Outline

- Object-orientation and databases
- Object-oriented model: ODL
- Object Query Language

Object-Oriented DBMS’s

- ODMG = Object Data Management Group: an OO standard for databases.
- ODL = Object Description Language: design in the OO style.
- OQL = Object Query Language: queries an OO database with an ODL schema, in a manner similar to SQL.

ODMG Compliant Databases

- Jasmine (Computer Associates)
- TITANIUM (MicroDB Systems)
- ObjectStore (EXcelon)
- Objectivity (Objectivity)
- POET Object Server (POET Soft.)
- Versant ODBMS (Versant Corp.)
- OO databases account for less than 2% of database market.

ODL Overview

- Class declarations (interfaces).
- Interface includes:
  1. Name for the interface.
  2. Key declaration(s), which are optional.
  3. Extent declaration = name for the set of currently existing objects of a class.
  4. Element declarations. An element is an attribute, a relationship, or a method.

ODL Class Declarations

interface <name> {
    elements = attributes, relationships, methods
}

- Element Declarations
  attribute <type> <name>;
  relationship <rangetype> <name>;
- Relationships involve objects; attributes involve non-object values, e.g., integers.
Method Example

float gpa(in name) raises(noGrades)

- float = return type.
- in: indicates the argument (a student name) is read-only.
- Other options: out, inout.
- noGrades is an exception that can be raised by method gpa.

ODL Relationships

- Only binary relations supported.
- Multiway relationships require a "connecting" class, as discussed for E/R model.
- Relationships come in inverse pairs.
- Example: Sells between beers and bars is represented by a relationship in bars, giving the beers sold, and a relationship in beers giving the bars that sell it.

More ODL Relationships

- Many-many relationships have a set type (called a collection type) in each direction.
- Many-one relationships have a set type for the one, and a simple class name for the many.
- One-one relations have classes for both.

Beers-Bars-Drinkers Example

```java
interface Beers {
  attribute string name;
  attribute string manf;
  relationship Set<Bars> servedAt inverse Bars::serves;
  relationship Set<Drinkers> fans inverse Drinkers::likes;
}
```

- An element from another class is indicated by <class>::
- Form a set type with Set<type>.

Bars ODL

```java
interface Bars {
  attribute string name;
  attribute Struct Addr (string street, string city, int zip) address;
  attribute Enum Lic {full, beer, none} licenseType;
  relationship Set<Drinkers> frequents inverse Drinkers::frequents;
  relationship Set<Beers> serves inverse Beers::servedAt;
}
```

- Structured types have names and bracketed lists of field-type pairs.
- Enumerated types have names and bracketed lists of values.

Drinkers ODL

```java
interface Drinkers {
  attribute string name;
  attribute Struct Bars::Addr address;
  relationship Set<Beers> likes inverse Beers::fans;
  relationship Set<Bars> frequents inverse Bars::customers;
}
```

- Note reuse of Addr type.
ODL Type System

- Basic types: int, real/float, string, enumerated types, and classes.
- Type constructors: Struct for structures and four collection types: Set, Bag, List, and Array.
- Limitation on nesting:
  - Relationships: class – collection
  - Attributes: basic – struct – collection

Many-One Relationships

- Don’t use a collection type for relationship in the “many” class.
- Drinkers have favorite beers:
  ```java
  interface Drinkers {
      ...
      relationship Beers favoriteBeer
          inverse Beers::realFans;
      ...
  }
  ...
  ```
- Also add to Beers:
  ```java
  relationship Set<Drinkers> realFans
      inverse Drinkers::favoriteBeer;
  ```

Example: Multiway Relationship

- Consider a 3-way relationship bars-beers-prices. We have to create a connecting class BBP.
  ```java
  interface Prices {
      attribute real price;
      relationship Set<BBP> toBBP
          inverse BBP::thePrice;
      }
  
  interface BBP {
      relationship Bars theBar inverse ...
      relationship Beers theBeer inverse ...
      relationship Prices thePrice
          inverse Prices::toBBP;
      }
  ```

Example (contd.)

- Inverses for theBar, theBeer must be added to Bars, Beers.
- Better in this special case: make no Prices class; make price an attribute of BBP.
- Notice that keys are optional.
  - BBP has no key, yet is not “weak.” Object identity suffices to distinguish different BBP objects.

Roles in ODL

- Names of relationships handle roles. E.g. Spouses and Drinking Buddies
  ```java
  interface Drinkers {
      attribute string name;
      attribute Struct Bars::Addr
          address;
      relationship Set<Beers> likes
          inverse Beers::fans;
      relationship Set<Bars> frequents
          inverse Bars::customers;
      relationship Drinkers husband
          inverse wife;
      relationship Drinkers wife
          inverse husband;
      relationship Set<Drinkers> buddies
          inverse Drinkers::buddies;
      }
  ```
- Notice that Drinkers:: is optional when the inverse is a relationship of the same class.

ODL Subclasses

- Follow name of subclass by colon and its superclass.
- Example: Ales are Beers with a Color
  ```java
  interface Ales:Beers {
      attribute string color;
  }
  ```
- Objects of the Ales class acquire all the attributes and relationships of the Beers class.
- While E/R entities can have manifestations in a class and subclass, in ODL we assume each object is a member of exactly one class.
Keys in ODL

- Indicate with key(s) following the class name, and a list of attributes forming the key.
- Several lists may be used to indicate several alternative keys.
- Parentheses group members of a key, and also group key to the declared keys.
- Thus, \((\text{key}(a_1, a_2, \ldots, a_n))\) = "one key consisting of all \(n\) attributes." (key \(a_1, a_2, \ldots, a_n\)) = "each \(a_i\) is a key by itself."

Example

```
interface Beers (key name)
{attribute string name ...}  
```

```
interface Courses
(key (dept, number), (room, hours)) 
{"..."}  
```

- Keys are optional in ODL. The object ID suffices to distinguish objects that have the same values in their elements.

Translating ODL to Relations

1. Classes without relationships: like entity set, but several new problems arise.
2. Classes with relationships:
   a) Treat the relationship separately, as in E/R.
   b) Attach a many-one relationship to the relation for the many.

ODL Class Without Relationships

- Problem: ODL allows attribute types built from structures and collection types.
- Structure: Make one attribute for each field.
- Set: make one tuple for each member of the set.
  • More than one set attribute? Make tuples for all combinations.
- Problem: ODL class may have no key, but we should have one in the relation to represent "OID."

Example: ODL

```
interface Drinkers (key name) {
  attribute string name;
  attribute Struct Addr
  {string street, string city, int zip}; address;
  attribute Set<string> phone;
}
```

Example: Relations

```
<table>
<thead>
<tr>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_1</td>
<td>s_1</td>
<td>c_1</td>
<td>z_1</td>
<td>p_1</td>
</tr>
<tr>
<td>n_2</td>
<td>s_2</td>
<td>c_2</td>
<td>z_2</td>
<td>p_2</td>
</tr>
</tbody>
</table>
```

- Surprise: the key for the class (name) is not the key for the relation (name, phone).
- name in the class determines a unique object, including a set of phones.
- name in the relation does not determine a unique tuple.
- Since tuples are not identical to objects, there is no inconsistency!
- BCNF violation: separate out name-phone.
ODL Relationships
- If the relationship is many-one from \( A \) to \( B \), put key of \( B \) attributes in the relation for class \( A \).
- If relationship is many-many, we'll have to duplicate \( A \)-tuples as in ODL with set-valued attributes.
  - Wouldn't you really rather create a separate relation for a many-many-relationship?
  - You'll wind up separating it anyway, during BCNF decomposition.

Example: ODL

```java
interface Drinkers (key name) {
    attribute string name;
    attribute string addr;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Beers favorite
        inverse Beers::realFans;
    relationship Drinkers husband
        inverse wife;
    relationship Drinkers wife
        inverse husband;
    relationship Set<Drinkers> buddies
        inverse buddies;
}
```

Example: Relations

```java
Drinkers (name, addr, beerName, favBeer, wife, buddy)
```
- Not in BCNF; decompose to:
  - Drinkers(name, addr, favBeer, wife)
  - DrBeer(name, beer)
  - DrBuddy(name, buddy)

Motivation for OQL
- Relational languages suffer from *impedance mismatch* when we try to connect them to conventional languages like C or C++.
  - The data models of C and SQL are radically different, e.g., C does not have relations, sets, or bags as primitive types; C is tuple-at-a-time, SQL is relation-at-a-time.

Object Query Language (OQL)
- OQL is an attempt by the OO community to extend languages like C++ with SQL-like, relation-at-a-time dictions.

OQL Types
- Basic types: strings, ints, reals, etc., plus class names.
- Type constructors:
  - Struct for structures.
  - Collection types: set, bag, list, array.
- Like ODL, but no limit on the number of times we can apply a type constructor.
  - Set(Struct()) and Bag(Struct()) play special roles akin to relations.
OQL and ODL

- ODL is the Schema-Definition language for OQL.
- For every class we can declare an extent = name for the current set of objects of the class.
  - Remember to refer to the extent, not the class name, in queries.

Example

interface Bar (extent Bars)
{ attribute string name;
  attribute string addr;
  relationship Set<Sell> beersSold
    inverse Sell::bar; }

interface Beer (extent Beers)
{ attribute string name;
  attribute string manf;
  relationship Set<Sell> soldBy
    inverse Sell::beer; }

Example (contd.)

interface Sell (extent Sells)
{ attribute float price;
  relationship Bar bar
    inverse Bar::beersSold;
  relationship Beer beer
    inverse Beer::soldBy; }

Path Expressions

- Let x be an object of class C.
- If a is an attribute of C, then x.a = the value of a in the x object.
- If r is a relationship of C, then x.r = the value to which x is connected by r.
  - Could be an object or a collection of objects, depending on the type of r.
- If m is a method of C, then x.m(…) is the result of applying m to x.

Examples

- Let s be a variable whose type is Sell.
  - s.price = the price in the object s.
  - s.bar.addr = the address of the bar mentioned in s.
  - Note: cascade of dots OK because s.bar is an object, not a collection.
- Illegal use of dot:
  - b.beersSold.price, where b is a Bar object.
  - Illegal because b.beersSold is a set of objects, not a single object.

OQL Select-From-Where

SELECT <list of values>
FROM <list of collections and typical members>
WHERE <condition>

- Collections in FROM can be:
  1. Extents.
  2. Expressions that evaluate to a collection.
- Following a collection is a name for a typical member, optionally preceded by AS.
Example

- Get the menu at Spoon.
  `SELECT s.beer.name, s.price
  FROM Sells s
  WHERE s.bar.name = "Spoon"
  ` Notice double-quoted strings in OQL.

Example

- Another way to get Spoon’s menu, this time focusing on the Bar objects.
  `SELECT s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Spoon"
  ` Notice that the typical object `b` in the first collection of FROM is used to help define the second collection.

Typical Usage

- If `x` is an object, you can extend the path expression, like `s` or `s.beer` in `s.beer.name`.
- If `x` is a collection, you use it in the FROM list, like `b.beersSold` above, if you want to access attributes of `x`.

Tailoring the Type of the Result

- Default: bag of structs, field names taken from the ends of path names in SELECT clause.
- Example
  `SELECT s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Spoon"
  ` has result type:
  Bag(Struct(  
    name: string,
    price: real)
  )

Rename Fields

- Prefix the path with the desired name and a colon.
  `SELECT beer: s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Spoon"
  ` has type:
  Bag(Struct(  
    beer: string,
    price: real
  ))

Change the Collection Type

- Use SELECT DISTINCT to get a `set` of structs.
  `SELECT DISTINCT s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Spoon"
  ` Use ORDER BY clause to get a `list` of structs.

  ```sql
  SELECT s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Spoon"
  ```
  ```sql
  SELECT DISTINCT s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Spoon"
  ```

Example

joeMenu =
    SELECT s.beer.name, s.price
    FROM Bars b, b.beersSold s
    WHERE b.name = "Spoon"
    ORDER BY s.price ASC;

- ASC = ascending (default); DESC = descending.
- We can extract from a list as if it were an array, e.g.
  cheapest = joeMenu[1].name;