ODL Notes

CP465 Databases II

Dr. Ilias S. Kotsireas

ikotsire@wlu.ca

Wilfrid Laurier University

Introduction

ODL is a specification language used to define the specifications of object types that conform to the ODMG Object Model.

Guiding Principles of ODL design:

- support all semantic constructs of the ODMG Object Model
- be programming language independent
- not intended to be a full programming language

ODL defines the characteristics of types, including their properties and operations.

ODL defines only the signatures of operations (does not define the methods that implement those operations)

What can we define in ODL

1. Type characteristics
   (supertype, name of extent, keys)

2. Instance Properties
   (attributes & relationships of the type's instances)

3. a relationship spec. names and defines a traversal path (designation of the target type and the inverse traversal path) for a relationship

4. Operations
   e.g. void, raise exceptions

We will illustrate the use of ODL to declare the schema for a sample university database application.

Object types (defined by classes) are shown as rectangles.

Object types (defined by interfaces) are shown as ovals.

Relationship types are shown as lines.
Cardinalities $1:1, 1:N, M:N$ are indicated by arrows.

Large grey arrows run from subtype to supertype.
(is-a, ISA)

Large black arrows denote **extends**.
Inheritance of state & behavior.

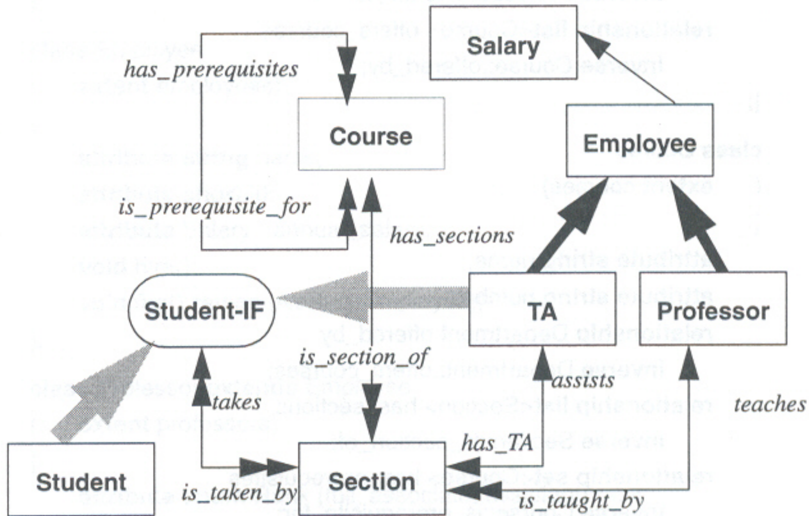

Figure 3-2. Graphical Representation of Schema

```
module ODMGExample {
    exception NoSuchEmployee();
    exception AlreadyOffered{};
    exception NotOffered{};
    exception IneligibleForTenure{};
    exception UnsatisfiedPrerequisites{};
    exception SectionFull{};
    exception CourseFull{};
    exception NotRegisteredInSection{};
    exception NotRegisteredForThatCourse{};
    struct Address {string college, string room_number; };
    class Department
    (   extent departments)
    {
        attribute string name;
        relationship list<Professor> has_professors
            inverse Professor::works_in;
        relationship list<Course> offers_courses
            inverse Course::offered_by;
    };
    class Course
    (   extent courses)
    {
        attribute string name;
        attribute string number;
        relationship Department offered_by
            inverse Department::offers_courses;
        relationship list<Section> has_sections
            inverse Section::is_section_of;
        relationship set<Course> has_prerequisites
            inverse Course::is_prerequisite_for;
        relationship set<Course> is_prerequisite_for
            inverse Course::has_prerequisites;
        boolean offer (in unsigned short semester)
            raises (AlreadyOffered);
        boolean drop (in unsigned short semester) raises (NotOffered);
    };
```

```
class Section
(    extent sections)
{
     attribute string number;
     relationship Professor is_taught_by
          inverse Professor::teaches;
     relationship TA has_TA
          inverse TA::assists;
     relationship Course is_section_of
          inverse Course::has_sections;
     relationship set<Student> is_taken_by
          inverse Student::takes;
};
class Salary
{
     attribute     float     base;
     attribute     float     overtime;
     attribute     float     bonus;
};
class Employee
(    extent employees)
{
     attribute string name;
     attribute short id;
     attribute Salary   annual_salary;
     void hire();
     void fire() raises (NoSuchEmployee);
};
class Professor extends Employee
(    extent professors)
{
     attribute enum Rank {full, associate, assistant} rank;
     relationship Department works_in
          inverse Department::has_professors;
     relationship set<Section> teaches
          inverse Section::is_taught_by;
     short grant_tenure() raises (IneligibleForTenure);
};
```

```
interface StudentIF
{
    attribute string name;
    attribute string student_id;
    attribute Address dorm_address;
    relationship set<Section> takes
        inverse Section::is_taken_by;
    boolean register_for_course (in unsigned short course,
                                 in unsigned short Section)
        raises (UnsatisfiedPrerequisites, SectionFull, CourseFull);
    void drop_course (in Course c)
        raises (NotRegisteredForThatCourse);
    void assign_major (in Department d);
    short transfer (in Section old_section,
                    in Section new_section)
        raises (SectionFull, NotRegisteredInSection);
};
classTA extends Employee : StudentIF
{
    relationship Section assists
        inverse Section::has_TA;
    attribute string name;
    attribute string student_id;
    attribute struct Address dorm_address;
    relationship set<Section> takes
        inverse Section::is_taken_by;
};
class Student : StudentIF
(   extent students)
{
    attribute string name;
    attribute string student_id;
    attribute struct Address dorm_address;
    relationship set<Section> takes
        inverse Section::is_taken_by;
};
};
```