Object-Oriented Database Systems
(ODDBS)

25/2-2002

Contains slides made by
Arthur M. Keller, Vera Goebel

Overview

✓ OO concepts and ODDBS’ properties
  object identity/object identifier (OID), objects & values, extent (instances of class), complex objects, type constructors, operators, types/classes, user-definable types/classes, programming language compatibility (seamlessness), encapsulation (information hiding), type/class hierarchies, inheritance, polymorphism

✓ ODL
  classes, attributes, relationships, methods, type system, extents, keys, inheritance
Data Models & Database System Architectures
- Chronological Overview -

✓ Network Data Models  (1964)
✓ Hierarchical Data Models  (1968)
✓ Relational Data Models  (1970)
✓ Object-oriented Data Models  (~ 1985)
✓ Object-relational Data Models  (~ 1990)
✓ Semistructured Data Models (XML 1.0)  (~1998)

Why Object-Oriented Database Systems?

- concepts of object-oriented systems
- concepts of traditional DBS
- new (nonstandard) requirements for DBS, e.g. complex objects, arbitrary relationships, complex data types, version, ...

object-oriented database systems
Object-Oriented Concepts (General)

- Model mini-world (Universe of Discourse, UoD) as collection of cooperating, related units, called *objects*

- **Abstraction and autonomy:**
  - object: `<value, {operators}>`
  - value: data structure
  - encapsulation (information hiding)
  - request of performance from other objects

- **Classification:**
  - common description (intension)
  - summary of similar objects (extension, class)

- **Taxonomy:**
  - super-/sub-classes
  - inheritance of properties
  - polymorphism
Task of the Data Model

- 

Advantages of OO Data Modeling

- “Naturalness” (compared to traditional data models)
  - meaningful abstraction / high modularity
  - control of complexity
  - separation of interface and implementation
- Evolutionary system design
  - incremental programming
  - reusability
Object-Oriented Database Systems

OODBS = DBS with OO Data Model

✓ OO Database = collection of objects
✓ Object = <OID, value, {operators}>
✓ An object has specific properties....

OODBS and Object Values

Object = <OID, value, {operators}>

Value example:

class athlete {
    text name;
    integer salary;
}

V1 = tuple of (name: "Solskjær", salary: 4.000.000)
V2 = tuple of (name: "Berg", salary: 1.000.000)
V3 = tuple of (name: "Dæhli", salary: 6.000.000)

O1 = <•, V1, •>
O2 = <•, V2, •>
O3 = <•, V3, •>
Object = <OID, value, {operators}>

✓ Value operators:
  ➢ Processing of entire objects - depending on actual structure
  ➢ Modify values

✓ Generic operators:
  ➢ Processing of individual object levels - structure irrelevant

✓ One-level / multi-level operators (composite objects)

✓ Also: selection, navigation, ....

Properties of OODBS

Must have:
✓ object identity
✓ complex (composite) objects
✓ types/classes
✓ user-definable types
✓ language completeness
✓ encapsulation
✓ inheritance: type/class hierarchies
✓ polymorphisms: overloading/overriding/late binding

⇒ ... all orthogonal.

Should have:
✓ object versions
✓ specific realization (implementation) of concepts
✓ distribution (client/server architectures)
✓ specific processing aspects, e.g., new transaction mechanisms
✓ rule-based mechanisms (active/deductive features)

⇒ ... and many more
Object Identifier (OID)

✓ Objects exist independently of their (current) values
  - Modifications of any kind result in “same” object
  - No misleading references to objects
  - “identity” /= “equality” (both needs to be expressible)

✓ OIDs cannot (reliably) be based on ordinary values provided by application (value orientation)
  ... but on surrogates: OIDs are
  - (system-wide) unique
  - unchanged during object lifetime
  - not reused after object deletion
  - generally system-managed

“generic” object operators:
- object comparison
- object retrieval
- ...
  based on OID

Complex (Composite) Objects

Two types of complex objects:
✓ Unstructured: long fields (BLOBs - binary large objects)
✓ Structured:
Types / Classes

- Intension (common description of similar objects)
- Creation of instances - instantiation
- Extension (set of actually existing instances)
- Users are allowed to define their own types

Encapsulation

- Visible for user of class
- Hidden for user of class

Definition of operator interfaces
Definition of value type (data structures for object state)
Implementation of operator bodies
Inheritance:
Type/Class Hierarchies

✓ Object types not always independent of each other:
   generalization/specification ⇒ subtypes/supertypes

✓ Instances of subtypes INHERIT properties from supertypes

✓ Forms: simple inheritance (type hierarchy),
   multiple inheritance (type lattice)

✓ Advantages of the inheritance principle:
   - code reusability (when operators are inherited)
   - representation of additional semantics
   - design discipline (stepwise refinement)

Polymorphism:
Overloading, Overriding, Late Binding

✓ Overloading:
   use of same name for different operators (in different types)

✓ Overriding:
   reimplemention of operator bodies on lower level of type hierarchy

✓ Late binding:
   bind an operator name to an associated implementation later, i.e.,
   made individually for each object at run-time

print_geometric_object (o: g_obj)
(implemented for circles, rectangles, triangles etc)
for all x do print_geometric_object(x);
vs.
for all x do case
   x is circle: print_circle(x);
   x is rectangle: print_rectangle(x);
   x is triangle: print_triangle(x);
   otherwise exception handling);
Object Description Language (ODL)

Object-Oriented DBMS Standardization

- **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.
ODL Class Declarations

- Class declarations include:
  - Name.
  - Key declaration(s), which are optional.
  - Extent declaration: name for the set of currently existing objects of a class.
  - Element declaration(s): an element is an attribute, a relationship, or a method.

- Syntax:
  
  ```
  class <name> {
    <elements = attributes, relationships, methods>
  }
  ```

ODL Element Declarations:
Attributes - I

- Attributes: involve non-object values, e.g., integers, strings

- Syntax:
  
  ```
  attribute <type> <name>
  ```

- Example:
  
  ```
  class Bars {
    attribute string name;
    attribute Struct addr {
      string street,
      string city
    } address;
    attribute Enum lic {
      full, beer, none
    } licenseType;
  }
  ```
Structured types have names and bracketed lists of field-type pairs.

```plaintext
attribute Struct addr {string street, string city} address;
```

Enumerated types have names and bracketed lists of values.

```plaintext
attribute Enum lic {full, beer, none} licenseType;
```

Why name structures and enumerations?

```plaintext
class Drinkers {
    attribute string name;
    
    attribute Struct Bars::addr address;
}
```

**NOTE 1:** Reuse of the Struct addr attribute in the class Bars

**NOTE 2:** Elements from another class is indicated by `<class>::<name>`

Relationships involve objects, is a reference

```plaintext
relationship <type> <name>;
```

Examples:

- relationship Set<Person> hasKids;
- relationship Person hasWife;
- relationship Set<Car> ownCars;
ODL Element Declarations: Relationships - II

✓ Relationships come in inverse pairs:

*Serves* 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.

```
class Bars {
    relationship Set<Beers> serves inverse Beers::servedAt;
}
class Beers {
    relationship Set<Bars> servedAt inverse Bars::serves;
}
```

ODL Element Declarations: Relationships - III

✓ Only binary relations supported:

Multiway relationships require a “connecting” class

**NOTE 1:** Inverses for *theBar*, *theBeer* must be added to *Bars*, *Beers*

```
class Prices {
    attribute real price;
    relationship Set<BBP> toBBP inverse BBP::thePrice;
}
class BBP { /*beers-bars-prices*/
    relationship Bars theBar inverse ... 
    relationship Beers theBeer inverse ... 
    attribute float price;
}
```

**NOTE 2:** might in this case use *only* attribute *price*
**ODL Element Declarations: Relationships - IV**

✓ *Many-many relationships* have a set type in each direction.

```java
class Bars {
    relationship Set<Beers> serves inverse Beers::servedAt;
}
class Beers {
    relationship Set<Bars> servedAt inverse Bars::serves;
}
```

**ODL Element Declarations: Relationships - V**

✓ *Many-one relationships* have a set type for the one, and a simple class name for the many.

```java
class Manufacturer {
    relationship Set<Beers> produceBeers inverse ...;
}
class Beers {
    relationship Manufacturer producedBy inverse ...;
}
```
ODL Element Declarations: Relationships - VI

✓ One-one relations have simple class names for both.

-Manufacturer - Best-seller - Beers

```java
class Manufacturer {
    relationship Beers isBestSellingBeer inverse ...;
}
class Beers {
    relationship Manufacturer hasBestSellingBeer inverse ...;
}
```

ODL Element Declarations: Methods

✓ A piece of executable code which may be applied to the objects of the class
✓ Parameters: in, out, and inout. The object itself is a “hidden” parameter
✓ The method itself may have return values
✓ May raise exceptions
✓ Only signatures are defined in ODL classes; the code is written in a host language – not part of ODL
✓ Example:
```java
class Bars {
    ....
    void availableBeers(out Set<Beers>);
}
```
ODL Type System

- **Basic types:**
  - integer, real, float, character, string, enumerated types, boolean, etc.

- **Type constructors:**
  - Struct for structures
  - Collection types:
    - `Set<T>` unordered set of different objects of type T.
    - `Bag<T>` unordered collection of objects of type T, duplicates allowed.
    - `List<T>` ordered collection of objects of type T.
    - `Array<T>` ordered, indexed collection of objects of type T.
    - `Dictionary<T>` set of object pairs of type T.

- Relationship types may only be classes or a collection of a class.

ODL Keys

- Keys are *optional* in ODL. The OID suffices to distinguish objects that have the same values in their elements.
- Indicate with `key` (or `keys`) 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.
- `key (a_1, a_2, ..., a_n) = “one key consisting of all n attributes.”`
- `key a_1, a_2, ..., a_n = “each a_i is a key by itself where again each a_i may be multi-valued, i.e., a_i = (b_1, b_2, ..., b_n).”`
- Example: single, single-valued key
  ```
  class Beers (key name) {
      attribute string name;
  }
  ```
- Example: multiple, multi-value keys
  ```
  class Courses (key (dept, number), (room, hours)) {
      ...
  }
  ```
- Properties other than attributes are allowed to appear in keys
ODL Extents (Instantiated Classes)

- Difference between the class definition and set of objects of a class.
- Indicate with `extent` and a name that we have an extent (set of existing objects) of a class.
- A class can be *instantiated*, i.e., it may have an associated set of objects (objects may be created), if it is defined with the keyword `class`.
- Example:
  ```java
class Student {extent students key SSN}{
    ...
  }
```  

ODL Signature Classes

- ODL provides support for *interfaces*, which are essentially classes without associated objects.
- Useful if we have several classes that have different extents, but similar properties, e.g., students and teachers are both persons.
- Signature classes are defined with keyword `interface`.
- Signature classes can not be instantiated and are used only as an auxiliary concept to define other classes. Thus, it is not meaningful to use the keywords `extent` and `key` in the class interface of a signature class.
- Example:
  ```java
  interface Person {
    attribute integer SSN;
  } 
class Student : Person (extent students key SSN){
    ...
  } 
class Teacher : Person (extent teachers key SSN){
    ...
  }
```
ODL Subclasses/Inheritance - I

- A subclass inherits all the properties from its superclass, e.g., objects of the Ales class acquire all the attributes and relationships of the Beers class.

- Indicate by the super-class “prefixed” by:
  - colon (:) in signature classes
  - keyword `extends` in instantiated classes

- Example: Ales are Beers with a Color
  ```
  class Ales extends Beers {
    attribute string color;
  }
  ```

- Signature classes can only inherit from other signature classes.

ODL Subclasses/Inheritance - II

- Multiple inheritance:
  - Indicate by keyword `extends` and a list of classes separated by colon (:).
  - Example:
    ```
    class Amphibian
      extends car:boat {
        ...
      }
    ```
  - Naming conflicts are not allowed, and it is the responsibility of the user to avoid them.
  - All classes can inherit from arbitrary many signature classes, but an (instanciated) class can only inherit from one instantiated class, i.e., a class can only be extension of one class.
interface Person {
    attribute integer SSN;
    attribute string name;
    attribute Struct address {
        string street, string city
    } addr;
}

NOTE:
Signature (interface) class
no extent or key
NOTE 1: inherits all the properties of signature class Person
NOTE 2: relationships come with inverse pairs
NOTE 3: “MANY” relationships have a set type
NOTE 4: “ONE” relationships don’t have a set type

class Employee : Person (extent employees key SSN) {
    attribute integer salary;
    relationship Department works_for inverse Department::has_Emp;
    relationship Department manages inverse Department::manager;
    relationship Set<Project> works_on inverse Project::members;
}

NOTE: reuse of address from the Person class

class Department (extent departments key number) {
    attribute integer number;
    attribute string name;
    attribute Struct Person::address addr;
    relationship Employee manager inverse Employee::manages;
    relationship Employee has_Emp inverse Employee::works_for;
    relationship Set<Project> controls inverse Project::contr_by;
}
class Project (extent projects key number) {
  attribute integer number;
  attribute string name;
  attribute integer money;
  relationship Department contr_by inverse Department::controls;
  relationship Set<Employee> members inverse Employee::works_on;
}

NOTE 1:
only binary relationships are supported

NOTE 2:
inverse must be added to Project, Employee

class WorkingHours {
  attribute integer hours;
  relationship Project theProject inverse Project::emp_hours;
  relationship Employee theEmployee inverse Employee::proj_hours;
}
Summary

✓ OO concepts and OODBS properties
✓ ODL
  ➢ classes: signature and instantiated classes
  ➢ superclasses/subclasses → inheritance
  ➢ relationships
  ➢ type system
  ➢ extents
  ➢ keys