Chapter 8

Object-Oriented Databases

8.1 New DB Applications

1. Traditional applications.

   (a) Four generations of traditional DB systems: file system, hierarchical, CODASYL, and relational. All are designed for business applications: inventory, employee, university, bank, library, air-line reservation systems, etc.

   (b) Common features of 'traditional' applications:

      i. Uniformity: large number of similarly structured data items, all of which have the same size,

      ii. Record orientation: the basic data items consist of fixed-length records,

      iii. Small data items: each record is short,

      iv. Atomic fields: fields within a record are short and of fixed length. There is no structure within fields. The 1st normal form holds.

      v. Short transactions: within fractions of a second. There is no human interaction with a transaction during its execution.

      vi. Static conceptual schemes: The database scheme is changed infrequently. Only simple changes are allowed, e.g., in relational systems: create relation, remove relation, add/remove attributes to/from a relation scheme.

2. New applications

   (a) Engineering databases, CAx: computer-aided design (CAD), manufacturing (CAM), engineering (CAE)), CIM (computer-integrated manufacturing).

   Tasks: A CAD database stores data required pertaining to an engineering design, including the components of the items being designed, the inter-relationship of components, and old versions of designs.

   (b) Computer-aided software engineering (CASE): A CASE database stores data required to assist software developers, including source code, dependencies among software modules, definitions and uses of variables, and the development history of the software system.

   (c) Multimedia databases: A multimedia database contains spatial data, audio/video data, and the like. DBs of this sort arise from geophysical data, voice mail systems and graphics applications.

   (d) Office Information Systems (OIS): Office automation includes workstation-based tools for document creation and retrieval, tools for maintaining appointment calendars, and so on. An OIS DB must allow queries pertaining to schedules, documents, and contents of documents.

(e) Hypertext databases: Hypertext is text enriched with links that point to other documents, e.g., WWW. Hypertext documents may also be structured in specific ways that help index them. Hypertext database must support the ability to retrieve documents based on links, and to query documents based on their structure.

3. Expected Features for New Applications

(a) Complex objects: A complex object is an item that is viewed as a single object in the real world, but that contains other objects (with an arbitrary complex internal structure). Often objects are stored hierarchically, representing the containment relationship. This leads to object-oriented DBs and nested relational DBs.

(b) Behavioral data: Distinct objects may need to respond in different ways to the same command. For example, the deletion of certain tuples may require to delete other tuples in the case for weak entities. In CAD and CASE applications the behavior of different objects in response to a given command may be widely different. This behavioral information can be captured by storing executable code with objects in the database. This capability is provided by the *methods* of OODBs and by the *rule base* of KB systems.

(c) Meta knowledge: General rules about the application rather than specific tuples (i.e., data about data) form an important part of expert databases.

(d) Long duration transactions: CAD and CASE applications involve human interaction with the data. "what-if" modifications that the user may wish to undo, concurrent designer efforts that may lead to conflicts among transactions. Important concepts: nested transactions, correct, nonserializable executions.

8.2 The Object-Oriented Data Model

1. A data model is a logic organization of the real world objects (entities), constraints on them, and the relationships among objects. A DB language is a concrete syntax for a data model. A DB system implements a data model.

2. A core object-oriented data model consists of the following basic object-oriented concepts:

(1) **object and object identifier**: Any real world entity is uniformly modeled as an object (associated with a unique id: used to pinpoint an object to retrieve).

(2) **attributes and methods**: every object has a state (the set of values for the attributes of the object) and a behavior (the set of methods – program code – which operate on the state of the object). The state and behavior encapsulated in an object are accessed or invoked from outside the object only through explicit message passing.

[An attribute is an instance variable, whose domain may be any class: user-defined or primitive. A class composition hierarchy (aggregation relationship) is orthogonal to the concept of a class hierarchy. The link in a class composition hierarchy may form cycles.]

(3) **class**: a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class (instance-of relationship). A class is similar to an abstract data type. A class may also be primitive (no attributes), e.g., integer, string, Boolean.

(4) **Class hierarchy and inheritance**: derive a new class (subclass) from an existing class (superclass). The subclass inherits all the attributes and methods of the existing class and may have additional attributes and methods. single inheritance (class hierarchy) vs. multiple inheritance (class lattice).

8.2.1 Object Structure

1. The object-oriented paradigm is based on encapsulating code and data into a single unit. Conceptually, all interactions between an object and the rest of the system are via messages. Thus, the interface between an object and the rest of the system is defined by a set of allowed *messages*.

2. In general, an object has associated with it:
   - A set of *variables* that contain the data for the object. The value of each variable is itself an object.
   - A set of *messages* to which the object responds.
   - A set of *methods*, each of which is a body of code to implement each message; a method returns a value as the *response* to the message.
3. Motivation of using messages and methods.

   All *employee* objects respond to the *annual-salary* message but in different computations for managers, tellers, etc. By encapsulation within the employee object itself the information about how to compute the annual salary, all employee objects present the same interface.

   Since the only external interface presented by an object is the set of messages to which it responds, it is possible to (i) modify the definition of methods and variables without affecting the rest of the system, and (ii) replace a variable with the method that computes a value, e.g., age from birth_date.

   The ability to modify the definition of an object without affecting the rest of the system is considered to be one of the major advantages of the OO programming paradigm.
4. Methods of an object may be classified as either *read-only* or *update*. Message can also be classified as *read-only* or *update*. Derived attributes of an entity in the ER model can be expressed as read-only messages.

8.2.2 Object Classes

1. Usually, there are many similar objects in a DB. By "similar", it means that they respond to the same messages, use the same methods, and have variables of the same name and type. We group similar objects to form a *class*. Each such object is called an *instance* of its class. E.g., in a bank DB, customers, accounts and loans are classes.
2. The definition of the class *employee*, written in pseudo-code. The definition shows the variables and the messages to which the objects of the class respond, but not the methods that handle the messages.

```
class employee {
    /* Variables */
        string name;
        string address;
        date start-date;
        int salary;
    /* Messages */
        int annual-salary();
        string get-name();
        string get-address();
        int set-address(string new-address);
        int employment-length();
};
```

3. Class: (i) captures the instance-of relationship, (ii) the basis on which a query may be formulated, (iii) enhance the integrity of OO systems by introducing type checking, and (iv) reducing replications of names and integrity-related specifications among objects in the same class.
4. The concept of classes is similar to the concept of abstract data types. There are several additional aspects to the class concept beyond those of ADTs. To represent these properties, we treat each class as itself being an object.

   Metaclass: the class of a class. Most OODB systems do not support the strict notion of metaclass. In ORION, CLASS is the root of the class hierarchy (the metaclass of all other classes). A class object includes
   - a set-valued variable whose value is the set of all objects that are instances of the class,
   - implementation of a method for the message *new*, which creates a new instance of the class.

8.2.3 Inheritance

1. An object-oriented database schema typically requires a large number of classes. Often, however, several classes are similar. For example, bank employees are similar to customers.

2. In order to allow the direct representation of similarities among classes, we need to place classes in a specialization hierarchy. E.g., Fig. 8.1 is a specialization hierarchy for the ER model.

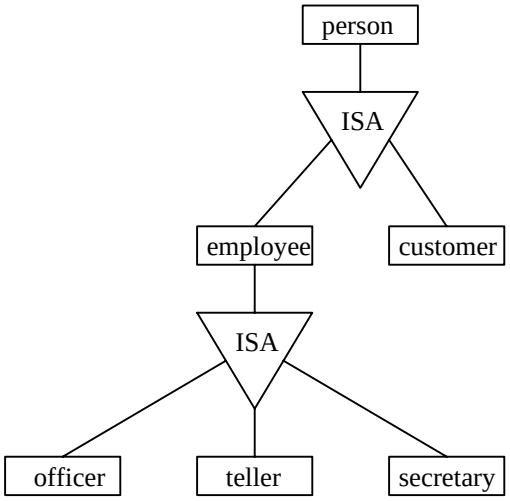

Figure 8.1: Specialization hierarchy for the banking example

The concept of a class hierarchy is similar to that of specialization in the ER model. The corresponding corresponding class hierarchy is shown in Fig. 8.2.

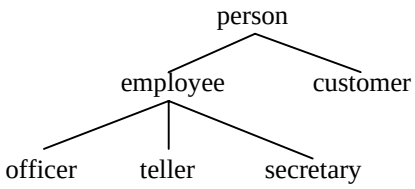

Figure 8.2: Class hierarchy corresponding to the banking example

The class hierarchy can be defined in pseudo-code in Fig. 8.3, in which the variables associated with each class are as follows. For brevity, we do not present the methods associated with these classes.

3. The keyword **isa** is used to indicate that a class is a specialization of another class. The specialization of a class are called *subclasses*. E.g., *employee* is a subclass of *person*; *teller* is a subclass of *employee*. Conversely, *employee* is a superclass of *teller*.

4. Class hierarchy and inheritance of properties from more general classes. E.g., an object representing an *officer* contains all the variables of classes *officer, employee*, and *person*. Methods are inherited in a manner identical to inheritance of variables.

5. An important benefit of inheritance in OO systems is the notion of *substitutability*: Any method of a class, A, can be equally well be invoked with an object belonging to any subclass B of A. This characteristic leads to code-reuse: methods and functions in class A (such as get-name() in class *person*) do not have to be rewritten again for objects of class B).

6. Two plausible ways of associating objects with nonleaf classes:

   - associate with the *employee* class all employee objects including those that are instances of *officer*, *teller*, and *secretary*.

```
class person {
        string name;
        string address;
};
class customer isa person {
        int credit-rating;
};
class employee isa person {
        date start-date;
        int salary;
};
class officer isa employee {
        int office-number;
        int expense-account-number;
};
class teller isa employee {
        int hours-per-week;
        int station-number;
};
class secretary isa employee {
        int hours-per-week;
        int manager;
};
```

Figure 8.3: Definition of class hierarchy in pseudo-code

- associate with the *employee* class only those employee objects that are instances neither *officer*, nor *teller*, nor *secretary*.

Typically, the latter choice is made in OO systems. It is possible to determine the set of all employee objects in this case by taking the union of those objects associated with all classes in the subtree rooted at *employee*.

7. Most OO systems allow specialization to be *partial*, i.e., they allow objects that belong to a class such as *employee* that do not belong to any of that class's subclasses.

8.2.4 Multiple Inheritance

1. In most cases, tree-structured organization of classes is adequate to describe applications. In such cases, all superclasses of a class are ancestors of descendants of another in the hierarchy. However, there are situations that cannot be represented well in a tree-structured class hierarchy.

2. Example. We could create subclasses: *part-time-teller, full-time-teller*, etc., as shown in Fig. 8.4. But problems: (1) redundancy leads to potential inconsistency on updates; and (2) the hierarchy cannot represent full/part- time employees who are neither secretaries nor tellers.

3. *Multiple inheritance*: the ability of class to inherit variables and methods from multiple superclasses.

4. The class/subclass relationship is represented by a rooted directed acyclic graph (DAG) in which a class may have more than one superclass.

5. Handling name conflicts: When multiple inheritance is used, there is potential ambiguity if the same variable or method can be inherited from more than one superclass.

6. Example. In our banking example, we may define a variable *pay* for each *full-time, part-time, teller* and *secretary* as follows:

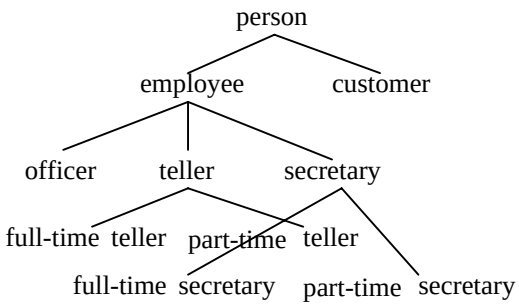

Figure 8.4: Class hierarchy for full- and part-time employees.

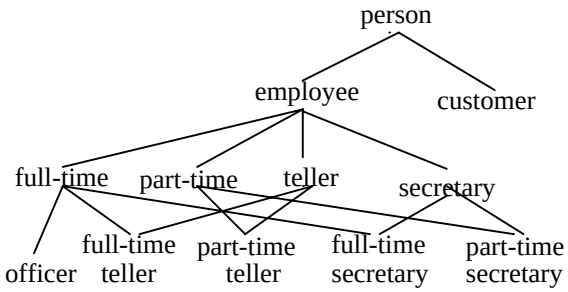

Figure 8.5: Class DAG for the banking example.

- *full-time*: pay is an integer from 0 to 100,000 containing annual salary.
- *part-time*: pay is an integer from 0 to 20 containing an hourly rate of pay.
- *teller*: pay is an integer from 0 to 20,000 containing the annual salary.
- *secretary*: pay is an integer from 0 to 25,000 containing the annual salary.

7. For *part-time-secretary*, it could inherit the definition of *pay* from either *part-time* or *secretary*. We have the following options:

   - Include both variables, renaming them to *part-time-pay* and *secretary-pay*.
   - Choose one or the other based on the order of creation.
   - Force the user the make a choice at the time of class definition.
   - Treat the situation as an error.

   No single solution has been accepted as best, and different systems make different choices.

8. Not all cases of multiple inheritance lead to ambiguity. If, instead of defining *pay*, we retain the definition of variable *salary* in class *employee*, and define it nowhere else, then all the subclasses inherit *salary* from *employee* (no ambiguity).

9. We can use multiple inheritance to model the concept of *roles*. For example, for subclasses, *student, teacher* and *footballPlayer*, an object can belong to several categories at once and each of these categories is called a *role*. We can use multiple inheritance to create subclasses, such as *student-teacher*, *student-footballPlayer*, and so on to model the possibility of an object simultaneously having multiple roles.

8.2.5 Object Identity

1. Object identity: An object retains its identity even if some or all of the values of variables or definitions of methods change over time.

   This concept of object identity is necessary in applications but doe not apply to tuples of a relational database.

2. Object identity is a stronger notion of identity than typically found in programming languages or in data models not based on object orientation.

3. Several forms of identity:

   - **value**: A data value is used for identity (e.g., the primary key of a tuple in a relational database).
   - **name**: A user-supplied name is used for identity (e.g., file name in a file system).
   - **built-in**: A notion of identity is built-into the data model or programming languages, and no user-supplied identifier is required (e.g., in OO systems).

4. Object identity is typically implemented via a *unique*, *system-generated* OID. The value of the OID is not visible to the external user, but is used internally by the system to identify each object uniquely and to create and manage inter-object references.

5. There are many situations where having the system generate identifiers automatically is a benefit, since it frees humans from performing that task. However, this ability should be used with care. System-generated identifiers are usually specific to the system, and have to be translated if data are moved to a different database system. System-generated identifiers may be redundant if the entities being modeled already have unique identifiers external to the system, e.g., SIN#.

8.2.6 Object Containment

1. Objects that contain other objects are called *complex* or *composite* objects. There can be multiple levels of containment, forming a *containment hierarchy* among objects.

2. Example: A bicycle design database:

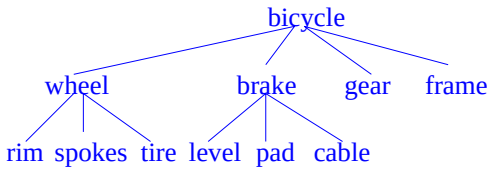

Figure 8.6: Containment hierarchy for bicycle-design database.

   Fig. 8.6 shows the containment relationship in a schematic way by listing class names. Thus the links between classes must be interpreted as **is-part-of**, rather than the **is-a** interpretation of links in an inheritance hierarchy.

3. Containment allows data to be viewed at different granularities by different users. E.g., *wheel* by wheel designer but *bicycle* by a sales-person.

   The containment hierarchy is used to find all objects contained in a *bicycle* object.

4. In certain applications, an object may be contained in several objects. In such cases, the containment relationship is represented by a DAG rather than by a hierarchy.

8.3 Object-Oriented Languages

The expression of object-orientation can be done in one of two ways.

1. The concepts of object-orientation can be used purely as a design tool, and are encoded into, e.g., a relational database. E.g., ER modeling.

2. The concepts of object-orientation are incorporated into a language that is used to manipulate the database. There are several possible languages into which the concepts can be integrated.

(a) Extend a DML such as SQL by adding complex types and object-orientation. Systems that provide object-oriented extensions to relational systems are called *object-relational systems*.

(b) Take an existing object-oriented programming language and extend it to deal with database. Such languages are called *persistent programming languages*.

8.4 Persistent Programming Languages

1. Persistent data: data that continue to exist even after the program that created it has terminated.

2. A persistent programming language is a programming language extended with constructs to handle persistent data. It distinguishes with embedded SQL in at least two ways:

   (a) In a persistent program language, query language is fully integrated with the host language and both share the same type system. Any format changes required in databases are carried out transparently.

   Comparison with Embedded SQL where (1) host and DML have different type systems, code conversion operates outside of OO type system, and hence has a higher chance of having undetected errors; (2) format conversion takes a substantial amount of code.

   (b) Using Embedded SQL, a programmer is responsible for writing explicit code to fetch data into memory or store data back to the database.

   In a persistent program language, a programmer can manipulate persistent data without having to write such code explicitly.

3. Drawbacks: (1) Powerful but easy to make programming errors that damage the database; (2) harder to do automatic high-level optimization; and (3) do not support declarative querying well.

8.4.1 Persistence of Objects

Several approaches have been proposed to make the objects persistent.

1. persistence by class. Declare class to be persistent: all objects of the class are then persistent objects. Simple, not flexible since it is often useful to have both transient and persistent objects in a single class. In many OODB systems, declaring a class to be persistent is interpreted as "persistable" — objects in the class potentially can be made persistent.

2. persistence by creation. Introduce new syntax to create persistent objects.

3. persistence by marking. Mark an object persistent after it is created (and before the program terminates).

4. persistence by reference. One or more objects are explicitly declared as (root) persistent objects. All other objects are persistent iff they are referred, directly or indirectly, from a root persistent object. It is easy to make the entire data structure persistent by merely declaring the root of the structure as persistent, but is expensive to follow the chains in detection for a database system.

8.4.2 Object Identity and Pointers

1. The association of an object with a physical location in storage (as in C++) may change over time.

2. There are several degrees of permanence of identity:

   - **intraprocedure**: Identity persists only during the execution of a single procedure, e.g., local variables within procedures.
   - **intraprogram**: Identity persists only during the execution of a single program or query, e.g., global variables in programming languages, and main memory or virtual memory pointers.
   - **interprogram**: Identity persists from one program execution to another, e.g., pointers to file system data on disk but may change if the way data is stored in the file system is changed.

- **persistent**: Identity persists not only among program executions but also among structural reorganizations of the data. This is the persistent form of identity required for object-oriented systems.

3. In persistent extension of C++, object identifiers are implemented as "*persistent pointers*" which can be viewed as a pointer to an object in the database.

8.4.3 Storage and Access of Persistent Objects

1. How are objects stored in a database?

   Code (that implements methods) should be stored in the database as part of the schema, along with type definitions, but many implementations store them outside of the database, to avoid having to integrate system software such as compilers with the database system.

   Data: stored individually for each object.

2. How to find the objects?

   (a) Give names to objects like we give names to files: works only for small sets of objects.

   (b) Expose object identifiers or persistent pointers to the objects:

   (c) Store the collections of object and allow programs to iterate over the collections to find required objects. The collections can be modeled as objects of a *collection type*. A special case of a collection is a *class extent*, which is a collection of all objects belonging to the class.

   Most OODB systems support all three ways of accessing persistent objects. All objects have object identifiers. Names are typically given only to class extents and other collection objects, and perhaps to other selected objects, but most objects are not given names. Class extents are usually maintained for all classed that can have persistent objects, but in many implementations, they contain only persistent objects of the class.

```
class Person:  public Persistence_Object {
public:
  String name;
  String address;
}

class Customer:  public Person {
public:
  Date member_from;
  int customer_id;
  Ref ⟨Branch ⟩home_branch;
  Set ⟨Ref ⟨Account ⟩⟩accounts inverse Account:  owners;
}
class Account:  public Persistent_Object {
private:
  int balance;
public:
  int number;
  Set ⟨Ref ⟨Customer ⟩⟩owners inverse Customer:  accounts;
  int find_balance();
  int update_balance(int delta);
}
```

Figure 8.7: Example of ODMG C++ Object Definition Language

8.5 Persistent C++ Systems

8.5.1 The ODMG C++ Object Definition Language

1. The ODMG (Object Database Management Group) has been working on standardizing language extensions to C++ and smalltalk to support persistence, and on defining class libraries to support persistence.

2. There are two parts to the ODMG C++ extension: (1) the C++ Object Definition Language (C++ ODL); and (2) the C++ Object Manipulation Language (C++ OML).

3. An example the ODMG C++ Object Definition Language is in Figure 8.7. `Person` and `Account` are subclasses of `Persistence_Object`, and objects in these classes can therefore be made persistent. `Customer` is a subclass of `Person` and is thus also a subclass of `Persistence_Object`.

4. Methods can be directly invoked and behave like procedure calls. The keyword `private` indicates that the following attributes or methods are visible to only methods of the class; `public` indicates that the attributes or methods are visible to other code as well.

5. Type `Ref ⟨Branch ⟩` is a *reference*, or persistent pointer, to an object of type `Branch`. Referential integrity constraint is specified using `inverse`. E.g., `inverse Account: owners` for the attribute *accounts* says that, for each `account` object referenced in the *accounts* set of `Customer` object, the field owners of the `account` object must contain a reference back to the `Customer` object.

8.5.2 The ODMG C++ Object Manipulation Language

1. Figure 8.8 is an example of ODMG C++ Object Manipulation Language.

```
int create_account_owner (String name, String address) {
  Database *bank_db;
  bank_db = Database::open("Bank-DB");
  Transaction Trans;
  Trans.begin();

  Ref ⟨Account ⟩account = new(bank_db) Account;
  Ref ⟨Customer ⟩cust new(bank_db) Customer;
  cust->name = name;
  cust->address = address;
  cust->accounts.insert_element(account);
  account->owner.insert_element(cust);
  ...Code to initialize customer_id, account number, etc.
  Trans.commit();
}
```

Figure 8.8: Example of ODMG C++ Object Manipulation Language

8.6 References

1. A. Silberschatz, H. F. Korth and S. Sudarshan, *Database System Concepts*, McGraw-Hill, 3rd ed., 1997.

2. J. D. Ullman, "*Principles of Database and Knowledge-Base Systems*", Vol. 1, Computer Science Press, 1988. Chapter 1 "Data Base, Object Base and Knowledge Base", Chapter 5. "Object-Oriented Database Languages".

3. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, Bemjamin/Cummings, 2nd ed. 1994.