Object Model

CSC 436 – Fall 2003

* Notes kindly borrowed from Dr. Azz Art-Rahash, School of Computing, IS & Math, South Bank University

Object Database Management Group

- Since its inception in 1991, the ODMG has grown to over 20 members and includes every ODBMS vendor, plus several tool vendors, consulting firms, and corporate end users.
- The voting member companies are GemStone Systems, IBEX Computing SA, O2 Technology, Object Design, Objectivity, POET Software, UniSQL and Versant Object Technology.
- Reviewer member companies are Andersen Consulting, CERN, Electronic Data Systems (EDS), Fujitsu Software Corporation, Hitachi, Lockheed Martin, Microsoft, MITRE Corporation, NEC Corporation, ONTOS, Persistence Software, Sybase, Unidata and VMARK Software.
- Voting membership requires that a company must have developed and must commercially ship an object DBMS product. The company must commit 20 percent of the time of a senior object DBMS expert and must commit to implement the ODMG standard.
- Reviewer membership requires a company to commit 10 percent of the time of a senior object DBMS expert, but does not require that the company ship an object DBMS product.
Object Model

//From ODMG-93
//The basic modelling primitive is the object
//The behaviour of objects is defined by a set of operations that can be executed on an object of the type (e.g., you can ‘discharge’ an object of type Patient)
//The state of objects is defined by the values they carry for a set of properties:
  – attributes of the object itself
  – relationships between the object and one or more other objects
//Objects can be categorised into types. All objects of a given type exhibit common behaviour and a common range of states

Type Interface

//A type has one interface and one or more implementations

//The type interface defines the external interface supported by instances of the type
  – the properties and
  – the operations that can be invoked on them

//An implementation defines data structures to physically represent instances of the type and the methods that operate on those data structures to support the state and behaviour defined in the interface
Types and Instances

A type defines the state (properties) and behaviour (operations) of its instances, collectively as its characteristics.

Types are themselves objects and have properties:
- **Supertypes**: types are related in a subtype/supertype graph (like ISA in the E-R model).
- **Extents**: the set of all the instances of a type.
- **Keys**: the property or set of properties whose values can uniquely identify the instances of a type.

Basic Type Hierarchy

- **Denotable Object**
  - **Object**
  - **Literal**
- **Characteristic**
  - **Property**
    - **Attribute**
    - **Relationship**
- **Operation**
Inheritance

- A subtype inherits all of the characteristics of its supertypes
- Subtypes also may define additional characteristics that apply only to its instances
- An instance of a subtype may be treated as an instance of its supertype
- Abstract types only define characteristics inherited by their subtypes; they cannot be directly instantiated
- A type can have multiple supertypes
  - may result in possible name clashes
  - subtype must redefine the name of one of the inherited characteristics

Example Type Hierarchy

interface Employee {
  supertype: Atomic_object;
  id: Integer;
  name: String;
  salary: Float; ...}
interface Staff {
  supertype: Employee;
  position: String;
  hours: Struct<from:Time, to:Time>;}
interface Student {
  supertype: Atomic_Object;
  Stud_id:integer; ...}
interface TA {
  supertype: Employee, Student;
  assists: Course inverse Course:ta;
  emp_name redefines Employee.name}
**Extents**

- The extents of a type is the set of all instances of the type
- Extents are optional; not all types must have an extent
- ODBMS automatically maintains the extent as instances are created and deleted
- Each instance of type A will be a member of the extent of A
- If type A is a subtype of type B, then the extent of A will be a subset of the extent of B

**Implementations and Classes**

- A type has one more implementations
- An implementation has a name and consists of a representation (data structures and a set of methods (procedures bodies)
- There is one method for each operation defined in the type interface
- There may be additional methods and data structures that have no counterpart in the type interface
- The combination of the type interface and one of its implementations is termed a class
- In a C++ class, the private part corresponds to the implementation
Why Multiple Implementations?

- To support databases that span networks which include machines with different architectures
- To support mixed-language and mixed-compiler environments
- To meet different performance versus space versus recoverability trade-offs (e.g., set as a B-tree or linked list)
- Which implementation an object uses is specified at object creation time

Objects

- Objects are things which characteristics are predicated; that is, objects have state and behaviour
- Objects also have identity
  - the identity of a literal is typically represented by its value
  - the identity of an object is an object identifier (OID)
- Two orthogonal lines along which objects can be decomposed:
  - mutable (Object) versus immutable (Literal)
  - atomic versus structured
Type Object

Instances of type Object are mutable
- the values of attributes may change
- the relationships in which they participate may change
- the identity of the object remains invariant

An object’s OID distinguishes the object from all other objects in the database in which it exists

Objects may have one or more names; each name can be used to identify a single object

Subtypes
- Atomic_Object
- Structured_Object

Type Object (cont.)

Built-in properties:
- has_name?: Boolean
- names: Set<String>
- type: Type

Built-in operations:
- delete()
- same_as? (oid: Object_id) -> b: Boolean

Object Lifetime
- transient
  » declared in a procedure and returned when the procedure returns
  » allocated by the PL runtime and removed when the process ends
- persistent: allocated by the ODBMS runtime
Structured Objects

// The type Structured_object are mutable
  – Structure<e₁: T₁, ..., eₙ: Tₙ, >
  – Collection<T>

// Structures have a fixed number of named slots each of which contains an object or a literal

// Collections contain an arbitrary number of elements, do not have named slots, and contain elements that are all instances of the same type

// Built-in Collection subtypes:
  – Set<T>: unordered collection of unique elements
  – Bag<T>: unordered collection of elements
  – List<T>: collection of elements ordered by insertion
  – Array<T>: collection of elements ordered by property

Literals

// Literals are objects whose instances are immutable

// Two subtypes
  – Atomic_Literal
  – Structured_Literal

// All instances of atomic literals implicitly pre-exist

// Subtypes of atomic literals
  – Integer
  – Float
  – Boolean
  – Character
StructuredLiteral

// Two subtypes
   /// Immutable_Collection<T>
   /// Immutable_Set<T>
   /// Immutable_Bag<T>
   /// Immutable_List<T>
   /// Bit_String
   /// Character_String
   /// Immutable_Array<T>
   /// Enumeration
   /// Immutable_Structure
   /// Date
   /// Time
   /// DateTime
   /// Interval

// Cannot update the value of a structured literal

Modelling State - Properties

// An object type defines a set of properties through which users of instances of the type can interrogate and manipulate the state of these instances

// Two kinds of properties:
  – Attributes: defined on one object type and take literals as their values
  – Relationships: defined between two object types, both mutable objects
Attributes

The declaration of an object type includes declarations of each of the attribute types for which an instance of the object type carries a specific value.

Example attribute type definitions for type Person:
- age: Integer
- sex: Enumeration(male, female)
- height: Integer

Built-in operations defined on attributes:
- set_value(new_value: Literal)
- get_value() -> existing_value:Literal

Relationships

Relationship types are defined between mutable object types.
Relationships have no names themselves; instead traversal paths are defined for each direction of traversal.
Inverse clause used to indicate the common relationship between traversal paths.

Example:
interface Student
{ ...
takes: Set<Course> inverse Course::is_taken_by
}
interface Course
{ ...
is_taken_by: Set<Student> inverse Student::takes
}
The potential behavior of instances of an object type is specified as a set of operations.

The operation signature of an operation in the object type definition includes:
- argument names and types
- exceptions potentially raised
- types of the values returned, if any

Operation names must be unique only within a single type definition.

Overloading operations are operations defined on different types using the same name.

Specific operation is selected based on the most specific type of the object supplied.

Overloaded Operations: Example

```java
interface Employee {
    supertype: Atomic_Object;
    extent: employees;
    id: Integer;
    name: String;
    hourly_wage: Float;
    salary();
    ...}

interface Salesperson {
    supertype: Employee;
    extent: salespeople;
    commissions: Set<Float>;
    salary();
    ...}
```

Consider the function to calculate the sum of all employees salaries (early versus late binding).
Type Compatibility

- Every object has a type
- Every operation requires typed operands
- Two objects have the same type only if they are instances of the same named type
- If B is a subtype of A, then an instance of Type B can be assigned to an object of type A, but not the reverse

Metadata

- All types are instances of type Type
- Type Type is both a subtype and an instance of Atomic_Object
- Can use standard DML to interrogate the metadata
- Type Type has instance properties, such as
  - has_operations: Set<Operations>
  - has_properties: Set<Property>
  - has_supertypes: Set<Type>
  - name: String
  - extent: Set<Atomic_Object>
- Each type definition specifies values for these properties
Type Database

A database provides storage for persistent objects of a given set of types

The database schema is the set of type definitions

Each database is an instance of the type Database, which supports the following operations
- open()
- close()
- contains_object?(oid:Object) -> b:Boolean
- lookup_object(oid:Object) -> b:Boolean

Transactions

The object model supports a nested transaction model
- simple way of undoing changes made to stored persistent data
- method to protect against network failure for remote operations

Transaction::begin() --> t:Transaction;
... Transaction::begin() --> x:Transaction;
... Transaction::begin() --> y:Transaction;
... if minor_error then y.abort()
... if major_error then y.abort_to_top_level()
... y.commit()
... x.commit()
... t.commit()
Catell R.G.G. ed.:

*The Object Database Standard: ODMG - 93,*