ODMG 3.0 Notes

CP465 Databases II

Dr. Ilias S. Kotsireas

ikotsire@wlu.ca

Wilfrid Laurier University

1. Overview
2. Object Model
3. Object Specification Languages
   (a) Object Definition language, ODL
   (b) Object Interchange Format, OIF
4. Object Query Language, OQL
5. ODMG C++/Java Bindings

Overview

ODMG* is a standard implemented by:

- object database management systems, ODBMSs (store objects directly)
- object-to-database mappings, ODMs (convert objects in relational and then store them)

ODBMSs and ODMs are referred to with the common term: Object Data Management Systems, ODMSs

Importance of a Standard

Lack of standard for storing objects in databases $\Rightarrow$ limitations to object application portability across database systems.

ODMG enables many vendors to support and endorse a common object interface to which customers write their database applications.

*Object Data Management Group

ODMG Goals:

1. define a set of spec/tions that allow a developer to write portable applications
2. data scheme, progr. lang. binding, data manip/tion, query languages must be portable
3. source code portability
4. combining the strongest features of the products currently available

ODMG member companies cover almost the entire spectrum of the ODMS industry.

The ODMG standard has become the de facto standard for this industry.

ODMG Scope/Aims:

An **ODMS** transparently integrates database capability with the application programming language.

We distinguish two kinds of ODMSs:

ODBMS: is a DBMS that integrates database capabilities with object-oriented programming language capabilities.

ODM: is a system that integrates relational (or other non-object DBMSs) with object-oriented programming language capabilities.

Both type of ODMS make database objects appear as programming language objects, in one or more existing programming languages.

An ODMS extends the progr. lang. with persistent data, concurrency control, data recovery, queries and other database capabilities.

ODMG 3.0 Major Components

1. **Object Model**:
   based on the OMG Object Model, with added components, such as relationships

2. **Object Specification Languages**:
   (a) **ODL**: specification language used to define the object types that conform to the ODMG Object Model
   (b) **OIF**: specification language used to dump/load the current state of an ODMS to/from a file or a set of files

3. **Object Query Language, OQL**:
   is a declarative (non-procedural) language for querying and updating ODMS objcts.
   based on SQL, but supports more powerful capabilities

4. **ODMG C++ (Java/Smalltalk) Binding**:
   how to write portable C++ code that manipulates persistent objects: C++ OML (object manipulation language)
   a version of ODL that uses C++ syntax
   a mechanism to invoke OQL
   procedures for operations on ODMSs and transactions

Object Model

The Object Model specifies the constructs supported by an ODMG-compliant ODMS.

The Object Model specifies the kinds of semantics that can be defined explicitly to an ODMS:

(a) determine the characteristics of objects

(b) how objects can be related to each other

(c) how objects can be named and identified

Constructs supported by an ODMS:

1. basic modeling primitives:
   object (has a unique ID), **literal** (has no ID)

2. obj./lit. are categorized by their **types**
   all elements of a given type have a common set of properties/behavior/operations
   An object is an **instance** of its type

3. the **state** of an object is defined by the values it carries for a set of properties (attributes of the object, relationships btw. the object and one or more other objects)
   The state of an object can change over time.

4. the **behavior** of an object is defined by the set of operations that can be executed on (or by) the object.
   Operations may have a list of I/O parameters, each with a specified time.
   Operations may also return a typed result.

5. An ODMS stores objects, enabling them to be shared by multiple users and applications.
   An ODMS is based on a **schema** that is defined in ODL.
   An ODMS contains instances of the types defined by its schema.

The ODMG Object Model specifies the meaning of:

objects, literals, types, operations, properties, attributes, relationships etc.

An application developer uses the constructs of the ODMG Object Model to define the object model for their application.

The application's object model specifies particular types,
(e.g. Document, Author, Publisher, Chapter)
and the operations/properties of each of these types.

The application's object model is the ODMS's (logical) schema.

The ODMG Object Model includes significantly richer semantics than the relational model:

it declares relationships and operations explicitly.

Types:
Specifications & Implementations

A definition of a type has two aspects:

(1) an (external) **specification**
(2) one or more (internal) **implementations**

The specification defines the external characteristics of the type, visible to the users:

(a) operations that can be invoked on it instances
(b) properties (state variables) whose values can be accessed
(c) exceptions that can be raised by its operations

An implementation defines the internal aspects of the objects of the type:
implementation of the type's operations.

An implementation of a type is determined by a language binding.

An external specification of a type consists of an implementation-independent, abstract description of the operations exceptions and properties that are visible to the user of the type.

An **interface definition** is a specification that defines only the abstract behavior of an object type.

A **class definition** is a specification that defines the abstract behavior and abstract state of an object type.

A **class** is an extended interface with information for ODMS schema definition.

A **literal definition** defines only the abstract state of a literal type.

Example:

```
interface Employee { ... };
class Person { ... };
struct Complex {float re; float im; };
```

`interface Employee` defines only the abstract behavior of Employee objects

`class Person` defines both the abstract behavior and the abstract state of Person objects

the `struct Complex` defines only the abstract state of Complex number literals

In addition to the **struct** definition and the primitive literal datatypes **boolean, char, short, long, float, double, octet, string**, ODL defines declarations for user-defined **collection, union, enumeration** literal types.

Implementation of an Object Type

An implementation of an Object Type consists of a **representation** and a set of **methods**.

representation = data structure derived from the type's abstract state by a language binding:
$\forall$ property contained in the abstract state
$\exists$ an instance variable of an appropriate type defined

methods = procedure bodies that are derived from the type's abstract behavior by the language binding:
$\forall$ operation defined in the type's abstract behavior
$\exists$ a definition of a method

This method implements the externally visible behavior of an object type.

A method can read or modify the rep. of an object's state, or invoke operations defined on other objects.

There can be methods in an implementation that have no direct counterparts to the operations in the type's specification.

The internals of an implementation are not visible to the users of the objects.

Each language binding also defines an implementation mapping for literal types.

Some languages (e.g. C++) have constructs that can be used to represent (structured) literals directly, i.e. `struct`.

Some other languages (e.g. Java/Smalltalk) do not have such constructs. These language bindings map each literal type into constructs that can be directly supported using object classes.

C++ and Java handle directly floating-point datatypes, so they would bind the `float` elements of the `Complex` literals accordingly.

Encapsulation

The disctinction/separation between specification and implementation is how the Object Model reflects **encapsulation**.

ODL is used to specify the external specifications of types in application object models.

Language bindings define the C++/Java constructs used to specify the implementations of these specifications.

A type can have more than one implementation.
(i.e. one C++ and one Java)
(i.e. two C++ for two different machine architectures)

Usually, only one implementation is used any particular program.

Separating the specifications from the implementations keeps the semantics of the type from being tangled with representation details.

Object-oriented languages, C++, Java, Smalltalk have **classes**.

These are implementation classes and should not be confused with the **abstract classes** defined in the Object Model.

Each language binding defines a mapping between abstract classes and its language's implementation classes.

Subtyping & Inheritance of Behavior

The ODMG Object Model includes inheritance-based type-subtype relationships, commonly represented in graphs.

Each node is a type and each arc connects one type (supertype) to another type (subtype).

The type-subtype relationship is also called an **is-a** or an **ISA** or a **generalization/specialization** relationship.

The supertype is the more general type, the subtype is the more specialized.

Example

```
interface Employee { ... };
interface Professor : Employee { ... };
interface AssociateProfessor : Professor { ... };
```

Professor inherits from Employee.

AssociateProfessor inherits from Employee and Professor.

An instance of the subtype is also an instance of the supertype.

An object's **most specific type** is the type that describes all the behavior and properties of the instance.

The most specific type of an AssociateProfessor object is the AssociateProfessor interface. This object carries additional type information from the Professor and Employee interfaces.

An AssociateProfessor instance has all behaviors defined in the AssociateProfessor interface and inherits all behaviors defined in the Professor, Employee interfaces.