

Databases Illuminated

Chapter 8

The Object-Oriented Model

Why OO?

- Traditional relational model does not represent complex data and relationships well
- Need additional support for advanced applications
- OO paradigm widely used for programming
- OO database provides persistent objects to correspond to temporary objects in programs
- Examples: Objectivity, GemStone, ObjectStore, Versant

OO Data Concepts

- An **object** has a **state** and a unique **identifier**
 - Similar to an entity, but has methods in addition to attributes (data elements)
 - Objects are encapsulated – data and methods form a unit with restricted access
 - Only object's methods have access to its data
 - Object's interface is visible to outside world
- A **literal** has a state but no identifier – can be atomic (values of built-in types) or structured

Classes

- A **class** is a set of objects having the same structure-
same variables with the same datatypes, same methods,
and same relationships
 - Roughly equivalent to an entity type
 - Includes data elements, operations and relationships
 - Datatypes can be predefined atomic types (integer, real, char, boolean), more complex types, or user-defined types
- Set of objects in a class is called the **extent** (like extension)
 - Class extent is roughly equivalent to an entity set
 - Objects in the class (**instances, object instances, objects**) are similar to entity instances

Defining a Class

- Defined by listing components
 - **Data members** (attributes, instance variables)
 - Member **methods** (functions or procedures that belong to the class)
 - **Relationships** that the class participates in

Simplified Example:

```
class Person {  
    attribute string name;  
    attribute string address;  
    attribute string phone;  
    void setName(string newName); // method  
    string getName( ); //method  
    relationship Job hasjob; //relates to Job class  
}
```

Writing and Using Methods

- Method written in OO language-Java, C++, etc.

- Ex:

```
void setName(string newName)
{
    self.name = newName;
}
```

- User's program may have a Person object
Person firstPerson = new Person();
- Invoke method using Person object
firstPerson.setName('Jack Spratt');
- firstPerson becomes "calling object", referred to as *self* in method

Class Hierarchies

- Classes organized into **class hierarchies**
 - **Superclasses (base classes and subclasses)**
 - Each subclass has an *isa* relationship with its superclass
- similar to specialization and generalization in the EER model
- Subclasses inherit the data members and methods of their superclasses, and may have additional data members and methods of their own
- Hierarchy diagram-See **Figure 8.2**
 - Represent classes as rectangles
 - Connect subclasses to *isa* triangle, and connect triangle to superclass

UML Class Diagrams-1

- **Unified Modeling Language (UML) class diagrams - See Figure 8.5**
 - Rectangles for classes- 3 sections: class name, attributes, methods
 - Relationships – 2 types
 - Association
 - for uni- or bi-directional relationships between distinct classes (Ex Student – Faculty relationship)
 - represented by directed line connecting rectangles, with optional name
 - Any descriptive attributes of association in box connected to association line by dotted line (Ex *grade* in Student - ClassSection)
 - Rolenames can appear on line, but placement is opposite that in E-R (Ex. Student *hasAdvisor* role appears next to Faculty)
 - Aggregation
 - Connects parts to a whole, described by “is part of” (Ex ClassSection to Course)
 - Represented by line with diamond on side of aggregate
 - Can be given a name, or interpreted as “has”
 - Reflexive association or reflexive aggregation shown by line back to same class, with rolenames (Ex Course)

UML Diagrams-2

- **Multiplicity indicators** show cardinality & participation
 - min..max, but place opposite to placement in E-R; no ()
 - Use * for M; use 1 for 1..1
- **Generalization hierarchies**
 - Lines connect subclasses to superclass, with triangle at end, pointing to superclass
 - Filled triangle for overlapping subclasses
 - Open triangle (outline only) for disjoint subclasses
 - Can write constraints in curly braces on line near triangle (Ex. {overlapping, optional} near Person class rectangle)
- Weak entity represented by rectangle with line to strong entity, with discriminator written in box below strong entity (Ex. *date* is discriminator for Evaluation)

ODMG Model

- Object Database Management Group
- Group of vendors
- Developed standards for OO databases
- Standards for
 - Object model itself
 - **Object definition language (ODL)**
 - **Object query language (OQL)**
 - Language bindings for C++, Java, Smalltalk

ODL-Class Declarations

- See Figure 8.6
- **Class declarations**
 - Begin with word **class**, then *classname*
 - Optional **extent** and **key** declarations in parentheses
 - List of **attributes**, methods, and relationships, all enclosed in curly braces
- **extent**
 - Set of object instances for that class that are stored in the database at a given time; the extension
 - Like the name of the file where the objects in the class are stored

ODL-Attribute Types

- Attribute types – atomic or structured
 - **Atomic types** - integer, float, character, string, boolean, and enumerated types
 - **Enumerated types** - keyword **enum**, name of the type, curly braces with a list of literals for the type, and the name of the attribute with that type

Ex: attribute enum FacultyRank{instructor, assistant, associate, professor} rank;

ODL-Structured Types

- Keyword **Struct**, the name of the type, curly braces with each attribute and its datatype, then the identifier of that type

Ex: **attribute Struct Addr**(string street, string city, string state, string zip) address;

- If type used again for other classes, identify the class it was defined in, using the **scoped name-class name**, double colon, and the type name

Ex: If we defined Addr type in Person class, and we need a NewAddress field in another class in the same schema, write **attribute newAddress Person::Addr**

Collection Types

- **Set**-finite number of unordered values of one datatype, specified in angled brackets, `Set<typename>`
- **List**-finite list of elements of a single type, written `List<datatype>`
- **Array**-set of elements all of the same type, with an index indicating position of each element; constructor requires datatype and number of elements, as in `Array<float, 5>`
- **Bag** or multiset-similar to a set, but permits duplicate values, written `Bag<datatype>`
- **Dictionary**-constructor has the form `Dictionary <K,V>` where `K` and `V` are some datatypes- used to construct pairs of values, `<k,v>` where `k` is a key type and `v` is some range type

Relationships

- Represented by **references**
- System stores and maintains the references
- Ex 1: in Faculty class
relationship Department belongsTo Inverse Department::hasFaculty;
 - Defines relationship *belongsTo* connecting Faculty to Department
 - A “one” relationship - only one Department reference per Faculty object
 - There is also an inverse relationship *hasFaculty* in Department (bidirectional relationship)
 - Not all relationships have inverses – unidirectional is OK
- Ex 2: in Student class
relationship Set<ClassSection> takesClass Inverse ClassSection::hasStudent;
 - Defines relationship *takesClass* connecting Student to ClassSection
 - Each Student object has a **set** of references to ClassSection objects-a “many” relationship
- Cardinality of relationship shown by whether or not the word “Set” appears in the relationship specification

Methods

- A function or procedure for members of the class
- Declarations specify the **signature** - the name of the method, the return type (if any), and the number and type of parameters, identified as IN, OUT, or IN/OUT
- Two methods for the same class may have the same name but if their signatures are different, they are different methods
- Actual code for the method is not part of the ODL, but written in a host language
- May be **overloaded** - same method name used for different classes, with different code for them
- Class member methods are applied to an instance of the class

Subclasses

- Keyword **class**, *subclass name*, keyword **extends** *superclass name*

Ex: **class Student extends Person**

- Subclass inherits all attributes, relationships, methods
- Can have additional properties of its own
- For multiple inheritance, add a colon and the name of the second superclass
- Second superclass must be an **interface**, a class definition without an associated extent

Ex: If Student also inherited from a Customer interface
class Student extends Person:Customer

Relationship Classes

- For binary M;M relationships without descriptive attributes, use relationship clause in classes, with Set specification in both directions
- Binary M:M relationships with descriptive attributes
 - Cannot be represented by sets in both directions, since that leaves no place for descriptive attributes
 - Set up a class for the relationship, place the descriptive attributes as attributes of the new class, and define two one-to-many relationships between the new class and the two original classes
 - See **Grade** class in **Figure 8.6**
- For ternary or higher-order relationships, create a class for the relationship itself
 - New relationship class definition includes three or more relationships that connect the new class to the originally-related classes
 - Also list any descriptive attributes

Keys

- Keys are optional in ODL
- System uses unique object identifier (OID), automatically given to each object instance, to tell instances apart
- Designer can identify any candidate keys as well
- Done at the beginning of the class declaration within the same parentheses as the extent declaration
- Key may be a single attribute or a composite, identified by parentheses around the component attribute names

OQL-Object Query Language-1

- Syntax similar to SQL, but operates on objects, not tables
- Form for queries is

```
SELECT expression list
FROM list of variables
WHERE condition;
```
- *expression list* can contain the names of attributes using dot notation, essentially invoking automatic get method, as in

```
SELECT s.stuld, s.credits
FROM students s;
```
- Can use methods in the expression list –get the result of applying the method

```
SELECT p.getName( )
FROM people p;
```
- Can use relationship in the expression list- retrieves the object or set of objects related to the calling object through the relationship

```
SELECT s.stuld, s.takesClass
FROM students s
WHERE s.stuld = 'S999';
```

OQL-FROM line

- List of variables -similar to defining an alias in SQL
- List the name of an extent, such as students or people, and an identifier for the name of the variable, such as s or p
- Variable is actually an **iterator variable** that ranges over the extent
- Alternate forms for declaring an iterator variable
 - FROM students s
 - FROM s in students
 - FROM students as s

OQL-WHERE line

- Must be boolean expression having constants and variables defined in the FROM clause
- Can use <, <=, >, >=, !=, AND, OR and NOT
- Does not eliminate duplicates; returns a bag
- To eliminate duplicates, add DISTINCT
- Can optionally add ORDER BY

Developing an OO Database

- See Figure 8.7
- Natural extension of application development in an object-oriented programming environment
- Language bindings specified in ODMG standard for C++, Java, and Smalltalk
- Difference between program objects and database objects is **persistence**
- OODBMS provides facilities to make program objects persist, and provides access to database objects for manipulation within programs

Defining the Schema

- Designer defines the schema using a data definition language such as ODL or an OO programming language such as C++
- Class definitions can be standard C++ (or other language) that has been extended to provide persistence and to support relationships between objects, as well as inheritance
- Persistence is provided by making all objects that are to be persistent inherit from a class provided by the OODBMS just for that purpose