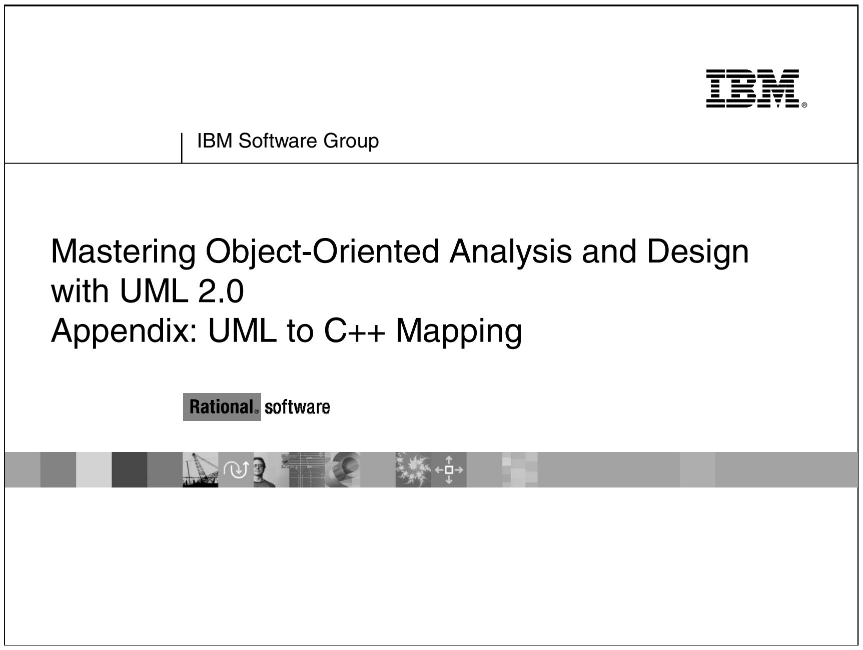
Instructor Notes:

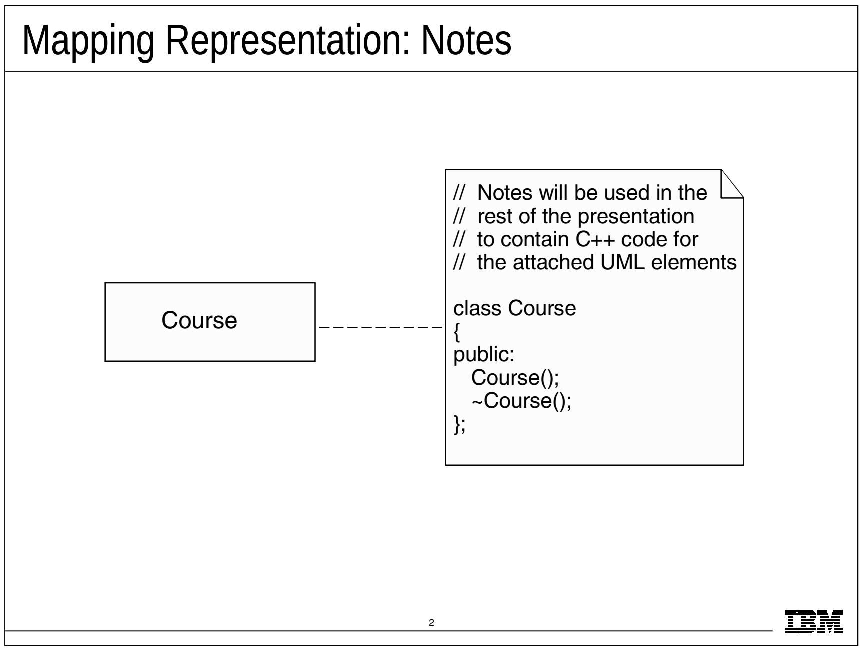


IBM Software Group

Mastering Object-Oriented Analysis and Design with UML 2.0
Appendix: UML to C++ Mapping

Rational. software

Instructor Notes:

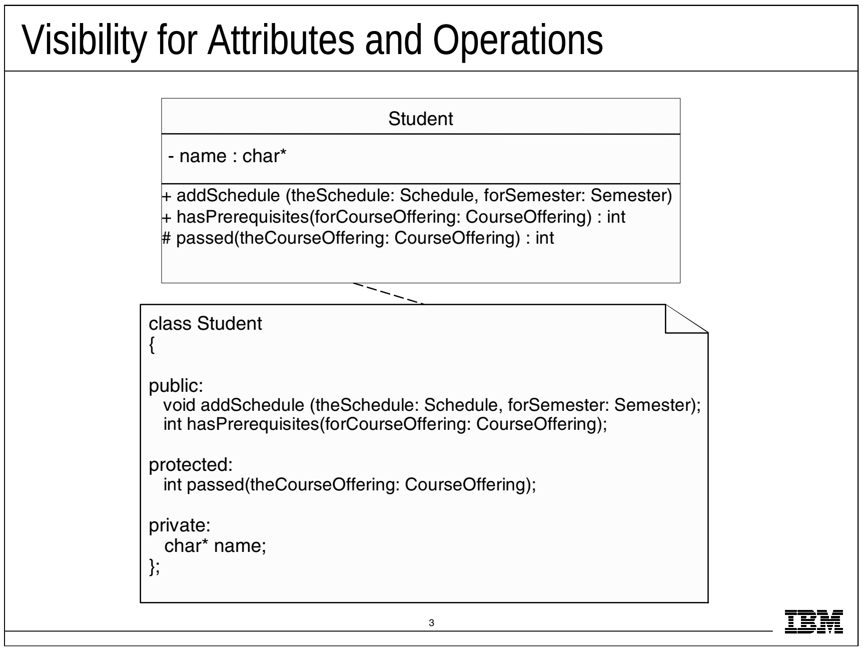


Mapping Representation: Notes

```
//  Notes will be used in the
//  rest of the presentation
//  to contain C++ code for
//  the attached UML elements

class Course
{
public:
   Course();
   ~Course();
};
```

If you remember from earlier in the course, a note can be added to any UML element. It is represented as a ‘dog eared’ rectangle. The note may be anchored to a specific element(s) with a dashed line

Instructor Notes:



Visibility for Attributes and Operations

Student
- name : char*
+ addSchedule (theSchedule: Schedule, forSemester: Semester)
+ hasPrerequisites(forCourseOffering: CourseOffering) : int
# passed(theCourseOffering: CourseOffering) : int

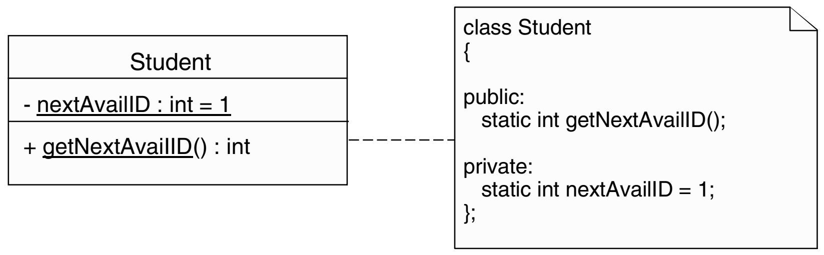
```
class Student
{

public:
   void addSchedule (theSchedule: Schedule, forSemester: Semester);
   int hasPrerequisites(forCourseOffering: CourseOffering);

protected:
   int passed(theCourseOffering: CourseOffering);

private:
   char* name;
};
```

3

A subset of the Student class operations and attributes are shown above. For this example, we included a subset to demonstrate the UML construct we are emphasizing.

Instructor Notes:

Class Scope Attributes and Operations

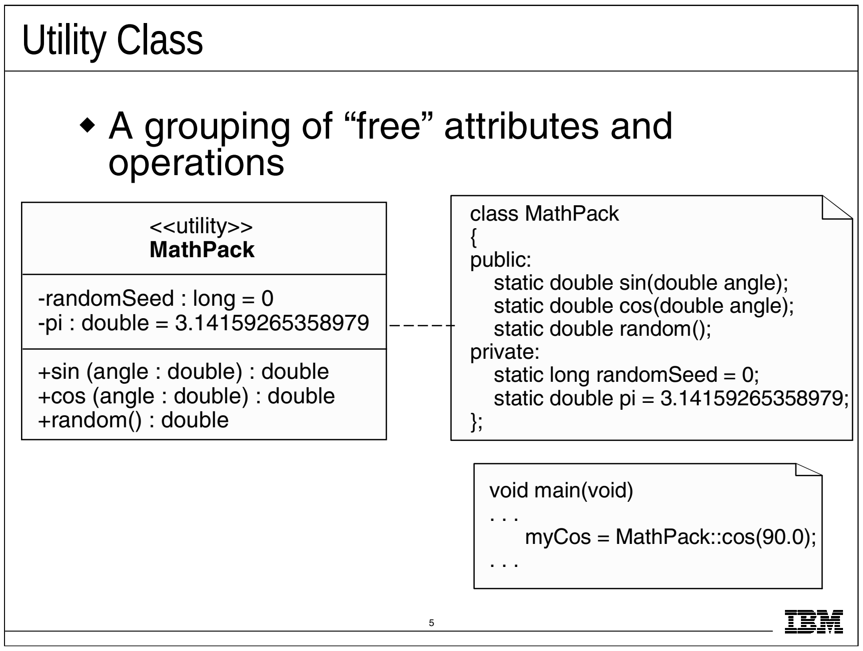


```
class Student
{

public:
   static int getNextAvailID();

private:
   static int nextAvailID = 1;
};
```

4

A subset of the Student class operations and attributes are shown above. For this example, we included a subset to demonstrate the UML construct we are emphasizing.

Instructor Notes:



Utility Class

- A grouping of “free” attributes and operations

```
<<utility>>
MathPack
-----------------------------------
-randomSeed : long = 0
-pi : double = 3.14159265358979
-----------------------------------
+sin (angle : double) : double
+cos (angle : double) : double
+random() : double
```

```
class MathPack
{
public:
   static double sin(double angle);
   static double cos(double angle);
   static double random();
private:
   static long randomSeed = 0;
   static double pi = 3.14159265358979;
};
```

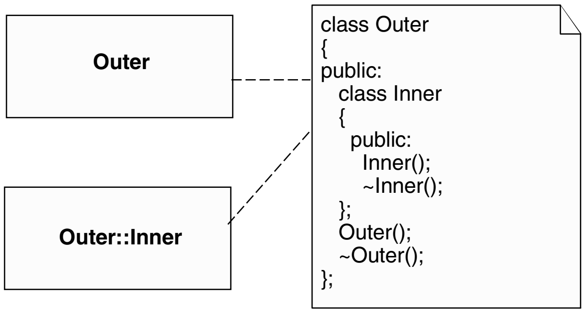
```
void main(void)
. . .
     myCos = MathPack::cos(90.0);
. . .
```

5

Instructor Notes:

Nested Class

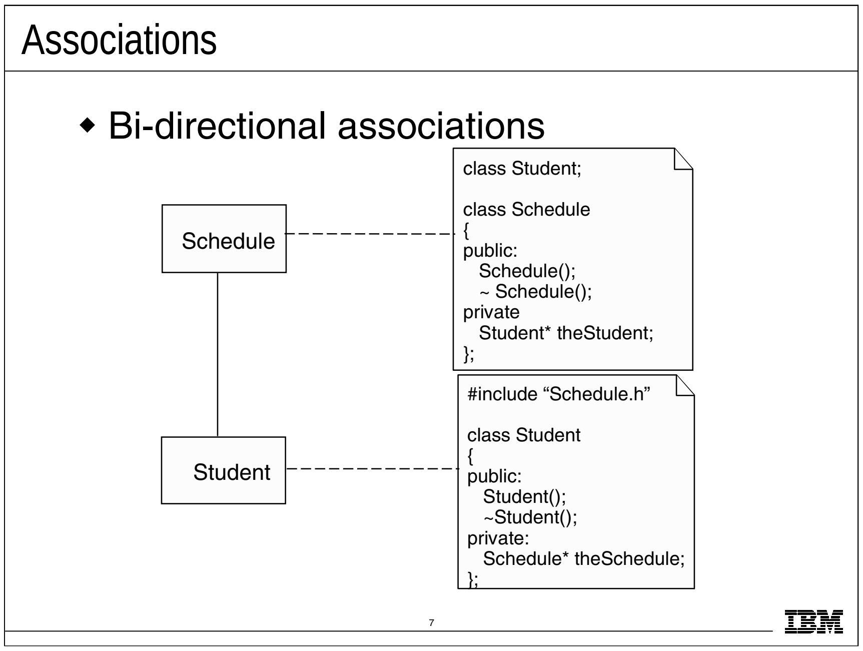
- Hide a class that is relevant only for implementation



```
class Outer
{
public:
   class Inner
   {
     public:
       Inner();
       ~Inner();
   };
   Outer();
   ~Outer();
};
```

6

Instructor Notes:



Associations

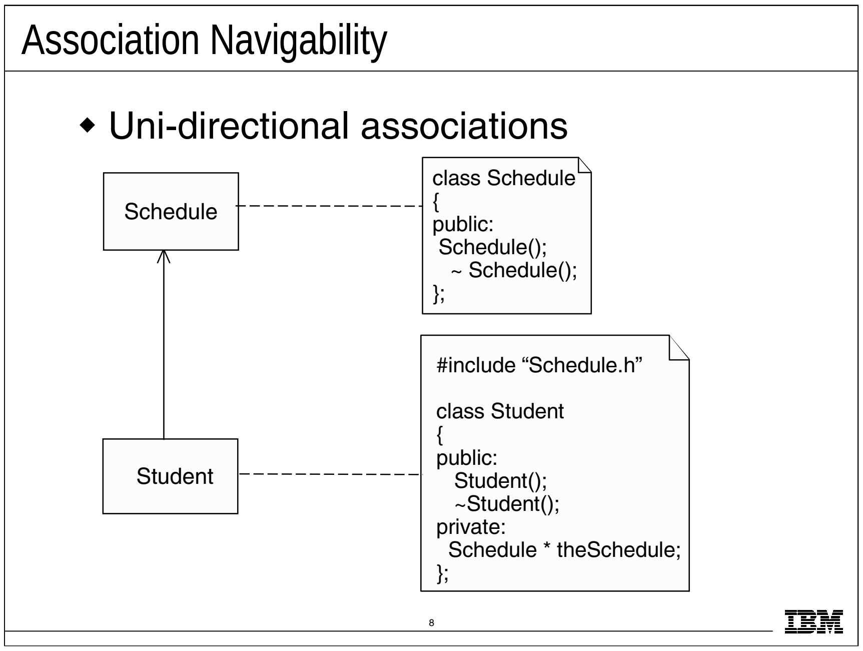
- Bi-directional associations

```
class Student;

class Schedule
{
public:
   Schedule();
   ~ Schedule();
private
   Student* theStudent;
};
```

```
#include "Schedule.h"

class Student
{
public:
   Student();
   ~Student();
private:
   Schedule* theSchedule;
};
```

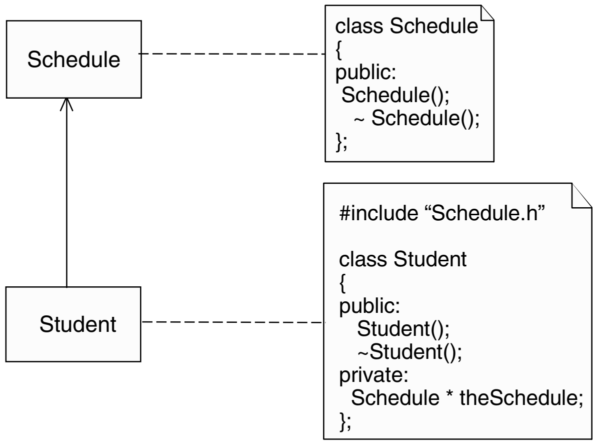
7

Instructor Notes:



Association Navigability

- Uni-directional associations

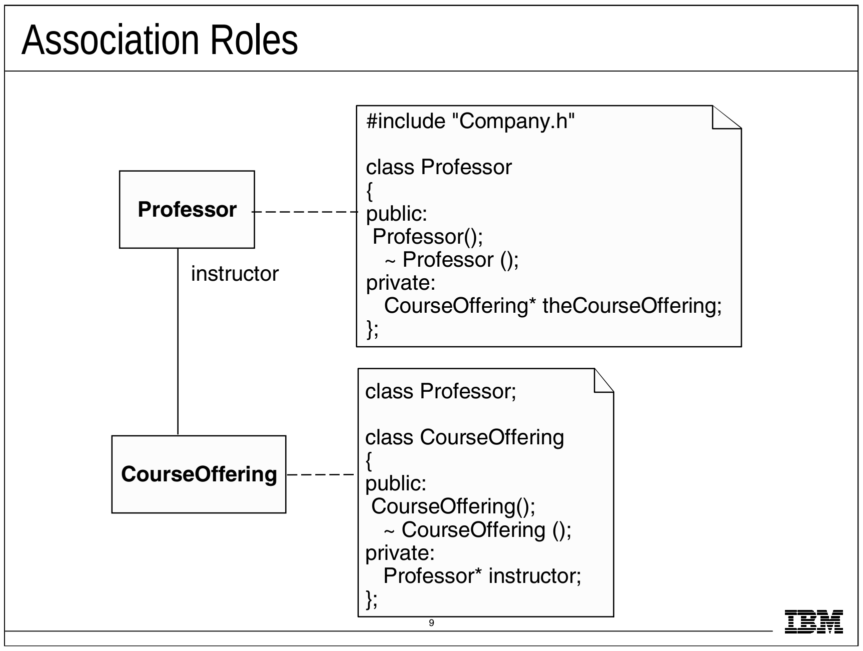


```
class Schedule
{
public:
 Schedule();
   ~ Schedule();
};
```

```
#include "Schedule.h"

class Student
{
public:
   Student();
   ~Student();
private:
  Schedule * theSchedule;
};
```

8

Instructor Notes:



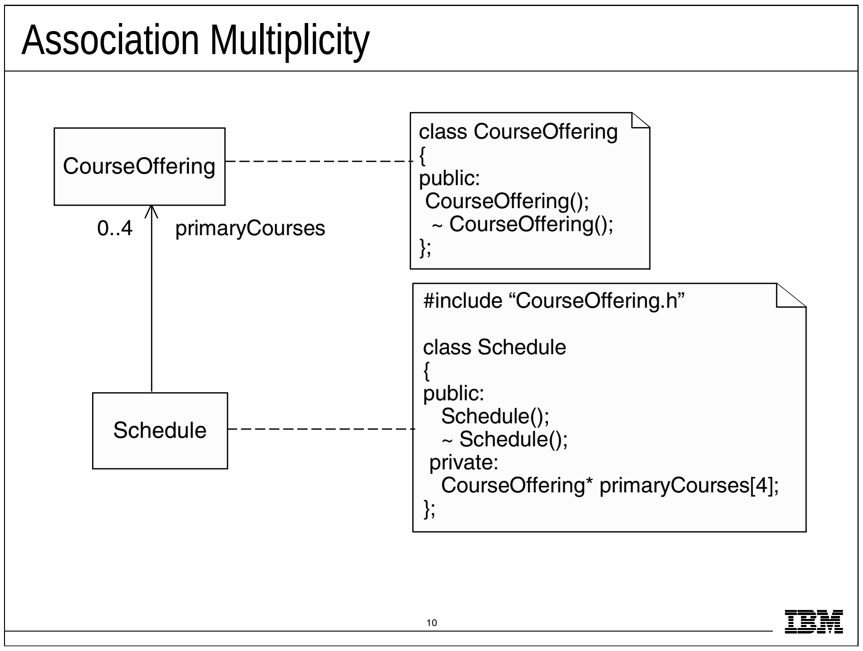
Association Roles

```
#include "Company.h"

class Professor
{
public:
 Professor();
   ~ Professor ();
private:
   CourseOffering* theCourseOffering;
};
```

```
class Professor;

class CourseOffering
{
public:
 CourseOffering();
   ~ CourseOffering ();
private:
   Professor* instructor;
};
```

Roles on the end of the association can add clarity.

Instructor Notes:



Association Multiplicity

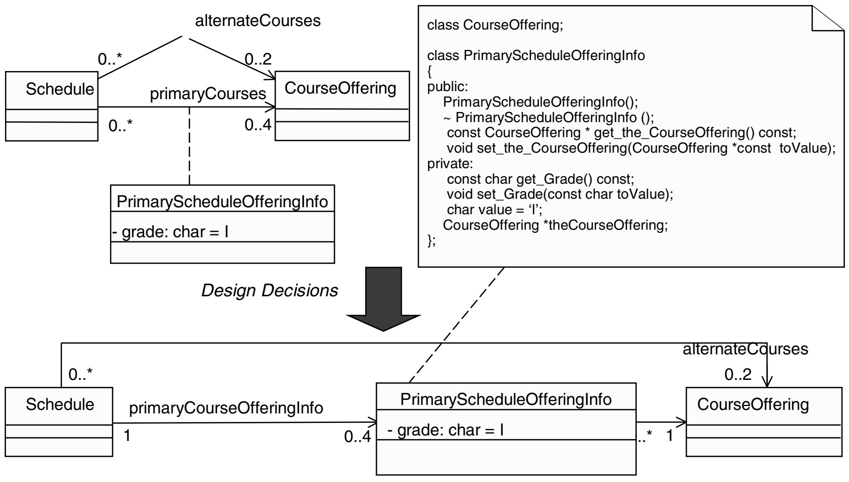
```
class CourseOffering
{
public:
 CourseOffering();
  ~ CourseOffering();
};
```

```
#include "CourseOffering.h"

class Schedule
{
public:
   Schedule();
   ~ Schedule();
 private:
   CourseOffering* primaryCourses[4];
};
```

Multiplicity is the number of instances of one class in relation to the other.

Instructor Notes:

Association Class



```
class CourseOffering;

class PrimaryScheduleOfferingInfo
{
public:
   PrimaryScheduleOfferingInfo();
   ~ PrimaryScheduleOfferingInfo ();
    const CourseOffering * get_the_CourseOffering() const;
    void set_the_CourseOffering(CourseOffering *const  toValue);
private:
    const char get_Grade() const;
    void set_Grade(const char toValue);
    char value = 'I';
   CourseOffering *theCourseOffering;
};
```

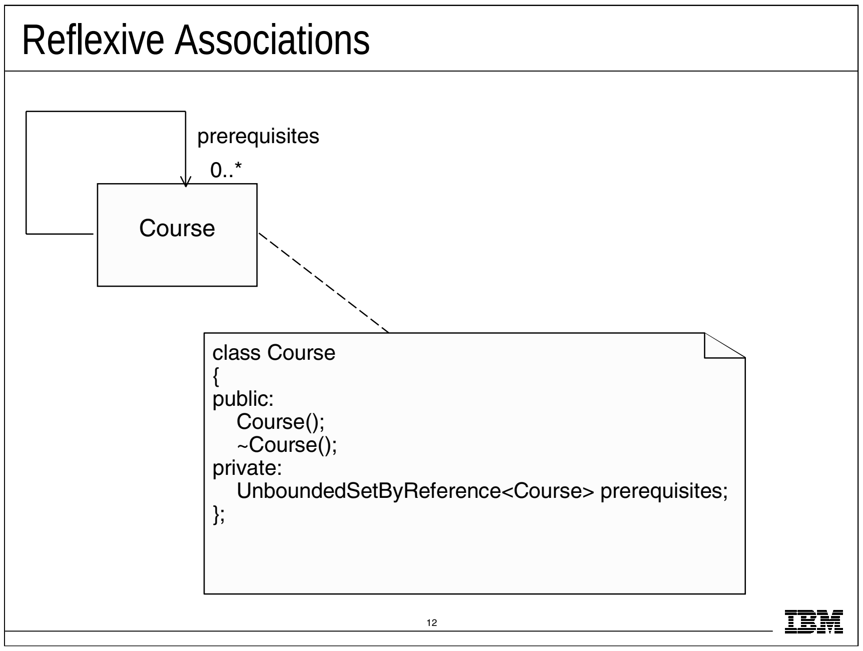
11

Remember, an association class is a class that is connected to an association. There is an instance of the association class for every instance of the relationship (for example, for every link).

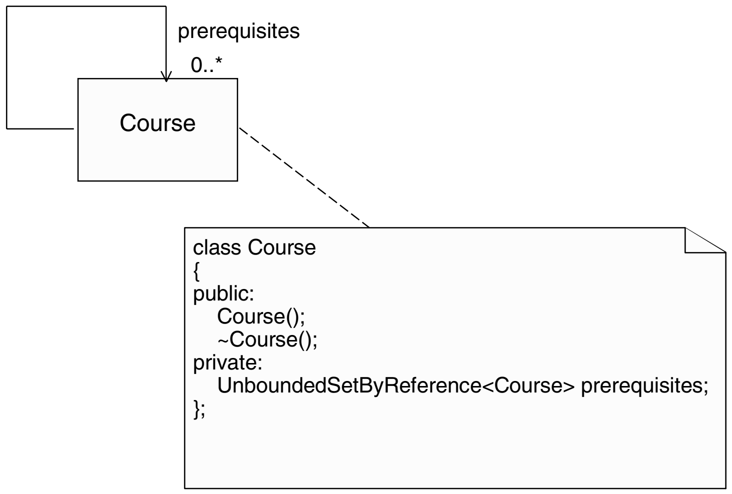
During design, some decisions are made regarding navigation between the involved classes.

A subset of the class operations and attributes are shown above. For this example, we included a subset to demonstrate the UML construct we are emphasizing.

Instructor Notes:



Reflexive Associations



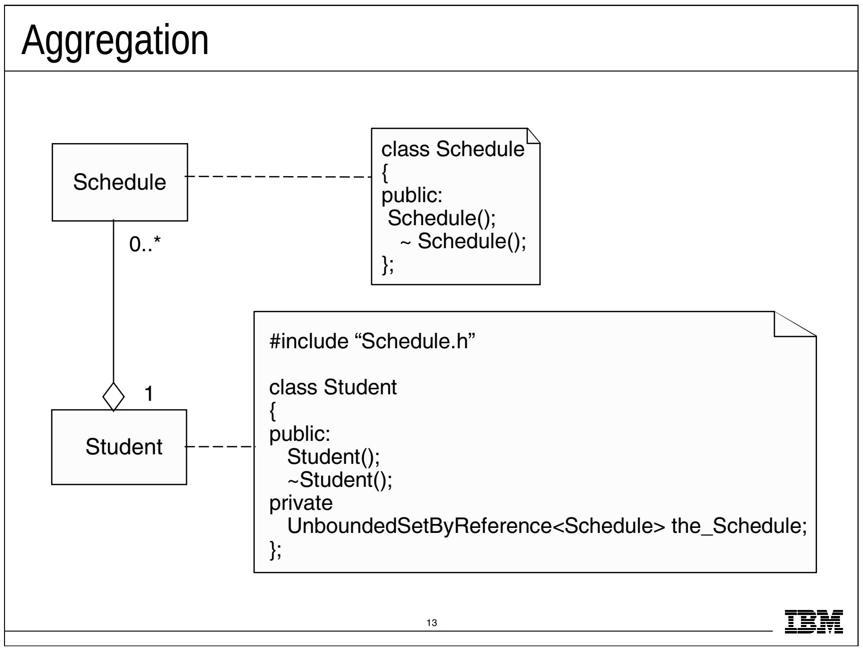
```
class Course
{
public:
    Course();
    ~Course();
private:
    UnboundedSetByReference<Course> prerequisites;
};
```

12

A class may have an association with objects of the same type.

In the above example, unboundedSetByReference is just an example of a template (that supports and unbounded list of courses).

Instructor Notes:



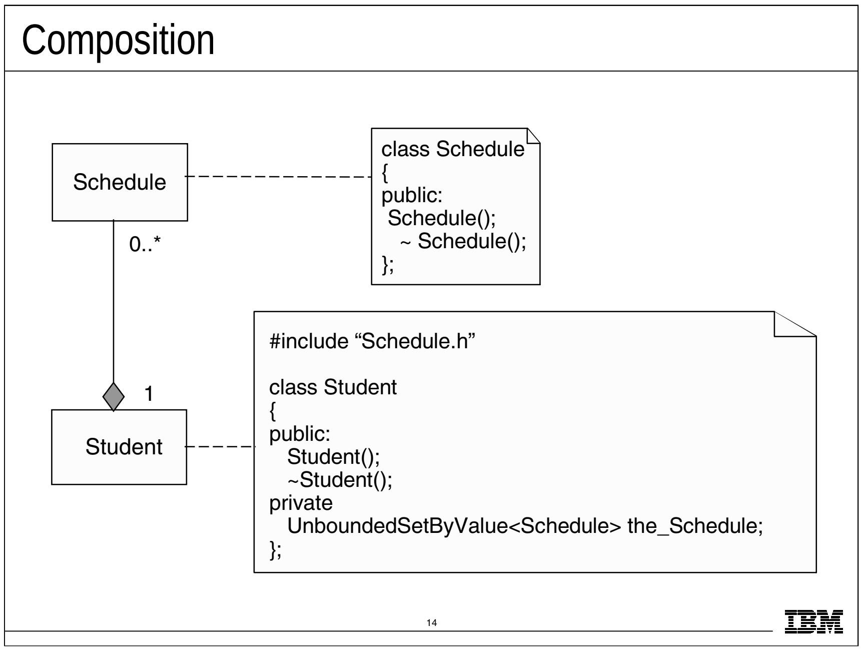
Aggregation

```
class Schedule
{
public:
 Schedule();
   ~ Schedule();
};
```

```
#include “Schedule.h”

class Student
{
public:
   Student();
   ~Student();
private
   UnboundedSetByReference<Schedule> the_Schedule;
};
```

Note: UnboundedSetByReference represents a parameterized class that has been instantiated with “Schedule”. It could be replaced by any reusable list class available in the programming environment.

Instructor Notes:



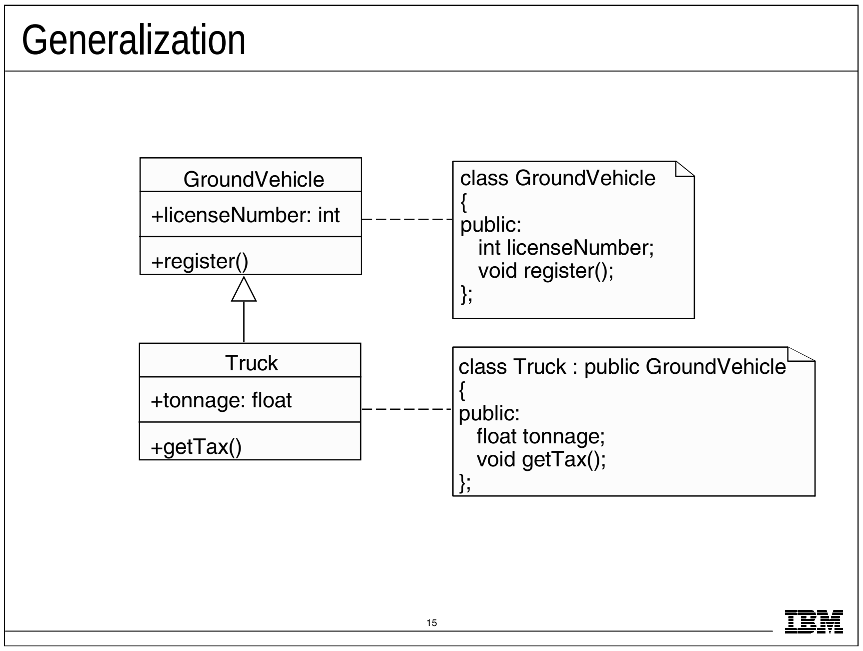
Composition

```
class Schedule
{
public:
 Schedule();
   ~ Schedule();
};
```

```
#include "Schedule.h"

class Student
{
public:
   Student();
   ~Student();
private
   UnboundedSetByValue<Schedule> the_Schedule;
};
```

Note: UnboundedSetByReference represents a parameterized class that has been instantiated with "Schedule". It could be replaced by any reusable list class available in the programming environment.

Instructor Notes:



Generalization

```
class GroundVehicle
{
public:
   int licenseNumber;
   void register();
};
```

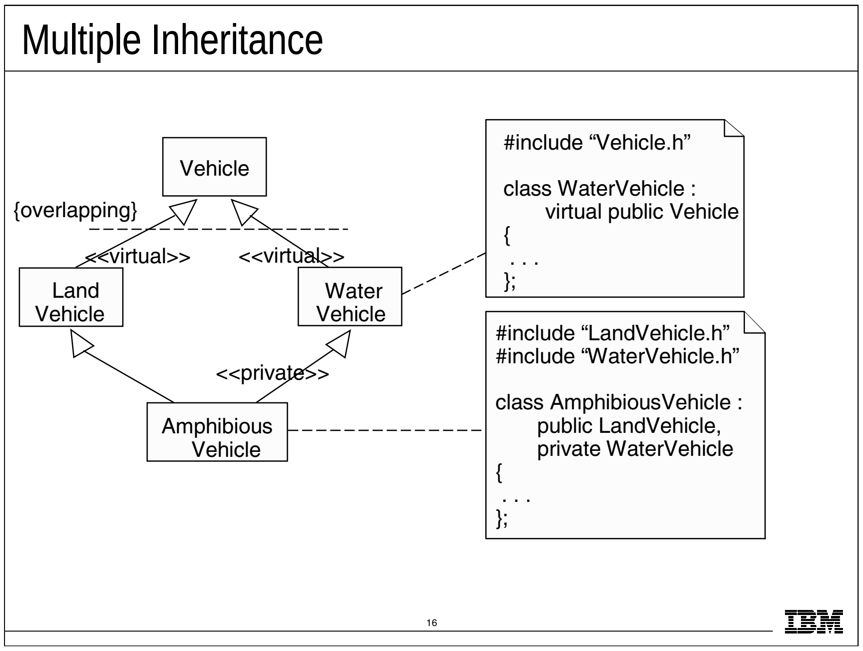
```
class Truck : public GroundVehicle
{
public:
   float tonnage;
   void getTax();
};
```

Remember generalization is a "is-a" or "kind-of" association. It is used to represent generalization / specialization.

Generalization is modeled as an open triangular arrowhead pointing to the base on the base class end.

In C++, inheritance is a language implementation mechanism for specialization. The use of inheritance does not guarantee substitutability (i.e., is-a programming), so the distinction is important. Delegation is another way to implement specialization.)

Instructor Notes:



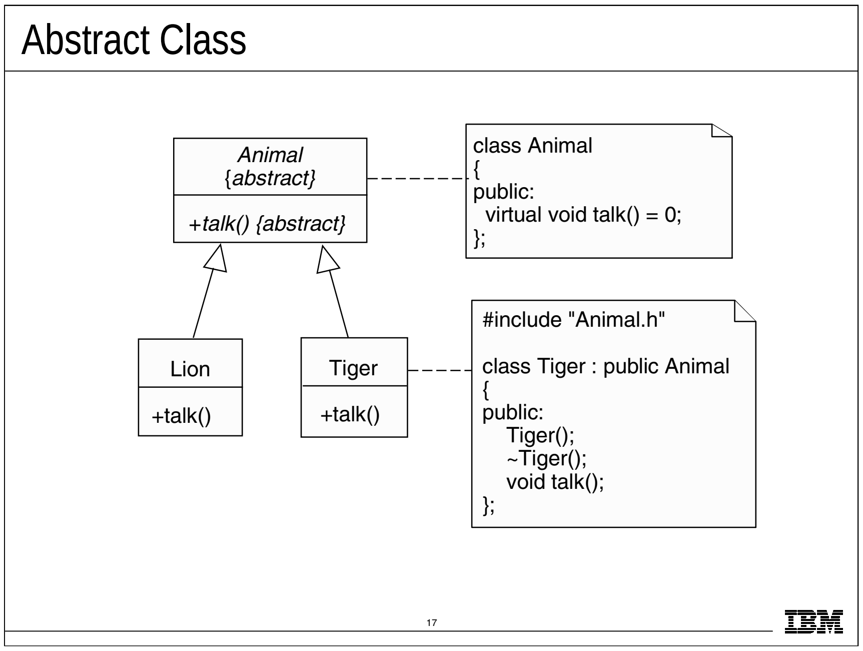
Multiple Inheritance

```
#include "Vehicle.h"

class WaterVehicle :
      virtual public Vehicle
{
 . . .
};
```

```
#include "LandVehicle.h"
#include "WaterVehicle.h"

class AmphibiousVehicle :
      public LandVehicle,
      private WaterVehicle
{
 . . .
};
```

Remember, subclasses that are not mutually exclusive can be annotated with the UML {overlapping} constraint. This supports multiple inheritance.

Instructor Notes:



Abstract Class

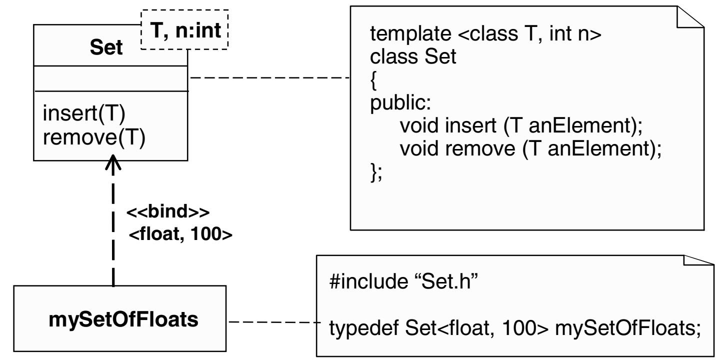
```
class Animal
{
public:
  virtual void talk() = 0;
};
```

```
#include "Animal.h"

class Tiger : public Animal
{
public:
    Tiger();
    ~Tiger();
    void talk();
};
```

Remember, an abstract class is a class for which no instances are created. In C++, an abstract class contains at least one pure virtual function.

Instructor Notes:

Parameterized Class



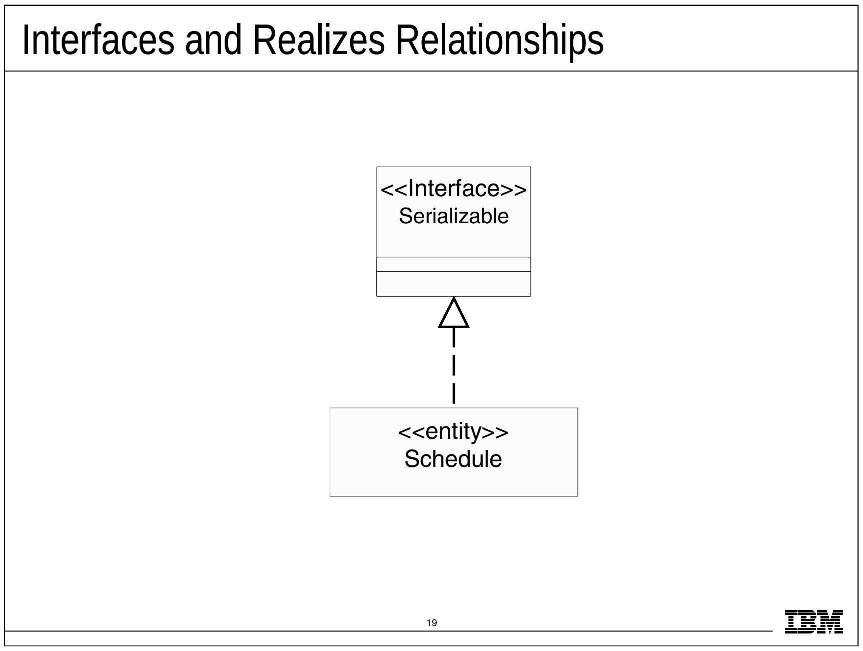
```
template <class T, int n>
class Set
{
public:
    void insert (T anElement);
    void remove (T anElement);
};
```

```
#include "Set.h"

typedef Set<float, 100> mySetOfFloats;
```

18

Remember, a parameterized class is a class which defines other classes. They are often used for container classes.

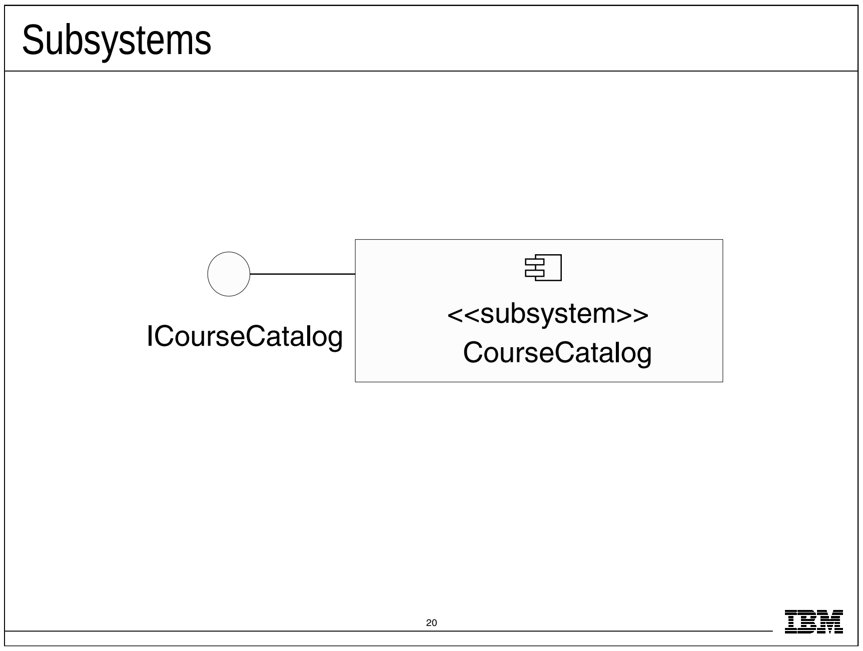
Instructor Notes:



Interfaces and Realizes Relationships

C++ does not directly support interfaces and the realizes relationship. Generalization and abstract base classes are used to simulate realizes and interfaces. Abstract base classes are used for the interfaces. To “realize” an interface, another class inherits from the abstract base class.

Instructor Notes:



Subsystems

20

As discussed within the OOAD course, subsystems are the Design Model representation for components.

Subsystems do not map to a specific C++ language construct.