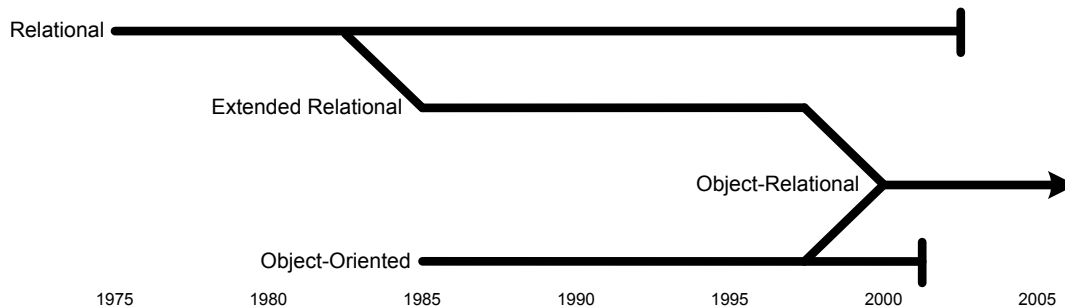


Figure 1: Approximate Timeline of Various Database Technologies

This paper will show that object-oriented features are important for many applications and that they are inadequately met in relational database management systems. The first section presents the features, technical characteristics and some sample applications of object-oriented database systems. This is followed with an analysis of the business and technical factors that prevented object-oriented database systems from making the expected inroads into the existing database market. Pure object-oriented database systems are contrasted with a parallel effort to extend relational database systems with object-oriented features. Finally, the merger of object-oriented and relational technologies in the form of object-relational databases is examined and some forecasts are made regarding this new hybrid. In particular, this paper predicts that both relational and object-oriented database systems have a limited future and instead will continue on in the combined form of object-relational database systems. A timeline corresponding to each of these developments is shown in Figure 1.

## Object-Oriented Database Systems

The relational database model has been pushed to its limits due to several real-world applications that demand more information from data. A relational database system is limited in the type of data it can handle—numbers, characters, dates, etc.—but now businesses demand that their databases handle many other types of data such as text, images, audio, video and time series data. Relational database systems are ill-suited to the needs of complex

information systems; they require all information to be modeled into tables (relations) of rows and columns where relationships between entities are defined by a set of atomic column values.

In contrast, object-oriented database management systems allow a more natural modeling of entities and the relationships between entities. These systems combine the features of object-oriented design with database technology to provide an integrated environment. Objects are defined in terms of data and the methods which operate on that data. The methods are not tied to particular database application (as they often are in relational database management systems) but to the data itself. This provides the advantage of an additional layer of abstraction and isolation between the applications and the data. Furthermore object-oriented systems can be extended to support complex objects such as multimedia content by defining new classes that have operations to support the new kinds of information.

Object-oriented management systems provide three powerful forms of relationships: inheritance, polymorphism and encapsulation. Encapsulation supports abstraction by hiding details that are unnecessary, and building complex data types. Inheritance allows solutions to problems incrementally by defining new objects in terms of previously defined objects. Polymorphism allows operations to be defined for one type of object and then shared with other types of objects. These operations can further be extended to provide behaviors unique to those objects.

This is one of the key differences between object-oriented and relational database management systems. Object-oriented database systems represent relationships between objects explicitly using specialization (instance-of), generalization (is-kind-of), composition (part-of), versioning (version-of) and references (refers-to). This allows objects to behave differently depending on how they are used and provides associative and navigational access to information. The more complex the relationships between information, the greater the advantages of being able to capture them explicitly in the logical structure of the database.

## Features and Strengths

Object-oriented database management systems offer many advantages over relational systems. These include:

<i>Representational Consistency</i>	All operations occur at the object level that avoids the inconsistency between set operations in the relational model and the record operations at the language level in relational database management systems.
<i>Fast Navigation</i>	The inherent references between objects make navigational queries nearly as fast as accessing data in memory. Relational systems perform slowly for applications that navigate from object to object.
<i>Accurate Modeling</i>	The rich variety of semantic constructs and relationships allows objects in the database to closely match their counterparts in the real world to any level of complexity. Relational systems that require a translation from the real-world information structure to the tables of relations used to represent data.
<i>Safety</i>	Encapsulation hides the details of implementation from the user and ensures that changes to objects can only be made in certain ways as defined by the object's interface. This helps safeguard the integrity of objects and data.
<i>Code Reuse</i>	The opportunity to reuse code is much greater because classes encapsulate the attributes and behaviors of their instances. Such code only needs to be developed once and is insulated from the objects use in applications. This leads to lower maintenance costs. Inheritance also contributes to code reuse because a subclass adds only new data or functionality to that of the superclass without reimplementing the superclass.
<i>Language Integration</i>	An object-oriented database management system integrates tightly with one or more object-oriented programming languages. The same objects manipulated in the language are stored in the database. Relational systems integrate poorly with other languages requiring the use of embedded SQL.

With the growing need in organizations to represent, store and manipulate complex data and relationships, object-oriented database management systems have an advantage over relational products. An example of this advantage is the representation of CAD or CASE data, such as a schematic electronics diagram, in a database. A schematic is a

model of modules composed of components composed of gates with complex series of interconnections and interrelationships that does not lend itself to a representation in the form of relations.

Other applications for which object-oriented systems are particularly useful include telecommunications data supporting requiring navigational access, finance and trading requiring time-series, multimedia, and geographic information systems.

## Technical Characteristics

The technical requirements of object-oriented database management systems were formally specified in the Object-Oriented Database System Manifesto [2] after many years of different interpretations of object-oriented features and different functionality between vendors. These features include:

<i>Classes</i>	A class (or abstract data type) captures the common features of a set of objects. It has two components: the interface and the implementation. Only the interface part is visible to the users of the class, the implementation of the object is seen only by the class creator thus incorporating the principles of abstraction and encapsulation.
<i>Objects</i>	An object is a representation of a real-world object. Objects can be physical items (car, plane, dog) or abstract concepts (shape, appointment). Objects are self-contained entities that include attributes (instance data or variables) and methods (code or functions) that act on attributes.  The state of an object can be defined as the values of an object's attributes at any point in time. These attributes may be simple base types (integers, strings, etc.), more complex types such as arrays, sets, or other complex objects. An object can participate in different types of relationships (specialization, generalization, aggregation, composition, versioning and referencing). Each object, therefore, can behave differently depending on how it is used.
<i>Object Identity</i>	An object has an existence which is independent of its value. This identity is maintained through object identifiers (OIDs). This allows objects to be tested for identity (the OIDs are the same) and equality (the OIDs are different but the attributes are the same). These different relationships are necessary to maintain and test inheritance relationships.
<i>Encapsulation</i>	The manipulation of an object is only possible through its defined external interface; the instance variables and methods are hidden. As a result, the implementation can be changed without affecting existing program code that uses the object's interface. This provides logical data independence.
<i>Single and Multiple Inheritance</i>	The relationship between classes is based on an inheritance hierarchy. A subclass (or subtype) is a specialization of its superclass (or supertype) and inherits the attributes and methods of its superclass; superclasses are generalizations of their subclasses. In single inheritance, a subclass has only one superclass; in multiple inheritance, a subclass can inherit from two or more superclasses.
<i>Strong Typing</i>	Strong typing refers to the ability to treat instances of a type or class that share the same base type or representation as distinct types.

All of these features help to more accurately model the real world and allow precise descriptions of objects and relationships. These features are all very much object-oriented concepts and could equally apply to languages (which is one reason for the tight integration with object-oriented languages). In addition, an object-oriented database management system needs to support object persistence. Most objects are transient in nature—coming into existence as necessary and disappearing when all referents are gone. A database system requires that objects (state, methods and relations) be permanently stored so that they can be accessed later. Persistence also introduces the need for transactions and locking to ensure the integrity of the object store. Persistence and transactions are the most significant characteristics that distinguish object-oriented database management systems from simply object-oriented programming environments.

## Example of Objects

Consider an object type that describes objects in plane geometry. The class Geometry is referred to as an interface class or abstract class that specifies variables and function definitions. It cannot be instantiated but instead serves as a base class for other classes, which can be instantiated. The relationship between the Geometry class and its subclasses is shown in Figure 2. The Geometry class is defined as follows:

```
Class Geometry {
  attributes::
    struct reference_point{ int x, int y} refpt;
    shape { 'R', 'T', 'C' } shape;
  Methods::
    float perimeter();
    float area();
};
```

Here shape can be 'R' for Rectangle, 'T' for Triangle and 'C' for Circle. The structure refpt contains the reference point with respect to a shape in a two-dimensional plane. The attribute refpt for a rectangle is the center point, for a circle it is the center point, and for a triangle it is the vertex point.

```
Class Rectangle {
  attributes::
    int length;
    int width;
    struct Point {
      int x, int y} refp;
};

Class Triangle {
  attributes::
    int sideAB, side BC,
    side CA;
    int angleAB, angleBC,
    angleCA;
    struct Point {
      int x, int y} refp;
};

Class Circle {
  attributes::
    int radius;
    struct Point {
      int x, int y} refp;
};
```

All functions of the Geometry supertype are inherited by each of the three subtypes (Rectangle, Triangle and Circle). When an object is created in a database, it belongs to one of these types. All objects whose shape is 'C' are of the subtype Circle which also inherit properties of the Geometry interface class. The functions **area()** and **perimeter()** are declared for all objects of type Geometry, but the implementation of the method for calculating the area differs for each subtype. The function to calculate area will be overridden by an implementation in each class and the appropriate method for the area function selected based on the object type.

This example is not only difficult to model using relational database systems, but also inefficient. In relational database systems, a generic class with features to be inherited by any other relation is not possible. At best, one can define separate relations for each Rectangle, Circle and Triangle. This involves duplication of data with additional storage. If an application needs to add another shape such as Ellipse, this will create another relation if a relation containing details for Ellipse does not already exist in the database. There is no generic relation from which the application can extend and inherit properties of generic shape class and then build on this class to provide functionality specific to Ellipse in relational database systems.

This contortion of separate relations for different shapes does not reflect the needs or relationships in the geometry example. Hence, relational databases cannot easily model some complex real-world scenarios. On the other hand, object-oriented database management systems can model scenarios such as the geometry example with accuracy and flexibility using the features of encapsulation and inheritance.

## Weaknesses

Despite their many advantages, object-oriented database management systems failed to capture the traditional database market. Part of this failure is due to business factors, but part of it is due to technical limitations of object-oriented database systems or their implementations.

The number one reason for object-oriented database systems remaining far behind relational systems is business inertia. Relational database systems have been in the market for more than two decades. A lot of research and understanding has been developed. They have been widely deployed. Many corporations have large investments in

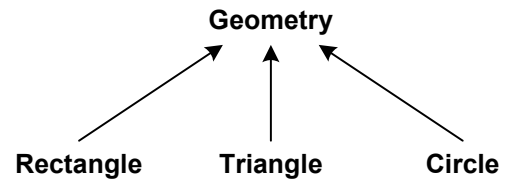


Figure 2: Geometry Class Hierarchy

software, applications and trained staff, not to mention in the databases themselves. A corporation migrating to an object-oriented database system needs to invest in not only developing new object-oriented models but also educating its employees and ensuring the integration of old applications with new databases.

The hesitation on the part of businesses to invest in this technology is complicated by the lack of standardization among the original object-oriented database system vendors. Systems supported different languages (Smalltalk, C++, Java) and had different sets of object-oriented features. Those features that were common were often implemented in different ways such that no single modeling approach would work with all products. Businesses were wary of depending on a specific vendor: there was no equivalent of SQL to isolate them from such product or vendor dependencies.

Several technical factors also contributed to the limited market penetration of object oriented database systems:

<i>Complexity</i>	With the flexibility of object-oriented systems also comes complexity. The relational model has a very simple, albeit limited, way of representing all entities and attributes and capturing the relationships between them. This is one of its greatest strengths. Object-oriented systems have many forms of relationships, which make it harder to know what to apply and when to apply it.
<i>Navigation Approach</i>	In object-oriented database systems, a procedural style of navigation is necessary as references are used to link objects to each other in various relationships. This makes querying more difficult, often inefficient, and limits the nesting of queries. Furthermore, the database may be subject to security violations or corruption by incorrect or obsolete references between objects. The relational models integrity constraints are not present or enforced in object-oriented databases because there isn't the isolation between programming and data.
<i>No Theoretical Foundation</i>	Object-oriented systems lack a mathematical foundation that results in loss of analysis, deduction and insight.

To meet various types of application requirements, many organizations are looking for a single database platform that applies scalability, transaction integrity, enforcement of business rules, and other robust relational database management functionality along with support for complex data types and relationships.

## Extended Relational Database Systems

In parallel with efforts to develop pure object-oriented database systems, supporters of the relational model were looking at ways to extend the relational model with object-oriented features. There was general consensus on the useful object-oriented features to be considered (user-defined complex data types, encapsulation and inheritance). There was not, however, agreement on how these features should be grafted on existing systems. The industry split into two camps: those who believed SQL was fundamental, and those who believed the relational model was fundamental.

### SQL as Fundamental

Those in favor of extending relational database systems through language extensions described their position in the "Third-Generation Database System Manifesto." [13] The claim made is that object-oriented features can be added without abandoning SQL, that SQL *must* be the basis for these features. The authors were less concerned with the underlying database system and model than with maintaining a uniform query language: "current systems that claim to be object-oriented generally are not faithful to any of our tenets and support propositions. To become truly third generation systems, they must add a query language and query optimizer, a rules system, SQL client/server support, and support for views."

Relational database systems, they argue, are much closer: "perhaps the most important disagreement we have with much of the OODB community is that we see a natural evolution from current relational DBMSs to ones with the capabilities discussed in this paper...To become true third generation systems they must add inheritance, and additional type constructors." However, the emphasis is on SQL as the means of providing these features: "For

better or worse, SQL is intergalactic dataspeak. SQL is the universal way of expressing queries today...Moreover, SQL is a reasonable candidate for the new functions suggested in this paper..."

## The Relational Model as Fundamental

The second approach to extending relational database systems is described in "The Third Manifesto." The authors concede the value of object-oriented features, but adamantly oppose any changes to the relational model:

"We fully acknowledge the desirability of supporting certain features that have been much discussed in more recent times, including some that are commonly regarded as aspects of Object Orientation. We believe that these features are orthogonal to the Relational Model, and therefore that the Relational Model needs no extension, no correction, no subsumption, and, above all, no perversion, in order for them to be accommodated in some database language that could represent the foundation we seek." [7]

The manifesto goes on to attack SQL as the limiting factor impeding the incorporation of object technology with relational databases: "We feel strongly that any attempt to move forward, if it is to stand the test of time, must *reject SQL unequivocally*." (Emphasis in original.) This deprecation of SQL stems from its obvious lack of support for the extensions described by the manifesto and, more importantly, from the forms that compromise the relational model. For example, anonymous columns created by a SQL statement like "SELECT X+Y FROM T," nulls and multi-valued attributes in tables (such that tables are not really relations), and the fact that SQL allows multiple rows to exist with identical attributes. To restore the purity of the relational model and add support for objects, the authors define the properties of the relational model in terms of a new hypothetical language, *D*, with an object-orientation.

The third manifesto defines a domain as a named set of values of arbitrary complexity that is interchangeable with a data type. Domains can have user-defined functions which allow values to be constructed and user-defined operators. Domains can be subdomains of superdomains creating a mechanism for single and multiple inheritance. In essence, domains are the classes in traditional object-oriented systems and through them most of the features of object-oriented systems can be provided.

## Combined Extensions

As might be expected from the two hard-line positions advocated in each manifesto, neither succeeded in wholly winning over followers in the opposite camp. But neither is it clear that both camps were that far apart from each other (although both are equidistant from the first manifesto). Differences between the two really amounted to whether SQL should or could form the basis of object-oriented features. The first camp did not object to maintaining relational underpinnings; they simply were flexible regarding the actual implementation as long as support for SQL was maintained. The second camp sought to defend the purity of the relational model from an ever-looser SQL. The reality was that SQL was not going away; companies had too much invested in databases, applications and education.

## Object-Relational Database Systems

An object-relational database system is the syncretic union of object-oriented features and the relational model. It is a compromise between the attempts to extend the relational model since it is based around a new version of the SQL92 standard yet incorporates the most powerful language feature described in the third manifesto, domains. At the same time it provides most of the object-oriented features originally proposed in the first manifesto.

## Standardization

The object-relational model is defined in the ANSI/ISO SQL99 standard, formerly known as SQL3 since it represents the third-generation of SQL. SQL99 is a superset of SQL92, which maintains backward compatibility while adding object-oriented features (as well as many extensions not specifically object-oriented). These features include user-defined types, methods, references, collections and large objects.

*Domains*                    A domain is a named user-defined object that can be specified as an alternative to a data type, wherever a data type can be specified. A domain consists of a data type, possibly a default option, and zero or more constraints.

*Structured Types* A structured type is a named, user-defined data type. The value of a structured type is the value of the attributes which define the type. Attribute values are encapsulated such that they are not typically visible to the user but may be made available through functions. Structured types support single and multiple inheritance through subtype and supertype relations. Limits on inheritance are possible using the FINAL keyword, which prevents a type from having subtypes. Polymorphism is also supported which allows an expression of a subtype to appear anywhere that an expression of any of its supertypes is allowed. For example:

```
CREATE TYPE EmployeeType UNDER PersonType AS (
    Salary DECIMAL(8,2) NOT FINAL);
```

This defines EmployeeType as a subtype of PersonType with the additional attribute of Salary.

*Distinct Types* A distinct type is a special user-defined data type that provides strong typing regardless of the underlying base type. This allows the use of the type to be precisely controlled. For example, consider the types Dollar and Rupee defined as:

```
CREATE DISTINCT TYPE Dollar AS DECIMAL(9,2)
CREATE DISTINCT TYPE Rupee AS DECIMAL(9,2)
```

The following statement will return an error despite identical representations in the database:

```
CREATE TABLE Items (
    ItemName VARCHAR(40),
    PriceInEuros Euro,
    PriceInDollars Dollar );
SELECT * FROM Items WHERE PriceInRupees = PriceInDollars
```

*Methods* User-defined types can have methods that are invoked to create or construct a type's value. These methods can be external (programmed in an object-oriented language such as Java or C++) or SQL routines. The following example illustrates the used of a method definition:

```
CREATE TYPE PersonType AS (... METHOD Age() RETURNS DECIMAL(3));
```

*Collections* SQL supports collection types that can represent entire groups of other types and be treated as a single type, which provides support for encapsulation. Two important collections are arrays and row types. Arrays are analogous to their counterparts in other languages: they allow multiple values of the same type to be referenced as a whole while still providing indexed access to the individual values. For example consider the following:

```
CREATE TABLE Student (
    Name VARCHAR(20),
    Grades INTEGER ARRAY[10] )
INSERT INTO Student
VALUES('Chris Lord', ARRAY[90, 95, 82])
```

Row types allow multiple columns to be treated as a single column entry. In this manner, entire relations can exist as attributes within another table. For example:

```
CREATE TABLE Persons (
    Name ROW(FirstName VARCHAR(20), Surname VARCHAR(20))
    Address ROW(Street VARCHAR(30), City VARCHAR(20)) )
INSERT INTO Persons VALUES(ROW('Sandhya', 'Gupta')
    ROW('5000 Forbes Ave', 'Pittsburgh'))
SELECT Address.city
FROM Persons WHERE Name.FirstName = 'Sandhya' AND
    Name.Surname = 'Gupta'
```

The new SQL99 standard achieves an object-orientation at the considerable cost of increased complexity of the specification (the original SQL standard released in 1989 was about 120 pages; the SQL99 standard is approximately 2200 pages) and some loss of relational purity as argued for in the third manifesto. The fact that it is a standard means that there will likely be consistency and support by major database management system vendors.

## An Example of Object Definitions

As an example of the flexibility of objects in object-relational database management systems, consider the following definition of a student.

```
CREATE TYPE student AS OBJECT (name VARCHAR2(30), major VARCHAR2(20));
CREATE TABLE student_table OF student;
```

It might appear that if a field contains an object such as student and an object contains several values, it might violate atomicity and the relational model's guaranteed access rule where all values are accessible via a table name, column name and primary key. But this does not pose a problem and in fact is no different than the way relational database systems treat some built-in data types. There are operations that can be performed on strings and operations that can be performed on integers. These operators are not the same and are defined in terms of their operands. Strings and integers therefore behave as encapsulated types: one can only operate on them by means of the operators defined for the type. Objects are also encapsulated types: one can only operate on them in terms of the methods defined for the object. The fact that it is a multi-valued construct is transparent to its use as an attribute just as the fact a string is in some sense a multi-valued attribute of many characters that is transparent to its use as a string value.

In some cases, however, such transparency isn't desired or necessary. The object types in SQL can be viewed in two ways: as single-column values and multi-column values. The student\_table above can be treated as a single-column table in which each row is a student object and only object-defined methods can manipulate it. The student\_table can also be viewed as a multi-column table in which each attribute of the object type student (name and major), occupies a column which allows relational operations.

```
INSERT INTO student_table VALUES ("Chris Lord", "MSIN");
SELECT VALUE(p) FROM student_table p WHERE p.name = "Chris Lord";
```

The first statement inserts a student object into student\_table, treating student\_table as a multi-column table. The second statement treats student\_table as a single-column table, using the VALUE function to return rows as objects.

With the object-oriented, row and column cells can have arbitrarily complex values. This makes database design much more difficult since there are multiple ways of logically representing entities and relationships. Such freedom necessitates new design methodologies, more powerful CASE tools, and the education of database designers.

## Conclusion

Despite a lot of early promise and some successful niche applications, object-oriented database systems failed to capture a significant part of the market, much less supplant relational database management systems. Much of this failure is due to the fact that an object orientation did not address the largest consumers of database management systems, businesses, in which most data could be (and was) organized using the relational model. While object-oriented features were convenient and allowed a more natural way to map entities and relationships from the real world on to a logical representation in a database, they were not mandatory. The convenience and flexibility came at the price of complexity and was insufficient to overcome the inertia of businesses already dependent on (and comfortable with) relational database management systems.

That has changed with the integration of object-oriented features in the SQL99 standard, which still preserves the relational underpinnings and is backwards compatible with existing applications. Major database vendors such as IBM, Oracle and Microsoft are already supporting this standard. This makes object-oriented technology available to the large audience of existing database customers who can phase in adoption of object-oriented features without sacrificing their existing investment in applications, expertise, and of course, the databases themselves.

Both relational and object-oriented database management systems therefore have a limited future and instead will continue on in the combined form of the object-relational database management system.