Assignment 3
Due Monday, 9/18/2000

- Identify a partner with whom you will work on this assignment.
- Using the set of use cases provided for the meeting scheduler system, develop a set of CRC (Classes, Responsibilities, Collaborations) cards.
- Also develop a traceability matrix that shows which use cases are handled by which classes and responsibilities.

Traceability Matrix Example (from the graph-drawing example)

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Drawing: remember components</th>
<th>Class: draw shape</th>
<th>Shape: remember position</th>
<th>Responsibility: Shape: remember size</th>
<th>Class: remember connectors</th>
<th>Connector: draw</th>
<th>Connector: remember start</th>
<th>Connector: remember end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draw shape</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move shape</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erase shape</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resize shape</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connect shapes</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erase connector</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Modeling Class Structure with UML

OMG = “Object Management Group”, a consortium, not a company
http://www.omg.org/

UML

- Unified Modeling Language
- unifies the approaches of the “three amigos”:
  - Grady Booch
  - Ivar Jacobson
  - James Rumbaugh
- Includes E-R (Entity-Relationship) diagrams from database world.
The two amigos

Classes are shown by boxes

Classes, not actual objects

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

<table>
<thead>
<tr>
<th>Student</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
</tr>
<tr>
<td>Date of birth</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course Offering</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td></td>
</tr>
<tr>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>Instructor</td>
<td></td>
</tr>
</tbody>
</table>

Operations (methods) may be listed

<table>
<thead>
<tr>
<th>Student</th>
<th>methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>enroll</td>
</tr>
<tr>
<td>Address</td>
<td>drop</td>
</tr>
<tr>
<td>Date of birth</td>
<td>assign grade</td>
</tr>
<tr>
<td>take course</td>
<td></td>
</tr>
<tr>
<td>graduate</td>
<td></td>
</tr>
</tbody>
</table>

*methods (more detail can be given, such as argument and result types, and visibility)*
What is an Association?

- An association exists when an object of one class *needs to know* about one or more objects in another class.
- Associations are abstract.
- Typically associations are implemented by pointers or references, although one should not infer that this is *the* implementation. We *could* have an association determined in some other way.
Directionality of Associations

- By default, associations allow “bidirectional” navigation:
  
  From an object in either class, one can get to an associated object 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 Textbook but not vice-versa.

- This is sometimes called a “navigation arrow”.

- If absent, then navigation is assumed to be bidirectional.
Ordered Reading of Associations

Arrowhead shows direction of **reading** the name of the association, e.g. “A Course Offering uses a Textbook”.

Ordered Reading of Associations

“A Textbook is used in a Course Offering”.
Ordered vs. Directional

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

Associations 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 \( n \).
- \( m .. * \) means \( m \) or more.
- \( * \) means the same as \( 0..* \) (0 or more).
- \( a, b, c, ... \) means one of \( a, b, c \ ... \)
- \( 0, 1 \) or \( 0..1 \) means optional.

Roles in Association

Since this is a class diagram and not an object diagram, it is not implied that start and end are the same Time.
Exercise: Identify Roles

Exercise: Identify Multiplicities
Note on Multiplicities

- Multiplicity should be the one that you wish the **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.*

Association Classes

An association may itself take the form of an object relating two or more other objects together.

- Course Offering
- Student
- Enrollment

Here an Enrollment is an association object relating a Student and a Course Offering.
Multi-Way Associations

Associations aren’t limited to 2-way.

![Diagram of 3-way association class]

Aggregation

An aggregation is a special form of association in which a collection of objects is associated with a single object.

![Diagram of aggregation]

Unfilled diamond means "aggregation": components exist independent of container.
An object can be in multiple distinct aggregations.

Filled diamond means "composition": components are inseparable, non-sharable, part of container. The container is composed of the components (and possibly others). Multiplicity 1 is thus implied.
Question

- Can an object be in an aggregation and a composition simultaneously?

Possible C++ comparison

- **Aggregation**
  ```cpp
class College
{
  list<Student*> students;

public:
  void addStudent(Student *s)
  {
    students.add(s);
  }
  ...
}
```

  - Students exist outside of the college.

- **Composition**
  ```cpp
class College
{
  Building* buildings;

public:
  College(int n)
  {
    building = new Building[n];
  }
  ...
}
```

  - Buildings don't exist outside of the college.
C++ Destruction Note

- With **composition**, contained objects should be created/destroyed when the containing object is.

- With **aggregation**, aggregate objects are created and destroyed independent of the aggregating object.

Exercise: Identify Likely Aggregations and Compositions

```
Room    Building
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Meeting</td>
<td>Department</td>
</tr>
<tr>
<td>Meeting</td>
<td>College</td>
</tr>
<tr>
<td>Time Slot</td>
<td>Course</td>
</tr>
<tr>
<td>Semester</td>
<td>Major</td>
</tr>
<tr>
<td>Course Offering</td>
<td>Enrollment</td>
</tr>
<tr>
<td>Instructor</td>
<td>Student</td>
</tr>
<tr>
<td>Textbook</td>
<td>Author</td>
</tr>
</tbody>
</table>

College
| Major|
| Building|
| Semester|
```
Identify Likely Aggregations and Compositions

Qualified Association

An attribute indicating how to locate the associated object.
Comparison:
Qualified vs. Unqualified Association

Exercise: Identify Opportunities for Qualified Association
Further Notes on Aggregation and Composition

- Composition cycles are not allowed

In some models, aggregation cycles are not allowed either; just why is unclear.
Further Notes on Aggregation and Composition

- Exactly why cycles are not allowed for pure aggregation is unclear (It is clear in the case of composition.)

- We will later see a way to get around this restriction if it is present.

Further Notes on Aggregation and Composition

- From "The Unified Modeling Language Reference Manual" (Rumbaugh, Jacobson, and Booch), p 148:
  - "The distinction between aggregation and association is often a matter of taste... Keep in mind that aggregation is association."
  - "The only real semantics that aggregation adds is the constraint that chains of aggregate links may not form cycles..."
  - "Think of it as a modeling placebo."
Inheritance/Generalization

- Open triangle read upward: "inherits from", "extends", or "specializes".

No multiplicity not an association

- Open triangle read downward: "generalizes".

Called the base class.

Called the derived class.

Liskov Substitution Principle
(Prof. Barbara Liskov, M.I.T.)

A member of a derived class must also make sense when used as a member of the base class.

For example, if a method has an object of a class as an argument, the same method should be able to work with an object of a derived class.

As originally stated: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.
Implied Associations
(by the Liskov Substitution Principle)

Designed association

Implied association

Derived Association Classes

2-way association class

possible derived 2-way association class
Usually there will be multiple derived classes

This notation is equivalent to that on the preceding slide.
“Multiple Inheritance” is possible, although should be avoided (not all languages support it)

```
Faculty Member

Department Chair

Administrator

Faculty Member
```

"Interface Inheritance" alternative

```
«interface» Administrator

Department Chair

Faculty Member
```

In Java, Interface is a part of the language definition; in C++ it is a matter of interpretation.

Dashed line denotes Interface inheritance.
Alternative Notation for Interface

Recursive Structure (1)

Aggregation cycles: bad in some cases

"A container contains zero or more items, each of which contains a container."

At most one of these is regarded as legal.
Use inheritance to articulate recursive structures.

“A container contains 0 or more items. An item can be either an atomic item or a container.”

Object vs. Class

- UML uses a notation similar to classes for objects

- Object diagrams are like class diagrams, except that
  - boxes are objects rather than classes
  - lines are links rather than associations
Object Notation

- An object is visually distinguished from a class by an **underline**. The fully named object has the form:
  
  `objectName : className`

- The class name may be omitted if we don’t know it yet.

- The object name may be omitted: anonymous object.

Objects, like classes, are shown by boxes

```
Shape

myShape : Shape

Class

Object
```
Objects, like classes, are shown by boxes

Attributes may be listed with values for objects

For classes, attributes don’t have values, (unless they are class-wide attributes i.e. "static" in C++/Java).
Object Links

**Classes**

- Shape
- Connector

**Objects**

- myShape: Shape
- myConn: Connector

A link is an element of an instance of an association. An instance of an association is a set of links.

Multi-way Links

- Abner: Man
- Daisy Mae: Woman
- Sam: Authority

Marriage is a 3-way link involving the groom, bride, and agent.
Recursive Structures

Classes

Objects
(all are items)

In addition to object diagrams, the object notation is used in:

- collaboration diagrams
- sequence diagrams
- and others
class Shape
{
    public:
        Shape(); // default constructor
        Shape(Shape & orig); // copy constructor
        ~Shape(); // destructor
        void setPosition(Position p); // setters
        void setSize(Size s);
        void setColor(Color c);
        Size getSize(); // getters
        void draw(Graphic g); // other actions
        Shape & operator=(Shape &original); // assignment
    private:
        Position position;
        Size size;
        Color color;
        ...
};

UML - > C++:
One-way navigability

class Shape
{
    public:
        ...
    private:
        ...
};
class Connector
{
    public:
        ...
    private:
        ...
};
Disclaimer

- The C++ code examples are samples of what can be done.
- They are generally not the only way a specific type of association can be implemented.
- A specific tool will generate a specific type of implementation; selection from a menu of implementations might be possible.
- Use of a standard library, such as STL, is possible.

UML -> C++:
Multiple associations with different roles

```cpp
class Shape {
public:
    ... 
private:
    ... 
};
```

```cpp
class Connector {
public:
    ... 
private:
    ... 
Shape *start;
Shape *end;
};
```
UML - >C++ Exercises

Generally choose one, not both. Each calling the other would be a problem.
```cpp
class Shape
{
    public:
        ...
    private:
        ...
};

class Connector
{
    public:
        ...
    private:
        ...
};
```

```cpp
class Shape
{
    public:
        ...
    private:
        ...
};

class Connector
{
    public:
        ...
    private:
        ...
};
```
class Shape
{
public:
...
private:
...
};

class Connector
{
public:
...
private:
...
};

class Connection
{
public:
...
private:
...
};

class Car
{
public:
    Car();
    ...
private:
    Chassis & chassis;
    ...
};

Car::Car() // constructor
    : chassis (new Chassis())
{}
Inheritance/Generalization

class Staff
{
public:
    Staff(String name);  // constructor
    ...  
};

class Faculty : public Staff
{
public:
    Faculty(String name);  // constructor
};

Faculty::Faculty(String name)
: Staff(name)
{
    ...  
}  

Recursive Structure, C++
// The base class

class Item
{
protected:
    char *name;

public:

    // construct an Item, copying name into it
    Item(char *name)
    {
        this->name = strcpy(new char[strlen(name)+1], name);
    }

    // destroy the item
    virtual ~Item()
    {
        delete [] name;
    }
};  // class Item

See turing:/cs/cs121/codeExamples/virt.cc for more details.

// Atom is an Item that cannot contain others
class Atom : public Item
{
public:

    // construct an Atom
    Atom(char *name)
    : Item(name)
    {
    }
};  // class Atom

// ItemCells are used to construct an open linked-list of Items
class ItemCell
{
private:
    Item *item;        // the first item in a list represented by this cell
    ItemCell *next;    // the next cell in the list, i.e. the rest of the items

public:

    // construct an ItemCell based on a first item and next ItemCell
    ItemCell(Item &newItem, ItemCell *next)
    : item(newItem)
    , next(next)
    {
        this->next = next;
    }

    // add an Item to a Container
    void add(Item &item)
    {
        items = new ItemCell(item, items);
    }
};  // class ItemCell

// Container is an Item that can contain others
class Container : public Item
{
private:
    ItemCell *items;  // list of Items

public:

    // construct a Container with a given name
    Container(char *name)
    : Item(name)
    {
        items = 0;  // empty list
    }

    // add an Item to a Container
    void add(Item &item)
    {
        items = new ItemCell(item, items);
    }
};  // class Container

// Atom is an Item that cannot contain others
class Atom : public Item
{
public:

    // construct an Atom
    Atom(char *name)
    : Item(name)
    {
    }
};  // class Atom

// ItemCells are used to construct an open linked-list of Items
class ItemCell
{
private:
    Item *item;        // the first item in a list represented by this cell
    ItemCell *next;    // the next cell in the list, i.e. the rest of the items

public:

    // construct an ItemCell based on a first item and next ItemCell
    ItemCell(Item &newItem, ItemCell *next)
    : item(newItem)
    , next(next)
    {
        this->next = next;
    }

    // add an Item to a Container
    void add(Item &item)
    {
        items = new ItemCell(item, items);
    }
};  // class ItemCell

// Container is an Item that can contain others
class Container : public Item
{
private:
    ItemCell *items;  // list of Items

public:

    // construct a Container with a given name
    Container(char *name)
    : Item(name)
    {
        items = 0;  // empty list
    }

    // add an Item to a Container
    void add(Item &item)
    {
        items = new ItemCell(item, items);
    }
};  // class Container