The Object-Oriented Model

An object-oriented model supports the following:

1. Object identity*.* The elements with which they deal are typically records with unique addresses, just as in the network and hierarchical models.
2. Complex objects*.* Typically, they allow construction of new types by record formation and set formation.
3. Type hierarchy*.* They allow types to have subtypes with special properties.

Object Structure

1. A data item of an elementary type, e.g., integer, real, or character string of fixed or varying length, is an object type T. Such a type corresponds to the data type for a “field” in networks or hierarchy.
2. If T is an object type, then SETOF(T) is an object type. An object of type SETOF(T) is a collection of objects of type T. The collection consists of pointers to the objects in the set.
3. If $T_1,...,T_k$ are object types, then RECORDOF($T_1,...,T_k$) is an object type. As with sets, an object of this type consists of pointers to one object of each of the k types in the record. However, if object type T_i is an elementary type, then the value of the object itself appears in the record.

Example: For our running example, we shall, for simplicity, assume that the only elementary types are string and int. Then the type of an item can be represented by the record

ItemType = RECORDOF (name: string, I#: int)

Notice the convention that a field of a record is represented by the pair (<fieldname>:<type>).
To handle orders, we need to represent item/quantity pairs. Thus we need another object type

IQType = RECORDOF (item: ItemType, quantity: int)
Here, the first field is an object of a nonelementary type, so that field should be thought of as a pointer to an item.
Now we can define the type of an order to be:

```
OrderType = RECORDOF (O#: int,
                      includes: SETOF (IQType))
```

Here, we have embedded the definition of another object type, SETOF (IQType)*,* within the definition of OrderType. That is equivalent to writing the two declarations:

SIQType = SETOF (IQType)
OrderType = RECORDOF (O#: int, includes: SIQType)

Either way, the field includes of OrderType is a representation of a set of pointers to objects of type IQType, perhaps a pointer to a linked list of pointers to those objects.
Customers can be represented by objects of the following type:

```
CustType=RECORDOF(name:string, addr: string,
              balance: int, orders: SETOF(OrderType))
```

while departments may be given the following declaration:

```
DeptType = RECORDOF (name: string, dept#: int,
           emps: SETOF (EmpType), mgr: EmpType,
           items: SETOF (ItemType))
```

Notice that this declaration twice makes use of a type EmpType, for employyes, once as a set and once directly. In both case, it is not the employees or manager of the department that appear here, but pointers to the actual employee objects. Those objects have the following type:

```
EmpType = RECORDOF (name: string, salary: int,
                    dept: DeptType)
```

Here, we should notice that DeptType is the type of a field of EmpType, just as EmpType and SETOF (EmpType) are types of fields of DeptType. That apparent mutual recursion causes no problems because the references are by pointers, rather than physical presence.

The entire definition of types for our example is:

```
IQType = RECORDOF (item: ItemType, quantity: int)
OrderType = RECORDOF (O#: int,
                       includes: SETOF (IQType))
CustType=RECORDOF(name:string, addr: string,
             balance: int, orders: SETOF(OrderType))
DeptType = RECORDOF (name: string, dept#: int,
              emps: SETOF (EmpType), mgr: EmpType,
              items: SETOF (ItemType))
EmpType = RECORDOF (name: string, salary: int,
                    dept: DeptType)
IPType = RECORDOF (item: ItemType, price: int)
SupType = RECORDOF (name: string, addr: string,
                    supplies: SETOF (IPType))
```

This database scheme is similar to, but not identical to the last hierarchical scheme. It includes a pathway from employees to their departments, since the field dept of EmpType is a pointer to the department. However, we do not have a way to get from items to their orders or suppliers. There is nothing inherit in either model that forces these structures. We could have added the additional pointers to item records by declaring

```
ItemType = RECORDOF (name: string, I#: int,
                     sups: SETOF (SupType),
                     orders: SETOF (OrderType))
```

The manager of the department is an object, not a set, and therefore can be only one manager of a department.

Classes and Methods

An object-oriented data model is not limited to the notation of an object type. The basic notation is really the class, which is an object type for the underlying data structure, and a set of methods, which are operations to be performed on the objects with the object-structure of that class.

Example: We can construct a class of all objects with the structure of EmpType. For this class we might create a set of methods:

```
GetName:
    return (name)

Raise (X):
    salary := salary + X
```

Class Hierarchies

Another essential ingredient in the object model is the notation of subclasses and hierarchies of classes, a formalization of isa relationship. There are two common approaches to the definition of class hierarchies:

1. In addition to record and set constructors for types, allow a third constructor, type union. Objects of type $U(T_1,T_2)$ are either type T_1 objects or type T_2 objects.
2. Define a notion of subtype for given types.

The first approach is used in programming languages like C and Pascal. In object-oriented database systems, it is preferable to use the second approach, because

a) It does not allow the union of unrelated types to considered a type, a capability that is useful in programming languages when defining data structures, but it counterproductive when trying to develop a meaningful database scheme.
b) It extends naturally from object structures to classes, i.e., from types to types with methods.

Suppose we have a class C, and we wish to define a subclass D. We begin with the same object structure for D as for C, and with the same methods for D as for C. We may then modify the class D as follows:

1. If the structure for C is a record type, i.e., of the form

RECORDOF $(T_1,\ldots,T_k)$

then we may add additional components to the record structure.

2. We may create new methods that apply only to subclass D.
3. We may redefine methods of class C to have a new meaning in D.

Example: It would be natural to define MgrType as a subclass of EmpType. We might give MgrType the additional field bonus, so the structure for MgrType would be

```
MgrType = RECORDOF (name: string, salary: int,
                    dept: DeptType, bonus: int)
```

We might also create a method for MgrType that returned the bonus. We could even create a method for MgrType that returned the department. If this method were not defined for the class of employee, then we could not use it on objects that were not of the manager class, even though the method “made sense”, since all employee objects have a dept field. Notice that each employee, whether or not a manager, corresponds to exactly one object of class EmpType. If the employee happens to be a manager, then that object has extra fields and methods, but there are not two objects for this employee.

Operations in the Object Model

Methods can perform any operation on data whatsoever. It is essential to allow navigation from an object O to the objects pointed to by fields of O; this operation corresponds to movement from parent to child, or analog a pointer in a virtual field in the hierarchical model. It is very useful to allow selection on fields that are sets of objects. Thus, we can navigate from an object O to a designed subset of the objects found in some set-valued field of O. A language OPAL includes these features.

Database Management System (DBMS)

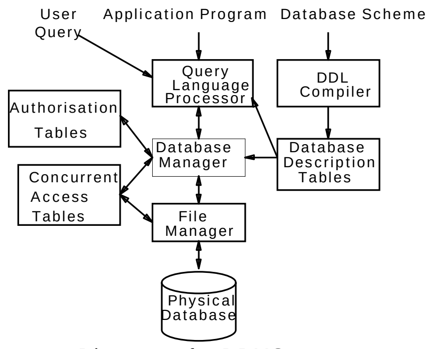

Diagram of a DBMS

On the right, we show the design, or database scheme, fed to the DDL compiler, which produces an internal description of the database. The modification of the database scheme is very infrequent in a large, multiuser database. This modification is normally the responsibility of a database administrator, a person or persons with responsibility for the entire system, including its scheme, subscheme (views), and authorization to access parts of the database.
We also see the query-language processor, which is given data manipulation programs from two sources. One source is user queries or other data manipulations, entered directly at a terminal. The second source is application programs, where database queries and manipulations are embedded in a host language and preprocessed to be run later, perhaps many times.

The portions of an aplication program written in a host language are handled by the host language compiler, not shown. The portions of the application program that are data manipulation language statements are handled by the query language processor, which is responsible for optimization of these statements.
DML statements, especially queries, which extract data from the database, are often transformed significantly by the query processor, so that they can be executed much more efficiently than if they had been executed as written. We show the query processor accessing the database description tables that were created by the DDL program to ascertain some facts that are useful for optimization of queries, such as the existence or nonexistence of certain indices.

Below he query processor we see a database manager, whose role is to take commands at the conceptual level and translate them into commands at the physical level, i.e., the level of files. The database manager maintains and accesses tables of authorization information and concurrency control information. Authorization tables allow the database manager to check that the user has permission to execute the intended query or modification of the database. Modification of the authorization table is done by the database manager, in response to properly authorized user commands.
If concurrent access to the database by different queries and database manipulation is supported, the database manager maintains the necessary information in a specialized table. There are several forms the concurrency control table can take.

For example, any operation modifying a relation may be granted a “lock” on that relation until the modification is complete, thus preventing simultaneous, conflicting modifications. The currently held locks are stored in what we referred to as the concurrent access tables.
The database manager translates the commands given it into operations on files, which are handled by the file manager. This system may be the general-purpose file system provided by the underlying operation system, or it may be a specialized system modified to support the DBMS. For example, a special-purpose DMBS file manager may attempt to put parts of a file that are likely to be accessed as a unit on one cylinder of a disk. Doing so minimizes “seek time”, since we can read the entire unit after moving the disk head once.

The file manager may use the concurrent access tables. We can allow more processes to access the database concurrently if we lock objects that are smaller than whole files or relations. For example, if we locked individual blocks of which a large file was composed, different processes could access and modify records of that file simultaneously, as long they were on different blocks.