ELEC 372

Week 2

Object-Oriented Development

- Development processes encode organized steps for developing software
- Processes help preserve quality attributes of interest
  - E.g., availability, security, modifiability, usability
- Object-Oriented processes provide a common notation for understanding context of use, design and implementation

Steps in Object-Oriented Development

- Object-Oriented Analysis (OOA)
  - Understanding program's domain
    - E.g., medical systems, production scheduling, travel planning
  - Understanding context of use
- Object-Oriented Design (OOD)
  - Architecture design
    - Dividing system into set of components and connectors
    - Each component is a *module*
  - Data structure design
    - Format of data within components, passed over connectors
- Object-Oriented Programming (OOP)
  - Realizing OO Design in language such as Java, C++

Object-Oriented Development

- Smooth flow from analysis to design to implementation may
  - help correctness
  - reduce development time
  - ensure requirements addressed in implementation
- OO mechanisms in Java/C++ help with code reuse
- Design patterns, architectural styles help with design reuse
  - Reuse saves time, provides debugged solution whose properties are well understood

Module Concept in OOD

- Good program design relies on splitting program function into a set of independent units (modules)
- Each module hides details of its implementation
  - Allows implementation to be changed without impacting rest of program
  - Simplifies code in rest of program
- In OO languages, *interfaces* and *classes* used to implement modules

Example: Implementation of a *TimeVal* class

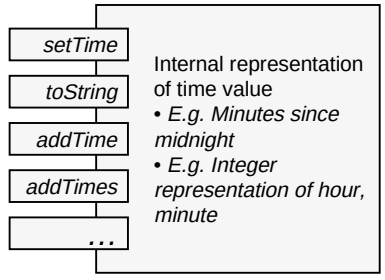
- Many applications require times to be represented
- E.g., a calendar application needs to represent the times of appointments
- Times might be represented in many different ways. E.g.,
  - A 24-hour format string ("10:00", "22:49")
  - A 12-hour format string ("10:00 AM", "10:49 PM")
  - Two integers for hours/minutes + Boolean flag for AM/PM
    - (<10, 0, false>, <10, 49, true>)
  - Integer representing number of minutes since midnight
    - (600, 1369)

Operations on Time Values

- Many operations are possible
  - Turn time into format suitable for printing/viewing
    - convert time to string representation in 12 hour clock
    - convert time to string representation in 24 hour clock
  - Add minutes to time/subtract minutes from time
    - (e.g., moving an appointment to 60 minutes later)
  - Subtract two times
    - (e.g., calculating how long an appointment is by subtracting start time from end time)
  - Comparing times
  - Etc.

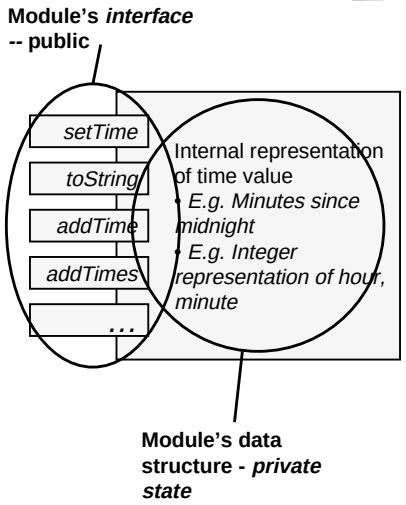
Goal in Modular Design

- Determine internal representation for data
  - Hidden inside module
  - Module's *data structure*
- Determine set of operations over data
  - Publicly available to users of data
  - Module's *interface*



Goal in Modular Design

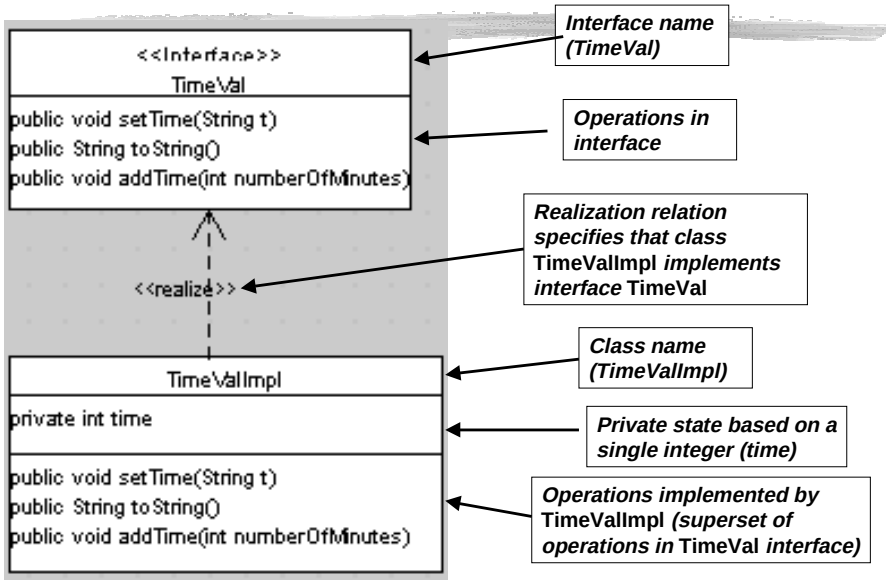
- Clients of module use the module's interface to access its internal state
  - I.e., parts of program that want to manipulate times invoke operations *setTime*, *addTime*, etc. rather than referring to data structure directly
- Consequence
  - Improves modifiability, since if we change internal data state, impact limited to module itself
  - Other parts of program don't depend on data structure decision within module



Modular Design in Java

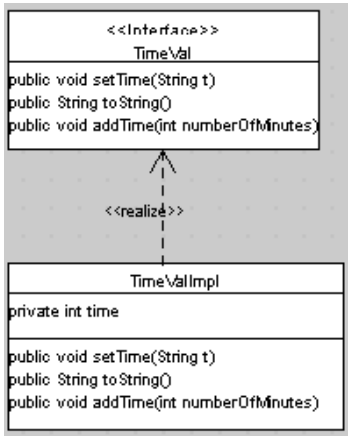
- **`interface`** construct used to define interfaces - the set of operations visible to the rest of the program
- **`class`** construct used to define the implementations of modules - the data structure and the implementation of operations in the interface
- **`new`** construct used to create instances of modules (called *objects*)

Description of Classes and Interfaces in UML



Classes and Interfaces

- An interface is a *specification* of the operations to be provided by a module
- A class is used to *implement* the interface



```
interface TimeVal {
   // Sets the time (24 hr string format)
   public void setTime (String t);

   // Returns the time in 24 hr string format
   public String toString ();

   // Adds numberOfMinutes to the time
   public void addTime (int numberOfMinutes);
}
```

```
class TimeValImpl implements TimeVal {

    private int time;
    private static int minutesPerHour = 60;
    private static int hoursPerDay = 24;
    private int stringToInt (String s) { ... }

    public void setTime (String t) { ... }
    public String toString () { ... }
    public void addTime (int numberOfMinutes) { ... }
}
```

Private data and operations used in implementation -- not visible outside class

Implementations of public operations, as specified in interface

```
class TimeValImpl implements TimeVal {

    // Represents time as the number of minutes since
    // midnight. Therefore ranges from 0 (12:00 AM)
    // to 1439 (11:59 PM)
    private int time;

    private static int minutesPerHour = 60;
    private static int hoursPerDay = 24;

    // Converts a string representation of an integer
    // into an integer. This method is private - not
    // visible outside this class.
    private int stringToInt (String s) {
        int i=0;
        try {
            i = Integer.parseInt (s);
        } catch (NumberFormatException e) {
            System.err.println ("String not an integer");
            System.exit (1);
        }
        return i;
    }
    ...
}
```

Hint: This method uses operations built into the Java library classes java.lang.String and java.lang.Integer -- details on these operations can be found at **http://www.javasoft.com/products/jdk/1.1/docs/api/packages.html**

```
class TimeValImpl implements TimeVal {
    ...
    public void setTime (String t) {
        // Find in the string where the ":" character appears
        int sep = t.indexOf (':');
        if (sep == -1) {
            System.err.println ("Attempt to set time to " + t);
            System.exit (1);
        }

        // Extract the part of the string for the hours and the
        // part for the minutes
        int hours = stringToInt (t.substring (0, sep));
        int minutes = stringToInt (t.substring (sep+1));

        // Check for errors
        if (hours < 0 || hours > 23 || minutes < 0 || minutes > 59) {
            System.err.println (t + " is an illegal time");
            System.exit (1);
        }

        // Set the time in minutes since midnight
        time = hours*minutesPerHour + minutes;
    }
    ...
}
```

```
class TimeValImpl implements TimeVal {
    ...
    public String toString () {
        // Converts the integer representation of the time to
        // string form.

        String hour = Integer.toString (time / minutesPerHour);

        String minute = Integer.toString (time % minutesPerHour);
        if (minute.length() == 1) {
            minute = "0" + minute;
        }

        return hour + ":" + minute;
    }

    public void addTime (int numberOfMinutes) {
        if (numberOfMinutes < 0) {
            System.err.println ("Parameter to addTime must be positive");
            System.exit (1);
        }

        time += numberOfMinutes;

        // Wrap around if we've gone past midnight
        time = time % (hoursPerDay * minutesPerHour);
    }
}
```

Instantiating a class

```
class Main {
    public static void main (String[] args) {
        // Test the TimeValImpl class

        // Create a time, add time, and print its value
        TimeVal t = new TimeValImpl();
        t.setTime ("3:00");
        t.addTime (70);

        // Correct output is 4:10
        System.out.println ("Time = " + t);
}
```

Instantiating Classes

- Main program creates a reference to a *TimeVal* called *t*
- *t* may be assigned any object that implements the interface *TimeVal*
- The *TimeValImpl* class implements *TimeVal*, so it is legal to assign an object of type *TimeValImpl* to t
- We create an object of type *TimeValImpl* using the *new* operation

```
class Main {
  public static void main (
                  String[] args) {

    TimeVal t = new TimeValImpl();

    t.setTime ("3:00");
    t.addTime (70);
    System.out.println (
                  "Time = "+t);
}
```

Instantiating Classes

- We create an object of type *TimeValImpl* using the *new* operation
- When we invoke operations on *t*, we are actually calling the operations defined in the *TimeValImpl* class

```
class Main {
  public static void main (
                  String[] args) {

    TimeVal t = new TimeValImpl();

    t.setTime ("3:00");
    t.addTime (70);
    System.out.println (
                  "Time = "+t);
}
```

Instantiating Classes

- In Java, printing an object (using **println**) invokes the object's **toString** method to convert the object into string form
- Therefore, if you want an object to be printable, you need to provide a **toString** method in its class definition

```
class Main {
  public static void main (
                  String[] args) {

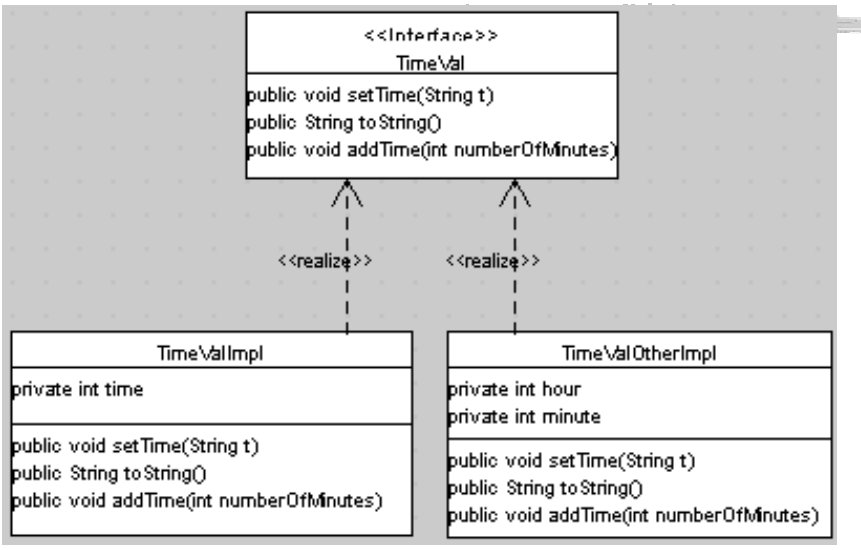
    TimeVal t = new TimeValImpl();

    t.setTime ("3:00");
    t.addTime (70);
    System.out.println (
                  "Time = "+t);
}
```

Alternative Implementations

- To provide a different implementation
  - Create a new class (e.g., **TimeValOtherImpl**)
  - That class must implement the **TimeVal** interface

Alternative Implementations



Changing the Main program

```
class Main {
    public static void main (String[] args) {
        // Test the TimeValImpl class

        // Create a time, add time, and print its value
        TimeVal t = new TimeValOtherImpl();
        t.setTime ("3:00");
        t.addTime (70);

        // Correct output is 4:10
        System.out.println ("Time = " + t);
}
```

Code using t does not change even when alternative implementation used.

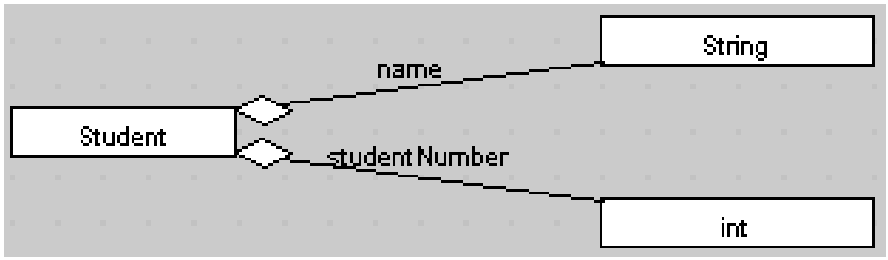
Object-Oriented Modeling

- Design of program is aided by an understanding of the domain to which it is to be applied
- First model application domain (OOA)
- Then transform this model to an implementation design (OOD)
- Then implement it in Java/C++/… (OOP)

Example: Student Data Base

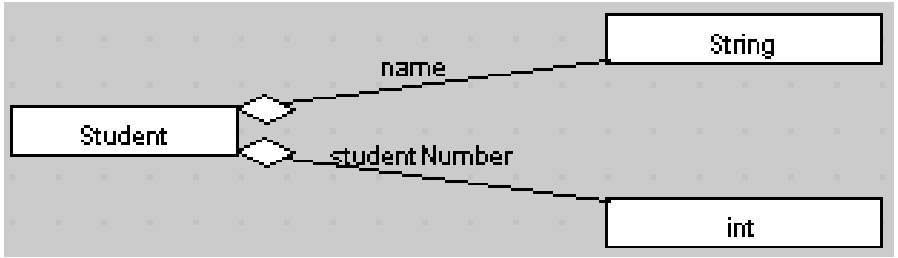
- Wish to develop a system that checks the integrity of student enrolments in courses
- If enrolment problems found, wish to inform students/administration so they can change their selections
- Rules:
  - Each course must have at least 3 students enrolled
  - Students may not be enrolled in both CISC 121 and CISC 124
  - Students may take at most 5 courses

Understanding the Domain



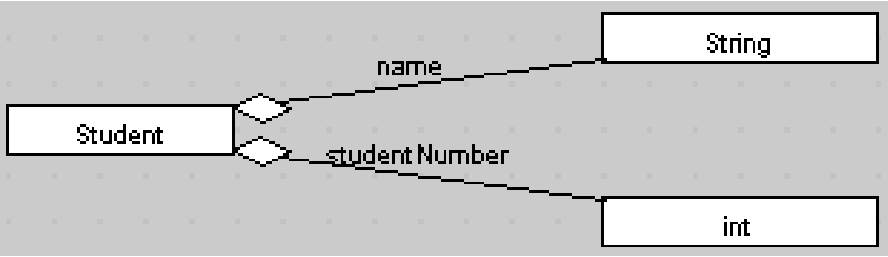
Modeling Students

- Identify entities in application domain
  - E.g., a student data base requires knowledge of students
- Identify relations amongst entities
  - E.g., students have names (which are strings), and student numbers (which are integers)



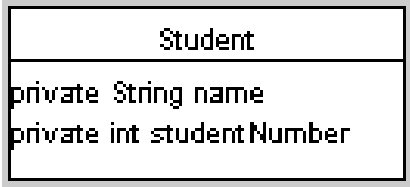
Modeling Students

- In UML, entities (or *classes*) are drawn as boxes
- Relations between classes are drawn as arcs between the boxes
  - ◇—— stands for aggregation, read as "has-a"
  - E.g., read "A student has a name, which is a String"



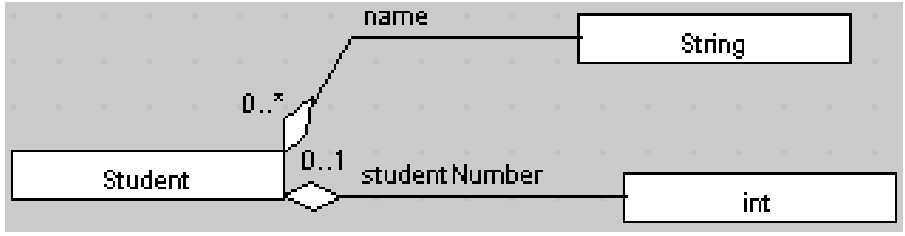
Modeling Students

- Alternatively, can express *has-a* relation by specifying attributes of class



Multiplicities

- Relations can convey cardinality information (called *multiplicity* of relation)
- E.g., student numbers are unique
  - At most one student has a given student number
- E.g., student names are not unique
  - Zero or more students might have a given name



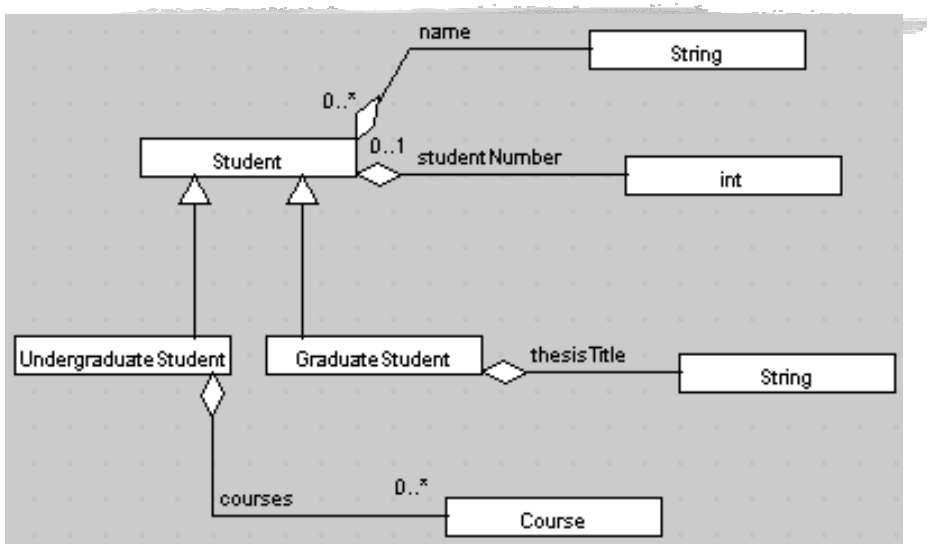
Multiplicities

- Multiplicities may be
  - *n* : some constant n≥0
  - 0..1 : zero or one
  - 0..* : zero or more (alternatively just *)
  - 1..* : one or more
- May appear at either end of aggregation arc
- Default multiplicity is many to one
  - 0..* at arc source, 1 at arc target

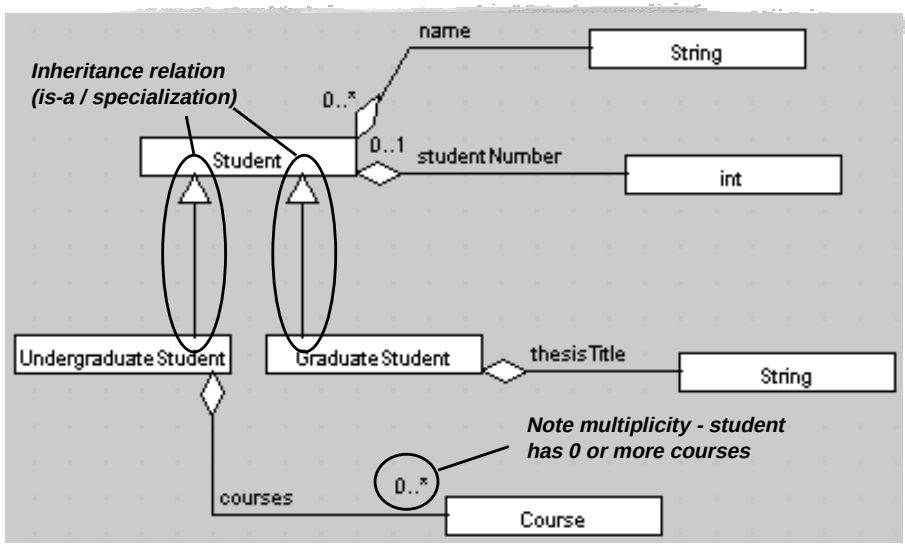
Modeling Students

- There are different kinds of students
  - Graduate students work on a thesis
  - Undergraduate students are enrolled in a set of courses

Modeling Students



Modeling Students

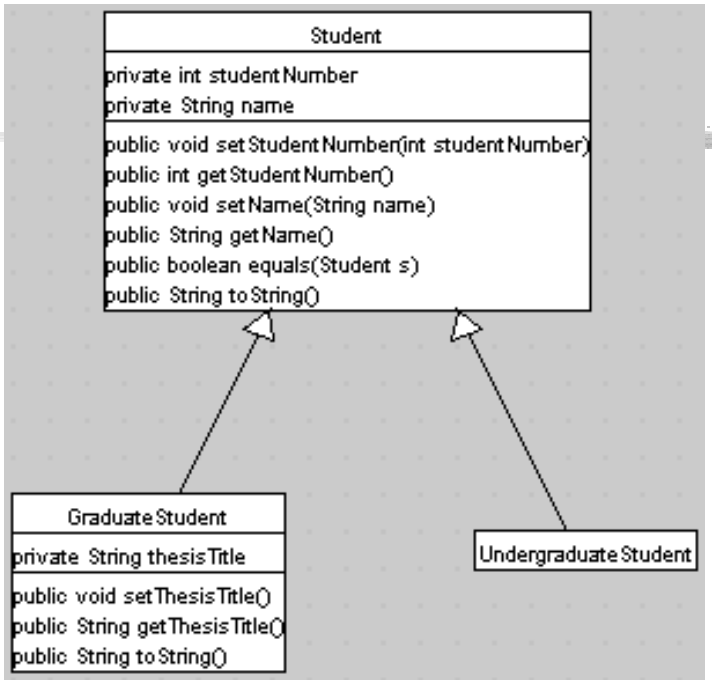


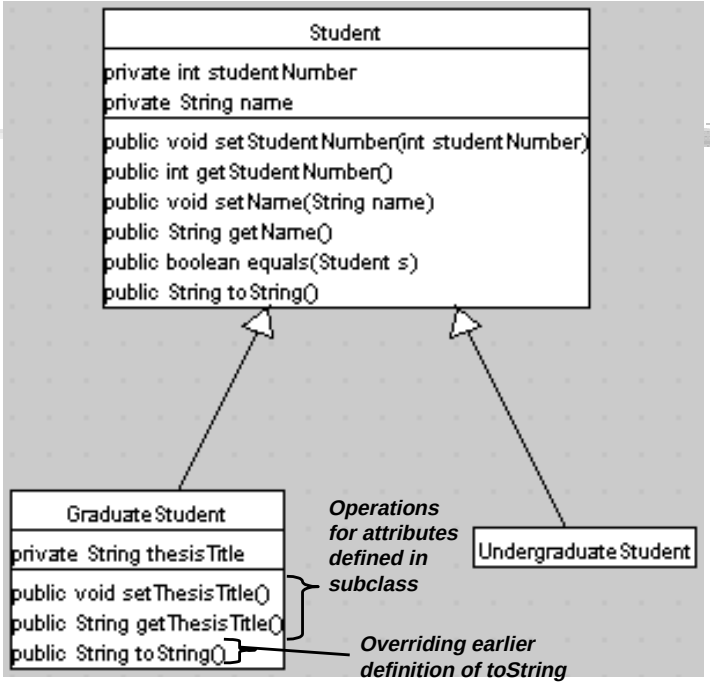
Inheritance

- Sometimes called *specialization*
- Read as "is-a" -- "An undergraduate student is a student"
- *UndergraduateStudent* and *GraduateStudent* are sub-classes of *Student*
- Sub-classes *inherit* all the attributes and operations of the parent class (or *super-class*)
- I.e., undergraduate students have a student number, a name and a set of courses; graduate students have a student number, a name and a thesis title

Operations

- get/set attributes of Student, UndergraduateStudent, GraduateStudent classes
- Test whether two students are the same (I.e. have the same student number)
- Turn the student's information into a string representation for display

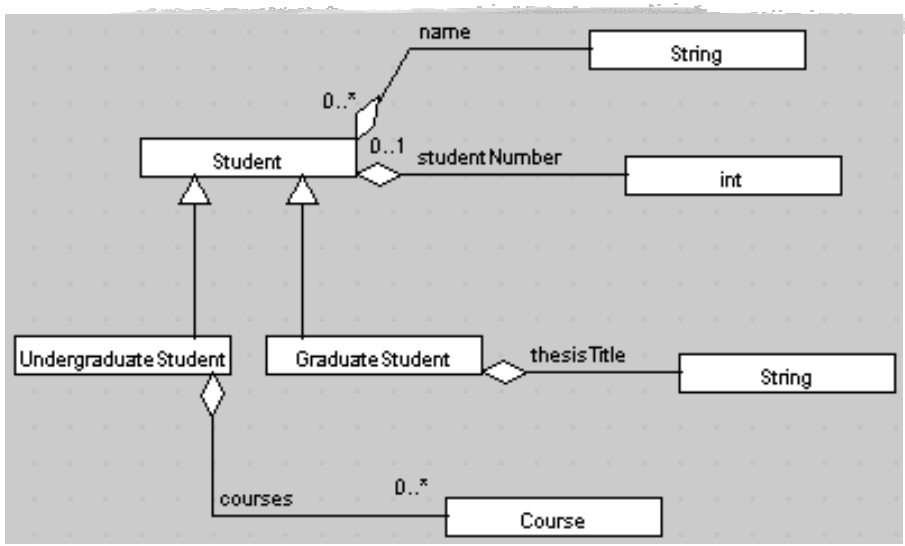




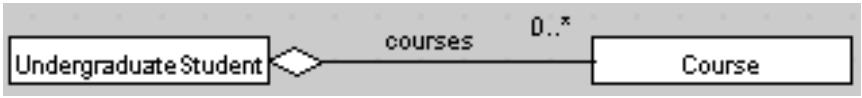
Refining from OOA to OOD

- OOA identifies domain concepts
  - Often abstracts design decisions such as details of datastructures implementing relationships between domain entities
- OOD refines OOA to pin down design decisions
  - Add explicit operations to classes
  - Specify attributes used to implement relations
    - E.g., in *Student* class, *studentNumber* attribute is implemented via a private attribute *studentNumber* of type *int*

Recall our Model of Students…

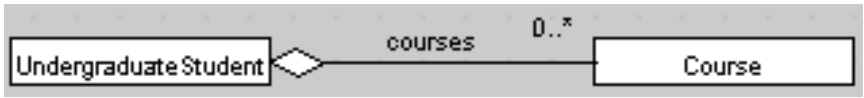


Refining the *courses* relation



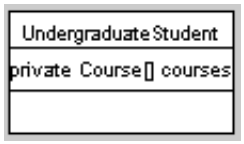
- Specifies that each student has three or more courses
- Need to decide how to implement this relation
- Implementations will trade off efficiency, ease of implementation, ease of modification

Refining the *courses* relation

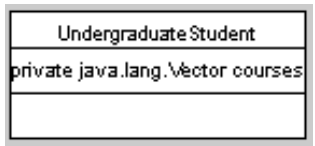


- Some choices
  - Use an array in *UndergraduateStudent*
  - Use the Java *Vector* class
  - Use a linked list
  - Create a *CourseList* class
    - Hides implementation of list
    - Itself uses array or *Vector* or linked list or …

Array and Vector Solutions




```
class UndergraduateStudent extends Student {
    …
    private Course[] courses;
    …
}
```



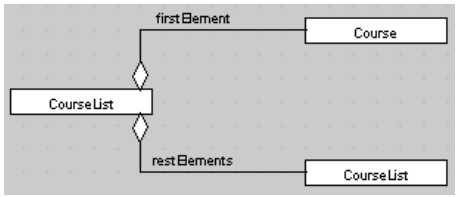

```
import java.util.Vector;
class UndergraduateStudent extends Student {
    …
    private Vector courses;
    …
}
```

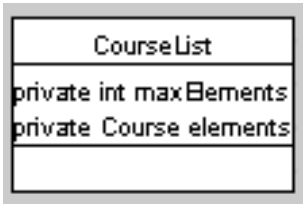
Course List Solution



- Hides implementation decision within a new *CourseList* class
- *CourseList* must itself be refined to an implementation

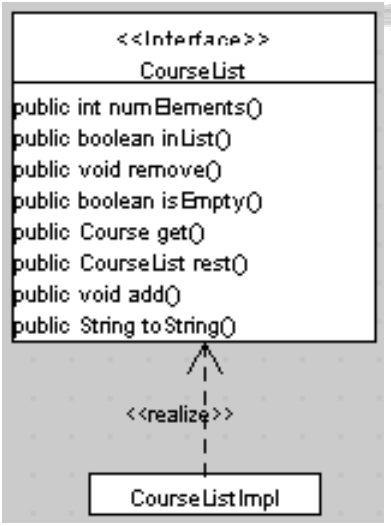
Choices for refining implementation of CourseList



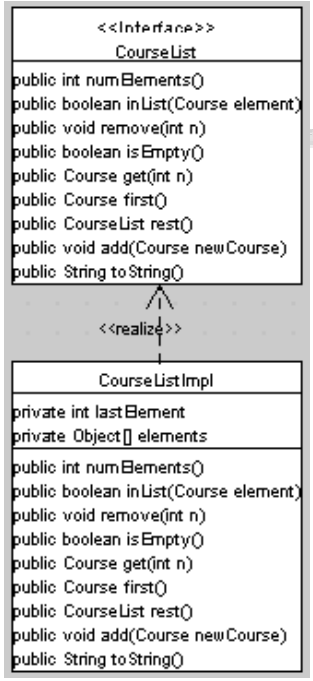


- Linked list
  - Each course list has reference to first element in list, and to another course list containing the remaining elements
- Array
  - The courses are represented in positions 0..maxElements-1 of an array

Operations in *CourseList*



- Hide implementation decision of *CourseList* through use of an interface, separate implementation(s)
- Could have one array-based implementation, another based on a linked list



- Operations
  - *numElements* - returns number of elements in the list
  - *inList* - determines whether given course is in the list
  - *remove* - removes the element at index n in the list (index 0 is the first element)
  - *isEmpty* - tests whether the list is empty
  - *get* - returns course at index n ($0 \le n < numElements$)
  - *first* - returns first course in list
  - *rest* - returns new list with all elements following the first one
  - *add* - adds the given course into the list
  - *toString* - returns a string representation of the contents of the list

Operations

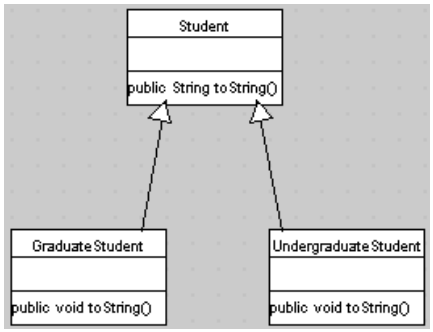
- Any implementation choice trades off efficiency of some operations versus others
  - E.g., array implementation allows efficient implementation of *get* operation
  - E.g., linked list implementation allows efficient implementation of *rest* operation
- In assignment 1, you will examine why

Refining to Code

- Object-oriented design exposes implementation decisions
  - Classes/interfaces map to Java classes/interfaces
  - Attributes map to fields
  - Operations map to methods

```
class CourseListImpl implements CourseList {
    private int lastElement;
    private Object [] elements;
    ooo
    public Course get (int n) {
        Util.assert (n >= 0 && n < numElements(),
                              "getElement out of bounds");
        return elements [n];
    }
    ooo
    public void remove (int n) {
        Util.assert (n >= 0 && n < numElements(),
                              "getElement out of bounds");
        // Move all elements following n down one position. Ie,
        // elements [n] = elements [n+1];
        // elements [n+1] = elements [n+2];
        // etc.
        for (int i=n; i<lastElement; i++) {
            elements [i] = elements [i+1];
        }
        lastElement--;
    }
    ooo
}
```

Polymorphism



- Subclasses may override methods of superclass
- When calling method in any of these classes, the method belonging to the actual object is called
  - even if object is currently typed as the superclass
- This process is called *polymorphism* or *dynamic binding*

Polymorphism

```
Student g = new GraduateStudent(“Diane”,123,“Medieval Basket Weaving”);
CourseList nicksCourses = … ;
Student u = new UndergraduateStudent (“Nick”, 456, nicksCourses);

// invokes toString in GraduateStudent class
System.out.println (g);

// invokes toString in UndergraduateStudent class
System.out.println (u);
```

