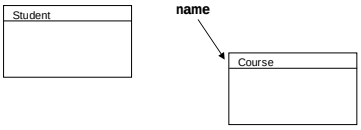
Designing with UML

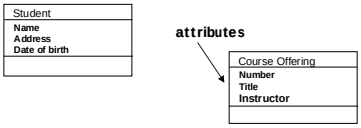
Review: Classes are shown by boxes



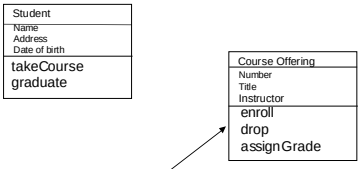
Classes, not actual objects

(Objects can also be shown by boxes;
For objects, names are always *underlined*.)

Review: Attributes may be listed

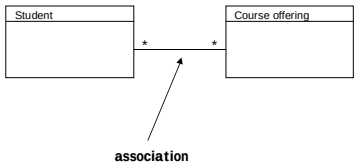


Operations (methods) may be listed



operations
(more detail can be given,
such as argument and result types, and visibility)

Review: *Associations* are shown by lines



Generally this means that there are 0 or more **pairings** of students with course offerings.

Many Possible Implementations of Associations

- Recall implementations of undirected graphs:
  - List of pairs
  - Arrays or list of references (or pointers) to other objects
  - Fixed reference or pointer variables
  - Implied associations

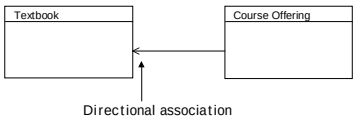
Directionality of Associations

- By default, associations allow "bi-directional" navigation:

  From an object in either class, one can get to the associated objects in the other class.

- Adding an open arrow-head ***restricts*** **navigation** to be one-way, in the direction of the arrow.

Directional Association

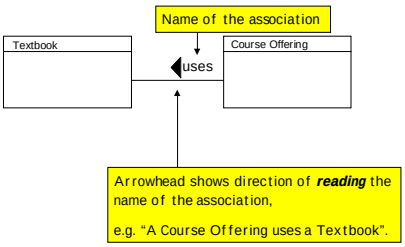


- Here a Course Offering knows about its Textbooks but not vice-versa.
- This is sometimes called a "navigation arrow".
- If **absent**, then navigation is assumed to be bi-directional.

Directionality

- Directionality is a "design detail" that need not be of concern in initial passes of the design.
- It will impact the choice of implementation techniques and performance.

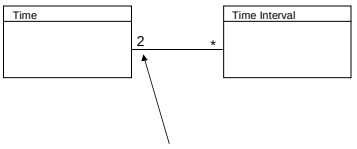
Review: Ordered *Reading* of Association Names



Ordered vs. Directional

- Ordered involves the **reading** interpretation of the association only.
- Directional determines the navigability.
- The two are totally *independent*.

Review: Associations may have a multiplicity



Multiplicity: says that each Time Interval has two Times (such as a start time and an end time).

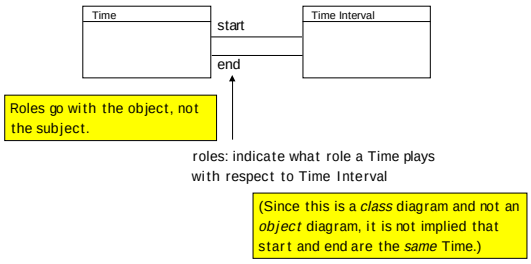
Association multiplicities

- The default multiplicity is 1.
- m..n means m through n (m and n fixed numbers).
- m..* means m or more.
- * means the same as 0..* (0 or more).
- a, b, c, ... means *one of* a, b, c ...
- 0,1 or 0..1 is a way of saying *optional*.

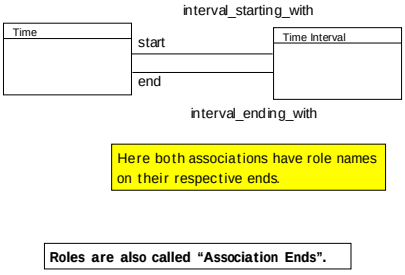
Note on Multiplicities

- Multiplicity should be the one that you wish the **software application** to address, rather than what *might* be the case in nature.
- For example, a major of a given *name* may exist in several colleges, suggesting * * association.
- However, * 1 association might be wanted (one college has multiple majors), but a given major belongs to a college.

Roles in Associations

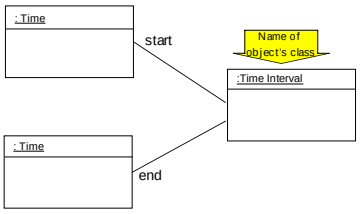


Roles in Associations

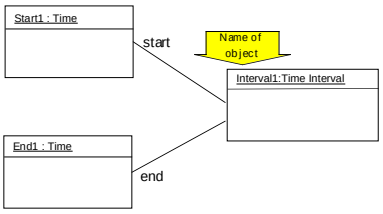


Roles are also called "Association Ends".

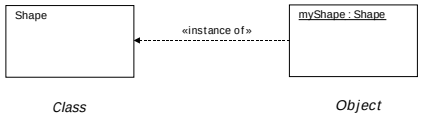
Corresponding *Object* Diagrams



Object Diagrams with Objects *Named*



Objects and Class in One Diagram



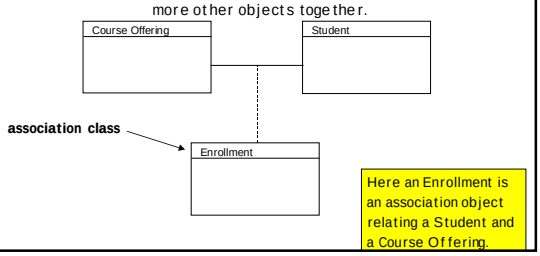
Scope of Object Notation

- In addition to object diagrams, the object notation is used in:
  - collaboration diagrams
  - sequence diagrams
  - and others

which will be described later.

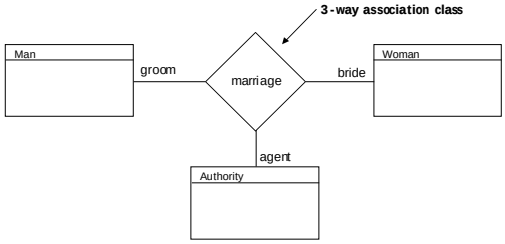
Review: Association Classes

We may wish to emphasize that an association may *itself* take the form of an object relating two or more other objects together.



Multi-Way Association Classes

Associations classes aren't limited to 2-way.

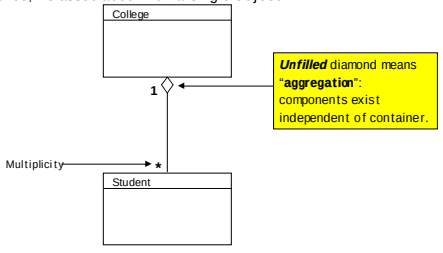


Aggregation and Composition

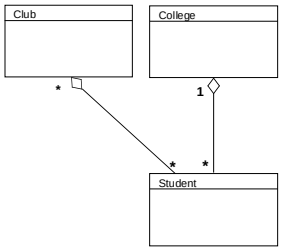
- These are both specialized forms of Association.
- They suggest whole/part relationships.
- They add certain kinds of constraints.

Aggregation

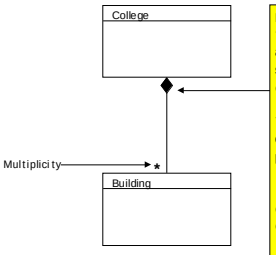
An *aggregation* is a special form of association in which a collection of objects, each having an independent existence, is associated with a single object.



An object can be in multiple distinct aggregations.



Composition



Filled diamond means "**composition**": components are **inseparable, non-sharable, part** of container.

The container is composed of the components (and possibly others).

In some sense, the container "controls" the components.

Multiplicity 1 is thus implied.

Question

- Can an object be in an aggregation and a composition simultaneously?
- Is it advisable to do this?

Possible C++ comparison

- Aggregation

```
class College
   {
   list<Student*> students;
public:
   void addStudent(Student* s)
     {
     students.add(s);
     }
     ...
   }
```

Students exist outside of the college.

- Composition

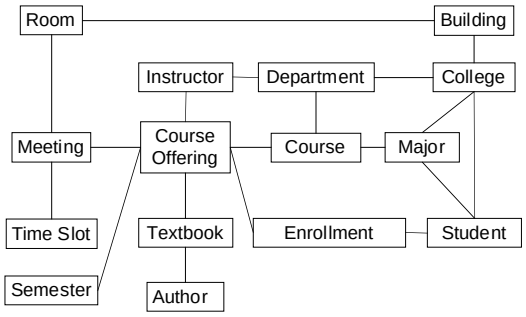
```
class College
   {
   list<Building*> buildings;
public:
   void buildBuilding(string name)
     {
     buildings.add(new Building[n]);
     }
     ...
   }
```

Construct inside; assuming buildings don't exist outside of the college.

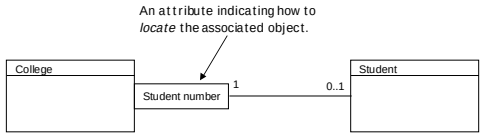
C++ Destruction Note

- With **composition**, contained objects are always created and known only "on the inside".
- With **aggregation**, aggregate objects are created and destroyed independent of the aggregating object.

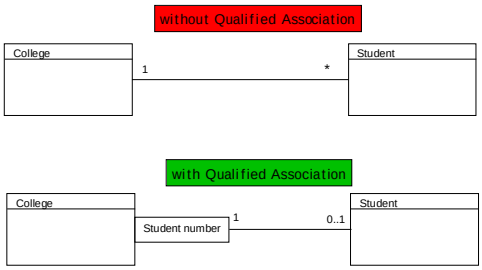
Exercise: Identify Likely Aggregations and Compositions



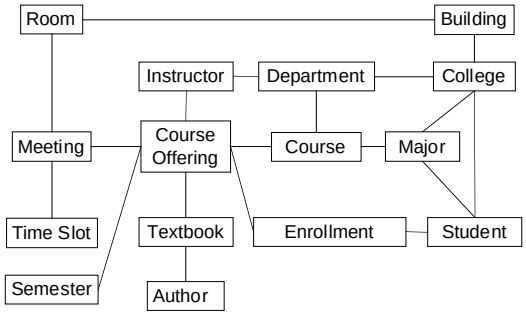
Qualified Association



Comparison: Qualified vs. Unqualified Association

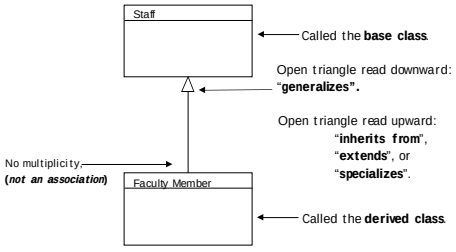


Exercise: Identify Opportunities for Qualified Association



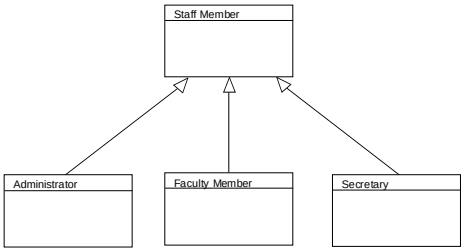
Inheritance and Generalization

Inheritance/Generalization

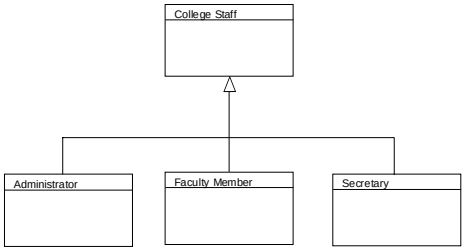


In this form of inheritance, a member of the derived class ***is-a*** member of the base class, as far as behavior is concerned.

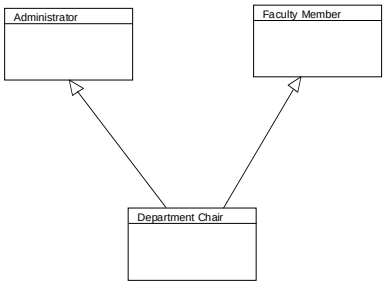
Usually there will be multiple derived classes if there is any.



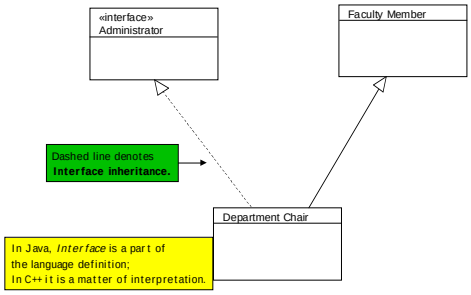
This notation is equivalent to that on the preceding slide.



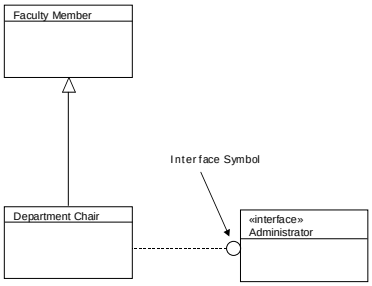
"Multiple Inheritance" is possible, although should be avoided since not all implementation languages support it well



"Interface Inheritance" alternative

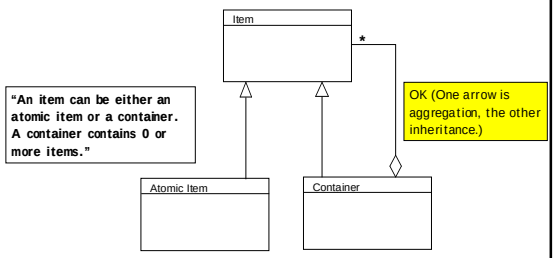


Alternative Notation for Interface



Recursive Structure

Use inheritance to articulate recursive structures.



Corresponding Object Diagram