Chapter 8: Object-Oriented Databases

- Need for Complex Data Types
- The Object-Oriented Data Model
- Object-Oriented Languages
- Persistent Programming Languages
- Persistent C++ Systems

Need for Complex Data Types

- Traditional database applications in data processing had conceptually simple data types
  - Relatively few data types, first normal form holds
- Complex data types have grown more important in recent years
  - E.g. Addresses can be viewed as a
    - Single string, or
    - Separate attributes for each part, or
    - Composite attributes (which are not in first normal form)
  - E.g. it is often convenient to store multivalued attributes as-is, without creating a separate relation to store the values in first normal form
- Applications
  - computer-aided design, computer-aided software engineering
  - multimedia and image databases, and document/hypertext databases.

Object-Oriented Data Model

- Loosely speaking, an **object** corresponds to an entity in the E-R model.
- The *object-oriented paradigm* is based on *encapsulating* code and data related to an object into single unit.
- The object-oriented data model is a logical data model (like the E-R model).
- Adaptation of the object-oriented programming paradigm (e.g., Smalltalk, C++) to database systems.

Object Structure

- 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; each message may have zero, one, or more *parameters.*
  - A set of **methods**, each of which is a body of code to implement a message; a method returns a value as the *response* to the message
- The physical representation of data is visible only to the implementor of the object
- Messages and responses provide the only external interface to an object.
- The term message does not necessarily imply physical message passing. Messages can be implemented as procedure invocations.

Messages and Methods

- Methods are programs written in general-purpose language with the following features
  - only variables in the object itself may be referenced directly
  - data in other objects are referenced only by sending *messages*.
- Methods can be read-only or update methods
  - Read-only methods do not change the value of the object
- Strictly speaking, every attribute of an entity must be represented by a variable and two methods, one to read and the other to update the attribute
  - e.g., the attribute *address* is represented by a variable *address* and two messages *get-address* and *set-address*.
  - For convenience, many object-oriented data models permit direct access to variables of other objects.

Object Classes

- Similar objects are grouped into a **class**; each such object is called an **instance** of its class
- All objects in a class have the same
  - Variables, with the same types
  - message interface
  - methods

  The may differ in the values assigned to variables
- Example: Group objects for people into a *person* class
- Classes are analogous to entity sets in the E-R model

Class Definition Example

```
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();
};
```

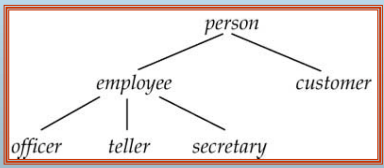
- Methods to read and set the other variables are also needed with strict encapsulation
- Methods are defined separately
  - E.g. **int** *employment-length*() { **return** *today*() – *start-date*;}
    int *set-address*(**string** *new-address*) { *address* = *new-address*;}

Inheritance

- E.g., class of bank customers is similar to class of bank employees, although there are differences
  - both share some variables and messages, e.g., *name* and *address.*
  - But there are variables and messages specific to each class e.g., *salary* for employees and *credit-rating* for customers.
- Every employee is a person; thus *employee* is a specialization of *person*
- *Similarly, customer* is a specialization of *person.*
- Create classes *person, employee* and *customer*
  - variables/messages applicable to all persons associated with class *person.*
  - variables/messages specific to employees associated with class *employee*; similarly for *customer*

Inheritance (Cont.)

- Place classes into a specialization/IS-A hierarchy
  - variables/messages belonging to class *person* are *inherited* by class *employee* as well as *customer*
- Result is a **class hierarchy**



Note analogy with ISA Hierarchy in the E-R model

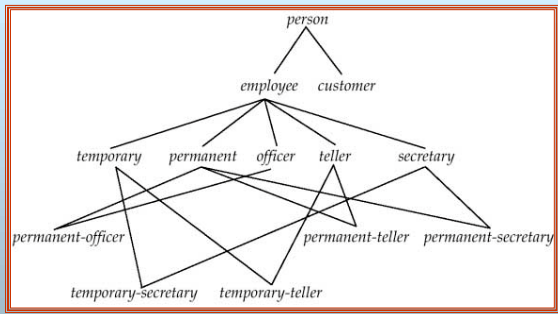
Class Hierarchy Definition

```
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 Hierarchy Example (Cont.)

- Full variable list for objects in the class *officer:*
  - *office-number, expense-account-number:* defined locally
  - *start-date, salary:* inherited from *employee*
  - *name, address:* inherited from *person*
- Methods inherited similar to variables.
- **Substitutability** — any method of a class, say *person,* can be invoked equally well with any object belonging to any subclass, such as subclass *officer* of *person.*
- **Class extent***:* set of all objects in the class. Two options:
  1. Class extent of *employee* includes all *officer, teller* and *secretary* objects.
  2. Class extent of *employee* includes only employee objects that are not in a subclass such as *officer, teller,* or *secretary*
     - This is the usual choice in OO systems
     - Can access extents of subclasses to find all objects of subtypes of employee

Example of Multiple Inheritance



Class DAG for banking example.

Multiple Inheritance

- With multiple inheritance a class may have more than one superclass.
  - The class/subclass relationship is represented by a **directed acyclic graph** (**DAG**)
  - Particularly useful when objects can be classified in more than one way, which are independent of each other
    - E.g. temporary/permanent is independent of Officer/secretary/teller
    - Create a subclass for each combination of subclasses
      - Need not create subclasses for combinations that are not possible in the database being modeled
- A class inherits variables and methods from all its superclasses
- There is potential for ambiguity when a variable/message N with the same name is inherited from two superclasses A and B
  - No problem if the variable/message is defined in a shared superclass
  - Otherwise, do one of the following
    - flag as an error,
    - rename variables (A.N and B.N)
    - choose one.

More Examples of Multiple Inheritance

- Conceptually, an object can belong to each of several subclasses
  - A *person* can play the roles of *student*, a *teacher* or *footballPlayer,* or any combination of the three
    - E.g., student teaching assistant who also play football
- Can use multiple inheritance to model "roles" of an object
  - That is, allow an object to take on any one or more of a set of types
- But many systems insist an object should have a **most-specific class**
  - That is, there must be one class that an object belongs to which is a subclass of all other classes that the object belongs to
  - Create subclasses such as *student-teacher* and *student-teacher-footballPlayer* for each combination
  - When many combinations are possible, creating subclasses for each combination can become cumbersome

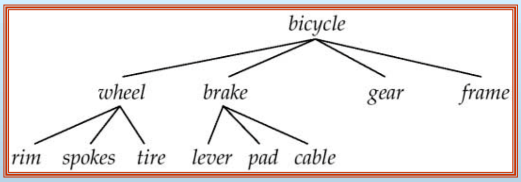
Object Identity

- An object retains its identity even if some or all of the values of variables or definitions of methods change over time.
- Object identity is a stronger notion of identity than in programming languages or data models not based on object orientation.
  - Value – data value; e.g. primary key value used in relational systems.
  - Name – supplied by user; used for variables in procedures.
  - Built-in – identity built into data model or programming language.
    - no user-supplied identifier is required.
    - Is the form of identity used in object-oriented systems.

Object Identifiers

- **Object identifiers** used to uniquely identify objects
  - Object identifiers are unique:
    - no two objects have the same identifier
    - each object has only one object identifier
  - E.g., the *spouse* field of a *person* object may be an identifier of another *person* object.
  - can be stored as a field of an object, to refer to another object.
  - Can be
    - system generated (created by database) or
    - external (such as social-security number)
  - System generated identifiers:
    - Are easier to use, but cannot be used across database systems
    - May be redundant if unique identifier already exists

Object Containment



- Each component in a design may contain other components
- Can be modeled as containment of objects. Objects containing; other objects are called **composite** objects.
- Multiple levels of containment create a **containment hierarchy**
  - links interpreted as **is-part-of,** not **is-a.**
- Allows data to be viewed at different granularities by different users.

Object-Oriented Languages

- Object-oriented concepts can be used in different ways
  - Object-orientation can be used as a design tool, and be encoded into, for example, a relational database
    - analogous to modeling data with E-R diagram and then converting to a set of relations)
  - The concepts of object orientation can be incorporated into a programming language that is used to manipulate the database.
    - **Object-relational systems** – add complex types and object-orientation to relational language.
    - **Persistent programming languages** – extend object-oriented programming language to deal with databases by adding concepts such as persistence and collections.

Persistent Programming Languages

- Persistent Programming languages allow objects to be created and stored in a database, and used directly from a programming language
  - allow data to be manipulated directly from the programming language
    - No need to go through SQL.
  - No need for explicit format (type) changes
    - format changes are carried out transparently by system
    - Without a persistent programming language, format changes becomes a burden on the programmer
      - More code to be written
      - More chance of bugs
  - allow objects to be manipulated in-memory
    - no need to explicitly load from or store to the database
      - Saved code, and saved overhead of loading/storing large amounts of data

Persistent Prog. Languages (Cont.)

- Drawbacks of persistent programming languages
  - Due to power of most programming languages, it is easy to make programming errors that damage the database.
  - Complexity of languages makes automatic high-level optimization more difficult.
  - Do not support declarative querying as well as relational databases

Persistence of Objects

- Approaches to make transient objects persistent include establishing
  - Persistence by Class – declare all objects of a class to be persistent; simple but inflexible.
  - Persistence by Creation – extend the syntax for creating objects to specify that that an object is persistent.
  - Persistence by Marking – an object that is to persist beyond program execution is marked as persistent before program termination.
  - Persistence by Reachability - declare (root) persistent objects; objects are persistent if they are referred to (directly or indirectly) from a root object.
    - Easier for programmer, but more overhead for database system
    - Similar to garbage collection used e.g. in Java, which also performs reachability tests

Object Identity and Pointers

- A persistent object is assigned a persistent object identifier.
- Degrees of permanence of identity:
  - Intraprocedure – identity persists only during the executions of a single procedure
  - Intraprogram – identity persists only during execution of a single program or query.
  - Interprogram – identity persists from one program execution to another, but may change if the storage organization is changed
  - Persistent – identity persists throughout program executions and structural reorganizations of data; required for object-oriented systems.

Object Identity and Pointers (Cont.)

- In O-O languages such as C++, an object identifier is actually an in-memory pointer.
- Persistent pointer – persists beyond program execution
  - can be thought of as a pointer into the database
    - E.g. specify file identifier and offset into the file
  - Problems due to database reorganization have to be dealt with by keeping forwarding pointers

Storage and Access of Persistent Objects

How to find objects in the database:

- Name objects (as you would name files)
  - Cannot scale to large number of objects.
  - Typically given only to class extents and other collections of objects, but not objects.
- Expose object identifiers or persistent pointers to the objects
  - Can be stored externally.
  - All objects have object identifiers.
- Store collections of objects, and allow programs to iterate over the collections to find required objects
  - Model collections of objects as **collection types**
  - **Class extent** - the collection of all objects belonging to the class; usually maintained for all classes that can have persistent objects.

Persistent C++ Systems

- C++ language allows support for persistence to be added without changing the language
  - Declare a class called Persistent_Object with attributes and methods to support persistence
  - **Overloading** – ability to redefine standard function names and operators (i.e., +, –, the pointer deference operator –>) when applied to new types
  - **Template classes** help to build a type-safe type system supporting collections and persistent types.
- Providing persistence without extending the C++ language is
  - relatively easy to implement
  - but more difficult to use
- Persistent C++ systems that add features to the C++ language have been built, as also systems that avoid changing the language

ODMG C++ Object Definition Language

- The Object Database Management Group is an industry consortium aimed at standardizing object-oriented databases
  - in particular persistent programming languages
  - Includes standards for C++, Smalltalk and Java
  - ODMG-93
  - ODMG-2.0 and 3.0 (which is 2.0 plus extensions to Java)
    - Our description based on ODMG-2.0
- ODMG C++ standard avoids changes to the C++ language
  - provides functionality via template classes and class libraries

ODMG Types

- Template class d_Ref<class> used to specify references (persistent pointers)
- Template class d_Set<class> used to define sets of objects.
  - Methods include insert_element(e) and delete_element(e)
- Other collection classes such as d_Bag (set with duplicates allowed), d_List and d_Varray (variable length array) also provided.
- d_ version of many standard types provided, e.g. d_Long and d_string
  - Interpretation of these types is platform independent
  - Dynamically allocated data (e.g. for d_string) allocated in the database, not in main memory

ODMG C++ ODL: Example

```
class Branch  :  public d_Object {
  ....
}
class Person  :  public d_Object {
   public:
      d_String    name;      // should not use String!
      d_String    address;
};
class Account : public d_Object {
   private:
      d_Long      balance;
   public:
      d_Long      number;
      d_Set <d_Ref<Customer>> owners;

      int         find_balance();
      int         update_balance(int delta);
};
```


ODMG C++ ODL: Example (Cont.)

```
class Customer  :  public Person {
    public:
       d_Date            member_from;
       d_Long            customer_id;
       d_Ref<Branch> home_branch;
       d_Set <d_Ref<Account>> accounts; };
```


Implementing Relationships

- Relationships between classes implemented by references
- Special reference types enforces integrity by adding/removing inverse links.
  - Type d_Rel_Ref<Class, InvRef> is a reference to Class, where attribute InvRef of Class is the inverse reference.
  - Similarly, d_Rel_Set<Class, InvRef> is used for a set of references
- Assignment method (=) of class d_Rel_Ref is overloaded
  - Uses type definition to automatically find and update the inverse link
  - Frees programmer from task of updating inverse links
  - Eliminates possibility of inconsistent links
- Similarly, insert_element() and delete_element() methods of d_Rel_Set use type definition to find and update the inverse link automatically

Implementing Relationships

- E.g.

```
extern const char _owners[ ],   _accounts[ ];
class Account : public d.Object {
        ....
        d_Rel_Set <Customer, _accounts> owners;
}
  // .. Since strings can't be used in templates ...
const char _owners= "owners";
const char _accounts= "accounts";
```


ODMG C++ Object Manipulation Language

- Uses persistent versions of C++ operators such as new(db)

  d_Ref<Account> account = new(bank_db, "Account") Account;
  - new allocates the object in the specified database, rather than in memory.
  - The second argument ("Account") gives typename used in the database.
- Dereference operator -> when applied on a d_Ref<Account> reference loads the referenced object in memory (if not already present) before continuing with usual C++ dereference.
- Constructor for a class – a special method to initialize objects when they are created; called automatically on new call.
- Class extents maintained automatically on object creation and deletion
  - Only for classes for which this feature has been specified
    - Specification via user interface, not C++
  - Automatic maintenance of class extents not supported in earlier versions of ODMG

ODMG C++OML: Database and Object Functions

- Class d_Database provides methods to
  - open a database: open(databasename)
  - give names to objects: set_object_name(object, name)
  - look up objects by name: lookup_object(name)
  - rename objects: rename_object(oldname, newname)
  - close a database (close());
- Class d_Object is inherited by all persistent classes.
  - provides methods to allocate and delete objects
  - method mark_modified() must be called *before* an object is updated.
    - Is automatically called when object is created

ODMG C++ OML: Example

```
int create_account_owner(String name, String Address){
    Database bank_db.obj;
    Database * bank_db= & bank_db.obj;
    bank_db =>open(“Bank-DB”);
    d.Transaction Trans;
    Trans.begin();

    d_Ref<Account> account = new(bank_db) Account;
    d_Ref<Customer> cust = new(bank_db) Customer;
    cust->name - name;
    cust->address = address;
    cust->accounts.insert_element(account);
    ... Code to initialize other fields

    Trans.commit();
}
```


ODMG C++ OML: Example (Cont.)

- Class extents maintained automatically in the database.
- To access a class extent:
  d_Extent<Customer> customerExtent(bank_db);
- Class d_Extent provides method
  d_Iterator<T> create_iterator()
  to create an iterator on the class extent
- Also provides select(pred) method to return iterator on objects that satisfy selection predicate pred.
- Iterators help step through objects in a collection or class extent.
- Collections (sets, lists etc.) also provide create_iterator() method.

ODMG C++ OML: Example of Iterators

```
int print_customers() {
    Database bank_db_obj;
    Database * bank_db = &bank_db_obj;
    bank_db->open (“Bank-DB”);
    d_Transaction Trans; Trans.begin ();

    d_Extent<Customer> all_customers(bank_db);
    d_Iterator<d_Ref<Customer>> iter;
    iter = all_customers–>create_iterator();
    d_Ref <Customer> p;
    while{iter.next (p))
          print_cust (p);  // Function assumed to be defined elsewhere
    Trans.commit();
}
```


ODMG C++ Binding: Other Features

- Declarative query language OQL, looks like SQL
  - Form query as a string, and execute it to get a set of results (actually a bag, since duplicates may be present)

```
d_Set<d_Ref<Account>> result;
d_OQL_Query q1("select a
                        from Customer c, c.accounts a
                        where c.name=‘Jones’
                            and a.find_balance() > 100");
d_oql_execute(q1, result);
```

- Provides error handling mechanism based on C++ exceptions, through class d_Error
- Provides API for accessing the schema of a database.

Making Pointer Persistence Transparent

- Drawback of the ODMG C++ approach:
  - Two types of pointers
  - Programmer has to ensure mark_modified() is called, else database can become corrupted
- ObjectStore approach
  - Uses *exactly* the same pointer type for in-memory and database objects
  - Persistence is transparent applications
    - Except when creating objects
  - Same functions can be used on in-memory and persistent objects since pointer types are the same
  - Implemented by a technique called pointer-swizzling which is described in Chapter 11.
  - No need to call mark_modified(), modification detected automatically.

Persistent Java Systems

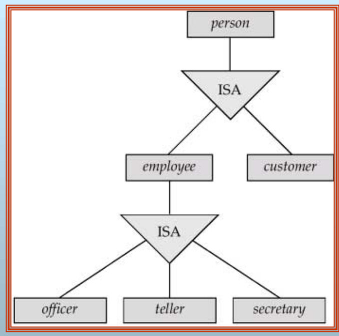
- ODMG-3.0 defines extensions to Java for persistence
  - Java does not support templates, so language extensions are required
- Model for persistence: persistence by reachability
  - Matches Java’s garbage collection model
  - Garbage collection needed on the database also
  - Only one pointer type for transient and persistent pointers
- Class is made **persistence capable** by running a post-processor on object code generated by the Java compiler
  - Contrast with pre-processor used in C++
  - Post-processor adds mark_modified() automatically
- Defines collection types DSet, DBag, DList, etc.
- Uses Java iterators, no need for new iterator class

ODMG Java

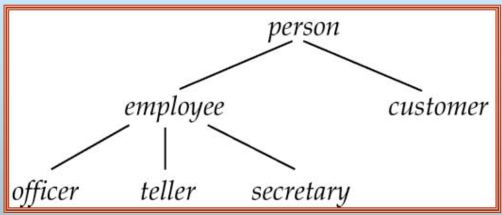
- Transaction must start accessing database from one of the root object (looked up by name)
  - finds other objects by following pointers from the root objects
- Objects referred to from a fetched object are allocated space in memory, but not necessarily fetched
  - Fetching can be done lazily
  - An object with space allocated but not yet fetched is called a hollow object
  - When a hollow object is accessed, its data is fetched from disk.

End of Chapter

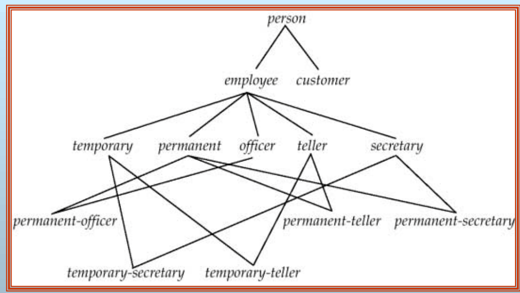
Specialization Hierarchy for the Bank Example



Class Hierarchy Corresponding to Figure 8.2



Class DAG for the Bank Example



Containment Hierarchy for Bicycle-Design Database