Vrije Universiteit Brussel
Faculteit Wetenschappen

A Survey of Object Oriented Databases

Serge Demeyer

Techreport vub-prog-tr-92-01

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3525
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

A Survey of Object Oriented Databases

Serge A. Demeyer

May 1992

This paper tries to give an overview of the current object oriented data base (OODB) technology. It is intended for readers that had occasional experience with computer programming, so technical details are avoided whenever possible. Rather, we did try to explain the meaning of certain key-concepts so the reader is able to understand the possibilities and capabilities of the technology. This will be done by sketching the evolution of the 'database' and 'programming language' communities, each of which has led to some important concepts. At the end an overview of various object oriented databases (both commercial systems and research prototypes) is included.

1. Introduction

In [KIM'90a] we found the following definition of an object oriented database: "An object-oriented database system is a database system which directly supports an object-oriented data model." We would reject this statement as a definition (it is not precise enough) but the sentence is a perfectly good explanation of what an object-oriented database is. If one understands what is meant by each the words one might understand better what is meant by "an object-oriented database". To acquire this knowledge, one needs to study the two fields object oriented databases emerged from: programming languages and databases.

Sketching the evolution of these fields will reveal the crucial aspects of object oriented databases, and will explain why the field is so promising. It will also allow to understand some of the difficulties, because -as also stated in [BL/ZD'87]- both fields are culturally biased and this leads to different viewpoints and, sometimes even, misunderstandings.

The following three chapters try to give an overview on the evolution of both communities. Only the aspects important to object oriented data bases are mentioned, so this is by no means a complete evolution of the two fields. In some cases interesting concepts are not covered, and in other cases it is hard to pin-point the exact time some idea emerged. As a consequence of this, priority was given to describe a clear time line instead of giving an accurate sequence of events.

2. The Database Community

The First Generation: Advanced File Access Systems

Very soon in computer history people tried to use their computers to manage large amounts of data. They built systems in order to help them manage all the data on disks and construct useful information out of it. The first generation of database systems was nothing more than a memory manager on disk (the so called *secondary storage*): the database system was a collection of linked lists to be used as an index on other files [SALZ'88]. Sometimes they are referred to as 'Inverted List Systems' [DATE'90].

It was impossible for different programs to access the database at the same time so the tasks that needed to be carried out ran in 'batch mode' (the programs were scheduled by the database system to run one after another and the results were not immediately available). This was recognised as a lack of *concurrency control* mechanisms.

It was also very difficult to organize the data: the database was a large pool of bytes, and the programmer was responsible to connect the pieces of data in the proper way (update indices, …). Once connected there was no support for using the connections to construct information out of the pieces of data. A *database schema* was needed.

The Second and Third Generation: Network and Hierarchical Database Systems

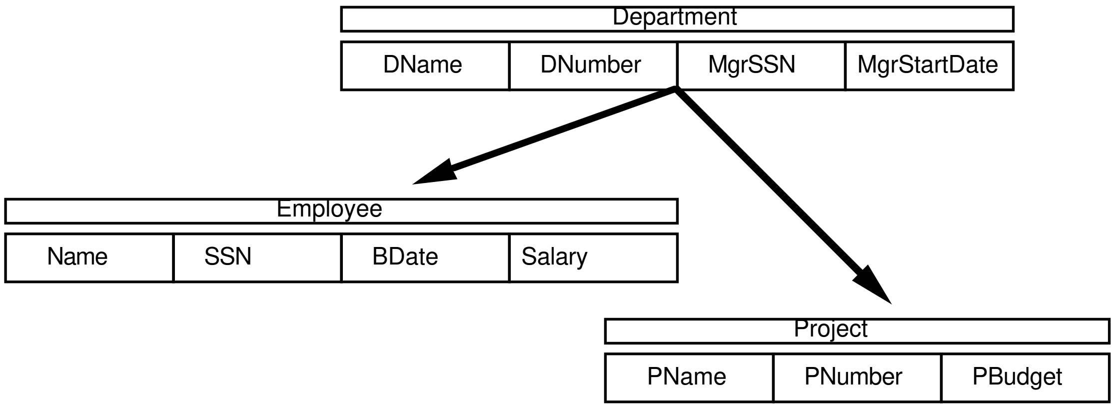


Figure 2.1.a: A hierarchical schema and a population

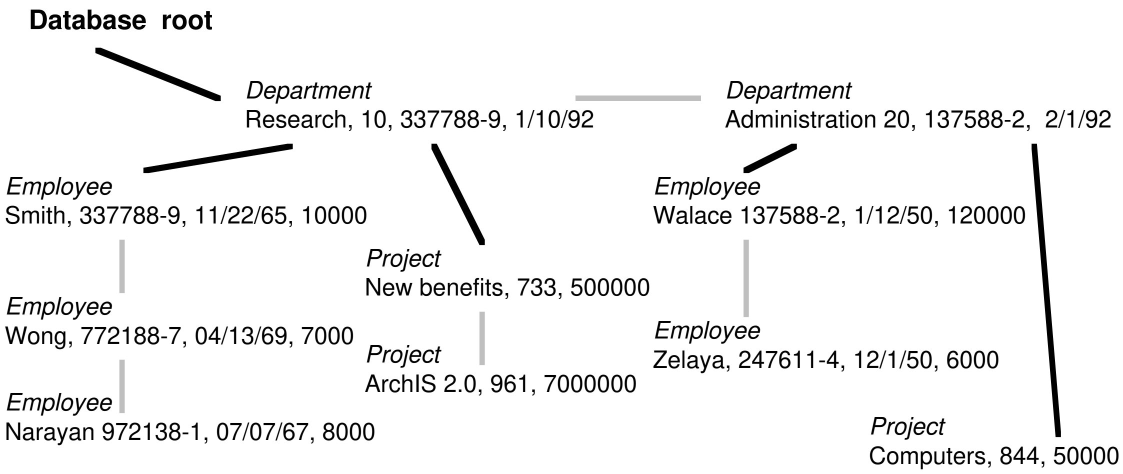


Figure 2.1.b: A hierarchical database population

Figure 2.1.b is a sample population of a hierarchical database. From the root one can find two departments (Research and Administration), each holding a list of employees and projects.
The black lines link records of different types, the gray ones group records of the same type.

Practical experience led to the network and hierarchical database systems [EL/MA'89], [DATE'90]. Information technology of these generations concentrated on various ways of linking pieces of data together (so called records), and traversing the so-built structures to find the needed information. A programmer wrote a database scheme (a framework on how the various records will be interconnected) and the database system would help building and traversing these systems according to the schema.

Those systems were able to give multi-user access to the database by means of a *transaction* manager. Small groups of reads and writes of data for one user were collected into so called transactions in such a way that every successfully completed transaction left the database in a consistent state (an *atomic transaction*). The serialisation theory led to the *two phase locking* protocol, which enables to recognize transactions that can be interleaved (the operations of the different transactions may be mixed). The transaction-manager scheduled transactions that did not interfere with each other to run together, but transactions that needed the same piece of data had to synchronize. The problem of deadlocks (two transactions waiting for each other forever) has various solutions varying from optimistic strategies to pessimistic ones. For a deeper study of transactions we refer to [BE/HA/GO'87] and [ULLM'88].

An important component of transaction management was the possibility to *recover* from system failures (hardware or software). The database must be left in a consistent state, so all of the effects of a transaction must be written into the database or none of them. This must be true, even when certain parts of the system behave exceptionally, or if the contents of main memory are lost after a system crash.

The concurrent access possibility was solved, but another problem emerged: *authorization control*. Not all of the users can manipulate (= read or change) all of the data. Somehow the database must record who can do what with a particular piece of data. It was recognised that this information ca not reside with the piece of data itself, but should also depend on the programs that manipulate it. In second and third generation systems this is very difficult because the database system does not have any control over the systems that manipulate them.

Another disadvantage of such systems was that, although the definition of the database was in one central place (the database schema), the maintenance of this structure was scattered all through the programs that manipulated it. As a consequence of that, programming was hard and tedious, code was duplicated many times, …: some form of *constraint* management was needed.

Furthermore the possible relationships between the data was pre-defined by the means of links. Tailoring the system to different needs was nearly impossible. Scheme management instead of scheme definition and linking of data without predefined links was needed. It was also recognized that having good schemas was of major importance to write database applications, so the need for schema design tools became a research issue.

The Fourth Generation: Relational Systems

The fourth generation of database systems evolved in a completely different way. For the previous systems the data model was formally defined after implementation (by means of the CODASYL-group), but in 1970 the mathematician E.F. Codd published a paper "A Relational Model for Large Shared Data Banks."[CODD'70]. Data was not linked anymore, but was collected in a set of *relations* (named relations are the so called *tables*). Codd defined a mathematically complete set of powerful operations (the relational algebra) on tables that could be used to construct information out of the data that resided in the different tables.

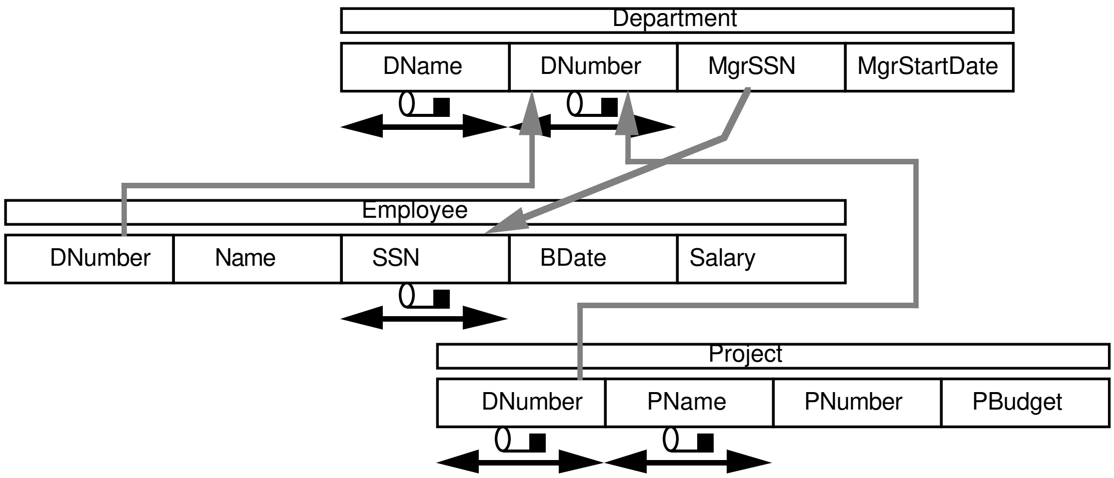


Figure 2.2.a: A relational schema (with key- and foreign key constraints) and population

The black arrows with a key on top of them are key-constraints. Columns holding such constraints are not allowed to hold the same value twice (e.g. in the department table the column DName has a key-constraint, meaning that every department has one unique name).
The gray arrows are foreign keys. A foreign key starting from column A and going to column B means that every value in column A should at least exist in column B (e.g. the foreign key between MgrSSN in the DepartmentTable and SSN in Employee, says that every manager of a department must be an Employee).

Department

DName	DNumber	MgrSSN	MgrStartDate
Research	10	337788-9	1/10/92
Administration	20	137588-2	2/1/92

Employee

DNumber	Name	SSN	BDate	Salary
10	Smith	337788-9	11/22/65	10000
20	Walace	137588-2	1/12/50	120000
10	Wong	772188-7	04/13/69	7000
10	Narayan	972138-1	07/07/67	8000
20	Zelaya	247611-4	12/1/50	6000

Project

DNumber	PName	PNumber	PBudget
10	New benefits	733	500000
10	ArchIS 2.0	961	7000000
20	Computers	844	50000

Figure 2.2.b: A relational database population

A population for a relational database is a set of rows. The database manager ensures that no operation on this population violates the constraints in the corresponding schema.

```
Project(
   select (
      join (Department.DNumber
          = Employee.Dnumber,
      Department, Employee),
   DNumber = 10)
DName, Name)
```

DName	Name
Research	Smith
Research	Wong
Research	Narayan

Figure 2.3: The result of applying some relational operators to the population

The join operation takes a join-expression and two tables. It returns a new relation with a set of columns that is the union of the column-sets of both operands and a set of rows that is a combination of possible rows that matches the join-expression.
The select operation takes a boolean expression and a relation an returns the relation holding the rows that satisfy the boolean expression.
The project operation takes a relation and a set of columns, and returns a new relation holding only these columns.

The simple yet powerful model gave birth to the concept of a *query-language* which makes it possible to write down a *query*. A relational query, when executed, combines the contents of one or more tables into a new relation (that will be thrown away after the results have been processed) according to certain user-defined specifications. A query language thus makes it possible to connect all the pieces of data in the database, as long as the records contain a common (or matchable) data value. Links, occuring in network or hierarchical systems are not needed any more.

Predefined links were replaced by predefined queries, the so called *views*. A view is a named query, which -after definition- acts like any other table, except that it contains no values. Only when a query that uses the view is executed, the actual contents are computed.

This powerful concept was also a solution to the authorization control problem. Only certain users can use named database objects like tables and views. But it is possible to limit the access on certain parts of certain tables by means of a view. This way the right combination between data- and program-authorization control was achieved.

In previous systems one had to program all the steps the machine needed to do to find the information needed, resulting in a *imperative* style of programming. With a query language it is possible to use a *declarative* style of programming: a programmer needs to describe the way the information must look like, and the system is responsible for returning the data that match this description. It is important to recognize that the relational operators (join, project, …) are imperative in nature (the order in which the operations are applied is very important), but that the clean model allows a declarative interface for these operators. Again the mathematical foundations led to a clean design: the declarative nature of the query language is based on the tuple-calculus, which itself emerged from the predicate calculus.

Hierarchical (imperative) query

```
$GET FIRST PATH Department, Employee
  WHERE (Department.DName = "Research");
WHILE DB_STATUS = 0 DO
  BEGIN
    writeln(Department.DName,
      Employee.Name);
    $GET NEXT PATH Department, Employee
      WHERE (Department.DName = "Research");
  END;
```

Equivalent relational (declarative) query

```
SELECT Dname, Name
  FROM Department, Employee
  WHERE (Department.DNumber
    = Employee.Dnumber) AND
    (Department.DName =
                  "Research");
```

Tuple calculus (based on predicate calculus)

```
{d.dname, e.name | ∀ d: ∀ e: Department(d) ^ Employee(e) ^
                  (d.dnumber = e.dnumber) ^ (d.dname = "Research")}
```

Figure 2.4: The different approaches to find information

The hierarchical query uses explicit navigation statements (GET FIRST, GET NEXT)
The relational query is another syntax for the tuple calculus query below. It has the same semantics as the relational algebra expression in figure 2.3.

To avoid misunderstandings, we would like to make clear that the relational algebra is by no means ‘low level’. In the relational algebra (see figure 2.3) one applies high level operations to powerful concepts (relations), so it is high level programming. But since the order in which the operations are applied is essential to get the result, people call it imperative (sometimes the word procedural is used here). On the other hand the tuple-calculus makes use of low level operations (and, or, universal and existential qualifiers) and low level concepts (tuple variables). Only together they form the basis for a high level language, people refer to as declarative. One can prove that the relational algebra and the relational calculus have the same computational power.

Experience has shown that a declarative style of programming leads to more correct and shorter programs that are a lot easier to maintain. The price for these advantages is that such systems need to be smarter, faster, bigger, … . But a lot a work has been done in the area of optimization techniques, varying from clustering techniques (to minimize average cost of accessing records) over intelligent cache management (to exploit knowledge of access patterns to prefetch data) to fast indexing (to get fast access to sets of records which contain certain values). Of course this area benefits from the precise formal model. Manageable relational systems can be built and today, relational database management systems (DBMS) have grabbed the commercial attention. Most new operating systems are even extended with relational technology.

Originally the central idea behind a query language was to make it possible for non-professional users to formulate their needs themselves instead of explaining an information scientist what he wanted, who then wrote a program that would do the job. *SQL* (the ‘standard’ relational query language) did not meet this goal, but research into the area of high-level query languages is still going on. A mainstream idea there is the query-by-example approach.

Despite the fact that a query language was meant for non-professional users, it was recognised that designing a database (deciding what pieces of information should reside in which tables) was a job for professionals. A whole *normalization* theory was developed to support schema designers in this complex task. A designer started from a “universal relation” (a relation that contained all of the possible data-items in the future database), tried to identify redundant information, decomposed the table into smaller ones and repeated the steps on the new tables. The theory guarantees this process to stop with a set of tables that were (to some extent) anomaly free. This theory gave birth to a whole family of software products (called CASE-tools (Computer Aided Software Engineering)) that help people design a database application.

Even with these tools the old adagium (separate the data definition part from the data manipulation part) was still in effect. But in modern relational systems the two are better integrated, and authorized users can change a populated database schema under certain circumstances. Dynamic scheme management was a fact.

In one of his “12 fidelity rules” Codd stated that "the description of the database is represented at the logical level, dynamically like ordinary data, so that authorized users can apply the same relational language to its interrogation" [BYTE'89], making it possible to ask information about the database using the database itself, in this way facilitating constraint management. This idea of a *data-dictionary* is, among others, explored in [DO/KI'87].

The kind of platforms the database systems run on have also changed during the development of commercial relational systems. There is a trend to shift from centralised towards distributed computing.

Originally database systems ran on mainframes and users were sitting behind dumb terminals, all controlled by the same (central) computer. The programs on the mainframe controlled the in- and output on the terminals. This meant that all software (and thus all the data) was centralized, making it relatively easy for a database administrator to manage. But with terminals getting smarter and personal computers being connected to the mainframe, a *client-server* model evolved. The mainframe was now a database server, which was responsible to manage the data and collects requests for data from the network and tried to respond as fast as possible. The personal computers (called clients) were responsible to send the right request to the server, present the data to the user and do some error-checking on data-entry.

The basis of this architecture is the *remote procedure call*. A program is running on the client and at some point in time it needs a service from a central server (for database applications this will be a request for data, but in general it can be a request for any computation). The client sends the request and the necessary information to the server, the server executes the appropriate computation and then control is returned to the client (sometimes with the result of the computation).

Now with the appearance of personal workstations, the trend is towards distributed databases. The difference between clients and servers still exists, but is not as strong anymore: clients have local private databases, multiple servers must cooperate to answer requests for data, … . Today the systems have been adapted to the kind of problems that appear when one leaves the centralized model of computation, but commercial realisations are hard to find. Mostly they are based on the notion of *location transparency*, meaning that users that interact with the data are unaware of the place it resides. *Repositories* are kept to keep track of the migration paths, pending updates and distant copies of the data.

Even when the relational solution has been adapted to a major evolution in computer architectures, there are still applications where it has been recognized that the relational model is not adequate to be used as the basic building technology. Those applications come from the domains of CAD/CAM, CAE, document management systems. Such applications (the list is exclusive nor exhaustive) manage very heterogeneous sets of data with very complex relationships between them in an cooperative environment.

A major critique is that the model itself is too simple. It starts with a limited set of basic datatypes (numbers and strings), each with its own limited set of operations. There should be more support for storage and retrieval of arbitrary long data; more possibilities for retrieving the data in smaller segments and other techniques (full text indexing) that support the manipulation of such data. Also there are no aggregation facilities (possibilities to combine heterogeneous datavalues into one item), except relations themselves. Nested relations are not allowed in the basic model, and extensions that allow columns to hold complex values (like sets, relations, …) are not well understood.

So relations can only hold lexical representations of real world entities. The value-based manipulation of records means that one must always peek inside before they can be used. The “everything is a table and nothing but a table”-idea makes it hard to abstract from the underlying implementation. The concept of *primary* and *foreign keys* together with the view concept is a way to get around this weak abstraction facility, but it needs more (schema)support than todays relational systems can offer.

Relations have a rigid structure: a relation has a fixed set of columns, each of which can hold a fixed datatype. In systems with many different heterogeneous types of records it is almost impossible to manage the complexity of the database schema. A completely normalized set of tables for such applications gets too big, and -by consequence of that- too slow: to pick up all the data-values in order to build a complex object spread over many tables, many joins need to be computed.

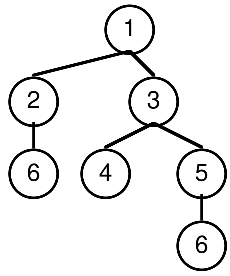
Relational query languages are all based on the first order propositional logic (see [ULLM'88]). This implies that relational systems are not computationally complete, meaning that not everything that can be expressed in traditional programming languages can be expressed in relational query languages. The solution to this problem is to embed query languages in programming languages (by means of precompilers, libraries, …), but this approach leads to what has been recognised as the *impedance mismatch*. The set oriented approach of query languages conflicts with the “one record at a time approach” of traditional programming languages resulting in a hard to use *cursor* mechanism.

PARTS

PART-NR	NAME
1	Car
2	Motor
3	Wheel
4	Tyre
5	Wheel II
6	Screw

PART-OF

MAJOR-P	MINOR-P
1	2
1	3
2	6
3	4
3	5
5	6



All the subparts from 'car'

```
SELECT minp.Name
FROM parts majp, parts minp, part-of p1
WHERE (majp.Name = 'Car') AND (majp.part-nr = p1.major-p)
  AND (p1.minor-p = minp.part-nr)
```

All the subsubparts from 'car'

```
SELECT minp.Name
FROM parts majp, parts minp, part-of p1, part-of p2
WHERE (majp.Name = 'Car') AND (majp.part-nr = p1.major-p)
  AND (p1.minor-p = p2.major-p) AND (p2.minor-p = minp.part-nr)
```

Figure 2.5: impossibility to compute transitive closures with relational query languages

One important consequence of the computationally incompleteness of a relational query language is the impossibility to compute transitive closures of relations. In figure 2.5 we find two relations: the PARTS relation that relates every part number (primary key) to a name. The PART-OF relations relates a MAJOR-PART-NO to a MINOR-PART-NO; every tuple in this relation states that the part with number MAJOR-PART-NO contains at least one part with number MINOR-PART-NO. As illustrated it is very easy to ask the subcomponents of some major part, or the subsubcomponents, or the subsubsubcomponents: it does not matter how deeply nested, as long as the level of nesting is known in advance. But the transitive closure of a relation is joining the relation with itself until it does not change anymore, thus without knowing in advance how many times this must be done ! This means that it is very hard to manage so called composite objects, the building blocks of CAE-systems. Some SQL-extensions provide a solution to this problem (the CONNECT BY clause in Oracle [ORAC'90]), but they are not part of the SQL-standard.

The concurrent access model (based on transactions) of current relational systems is created for applications that run independently. It is based on short term transactions, involving few tables, having short duration times (throughput of hundreds of transactions per second are routinely measured in vendor bench-marks) and always leaving the database in a consistent state. In cooperative environments there is a need for long term transactions manipulating complex and deeply nested structures, measuring duration times in hours or weeks and traversing many intermediate inconsistent states. These kind of transactions require database systems to have more knowledge on what is happening in the applications that use them. One of the promising ideas to solve this problem is *version* management .

The constraint definition in relational systems is part of the data-definition part and thus static in nature. *Rules* and *triggers* are seen as possible answers to dynamic constraint definition.

3. The Programming Language Community

This section begins with the explanation of some widely accepted concepts, and therefore some of this text may be viewed as overdone. Nevertheless this paper was intended for people with occasional programming experience, so we decided to spend a few words on such basic knowledge.

Sketching the evolution of programming languages is even harder than describing how database models evolved: the activities in the field are more diverse in nature and the goals to meet differed a lot. One can speak of ‘generations’ of database systems as opposed to programming languages which are divided into classes (functional, imperative, declarative, …). We refer to [MACL'87] and [MASal'91] for a completer historical evolution and concentrate on how ‘abstraction’ evolved in various programming languages.

Indeed this is the driving force behind the programming language community. The need for reliable and maintainable software makes abstraction facilities in program design and construction a crucial factor. We also believe that ‘abstraction’ is what the database community was looking for when they first made contact with the object oriented world.

Control Flow Abstraction Leads to Procedural Abstraction

Program sources are static descriptions of dynamic processes manipulating data. To understand what happens when such a process ‘runs’ it is very important to know when each part of the process passes control to another part. This means that a programmer must be able to read in the program source where and when the control is transferred. This is what is called the control flow of a program. Control flow abstraction means that it is possible to concentrate on certain aspects of the overall process (and abstract from others).

A big step forward was taken by the introduction of the so called *block structured* languages, with its first member the Algol-60 language (later Pascal, Modula-2, C, Ada, … joined the family). Before the invention of Algol, the control flow of programs was defined by ‘*gotos*’. From every part of the program, control could be transferred to every other part (the metaphor of spaghetti was used), making it very hard to read and understand program texts. People agreed that the static structure of the program should correspond in a simple way to its dynamic behaviour.

Block structured languages ensure that programmers divide their programs into hierarchically structured blocks. Such a nested block has one entry and one exit point, making it easier to trace the control flow and allowing to zoom in on certain parts. Each block is introduced by a control statement, all having a clear syntax constructed with a limited set of key-words, this way improving the readability of programs.

An important principle in writing programs is that it should be avoided to state something more than once; recurring patterns should be factored out (the abstraction principle). According to this principle, it was recognised that it should be possible to treat certain blocks as separate entities. This way duplication of blocks could be avoided as one block could be used in several places in the program. Therefore the concept of a named block (a *subroutine* or *procedure*) was introduced and a syntactical construct (in most languages it is called a procedure definition) was invented to bind a name to a block of code. Another construct (the procedure call) defined how a programmer can write the transfer of control from a block to a named block.

It was also recognised that subroutines must have parameters. As blocks are static descriptions of (sub)processes manipulating data, blocks must be aware of what data they should work on. But since named blocks can be called from different places, they should work on different data depending on the caller. So a calling block must pass to the subroutine the data it should work on. This is done by means of parameter passing. On the other side the subroutines must specify what data they expect to work on to ensure that callers pass the right kind of data. In other words: the interface for a procedure must be declared so that both caller and callee know what they can expect from each other.

With this tools, *procedural abstraction* was seen as a good way to organize things. A program should be constructed out of black boxes: from the outside it is impossible to see what is happening inside. All a programmer knows is that if such a black box is called with the right kind of values, eventually it will return with some other values of certain kinds. This results in information hiding, which has the advantage that small changes only have local effects. Software maintenance became easier and team writing became possible as each team member could work on his own black box.

Types and Values: the Declarative Parts of Programs

As already stated, programs are static descriptions of dynamic processes manipulating data. We will concentrate on the manipulation of data now.

When programs want to use a value, they must allocate an area of memory of specified size to hold it. Memory is divided into bits so an interpretation for a group of bits (as a number, a character, …) must be defined. A memory location is identified by an address.

Since the available memory is limited, storage must be reclaimed whenever possible. Therefore programs store different values in the same memory locations. Since managing memory locations is hard, early programming languages (like Fortran) tried to work around memory addresses whenever possible by means of the concept of a *variable*. A variable is a named memory location large enough to hold all future values of a certain maximum size with a known interpretation of the bits. The maximum size and bit-interpretation of a variable where concepts that got unified in the concept of a *type*.

Beside the basic types (numbers, characters, …) it was seen that programmers needed ways to combine existing types into larger blocks. There are two major type construction possibilities to be used in conjunction: the combination of different kinds of things into one (e.g. records) and the combination of the same kind of things into one (e.g. arrays, lists). This constructor restricts the way the values in the type may be accessed (by name, by index, by position).

Programs were divided in declaration and implementation parts. The implementation parts define the control flow of the process, while the declaration parts declare the variables and types the process may use. Niklaus Wirth summarized this principle well in the title of his book "Algorithms + Data Structures = Programs" [WIRT'76].

As stated in [DA/TO'88] "Types in programming languages are generally used and thought of as means of characterizing values that arise dynamically in the course of computation". From this perspective the concept of a type evolved from an abstraction facility (abstract from memory size and bit-interpretation) and compiler utility towards a constraint mechanism. Variables of certain types can only hold certain kinds of values, and may participate in a limited set of operations (ex. characters ca not be multiplied).

This leads to the notion of *strong typing*. Languages were designed to support strong typing, stating that only meaningful operations can be applied to the data. The operations built in the language where defined on the certain basic types and it was an error to apply an operation to types it was not defined for. When programmers define their own operations (by means of subroutines) they must declare what operators are allowed and it is an error if a subroutine is called with the wrong parameters. This declarative approach leads to more safety for programmer because the compiler can check the correctness of the use of operations. The compiler can also perform certain optimizations to produce fast (machine) code.

In most languages strong typing was recognised as a good thing, but in some cases type constraints are seen as a problem. Most strongly typed languages provide an explicit loophole in their typing system by means of a *type coercion* operation, always documented as an operator that should be used with extreme care.

Other languages explicitly deny the strong typing principle, stating that it limits the freedom of programmers and that its hard to use for prototyping and fast development of test programs. In Common Lisp for example, type declarations are allowed but not necessary. When a variable or parameter is declared to be of certain type, it is an error to assign it an illegal value, but the compiler is not recommended to check it. Type declarations are there to increase the readability of programs and to allow compiler optimizations, but the same program with or without declarations must produce the same results.

Declaring variables allows a compiler to allocate memory resources, but not all of the memory needs are known in advance (= at compile time). So a memory manager, responsible for managing run-time memory resources, is a necessary tool in software development. A program asks the memory manager for a certain amount of memory space, and should release it as soon as possible (allowing other parts of the program to reuse the same space). It is generally accepted that it is hard to write correct programs with explicit memory management and this led to memory managers with *garbage collectors*. When a programs start to run, all variables are empty. As values are assigned to them, memory locations are allocated and assigned to variables. Sometimes a value is larger than the current memory location assigned to a variable, and then a new memory location is allocated and the old one is lost. The process of collecting these lost memory cells is called garbage collection. Basically garbage collection algorithms are based on the idea that all the memory locations reachable from some root (the variables) are in use, and all the other ones are free and need to be collected for reuse.

One factor of great importance in the strong typing debate is the concept of *overloading* (more generally called polymorphism). There are many cases in which the same kind of operation is applicable to different types (e.g. the + (plus) operation is defined on reals and integers, …). Programmers want to use the same symbol for these slightly different operations, but this conflicts with the strong typing principle. Most strongly typed languages work around this problem for their build in operations (the compiler knows the type of the operators and produces different code depending on the case) but it remains a problem for operations defined by programmers.

Abstract Data Types

Practical use of subroutines soon led to the viewpoint that a procedure should not work on its own. Since a procedure has a well defined interface for the data it works on, several procedures can be combined to define a clean interface on how certain kinds of data can interact with each other.

On the other hand, the static nature of types as constraints did not satisfy programmers. Using types allowed compilers to check for errors, but more support for declaring run-time checks was needed to allow for better constraints.

Together this led to the notion of abstract data types. An abstract data type (*ADT*) is a set of values (a type) only accessible by the operations defined on it. A user of an ADT need not know how a concept is implemented, all he needs to know is its interface. As this interface is known to be the only way to access what's inside, defining a secure ADT is possible.

Using an ADT-style of programming can be done in any programming language with a procedure call mechanism, as long as programmers obey certain protocols. Of course special languages are developed to enforce such protocols, among others CLU [LISal'77] and FOOPS [GO/WO'90]. In the paper "Type theories and Object-Oriented programming" [DA/TO'88] an overview of different ADT approaches is given. In such special languages ADTs can be viewed as the way to ‘declarative programming’. A type is defined together with a set of applicable operations, and results of the interaction of operations (e.g. operation A after operation B produces the same result as operation B and operation C). The compiler should be able to infer an implementation from these specifications, but sometimes programmers can give hints.

Readers will have noticed that the interface of an ADT is very important. According to the information hiding principle, it is used to hide crucial implementation decisions allowing these to be changed afterwards without affecting the rest of the program.

Object Orientation: Active Abstract Data Types

The idea of an internal state results in yet another view on the abstraction facilities of a language. Incorporate the data and the procedures manipulating it into one entity, namely an *object*, results in a *object-oriented* style of programming. Data participates more actively in programs; this style is sometimes referred to as data-driven programming. According to P. Wegner in [WEGN'90] "Objects are collections of operations that share a state. The operations determine the messages (calls) to which the object can respond, while the shared state is hidden from the world and is accessible only to the objects operations. Variables representing the internal state of an object are called *instance variables* and its operations are called *methods*".

Programmers using objects are only interested in the behaviour of the object, not on how this behaviour is accomplished (defined in the methods). Object oriented programming consists of defining and creating a set of objects and asking them to do certain things by sending them *messages*. Objects are free to answer this message (by sending messages to other objects) or passing it to another object (depending on the definition of an object and its methods) or a combination of the two.

Note that the message-passing paradigm is a beautiful solution for the problem of overloading. Since an object itself decides whether it will answer a message, it is perfectly valid for two different objects to answer the same message. The methods in both receivers may differ internally, but the interface on the outside will look the same.

It is important to know that ADTs and objects are not the same concepts, however it is generally agreed upon that object oriented programming languages are well suited to implement ADTs. As argued in [COOK'91] "The basic distinction is that object-oriented programming achieves data abstraction by the use of procedural abstraction, while abstract data types depend upon type abstraction. Object-oriented programming and abstract data types can also be viewed as complementary implementation techniques: objects are centred around the constructors of a data abstraction, while abstract data types are organized around the operations. [...] This characterization is not completely strict, in that the type of a procedural data value (= an object) can be viewed as being partially abstract, because not all of the interface may be known; in addition, abstract data types rely upon procedural abstraction for the definition of their operations".

Again, like with ADTs, any language that allows to store procedures in variables can be used in object-oriented style, but special languages exist that offer mechanisms for direct use of objects. There are many languages that call themselves object-oriented, and as a consequence of that it is hard to define what an object oriented language really is. This is also noted in [DA/TO'88]: "In the case of object-oriented programming, this metaphor is rarely introduced with the mathematical precision available to the functional or logic programming models. Rather, OOP is generally expressed in philosophical terms, resulting in a natural proliferation of opinions concerning what object oriented programming really is". In most cases object orientation is divided in a set of features, and some are considered more important than others. We will try to explain some of these features now, and then return to the question of what object orientation really is.

Encapsulation

The fundamental idea behind object orientation is *encapsulation*. "Encapsulation is a powerful system structuring technique in which a system is made up of a collection of modules, each accessible through a well -defined interface." [ZD/MA'90]. In object oriented systems data and procedures are incorporated into one entity (the object) and accessed through a known external interface. This way a program is build from stand-alone entities that are encapsulated from each-other.

Classes

Applying the abstraction principle (avoid to state something more than once; recurring patterns should be factored out) to objects results in the concept of a *class*. "A class is the description of a family of objects having the same structure and behaviour. It describes a set of data and a set of procedures or functions". [MASal'91] Then every object is an *instance* of one unique class. A class thus has two components, behaviour and structure, and so the definition of a class must describe these two components. The structural component is generally called instance variables and used to hold the state of the object. As different instances of the same class should be able to represent different states, the structural component of a class is not shared between its instances. The behaviour of a class is collected in a set of methods (the method dictionary), and this dictionary is shared between the instances of a class.

Other techniques to group objects are used as well, but they have not received much success. Today the three languages that seem to survive (Smalltalk, C++ and CLOS; more and more people want to include Eiffel in this list) are all class-based. Nevertheless it is interesting to take a look at those alternatives, but before we must study some other important concepts. In the section “Frames, actors and prototypes” we will return to this matter.

Inheritance

It is generally recognised a good thing when programming systems support the reuse of code, and one way to accomplish this is by means of inheritance. When a message is sent to an object the system (possibly the object itself) is able to decide when it is not capable to answer a message and pass it to another object.

In class-based languages this is done by introducing *subclasses*. When defining a new class one can decide to define it as a subclass of another class. This way, the subclass will have the structure and behaviour of its superclass. To reflect the fact that a subclass is different from its superclass extra instance variables and/or extra operations can be defined, existing operations can be redefined (overridden) and so on.

By stating that a class can have several superclasses, *multiple inheritance* is introduced. This results in a few problems (mainly what should be done when different superclasses define the same instance variable or operation) that can easily be resolved. It is generally recognized that multiple inheritance introduces additional complexity, but that the additional modelling power justifies it.

In some ways, classes resemble types, but they should be viewed as different concepts. As stated in [WEGN'90] types are specified by predicates and classes are specified by templates used for generating and managing objects with uniform properties and behaviour. Types determine a type-checking interface for compilers, while classes determine a run time interface for programmers. With every class a type is associated, defined by the predicate that checks its template. In CLOS this is reflected in the principle that every class (in fact it is with every class that has a proper name) is associated with a type but some types are not associated with classes [STEE'90], [BOBal'88]. In Smalltalk the notion of a type does not exist, but the class of an instance may be checked so type-checking can be simulated.

Metaclasses

In class-based object-oriented languages, classes are used as templates for the creation of instances. In many of these languages, classes are somehow an active component: objects are created by asking a class to return an instance. This is interpreted as a class being an object which understands at least a message 'make-instance'. If a class is an object, then it is an instance of a class, and this class is called the metaclass. The introduction of a metaclass gives rise to a uniform object-oriented data-model, but it introduces the problem of a possibly infinite chain of metaclasses.

Metaclasses are powerful tools, and are especially useful to combine object oriented systems with databases. The paper "PCLOS: stress testing CLOS" [PAEP'90] gives an interesting viewpoint concerning this combination of fields. "Ideally we would have our systems be both portable and flexible at the same time. Unfortunately, these goals are generally in conflict with each other. Designing for portability leads us to constraining systems to ensure the ability of mapping onto the weakest targets. The final stage of this tendency is usually standardization. This is contrasted by the tendencies induced by the flexibility goal. This goal will tend to push the design towards expansion, towards open, highly customizable systems. Instead of standardization, the goal of flexibility has a randomizing effect.But a reconciliation of the conflicting design goals of flexibility and portability can be achieved. The solution is based on the realization that if the mechanisms for changing a portable system are themselves part of that system, then the idiosyncrasies of disparate design instantiations will port along with the system."

That's why the CLOS Meta-object protocol mechanism defines the internal static structure of the language, as well as the dynamic, run-time behaviour. System programmers may adapt the language by subclassing meta-object classes and by selectively shadowing methods that operate on their instances. In PCLOS this mechanism is used to implement objects that persist the Lisp session that created them by mapping CLOS objects to information in a database. Access to instance variables are intercepted so that values can be retrieved from or written to the database.

Frames, Actors and Prototypes

Frames were introduced by Marvin Minsky as a tool for knowledge representation and have known a major evolution since. It was claimed that predicate calculus was not suited for knowledge representation since things as default values, exceptions, incomplete and redundant information is hard to represent with logic. The formalism is based on the dynamic reuse of structures (= frames) which represent patterns of situations. A frame is a hierarchical network of nodes, the upper ones representing typical objects and the lower ones are more specific. Nodes have slots which can hold values (which can, but need not be, frames) or can be empty. When empty, slots inherit default values from higher nodes. Frame-based languages introduced the notion of a trigger (sometimes called a reflex or a daemon (not to be confused with the Unix daemons)): a procedure that gets executed just before or just after a slot is accessed. A powerful concept, especially when we think about constraint checking or handling exceptional values.

Normally, inheritance is seen as static: the object being inherited from remains the same. Opposed to that one can define dynamic inheritance or delegation. The major proponent of this class of systems are actor systems, based on a concurrent data model. Systems are built from autonomous agents (called actors) knowing some other actors (called acquaintances) and having a script which describes the behaviour of an actor. The set of messages understood by an actor is called its intention. When receiving a message the script filters it and activates the appropriate part, which may send messages to the acquaintances. Actors are not organised hierarchically and actors do not have classes. An actor can create another one by duplicating itself. The clone is able to change his acquaintances and his script, this way becoming a completely new kind of object.

Another way to see the difference between inheritance and delegation is to look at the granularity of sharing. Inheritance is then viewed as sharing at the level of classes, and delegation is sharing at the level of objects. The delegation in actor systems is organized by means of a graph. Systems which enforce delegation in a tree-structure are called *prototype* systems. Prototype systems do not know classes: one object (the prototype) is chosen as the representative of the set of objects that share structure and behaviour. Note that a prototype can have several descendants, but that an object can have but one prototype.

Object Identity

An integral part of an object oriented data model is the ability to make references through an *object identity*. This means that every object has some value that remains invariant through all possible modifications of the objects internal state. This object identifier (OID) is unique for all objects in the system so it can be used as an identification. Testing for equality can be done in two ways: comparing the OIDs (identity test) or comparing the internal state (equivalence test).

An OID is not the same as a variable name: one object can have many names (aliases) in one program, for example when it is passed as a parameter. An OID is also distinct from the address of the memory location where the object resides: the memory address is an external, accidental attribute while its identity is an internal essential attribute. It is not a primary key (database concept) either: a primary key is unique within a single relation while an OID is unique across the universe.

The notion of an OID makes it very easy to create shared structures. Two objects x and y can share state by holding a reference to a third object z responsible for managing the shared state. It is also very easy to build complex objects out of smaller ones: the former holds references to the latter and the encapsulation will ensure that the behaviour of the smaller objects is invisible on the outside of the top-level complex object.

4. When Two Worlds Collide

Some of the statements in this section are based upon the "Issues in the design of object-oriented Database programming languages" paper by Bloom, T and Zdonik, S. B. [BL/ZD'87].

The Database Culture

The database culture is such that it tries to centralize data and information on the data, while the functionality is separated in well-defined compartments. A DBMS should be viewed as a separate entity in the whole picture, sometimes even a stand alone application. In a way a DBMS behaves more like an operating system than a program development tool.

The power lies with the data rather than with the applications: constraints are used to put semantics in the database. Data-independence is important: applications should rely only on the database schema and the DBMS (instructed by the database administrator) is free to choose the representation. The main symptom for this data-orientation is the fact that the data base is designed before programs are written. This means that a database schema should be designed with the needs for all present and future applications perfectly balanced. As databases are designed to manipulate large amounts of heterogeneous data, the data structuring facilities are simple and power comes from uniformity.

The various tools that work with the data are meant to be used in a non-integrated fashion. The only communication channel between the tools is the database itself. Each tool is to be used for its own special purpose: scheme designers are tools for the programmer to think about the problem, the DDL (data definition language) defines the database structures, the DML (data manipulation language) manipulates the data in those structures, report generators summarize the information, … . The division between the DDL and DML part of a DBMS is seen as the major way to achieve a clean data base design.

The query language is the only way to get data in and out of the DBMS. Therefore optimization techniques concentrate on handling large amounts of data on slow secondary storage media, independent of the way the data will be used. This results in well understood technology for query optimization that benefits from the formal grounds of database models. Still it fails to meet application requirements since the query language has no notion of the surrounding applications, this way excluding techniques like data flow analysis.

Database research is focused on the notion of persistent data. In [SI/ST/UL'91] persistence is defined as "the maintenance of data over long periods of time, independent of any programs that access the data". It must be possible to extract data from the database, even when explicit links to the data do not exist anymore. As a consequence, data should be removed from the database explicitly.

Close to persistence is the notion of simultaneous data-access. In multi-user environments databases ensure that processes run independently. Different processes can share data-items as long as they do not interfere with each other. So the data access model is based on independent processes that share data, not on cooperative processes that communicate through data.

Since the introduction of the relational model, the database research community is devoted to a formal approach. Indeed the mathematical foundations for the model, the query language and the normalisation theory resulted in some well understood solutions for old problems. New approaches without such formal grounds are not easily accepted in the database community. Moreover this resulted in a clear definition on what is relational and what is not.

The Programming Language Culture

In the programming language community, integration is the key concept. Functionality and data is grouped in modules, several modules can be combined in larger ones until finally a root module (the application itself) evolves.

The power and semantics of such modules come from the combination of operations and data. Programs are constructed step by step and the responsibility over the data is distributed over the modules. The structuring and combination of data is easy and well integrated with the operations that act upon the data. Clean programs come from clean and well defined interfaces for the different modules.

As data access is already fast, optimization techniques must concentrate on the interaction between data and operations. Data flow analysis is the solution but since the number of possibilities are so enormous, serious optimization is almost impossible. Peephole optimization (optimizations are local on small pieces of programs) is the only technique that is well understood. Data is transient (= does not survive the process that created it) so techniques like garbage-collection are necessary. It is not possible to share data with other programs.

The programming language community is less devoted to formal techniques. As a consequence, there are no clear definitions on what is object-oriented and what is not.

Adding Persistence to Programming Languages

Building complex systems (like computer aided engineering (CAE), computer aided design/computer aided manufacturing (CAD/CAM) applications, multimedia information systems, …) revealed the shortcomings of both the fields. Programming languages lacked the notion of persistence, and the ability to share data. Database management systems needed things like dynamic constraint facilities, type systems, data-abstraction, powerful query-languages, stronger and more concurrency facilities. Moreover, the possibilities of both hardware and software (personal computers & workstations, better networking facilities) tends towards “distributed computing”. Working together was a logical step.

This “joining of forces” idea was recognized for a certain period of time. Capturing the power of the relational model in traditional programming languages was a research issue and the rationale behind it was very simple.

The relational model is powerful because of its simplicity: all manipulable objects are relations, and all relational operations return relations. It is not Turing equivalent (not everything is computable) but it is very expressive (with a few lines very complex operations can be defined and executed). On the other hand, traditional programming languages are computationally complete, but miss the expressivity of query languages like SQL.

Therefore, several experiments to encapsulate the relational model into traditional languages have been undertaken. On the academic side with extensions like Pascal/R and PS-algol (see [AT/BU/MO'88] and [HUGH'91] as well as on the commercial side (precompilers for Cobol, C, …). All approaches have problems with what has been recognised as the impedance mismatch.

The impedance mismatch problem has been reported on two levels. The first, and easiest to resolve, is at the type-level. Connecting two systems with different type systems means that values from one type must be translated into values of the other. What must be done when the ranges of corresponding types do not match and what must be done when a type in one system has no corresponding type in the other are questions that must be answered. Solutions are easy to find, but result in large amounts of code that basically add nothing to the functionality of the total system. A typical example of this mismatch can be found in [SM/ZD'87], where very long and variable length strings must be translated in the fixed-length strings of the underlying database. The proposed solution resulted in an extra table where small pieces of the string where stored, and a very complex operation to get the contents out of the database.

The second level of impedance mismatch is more fundamental in nature. The paradigm mismatch is the problem of converting two different programming paradigms. Traditional programming languages are not very well suited to a many-record-at-a-time approach and a run-time definition of datastructures. This means that it is not easy to represent the concept of a relation or query in languages like Pascal or C. The solution has been found in the concept of a ‘cursor’ (standard ANSI-SQL) which holds one (current) row in a relation and can be moved through all the rows in the query-result. This approach has several drawbacks: one must predefine the fields of the query-result (this way returning to the predefined links of the hierarchical model); one must explicitly enumerate the query-result (going back to the procedural style); … . The elegant simplicity of the relational model is lost, and with it the original beauty.

Object Oriented Databases: Object Orientation + Persistence

But since modelling facilities of object oriented languages are better than those of classical imperative languages, it was worth the effort of trying to capture database semantics in those languages. G-Base and Gemstone both added persistence to object oriented environments (G-Base started with frames, Gemstone with a Smalltalk like data model) and this idea has shown its benefits.

Various other experiments had been conducted (Encore/ObServer, Orion, IRIS, Statice, VBase, O2, …) leading to better understanding, new insights, other application domains, … . Soon everybody realised that this field was a very promising one, and the need for standardization became more and more apparent. But in the object-oriented world there is no such thing as an standard object oriented data model, so it is impossible to specify what constitutes a true object-oriented database. Finally a group of people working in the field (Atkinson, M / Bancilhon, F. / DeWitt, D. / Kittrick, K. / Maier, D. / Zdonik, S.), decided to define a collection of mandatory features. They published their ideas in a paper called "The Object-Oriented Database System Manifesto" [ATKal'89]. Figure 4.1. lists the “13 commandments”.

1.	*Thou shalt support complex objects*
2.	*Thou shalt support object identity*
3.	*Thou shalt encapsulate objects*
4.	*Thou shalt support either types or classes*
5.	*Thine classes or types shall inherit from their ancestors*
6.	*Thou shalt not bind prematurely*
7.	*Thou shalt be computationally complete*
8.	*Thou shalt be extensible*
9.	*Thou shalt remember thy data (persistence)*
10.	*Thou shalt manage very large databases*
11.	*Thou shalt accept concurrent users*
12.	*Thou shalt recover from hardware and software failures*
13.	*Thou shalt have a simple way of querying data*

Figure 4.1: Mandatory properties as proposed in the Object-Oriented Database System Manifesto

Their are other propositions for defining object-oriented databases. In the paper "Object-Oriented Databases: Definition and research directions" by Kim, W. [KIM'90a] a core object-oriented data model is presented and any database system that supports such a model is called an object-oriented data base. The core model is based on four components:

- object and object identifier
- attributes and methods (state and behaviour)
- class
- class hierarchy and inheritance.

In the book "Introduction to object-oriented databases" Kim writes "A data model is a logical organization of real-world objects (entities), constraints on them, and relationships among objects. A data model that captures object-oriented concepts is an object-oriented data-model. An object-oriented database is a collection of objects whose behaviour and state, and the relationships are defined in accordance with an object-oriented data model. An object oriented database system is a database system which allows the definition and manipulation of an object-oriented database" [KIM'90b].

In the introduction to "Readings in object-oriented database systems" Zdonik and Maier define a threshold model, but any real system is free to include other features. According to them

"an object-oriented database must, at a minimum, satisfy the following requirements:

1. It must provide database functionality […]
2. It must support object identity
3. It must provide encapsulation. This encapsulation should be the basis on which all abstract objects are defined
4. It must support objects with complex state. The state of an object may refer to other objects, which in turn may have incoming references from elsewhere." [ZD/MA'90]

Database functionality means at least the following features:

1. Having a (non-trivial) model and language
2. Able to represent relationships between entities
3. Having a permanent data store (= persistent + stable)
4. The possibility to share data (simultaneous access by multiple users)
5. Arbitrary size

A a list of possible extensions of the threshold model is included, and some of these extensions are grouped in a 'reference model'.

At the database level:

- Integrity constraints (static (DDL) + dynamic (triggers))
- Authorization control
- querying (may be declarative but should at least allow associative access to data)
- indexing
- separate schema (meta-information) and dynamic scheme management
- views (virtual data computed from stored data)
- database administration (reorganizing the logical or physical structure of data; gathering statistics for tuning performance; auditing for security and accounting; user administration; archiving; …)
- Distribution
- Report and form management
- Data-dictionary
- Database name spaces
- Active data (changes to one item may cause changes to other items in the database) and active databases (changes to the database can cause messages to the application)
- Derived data: some fields are declared as being derived from others by executing certain methods. Views are one form of derived data.
- Rules

At the data model level:

- Inheritance (Not in their mandatory list !!)
- Typing (may be parameterized)
- Polymorphism
- Hierarchies (three possible hierarchies are identified: a type hierarchy (specification), an implementation hierarchy (representation and methods) and a classification hierarchy
- Collections
- Versions

The Next Generation: Semantic Database Systems

Since relational systems pushed all other systems off the market, every database vendor is afraid to miss the next database-train. In the article "Relational vs. Object-oriented" Herb Eidelstein [EDEL'91] explains why. (After an introduction where Dr. Codd and his model are presented) "The rest is history. No matter what kind of DBMS a system was, it had to call itself relational in order to survive. The largest independent DBMS vendor of the day, Cullinet, spent tens of thousands of dollars on fruitless technical tricks and brochure engineering on its IDMS/R to no avail, and eventually the company was acquired at a bargain-basement price by Computer Associates"

The relational response to the object oriented database was formulated at the "Object-Oriented Databases" conference organized by the IFIP Working group 2.6 D.S.4 in July 1990 by the invited speaker Michael Stonebraker (Ingres), who presented the "Third-Generation Database system manifesto". The committee for Advanced DBMS function (Stonebraker, M. / Rowe, L. / Lindsay, B. / Gray, J. / Carey, M. / Brodie, M. / Bernstein, P. / Beech, D.) published this paper also in the ACM SIGMOD Record, Sept, 1990 [STOal'90].

Basically, object oriented database systems were seen as part of the larger next generation of database management systems (actually they called it the third generation, counting network and hierarchical system as the first generation and relational systems as the second). Nevertheless there are some remarkable differences between the two papers. Most eyecatching is the importance of the query language. "Essentially all programatic access to a database should be through a non-procedural, high-level access language (proposition 2.1)" and "Queries and their resulting answer should be the lowest level of communication between a client and a server" which is completely different than saying "Thou shalt have a simple way of querying data". Moreover the third generation manifesto states that "for better or worse, SQL is intergalactic dataspeak".

Another difference is the position of the object-identifier: third generation systems should assign unique identifiers to records only if a user-defined primary key is not available. Also rules (triggers and constraints) are seen as a major future feature in next generation systems and updatable views (virtual data-collections) are essential.

The scope of this next generation of database systems is much wider than the CAE-applications object oriented databases were invented for. In the special issue of the Communications of the ACM on next generation DBMS, the article "Database systems: achievements and opportunities" [SI/ST/UL'91] thinks about image databases, CAD-systems for skyscrapers (with hundreds of subcontractors), …. Problems and solutions will meet in new kinds of data (images ≠ bitmaps), rule processing, new data models (spatial data, temporal data, uncertainty, …), new algorithms (scaling up is impossible), parallelism, tertiary storage (archives), other transaction protocols (collaborative databases), object versions and maintenance of consistent configurations, heterogeneous and distributed databases, incomplete and inconsistent databases, …

5. Some Object Oriented Database Management Systems

G-Base

References: [JE/HA/DO'89], [OOST'91], [KIM'90b] + documentation and leaflets

G-Base was the first commercially available OODBMS (1984), originally developed by Graphael (France) it is now marketed by Object Databases. It is based upon the frame-model and its main programming language is Lisp (though a C++ version is sheduled for early 1992). G-Base comes in two versions: G-Base-W (single user) and G-Base/GTX (multi-user). There exists a Macintosh version on top of the Allegro Common Lisp environment (the predecessor of Macintosh Common Lisp), but according to our documentation it is the single-user G-Base-W.

Gemstone

References: [JE/HA/DO'89], [OOST'91], [AHMal'91], [KIM'90b], [BU/OT/ST'91], [MA/ST'90], [MAIal'86], [PE/ST'87], [PU/SC/MA'91] + documentation and leaflets

An OODBMS which has found widespread use. Based upon early research and Smalltalk-80 it is one of the older systems (1987), but as such available on a wide variety of platforms including Macintosh and IBM-PC clients. The supported languages are Smalltalk, C en C++ and there is also a C interface for other languages (Cobol, Fortran, Lisp).

Gemstone lacked some useful tools but with the new release (3.0 expected for the summer 1992) this problem should be fixed.

Versant

References: [OOST'91], [AHMal'91], [DESA'91] and training manual, documentation and leaflets

Versant is a newer product not based on well known research prototypes. Their experience is based on traditional database technology, extended with a variety of features (long term transactions, many different lock types, version control, heterogeneous and distributed databases). Versant is available on different Unix-platforms, not on Macintosh or IBM-PC. They support C, C++ and Smalltalk (not on all of the platforms) but this list may be extended in the future.

O2

References: [KIM'90b], [DEUal'91], [LE/RI/VE'90] + documentation and leaflets

O2 grew from a French research project aimed to build a database product for the 90s by combining that state of the art of different fields. After five years (from September 1995 until September 1991) of R&D efforts, they came up with a DBMS that supports C and C++ and is available on some UNIX platforms. They will extend this list of platforms before supporting other languages. O2 had a serious impact on the research community, especially the clean model (object orientation, persistence and query language are well integrated) is widely appreciated.

Orion & Itasca

References: [OOST'91], [AHMal'91], [BANal'90], [WO/KI/LU'90], [KIMal'90], [KIMal'87](136), [THUR'89] + documentation and leaflets

Orion started in 1985 as a research project at the MCC (Micro electronics and Computer Technology Corporation) based upon a object oriented extension of Common Lisp. The Itasca product is an extension of the third ORION prototype and supports C++, C and Lisp. Like O2, Orion had a major influence on the research community. It included the widest variety of features (especially for distributed and multimedia systems) and it has powerful dynamic scheme management facilities.

ObServer/ENCORE

References: [AHMal'91], [KIM'90]b (112), [SK/ZD'86], [SKAR'89], [SM/ZD'87] + documentation and leaflets

ObServer is a general purpose object server developed at Brown University for pure research purposes. ENCORE is the front-end environment supporting C. The system is available on a few Unix platforms. ObServer is recognised for its original work on extended concurrency control (long term transactions, version control).

VBase & Ontos

References: [JE/HA/DO'89], [OOST'91], [AHMal'91], [KIM'90]b (112), [ANDR'90], [ANDR'91], [DAMO'91], [AN/HA'87], [AN/HA'90]

Ontos, formerly VBase is a commercial OODBMS developed by Ontologic. VBase started with its own object oriented extension of C, with two major parts: TDL (Type definition language) and COP (C Object Processor). Later on they decided to move to C++ and from then on they called the product Ontos. Ontos runs on UNIX platforms and OS-2 machines.

Statice

References: [JE/HA/DO'89], [OOST'91], [KIM'90]b (112), [WEIal'91] + documentation and leaflets

Statice is implemented in Common Lisp to run on Symbolics LISP machines but SUNs and Macintoshes with an IVORY-coprocessor board are also supported. Being commercially available since 1988 it is one of the first OODBMS.

IRIS

References: [KIM'90]b (112), [FISal'90], [FISal'90]

IRIS is an industrial research prototype from Hewlett Packard. It has a very clean data model with a consistent use of functions. IRIS supports Lisp and C++.

Postgres (Ingres)

References: [KIM'90b], [RO/ST'90], [ST/KE'91]

Postgres is a sequel to the Ingres relational database system. The philosophy of the Postgres designers is to conservatively extend the well-founded relational model, and as such they were a member of "The committee for Advanced DBMS function" that wrote the "Third-Generation Database system manifesto".

ObjectStore

References: [OOST'91], [AHMal'91], [LAMal'91]

An OODBMS devoted to C++ and focusing on Unix and IBM-PC platforms.

ODBMS

References: Documentation and leaflets

An German system that has been implemented using Smalltalk/V to support for OS/2 and Windows 3.0.

6. Acknowledgements

Sponsors for this research are the "Brussels hoofdstedelijk gewest via het Instituut voor Wetenschappellijk onderzoek in Nijverheid en Landbouw (IWONL)" "The Brussels Free University" and "SoftCore".

7. References

[AHMal'91] Ahmed, S. / Wong, A. / Sriram, D. / Logcher, R. "A comparision of Object-Oriented Database Management Systems for Engineering Applications"
Massachusetts Institute of Technology - Research Report R91-12

[AN/HA'87] Andrews, T. / Harris, C. "Combining Language and Database advances in an Object-Oriented Development environment"
OOPSLA '87 proceedings

[AN/HA'90] Andrwes, T. / Harris, C. "Combining language and Database Advances in an Object-Oriented Development Environment"
From "Readings in object-oriented database systems (ed. Zdonik, S. B. / Maier, D.)" ISBN 0-55860-000-0; Morgan Kaufmann publishers.

[ANDR'90] Andrews, T. "The Vbase Object Database Environment"
Paper from "Research foundations in object-oriented and semantic database systems (ed. Cárdenas, A. / McLeod, D.)" ISBN 0-13-806340-0; Prentice Hall

[ANDR'91] Andrews, T. "Programming with VBase"
From "Object-Oriented Databases with applications to CASE, Networks, and VLSI" Gupta, R. / Horowitz, E. (Editors) ISBN 0-13-629833-8; Prentice hall

[AT/BU/MO'88] Atkinson, M. P. / Buneman, P. / Morrison, R. (Editors) "Data Types and Persistence"
ISBN 0-387-18785-5; Springer Verlag New York

[ATKal'89] Atkinson, M / Bancilhon, F. / DeWitt, D. / Kittrick, K. / Maier, D. / Zdonik, S. "The Object Oriented Database System Manifesto"
Deductive and Object-Oriented Databases; Elsevier Science Publishers

[BANal'90] Benerjee, J. / Chou, H. / Garza, J. F. / Kim, W. / Woelk , D. / Ballou, N. / Kim, H. "Data model issues for Object-Oriented Application"
From "Readings in object-oriented database systems (ed. Zdonik, S. B. / Maier, D.)" ISBN 0-55860-000-0; Morgan Kaufmann publishers.

[BE/HA/GO'87] Bernstein, P. A. / Hadzilacos, V. / Goodman, N. "Concurrency control and recovery in Database Systems"
ISBN 0-201-10715-5; Addison-Wesley

[BL/ZD'87] Bloom, T. / Zdonik, S. B. "Issues in the design of object-oriented Database programming languages"
OOPSLA '87 proceedings

[BOBal'88] Bobrow, D. G. / DeMichiel, L. G. / Gabriel, R. P. / Keene, S. E. / Kiczales, G. / Moon, D. A. "Common Lisp Object System specification X3JI3 Document 88-002R"
SIG PLAN Notices Vol. 23, Sept. '88. Special Issue

[BU/OT/ST'91] Butterworth, P / Otis, A. / Stein, J. "The Gemstone object database management system"
Communications of the ACM, Vol. 34(10), Oct. 91

[BYTE'89] Pascal, F. "A brave new world ?"
BYTE, September 1989 (In depth: Database trends)

[CODD'70] Codd, E.F. "A Relational Model for Large Shared Data Banks."
Communications of the ACM, Vol. 13(6), June 1970.

[COOK'91] Cook, W. R "Object-Oriented Programming versus Abstract Data Types"
Foundations of Object-Oriented Languages. ISBN 0-387-53931-X; Springer-Verlag

[DA/TO'88] Danforth, S. / Tomlinson, C. "Type Theories and Object-Oriented Programming"
ACM Computing surveys, Vol 20(1), March '88

[DAMO'91] Damon, C. "C++ and COP: abrief comparision"
From "Object-Oriented Databases with applications to CASE, Networks, and VLSI" Gupta, R. / Horowitz, E. (Editors) ISBN 0-13-629833-8; Prentice hall

[DATE'90] Date, C. J. "An introduction to database systems, Volume 1 (Fifth edition)"
ISBN 0-201-52878-9; Addisson-Wesley

[DESA'91] DeSanti, M. "OODBMS Pays off"

	DBMS Developing Corporate application Vol4 (12) Nov. '91
[DEUal'91]	Deux, O. et al. "The O2 system"
	Communications of the ACM, Vol. 34(10), Oct. 91
[DO/KI'87]	Dolk, D. R. / Kirsch, R. A. "A Relational Information Resource Dictionary System"
	Communications of the ACM, Vol 30 (1), January '87
[EDEL'91]	Edelstein, H. "Relational vs. Object-Oriented"
	DBMS Developing Corporate application Vol4 (12) Nov. '91
[EL/MA'89]	Elmasri, R. / Navathe, S. B. "Fundamentals of database system"
	ISBN 0-8053-0145-3; Benjamin/Cummings Publishing Company, Inc.
[FISal'90]	Fishman, D. H. / Beech, D. / Cate, H. P. / Chow, E. C. / Connors, T. / Davis, J. W. / Derrett, N. / Hoch, C. G. / Kent, W. / Lyngbaek, P. / Mahbod, B. / Neimat, M. A. / Ryan, T. A. / Shan, M. C. "Iris: an object-oriented database management system"
	From "Readings in object-oriented database systems (ed. Zdonik, S. B. / Maier, D.)" ISBN 0-55860-000-0; Morgan Kaufmann publishers.
[FISal'90]	Fishman, D. H. / Beech, D. / Annevelinck, J. / Chow, E. / Connors, T. / Davis, J. W. / Hasan, W. / Hoch, C. G. / Kent, W. / Leichner, S. / Lyngbaek, P. / Mahbod, B. / Neimat, M. A. / Risch, T. / Chan, M. C. / Wilkinson, W. K. "Overview of the Iris DBMS"
	Paper from "Research foundations in object-oriented and semantic database systems (ed. Cárdenas, A. / McLeod, D.)" ISBN 0-13-806340-0; Prentice Hall
[GO/WO'90]	Goguen, J. A. / Wolfram, D. "On Types and FOOPS"
	Proceedings of the IFIP 1990 Conference on OODB (Windermere, U.K.); Elsevier Science Publishers
[HUGH'91]	Hughes, J. G. "Object-Oriented Databases"
	ISBN 0-13-629874-5; Prentice Hall
[JE/HA/DO'89]	Jeffcoate, J. / Hales, K. / Downes, V. "Object-oriented systems: the commercial benefits (Ovum Report)"
	Ovum report. ISBN 0-903969-42-4; Ovum, Ltd
[KIM'90a]	Kim, W. "Object-Oriented Databases: Definition and Research directions"
	IEEE Transactions on knowledge and data engineering, Vol. 2 No. 3, September 1990
[KIM'90b]	Kim, W. "Introduction to object-oriented Databases"
	ISBN 0-262-11124-1; MIT Press
[KIMal'87]	Kim, W. / Banerjee, J. / Chou, H. / Garza, J. F. / Woelk, D. "Composite Object support in an object-oriented Database system"
	OOPSLA '87 proceedings
[KIMal'90]	Kim, W. / Ballou, N. / Banerjee, J. / Chou, H. / Garza, J. F. / Woelk, D. "Integrating an Object-Oriented Programming System with a Database System"
	Paper from "Research foundations in object-oriented and semantic database systems (ed. Cárdenas, A. / McLeod, D.)" ISBN 0-13-806340-0; Prentice Hall
[LAMal'91]	Lamb, C. / Landis, G. / Orenstein, J. / Weinreb, D. "The Objectstore database system"
	Communications of the ACM, Vol. 34(10), Oct. 91
[LE/RI/VE'90]	Lécluse, C. / Richard, P. / Velez, F. "O2: an object-oriented data model"
	From "Readings in object-oriented database systems (ed. Zdonik, S. B. / Maier, D.)" ISBN 0-55860-000-0; Morgan Kaufmann publishers.
[LISal'77]	Liskov, B. / Snyder, A. / Atkinson, R. / Schaffert, C. "Abstraction mechanisms in CLU"
	From "Readings in object-oriented database systems (ed. Zdonik, S. B. / Maier, D.)" ISBN 0-55860-000-0; Morgan Kaufmann publishers.
[MA/ST'90]	Maier, D. / Stein, J. "Development and implementation of an Object-Oriented DBMS"
	From "Readings in object-oriented database systems (ed. Zdonik, S. B. / Maier, D.)"I SBN 0-55860-000-0; Morgan Kaufmann publishers.
[MACL'87]	Maclennan, B. J. "Principles of programming languages (Design, evaluation and implementation)"
	ISBN 0-03-005163-0; CBS College publishing
[MAIal'86]	Maier, D. / Stein, J. / Otis, A. Purdy, A. "Development of an Object-Oriented DBMS"
	OOPSLA '86 proceedings
[MASal'91]	Masini, G. / Napoli, A. / Colnet, D. / Leonard, D. / Tombre, K. "Object oriented languages"
	ISBN 0-12-477390-7 (A.P.I.C. series, no. 34); Academic Press
[OOST'91]	Object-Oriented Strategies "The Object-Oriented Market in the fall of 1991"

	Object-Oriented Strategies - 1st issue - Oct '91
[ORAC'90]	ORACLE "Tree Walking"
	Oracle training services Course SQL
[PAEP'90]	Paepcke, A. "PCLOS: stress testing CLOS"
	ECOOP/OOPSLA'90 Proceedings
[PE/ST'87]	Penney, D. J. / Stein, J. "Class modification in the Gemstone Object-Oriented DBMS"
	OOPSLA '87 proceedings
[PU/SC/MA'91]	Purdy, A. / Schuchardt, B. / Maier, D. "Integrating an Object Server with Other Worlds"
	From "Object-Oriented Databases with applications to CASE, Networks, and VLSI" Gupta, R. / Horowitz, E. (Editors) ISBN 0-13-629833-8; Prentice hall
[RO/ST'90]	Rowe, L. A. / Stonebraker, M. R. "The POSTGRES Data Model"
	From "Readings in object-oriented database systems (ed. Zdonik, S. B. / Maier, D.)" ISBN 0-55860-000-0; Morgan Kaufmann publishers.
[SALZ'88]	Salzberg, B. "File Structures, An analytic approach"
	ISBN 0-13-314691-X; Prentice Hall.
[SI/ST/UL'91]	Silberschatz, A. / Stonebraker, M. / Ullman, J. "Database systems: achievements and opportunities"
	Communications of the ACM, Vol. 34(10), Oct. 91
[SK/ZD'86]	Skarra, A. H. / Zdonik, S. B. "The management of changing types in an Object-Oriented Database"
	OOPSLA '86 proceedings
[SKAR'89]	Skarra, A. H. "Concurrency Control for cooperating transactions in an object-oriented database"
	Sigplan notices, Vol 24(4), Apr. '89
[SM/ZD'87]	Smith, K.E. / Zdonik, S.B. "Intermedia: A case study of the Differences Between Relational and Object-Oriented Database Systems"
	OOPSLA'87 proceedings
[ST/KE'91]	Stonebraker, M. / Kemnitz, G. "The postgres next-generation database management system"
	Communications of the ACM, Vol. 34(10), Oct. 91
[STEE'90]	Steele, G. L. jr. "Common Lisp, The Language (second edition)"
	ISBN 1-55558-041-6; Digital Press
[STOal'90]	Stonebraker, M. et al. (The committee for Advanced DBMS function: Stonebraker, M. / Rowe, L. / Lindsay, B. / Gray, J. / Carey, M. / Brodie, M. / Bernstein, P. / Beech, D.) "Third-Generation Database system manifesto"
	Proceedings of the IFIP 1990 Conference on OODB (Windermere, U.K.); Elseviers Science Publishers
[THUR'89]	Thuraisingham, M. B. "Mandatory security in object-oriented database systems"
	OOPSLA '89 proceedings
[ULLM'88]	Ullman, J. D. "Principles of Database and Knowledge-based systems, Volume 1"
	ISBN 0-7167-8158-1; Computer Science press, Inc.
[WEGN'90]	Wegner, P. "Concepts and Paradigms of Object Oriented Programming"
	OOPS Messenger Volume 1(1), August '90
[WEIal'91]	Weinreb, D. / Feinberg, N. / Gerson, D. / Lamb, C. "An object-oriented database system to support an integrated programming environment"
	From "Object-Oriented Databases with applications to CASE, Networks, and VLSI" Gupta, R. / Horowitz, E. (Editors) ISBN 0-13-629833-8; Prentice hall
[WIRT'76]	Wirth, N. "Algorithms+data structures=programs"
	ISBN 0-13022-418-9; Prentice Hall
[WO/KI/LU'90]	Woelk, D. / Kim, W. / Luther, W. "An object-oriented approach to Multimedia Databases"
	From "Readings in object-oriented database systems (ed. Zdonik, S. B. / Maier, D.)" ISBN 0-55860-000-0; Morgan Kaufmann publishers.
[ZD/MA'90]	Zdonik, S. B. / Maier, D. "Fundamentals of Object-Oriented Databases"
	From "Readings in object-oriented database systems (ed. Zdonik, S. B. / Maier, D.)" ISBN 0-55860-000-0; Morgan Kaufmann publishers.