ODMG: The Object Database Management Group

As commercial OO-DBMSs developed, each vendor based their system on slightly different object models:

Versant, ONTOS: persistent C++ objects
O2: object model based on complex value theory
Others: persistent Smalltalk objects, Objective-C, etc.

ODMG standardization developed a common model for all OODBs, similar to the standard relational model.

This allows for portability of applications and sharing of objects between systems.
ODMG Standard Components

Object Model:
- Defines the concepts available for defining a schema
- OO things: classes, attributes, methods, inheritance
- database things: relationships, extents, collections, transactions, DBMS control

Object Definition Language:
- ODL is the language for defining an object schema.
- It may be the ODMG language neutral ODL
  or an OOPL specific ODL (C++/ODL, Java/ODL)

Object Query Language:
- OQL allows for declarative queries against the database.
- Queries return sets of objects.
- Based on SELECT/FROM/WHERE.
- Method calls allowed in queries.

Object Manipulation Language:
- The OMLs are supersets of standard OOPLs
- The OOPL is augmented with support for the database unique concepts in the object model.
- Referred to as a “binding” of the OOPL and the ODMG object model.

Comparison between OO model and relational model:

Benefits:
- More expressive data model
- Programmers/designers familiar with a popular OOPL can work directly in a familiar environment
- No “impedence mismatch”: database model and language model are the same

Deficits:
- Query optimization becomes undecidable, due to method calls in queries
- good physical layer design is difficult to determine: object applications tend to do a lot of “pointer chasing”
- lack of a solid theoretical basis
The Database Design Process for an OODB Application

1. Data Requirements
2. Processing Requirements
3. Conceptual Schema (ER model)
4. Transaction Design
5. Schema Design: class declarations (ODL)
6. Method Implementation (OML)
7. Pragmas give DBMS “hints” regarding data layout
8. Application program coding (OML with embedded OQL)
9. Transaction implementation (embedded in application program)
10. Executable application
**ODL Schema Definition**

class Professor
(extent professors)
{
    attribute string name;
    attribute short ssn[9];
    relationship set<Student> advises
        inverse Student::advisor;
    relationship Department dept
        inverse Department::faculty;
    float avgGPAofAdvisees();
};

class Student
(extent students)
{
    attribute string name;
    relationship Professor advisor
        inverse Professor::advises;
    float GPA();
};

class Department
(extent departments key name)
{
    attribute string name;
    relationship College college
        inverse College::depts;
    relationship Set<Professor> faculty
        inverse Professor::dept;
};

Key = attribute must be unique for all objects.
This is **not** a primary key.

The primary key for all classes is the system defined **OID**.
OID = Object Identifier
Object Query Language

```
select s.name
from students s
students is the extent of class STUDENT

select s.name
from students s
where s.gpa() >= 3.0

select struct(name: s.name,
              GPA: s.GPA())
from students s

select s.name
from professors p,
    p.advises s
where p.name = "Jill Pebble"
    and s.GPA() >= 3.0

The problem with query optimization:
What objects are accessed by the following query?

select p.name
from professors p
where p.avgGPAofAdvisees() >= 3.0
```
C++/ODL Schema Definition

class Professor : public d_Object
{
    d_String name;
    short ssn[9];
    d_Rel_Set<Student, _advisor> advises;
    d_Rel_Ref<Department, _faculty> dept
    float avgGPAofAdvisees();
};
const char _advisor[] = "advisor";
const char _faculty[] = "faculty";
d_Set<d_Ref<Professor> > professors;

class Student : public d_Object
{
    d_String name;
    d_Rel_Ref<Professor, _advises> advisor;
    relationship Professor advisor
    float GPA();
};
const char _advises[] = "advises";
d_Set<d_Ref<Student> > students;

class Department : public d_Object
{
    d_String name;
    d_Rel_Ref<College, _depts> college;
    d_Rel_Set<Professor, _dept> faculty;
};
const char _depts[] = "depts";
const char _dept[] = "dept";
d_Set<d_Ref<Department> > departments;

- professors, students and departments are global set objects that implement the extent of each class.
- all classes inherit from d_Object: this gives them an OID, inherited constructors and destructors and other functionality to behave as proper database objects
- d_Ref<T> replaces pointers with OID based "smart-pointers"
- d_Rel_Ref<T> and d_Rel_Set<T> understand and maintain inverse relationship semantics
Database Control and Transaction Control

class d_Database
{
    public:
        d_Database();

        enum access_status { not_open, read_write,
            read_only, exclusive };

        void open(char *database_name,
            access_status _status = read_write);

        void close();

        void set_object_name(d_Ref_Any &theObject,
            char* theName);

        void rename_object(char *oldName,
            char *newName);

        d_Ref_Any lookup_object(char * name);
};

• Every application has exactly one global instance of d_Database.

class d_Transaction
{
    public:
        d_Transaction();
        ~d_Transaction();

        void begin();

        void commit();

        void abort();

        void checkpoint();
};

• Transactions determine when database state changes are made permanent (committed) or removed (aborted).
Relationship Maintenance

Suppose Moh changes advisors from Dr. Brown to Dr. Green:

d_Ref<Student> moh =
    students.select_element("name='Moh'");

d_Ref<Professor> green =
    students.select_element("name='Dr. Green'");

moh.advisor = &green;

Is the database consistent?
Object Definition Language (ODL):

class PERSON
  (extent persons)
  {
    attribute string name;
    attribute string addr;
    relationship set<PET> owns inverse PET::owner;
  };

class PET
  (extent pets)
  {
    attribute string name;
    attribute string license;
    relationship PERSON owner inverse PERSON::owns;
  };

class DOG extends PET
  (extent dogs)
  {
    // add attributes for DOG
  };

ODL is a “language neutral” specification language for objects.
Language Specific Bindings

For supported OOPLs (C++, Java, Smalltalk) there is a defined translation from the ODMG object model and ODL to the object model of the OOPL.

The ODL preprocessor translates the ODL class declarations to declarations in the target language.

**ODL:**

class PERSON
(extent persons)
{
    attribute string name;
    attribute string addr;
    relationship set<PET> owns inverse PET::owner;
};

class PET
(extent pets)
{
    attribute string name;
    attribute string license;
    relationship PERSON owner inverse PERSON::owns;
};

**C++:**

const char _owns_inv = “owner”;
const char _owner_inv = “owns”;

class PERSON : public d_Object {
private:
    d_String name;
    d_String addr;
    d_Rel_Set<PET, _owns_inv> owns;
};
d_Set<d_Ref<PERSON> > persons;

class PET : public d_Object {
private:
    d_String name;
    d_String license;
    d_Ref_Ref<PERSON, _owner_inv> owner;
};
d_Set<d_Ref<PET> > pets;
**Inheritance of “Persistence Capability”**

All classes inherit from the system defined class `d_Object`.

This class defines everything that the DBMS needs to know about an object to make it behave as a database object.

- **object identifier (OID):** This replaces the pointer as the means of identifying a particular object.
- **object size**
- **modified or dirty flag:** indicates if an object’s state has changed from the value currently stored on disk
- **locking information:** controls access to objects from multiple applications
- **object’s class**

```
OID: SETDOG003332
{  DOG990022,
   DOG443200 }
OID: DOG990022
name: “Fido”
license: “A11233”

OID: SETDOG003332
{  DOG990022,
   DOG443200 }
OID: DOG990022
name: “Fido”
license: “A11233”
OID: DOG443200
name: “Spot”
license: “B99233”
```

*buffer manager* is responsible for maintaining memory/disk transfer (paging)
References: “Smart Pointers”

References behave like C++ pointers, but they use OIDs rather than memory addresses.

d_Ref<PET> pet1;

The object pet1 is similar to a C++ pointer, except it holds the OID of some object of type PET.

It is also smart enough to know if the object that it references has been copied to the application program’s memory.

pet1->DisplayName();

If pet1 refers to the object with OID DOG443200, then that object needs to be copied into memory before the method can be executed.

dereferencing pet1 causes the buffer manager to copy the object it refers to into memory
**Extents**

The ODL preprocessor automatically generates Set objects that hold all instances of a class:

```
Set<Ref<PERSON> > persons;
Set<Ref<PETS> > pets;
```

This allows us to easily get at all objects for any class, by iterating on the extent.

In C++ application code:

```cpp
d_Iterator<d_Ref<PET> > pet_iter = pets.create_iterator();

d_Ref<PET> p;
while (pet_iter.next(p))
    cout << p->name;
```

The constructors and destructors for each class need to interact with the extent object.
Using a C++ OODBMS