CS211 Lecture: Relationships Between Objects: Association, Aggregation, and Composition; Class and Object Diagrams in UML; Collections

Last revised July 22, 2003

Objectives:

1. To introduce UML Class and Object diagrams
2. To explain the association relationship between objects, adornments possible on such relationships, and ways of using these relationships
3. To introduce aggregation and composition associations
4. To introduce the use of Collections when modeling object relationships

Materials:

1. Handout of class diagram for ATM Example

I. Introduction

A. We have seen that, regardless of what software development model is followed, certain tasks will need to be done as part of the development process *per se* - whether all at once, iteratively, or incrementally.

1. Analysis. The goal of this task is to *understand* the problem.

2. Design. The goal of this task is to develop the *overall structure* of a solution to the problem in terms of individual, buildable components and their relationships to one another.

3. Implementation. The goal of this task is to actually *build* the system as designed.

4. Quality Assurance. The goal of this task is to *ensure* that the individual components and the system as a whole do what they are supposed to do (which involves identifying their shortcomings and fixing them.)

5. Deployment and Maintenance. Actually using the software and making changes as necessary.

B. Design is sometimes divided into high-level design and detailed design. (We are focussing on the former now) In performing high-level design - i.e. developing the overall structure of the system, we have three subtasks to perform:

1. Identify the classes that need to be part of the system.

2. Assign responsibilities to each class.

3. Identify the relationships between various classes

C. We have already discussed identifying classes and assigning responsibilitie to them. Today, we begin discussing the topic of identifying the *relationships* between the various classes. We will document these relationship by the use of UML Class Diagrams.

1. HANDOUT: Class Diagram for ATM Example

2. Actually, UML has two similar kinds of diagrams, called *class diagrams* and *object diagrams*. (Our text calls the latter instance diagrams.)

a) In a class diagram, the rectangular boxes stand for *classes*.

b) In an object diagram, the rectangular boxes stand for *individual objects*.

c) How do we distinguish between these two kinds of diagrams?

In a class diagram, each box contains one name - the name of the class. In an object diagram, each box contains two names, separated by a colon - the name of the individual object and the name of the class to which it belongs. Furthermore, in an object diagram the pair of names is (generally) underlined

(1) Example: Student - might appear in a class diagram

(2) Example: joe : Student - might appear in an object diagram

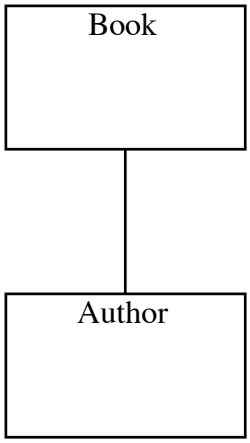
(3) In object diagrams, an object is often anonymous - i.e. it has just a class name. We still include the colon, however, to make it clear whether we are specifying an anonymous object of a specified class (colon before the name) or - more rarely - a named object of an unspecified class (colon after the name).

Example: : Student - might appear in an object diagram to stand for a Student whose name is unknown or unimportant

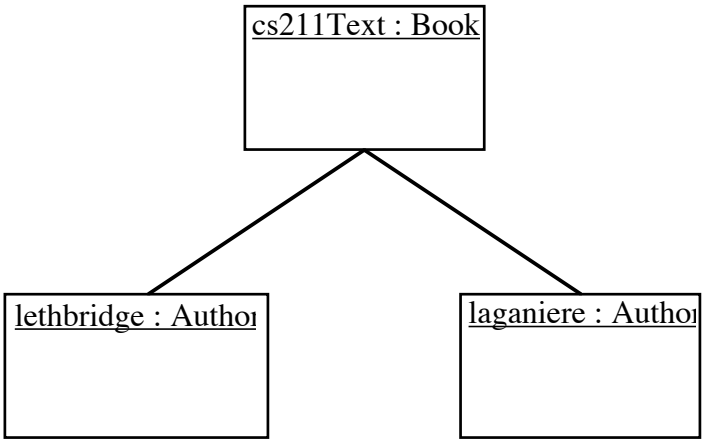
d) Another distinctive feature of the two kinds of diagrams arises when there is more than one object of a particular class involved. In a class diagram, one rectangle is drawn showing the class; in an object diagram, each object has its own rectangle.

EXAMPLE: Suppose we wanted to draw a diagram showing the relationship between a book and its author(s), using the classes Book and Author.

(1) As a class diagram, this might be drawn as:

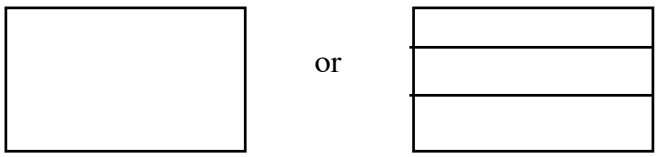


(2) As an object diagram, the specific case of the text for our course could be depicted as:



e) Another distinction is that, in a class diagram, each class can be represented by a box with three compartments, representing its name, its attributes, and its operations. Because all this detail tends

to produce a cluttered diagram (unless very few classes are involved), the compartments are often omitted. E.g.



Note: an *empty compartment* means there are *no* attributes or operations (as the case may be). So, if you don't intend to spell out the list of attributes or operations, don't draw the compartment either!

f) Note that a given diagram is *either* a class diagram or an object diagram - only one or the other kind of symbol occurs in it.

g) The diagram just distributed is obviously a class diagram. We will see examples of object diagrams when we discuss interaction diagrams later in the course.

D. At the outset, we note that there are two different sorts of relationship, that we handle similarly but need to keep distinct in our thinking.

1. There are relationships between *individual objects*. Such a relationship describes how a particular object of one class relates to a particular object of another class.

a) Among humans, the relationship known as marriage is such a relationship. It relates one individual to another specific individual. You may know many married people, but each has a different spouse.

b) In the OO world, the link along which a message is sent from an object to one of its collaborators is such a relationship - a particular sender sends a message to a particular receiver. (That is, the Collaborators column of a CRC card is documenting associations.)

c) In this case, then, each individual object participates in the relationship (or doesn't participate in the relationship, as the case may be) with its own particular partner or partners.

d) Where things get a bit confusing is that when we identify an individual relationship between objects, we are also identifying a relationship between the corresponding classes. The fact that an object of class Book is related to one or more objects of class

Author implies that there is a relationship between the *classes* Book and Author such that a member of the one class can participate in this relationship with a member of the other class.

2. There are relationships *between classes*. Such a relationship describes how one whole class of objects is related to another class.

   a) Among humans, the fact that all CS majors are also students is such a relationship.

   b) In the OO world, generalization, or inheritance, is such a relationship.

   c) In the case of a class relationship, all the objects that belong to a given class participate in the relationship in the same way.

3. In drawing a class diagram, we can depict *all* kinds of relationships - even those that are actually relationships between individual objects. (Recall the example of a book and its authors - the relationship is one between individual objects, but it can be shown in a class diagram.) In an object diagram, we can depict *only* relationships between individual objects, not those between classes. Thus, we will be using Class Diagrams exclusively in these lectures. (Indeed, the class diagram is the more frequently used type of diagram in UML in general.).

E. In this series of lectures and the next, we will discuss four kinds of relationships (three of which are exemplified in the ATM class diagram handed out.). We will consider object relationships (the first kind) in this lecture, and class relationships (the next three) in the next lecture.

   1. *Association* - a relationship between objects. In a UML diagram, this is represented by a solid line, possibly with a plain arrow head on one end, and possibly with an open or filled diamond on one end.

   EXAMPLES FROM DIAGRAM

   2. *Dependency* - a relationship between classes. In a UML diagram, this is represented by a dashed line with an arrowhead on one end.

   EXAMPLES FROM DIAGRAM

   3. *Generalization* (inheritance) - a relationship between classes. In a UML diagram, this is represented by a solid line with a triangle on one end.

   EXAMPLES FROM DIAGRAM

4. *Realization* - a relationship between a class and an interface. In a UML diagram, this is represented by a dashed line with a triangle on one end. (Note that the symbol is similar to that for generalization, because realization is similar to inheritance.)

   NO EXAMPLES IN DIAGRAM - WILL DISCUSS BELOW

II. Relationships Between Objects: Associations

A. Relationships between individual objects are called *associations* in UML. They are depicted by a solid line on a class diagram, or an object diagram.

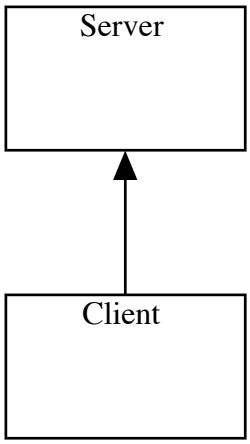
B. Technically, the fact that an object of class A can be associated with an object of class B is called an *association* and the corresponding connection between a specific object of class A and a specific object of class B is called a *link*. That is, an association is conceptually a *set* of links.

C. In the simplest case, an association may simply be drawn as line. But often, the line has one or more *adornments* that provide further information about the association.

1. Navigability (directionality):

   a) Ordinarily, associations are conceived of as being bidirectional - e.g. in the diagram showing the association between a Book and its Author(s), we probably intend for it to be possible to go from a Book object to its Author object(s), and likewise to go from an Author object to the Book(s) it is the author of.

   b) Sometimes, though, an association is conceptually unidirectional - e.g. if were to try to depict the relationship between a Server system and a Client system that uses it, we might draw it this way:

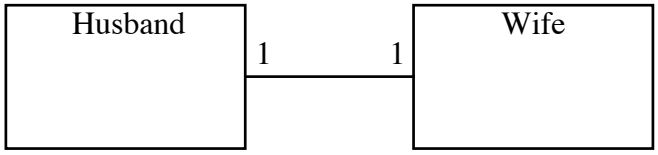


The arrow says that the Client must know about the Server, but the Server does not need to know about the Client (except briefly, during the time it is responding to a message received from the Client.)

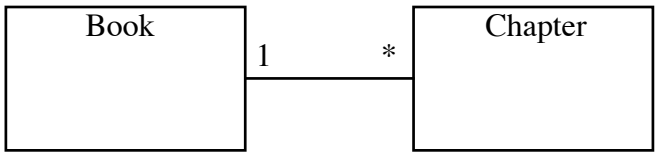
c) Why would we want to identify an association as being unidirectional where this is appropriate is? The presence of an association in the class diagram implies that the implementation will need to maintain information about this association. Keeping information about a bidirectional association means that both objects will have to maintain information about the association. If this is not necessary, maintaining the association in only one direction will simplify the implementation.

2. Multiplicity: Some associations are conceptually one to one - one object of a given type relates to one object of another type. Others allow one object of a given type to be related to many objects of another type. Here are some different situations that often arise, and the corresponding UML representation:

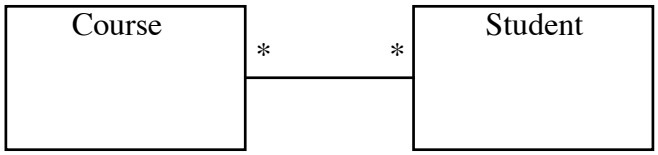
a) One-to-one. Example: marriage (at least as intended!)



One-to-many: Example: the relationship between a book and the individual chapters that are part of it.

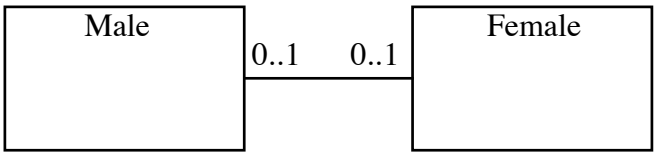


b) Many-to-many: Example: students and courses

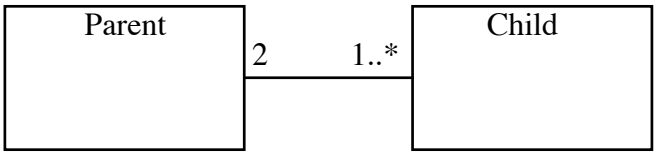


c) Often, the multiplicities will be expressed as *ranges*, rather than as simple values

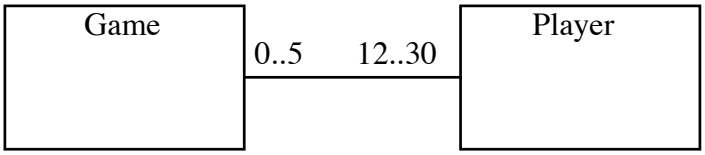
(1) Example: the marriage relationship above was shown as 1 to 1 between the classes Husband and Wife. If it were shown as a association between class Male and Female, the multiplicities would need to be expressed as ranges. (One cannot be a Husband without being married, but one can be a Male without being married, so a Male can have 0 or 1 wives!)



(2) Example: a person has exactly two birth parents. A parent has at least one child, but can have any number:



(3) Example: the annual volleyball competition between the Math and CS wings of our department involves up to 5 games. In each game, at least 12 but no more than 30 students can participate.

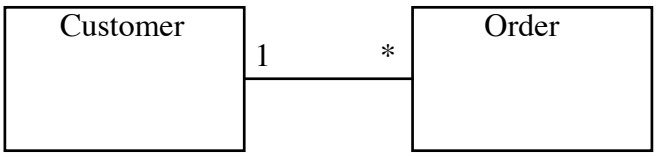


(This one's a bit contrived to illustrate a point, I admit :-).

(4) The symbol * we have previously used means "0 or more" - hence it is equivalent to 0..*

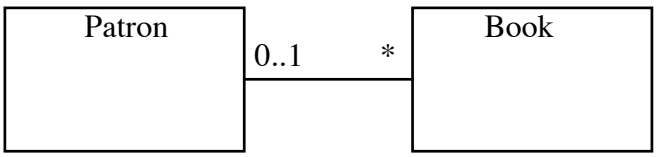
d) If the lower limit on the multiplicity of a certain relationship is 0, we say that the relationship is *optional*. If the lower limit is greater than 0, we say that the relationship is *mandatory*. Note that the same relationship may be optional in one direction, and mandatory in the other.

(1) Example: the relationship between a customer and the orders he/she has placed with a company. Assuming a person can register as a customer before placing an order, we have the following scenario:



The relationship from an order to a customer is mandatory - every order *must* be associated with a customer. The relationship from customers to orders is *optional* - a customer does not need to have any orders.

(2) It's certainly possible to have a relationship that's optional both ways - e.g. the relationship between a library patron and books. he/she currently has checked out. A patron does not have to have any books checked out at a given time, nor does any particular book have to be checked out at a given time.



(3) Recall that the notation "*" is short for "0..*", and so stands for a relationship that is inherently optional. If the relationship is mandatory, but of unlimited multiplicity, we must use the form "1..*".

(4) Also note that some writers (including our text), use the notation "n" instead of * in a range - so * (= 0..*) is written as "0..n" and 1..* is written as "1..n".

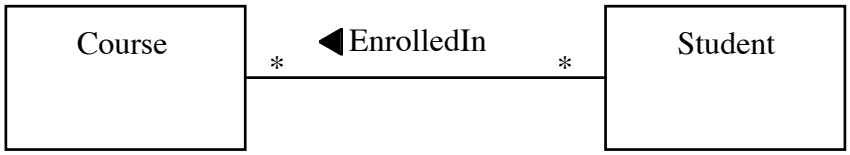
e) Note: Often in the literature the term "cardinality" is used for what we have called "multiplicity". The authors of the UML reference point out that - technically - cardinality refers to the properties of a *particular* instance of an association, while multiplicity refers to the overall properties of the association itself.

E.g. if there are 22 students enrolled in a given course, then the cardinality of the relationship between the object representing that course and the students in it is 22; but the multiplicity of the relationship between the class Course and the class Students might be, say 0..200 - assuming a course might have no students in it but cannot have more than 200.

We'll use the term "multiplicity" here - but understand that you will often see the term "cardinality" used to mean the same thing

3. Name: Often, the meaning of the association is implicit in the classes that are related, but sometimes an association will be given a name to make its meaning explicit.

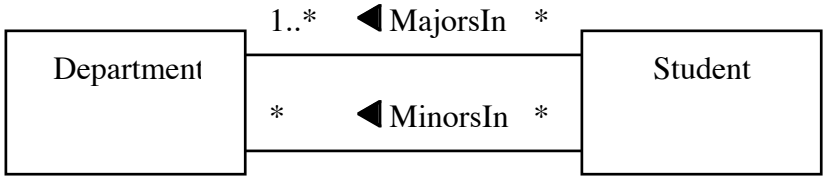
a) EXAMPLE:



(Note the arrow on the name, which indicates how it is to be read: "a student is enrolled in a course". It has nothing to do with navigability of the association itself, which is bidirectional in this case.)

b) Giving a name to an association is especially important in cases where there are two different relationships between the same pair of classes.

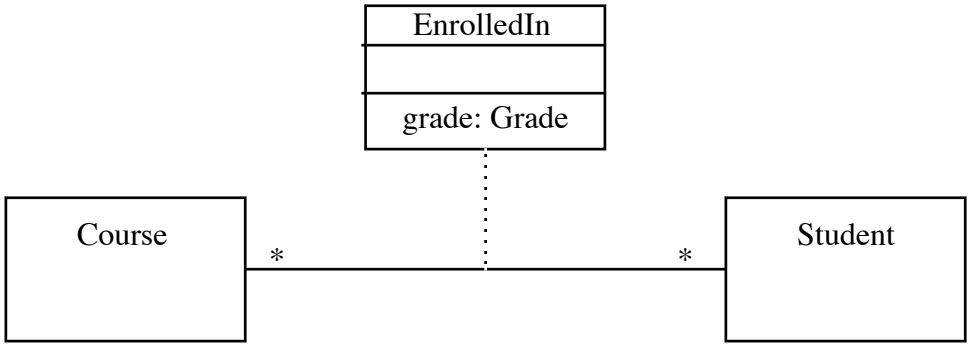
EXAMPLE



(Note that a student must have at least one major, but can have zero or more minors)

c) Note that association names typically begin with an upper-case letter, denoting that they are "class like". In fact, in some cases an association may need to be represented by an *Association Class*. This is particularly true when there are one or more attributes that are attributes of the association itself, rather than of the participating object.

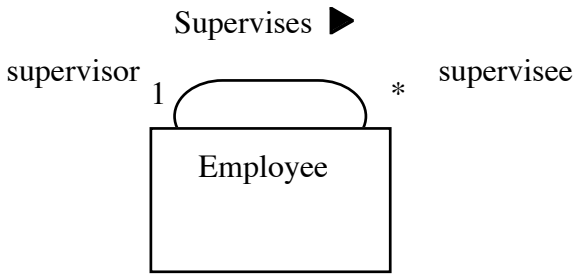
Example calling for an association class - the association between a student and a course, which has a grade attribute that is a property of the association - not of the student (who has individual grades for each course) or of the course (since there are individual grades for each student.)



(Note the use of the three sets of lines in the box representing the association class, to make it crystal clear that this is a class.)

4. Role: Often, the specific roles played by the two objects in a relationship is implicit in the classes to which they belong; but sometimes the roles are named explicitly: This is especially necessary in cases where an association connects objects of the same class to each other.

   EXAMPLE:



   Note: Care must be used in drawing a diagram to distinguish between the name of an association and role names. The latter should be drawn near the end of the association line; the former far enough from the ends to be clear that it is not a role.

5. Aggregation/Containment: Sometimes, an association is stronger than an ordinary association, in that one of the objects can be thought of as being *part of* the other - i.e. the relationship is one between a whole and its constituent parts. We call such an association *aggregation*.

   a) Aggregation is appropriate when we can meaningfully use the phrase "is a part of" to describe the relationship between the part and the whole, or "has a" to describe the relationship between the whole and the part.

   b) EXAMPLES:

      (1) In the ATM system, the CardReader, CustomerConsole, etc. objects are *parts of* the ATM object. This is a stronger connection than most of the examples of associations we have considered thus far.

      (2) The relationship between a course and its students might also be thought of as an aggregation, though this is perhaps a bit more debatable. (Perhaps most appropriate in a situation were we are modeling student registrations in a course.)

c) Aggregation is denoted in a UML diagram by putting a diamond on the "whole" part of the relation.

d) Aggregation actually comes in two forms: simple aggregation, and a stronger form, called *composition*.

(1) Composition has the additional characteristic that the "part" has no existence independent of the "whole". This leads to two additional characteristics:

(a) Each "part" can belong to only one whole.

(b) The "whole" is responsible for creating and destroying the "parts". Thus, the "parts" come into existence when the "whole" comes into existence; and if the "whole" is destroyed, the "parts" are destroyed too.

(c) Composition is denoted by using a *filled-in* diamond; whereas simple aggregation uses a hollow diamond.

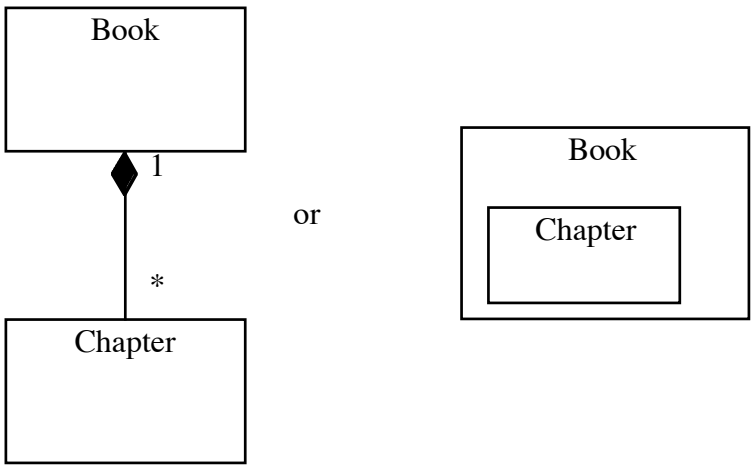
(d) Of the two examples we have considered:

i) The relationship between the ATM and its component parts is composition. One cannot imagine a component like a CardReader having an independent existence apart from an ATM (at least as far as the software is concerned), nor can a CardReader belong to two different ATM's.

ii) On the other hand, the relationship between courses and students is simple aggregation: students exist apart from their courses, and a given student can be - and typically is - a part of more than one course as the same time.

e) In the case of composition, there is an alternative representation possible in UML. That is to put the box representing the "part" class *inside* the box representing the "whole" class.

EXAMPLE: Consider the relationship between chapters of a book and the book itself. Clearly, each chapter is a part of one and only one book, and its existence is directly tied to the book of which it is a part. Thus, the association between a book and its chapters is a composition. *Either* of the following UML representations can be used:



The latter representation might be particularly appropriate if the Chapter objects are accessible to the outside world only by *going through* a Book object - i.e. if they don't enter into any relationships with outside objects on their own.

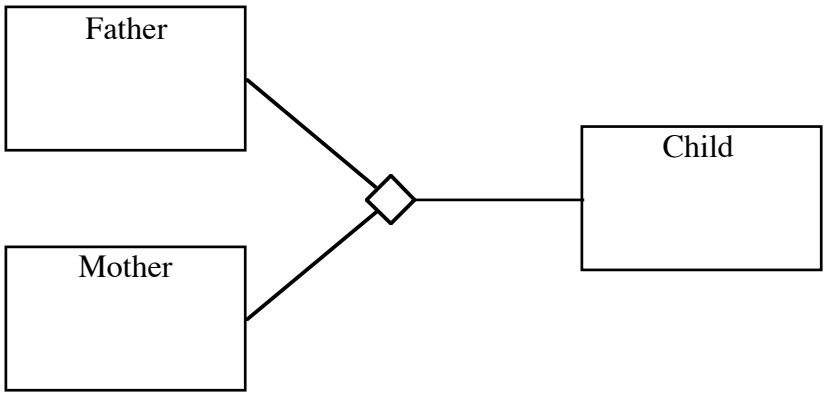
D. Almost all associations (including all the examples we have considered) are *binary associations*.

1. A binary association is one that has the following properties:

   a) Two classes are involved (or one class is involved in two ways)

   b) Each instance of the association links exactly two objects

2. It is also possible to have an n-way association that associates more than 2 classes. We will look at just one example: the relationship between a child and his/her two parents (a 3-way association):



E. Associations are used for three general purposes:

1. We have already seen that associations can be used to represent a situation in which an object of one class *uses* the services of an object of another object, or they mutually use each others services - i.e. one object sends messages to the other, or they send messages back and forth. (In the former case, the navigability can be monodirectional; in the latter case it must be bidirectional.)

2. We have also already seen that associations can be used to represent aggregation or composition - where objects of one class are wholes that are composed of objects of the other class as parts. In this case, a uses relationship is implicitly present - the whole makes use of its parts to do its job, and the parts may also need to make use of the whole.

3. As a third possibility, associations can also be used to represent a situation in which objects are related, even though they don't exchange messages. This typically happens when at least one of the objects is basically used to store information - e.g. in the AddressBook problem we did in CS112, this is the relationship between the AddressBook object and the various Person objects it stores. (The AddressBook doesn't directly send messages to Persons, though it can be used to retrieve a Person that some other object can then send a message to.)

   (Some writers call this a *weak relationship*. This is not a standard UML term, however.)

F. ON HANDOUT:

Discuss the various associations in the ATM example class diagram.

Note that the relationship between the ATM and its component parts could have been expressed by using the "box within box" representation.

III. Implementing Associations using Java References and Collections

A. Of course, the associations that are identified during the design phase will eventually have to be implemented in the Java code implementing the various classes. This typically takes the following form:

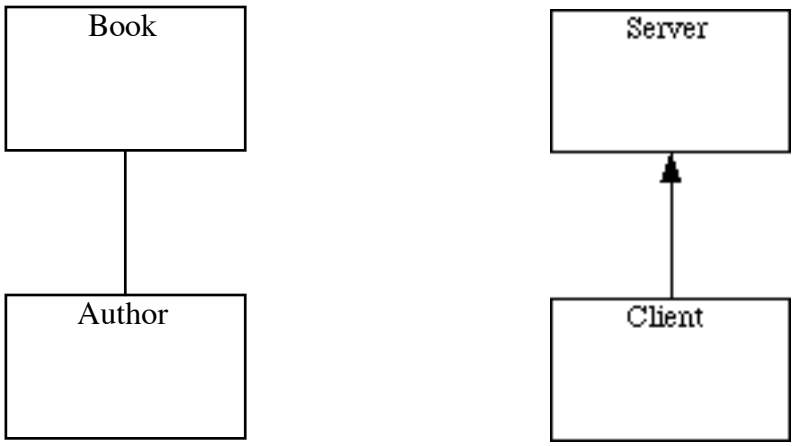
1. For each different association that relates objects of a given class to other objects, there will be a field in each object containing a link to the appropriate object(s).

a) If the association is bidirectional, *each* participating class will need such a field.

b) If the association is unidirectional, only the class whose objects need to know about their partner(s) will have such a field.

EXAMPLE:

Consider two cases that we looked at earlier:



In the first case, each Book object will need a field linking to the associated Author object(s), and each Author object will need a field linking to the associated Book object(s).

In the second each Client object will need a field linking to the associated Server object(s), but *not* vice-versa.

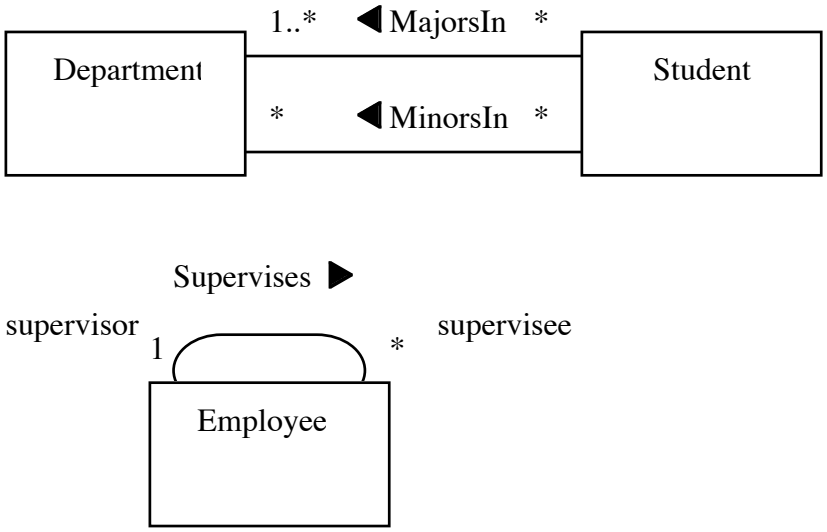
(This reduction in information that needs to be maintained is why we consider the possibility of unidirectional associations.)

2. Frequently, the field name will be derived from the name of the association, or from the role names, if such are present.

EXAMPLE: A Book object may contain a field called authors, and an Author object may contain a field called books.

EXAMPLE: A Client object may contain a field called server. The Server object, however, would <u>not</u> contain a field called client.

EXAMPLE: Consider two cases we looked at earlier



In the first case, a Student object might have fields called majors and minors. A Department object might likewise have fields called majors and minors.

In the second case, an Employee object might have fields called supervisor and supervisees. (Note the plural in the case of the latter name - the role is supervisee, but one supervisor can supervise multiple people.

B. There are a variety of different implementation approaches that can be used to actually realize the links.

1. If a given object can relate to only one other object in a given association (there is a "1" at the other end of the link), the easiest approach is to use a Java reference to the other object.

2. If the multiplicity is "0..1", the same strategy can be used, with the reference being null if there is no related object for this association.

3. If the multiplicity is some fixed, small integer, or is limited by some small fixed integer, then multiple fields can be used, or a field whose value is an array.

   EXAMPLE: Suppose we assume that a given student can have at most three majors and at most two minors. Then we might include fields like the following in a Student object:

```
Department major1, major2, major3;
Department minor1, minor2;
```

or

```
Department [] major;
Department [] minor;
```

(Of course, there are dangers if you cannot be sure that the upper limit is definite. However, three majors is probably enough for anyone!)

4. When (the upper end of) the multiplicity range is "*" (or some large integer), the first approach won't work, and the second is tricky unless you know when the object is created how many other objects it will be related to (since arrays in Java are created with a fixed size). A more flexible approach results from using Collections, which we will discuss next.

C. The Collections facility was added to Java as a part of JDK 1.2 (Java 2). However, it was "retrofitted" to also be usable with JDK 1.1.

1. A Collection is a group of objects that supports operations like:

   a) Adding objects to the Collection.

   b) Removing an object from the Collection.

   c) Accessing individual objects in the Collection.

   (Note that an array can be thought of as a very simple and limited form of Collection, but doesn't offer the full elegance of the Collections facility in the Java library)

2. Java Collections are of three basic types:

   a) *Sequences* are collections in which the contents are regarded as having some sequential order. (Note: the Java library calls these "Lists")

   (1) If a collection is a sequence, it is legitimate to ask questions like "what is the first object in the sequence?" or "what is the last object?" or "what is the ith object?". (Provided the collection is non-empty, in the first two cases - or has at least i+1 elements, in the last case - since elements are numbered starting at 0 - so to get, say, item 2 the collection must contain at least three elements.)

(2) It is also legitimate to make requests like "add this object at the very front" or "add this object at the very end" or "add this object in position i". (Provided the collection has at least i elements in the last case.)

(3) Finally, it is legitimate to make requests like "remove the first object" or "remove the last object" or "remove the ith object". (Provided the collection is non-empty, in the first two cases - or has at least i+1 elements, in the last case.)

b) Sets are collections in which a given object may appear at most once, and there is no ordering.

(1) If a collection is a set, it is legitimate to ask the question "is this particular object in the collection?" (yes or no).

(2) It is also legitimate to make the request "add this object to the collection". (Provided it's not already there.)

(3) Finally, it is legitimate to make the request "remove this object from the collection". (Provided it is in the collection to begin with.)

c) Maps are collections of key-value pairs. (E.g. a phone book is a kind of map, in which the keys are names and the values are phone numbers.)

(1) If a collection is a map, it is legitimate to ask questions like "what value - if any - is associated with the following key?" or "does this map contain the following key?".

(2) It is also legitimate to make the request "put the following key-value pair in the map". This can have one of two effects:

(a) If the key was not in the map to begin with, it is added with the specified value.

(b) If it was in the map, but with a different value, the old value is removed and the new value is associated with the key.

(3) Finally, it is legitimate to make the request "remove the following key from the map". If the key was in the map, it and its associated value are removed; if not, nothing happens.

d) For all types of Collections, it is possible to create an Iterator object that makes it possible to access each item in the collection once.

(1) For sequences, the order in which items are accessed by an Iterator is the sequential order first, second ...

(2) For sets and maps, the order is implementation-determined.

3. The Java Collections library contains two or more implementations for each of the different types of collection. The different implementations of a given type of Collection have the same behavior, but have different performance characteristics.

a) For List (Sequence), the Java library supplies:

(1) LinkedList - good if the list changes size (grows or shrinks) frequently), good for accessing either end of the list, but slower when accessing items in the middle of the list

(2) ArrayList - good if accessing elements by specific position, but slower for adds and removes.

b) For Set, the Java library supplies:

(1) HashSet - more efficient in most cases

(2) TreeSet - an iterator will access the elements of the set in a specific order based on their value (e.g. Strings would be kept in alphabetical order.)

c) For Map, the Java library supplies:

(1) HashMap - more efficient in most cases

(2) TreeMap - an iterator will access the elements of the map in key order.

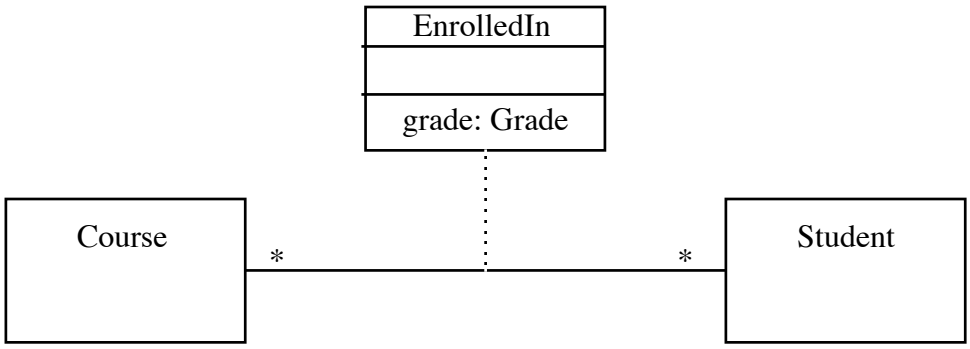
4. We will devote a lab to working with Java Collections

D. We noted earlier that if an association has attributes associated with the association itself (not just the participating objects), an association class can be used. In this case:

1. Each participating object contains a reference to the association class object.

2. The association class object contains references to each of the related objects.

   EXAMPLE:




```
class Course
  {
    ...
    (Some sort of collection of references to
      EnrolledIn objects) enrolledIn;
    ...
  }

class Student
  {
    ...
    (Some sort of collection of references to
      EnrolledIn objects) enrolledIn;
    ...
  }

class EnrolledIn
  {
    ...
    Course course;
    Student student;
    Grade grade;
    ...
  }
```

E. Finally, let's think about the various associations in the Video Store problem and how they might be represented by Java collections:

   1. The association between the store and its customers is clearly 1:*. When a customer wants to rent one or more items, he/she presents a

card, and the ID number on the card is used to access the stored information on the customer.

What type of collection is appropriate in the Store object to hold references to Customer objects for this case?

ASK

2. With regard to the rentable items, the Video Store class structure needs to make use of a design pattern. What pattern?

   ASK Class

   The Abstraction-Occurrence pattern described in chapter 6 of the book

   a) What is the Abstraction class in this case?

      ASK

      A RentableItem

   b) What is the Occurrence class?

      ASK

      A Copy of an item

   c) Why can't we just use one class for both purposes?

      ASK

      (1) If we have multiple copies of a title, avoid storing information redundantly (e.g. Title, producer, director, actors, running time, etc. for a movie)

      (2) A checkout is a relationship between a customer and a specific copy - e.g. if a copy shows up in the overnight return box, we want to know which customer to credit with returning it.

      (3) OTOH, a reservation is a relationship between a customer and a title - the customer doesn't reserve a specific copy - he/she just gets whichever copy comes in next

3. We can now consider how to maintain the record of copies checked out to a customer.

ASK

a) This is a 0..* relationship - each copy is checked out to at most one customer at a time, but a customer can have many copies.

b) Observe that a checkout has attributes - e.g. date due. We can represent this either by using an association class object, or by recording the date due as an attribute of the copy (since it can only be checked out to one person at a time)

   ASK class for how each approach would be set up in terms of Java collections

4. The record of reservations associated with each rentable item (movie, game) - again, clearly 1:*. When a copy of the rentable item is returned, it is used to satisfy the first reservation on the list; when a new reservation is made, it is added to the end of the list.

   Clearly, the Reservation object can contain a simple reference variable to connect it to the RentableItem object representing the movie or game which is being reserved.

   What type of collection is appropriate in the RentableItem object to hold references to Reservation objects for this case?

   ASK

5. What about the association between customers and reservations they hold?

   ASK