CS211 Lecture: Class Diagrams in UML

Last revised September 7, 2007

Objectives:

1. To introduce UML Class 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
5. To introduce the dependency relationship between classes
6. To review the inheritance relationship between classes, and consider how to use inheritance in design
7. To introduce the realization relationship between a class and an interface

Materials:

1. Handout of class diagram for ATM Example
2. Handout of class/object diagram symbols

I. Introduction

A. As we pointed out at the start of the course, there are many different processes that can be followed in software development (e.g. waterfall life cycle, RUP, etc).

B. Regardless of what process is followed, however, certain tasks will need to be done as part of the development process *per se* - whether all at once, iteratively, or incrementally. In fact, activities like these will be part of any situation in which one uses his/her professional skills to help solve someone else's problem - not just when creating software or even in a computer field.

1. Establishing Requirements: The goal of this is to spell out what constitutes a satisfactory solution to the problem.

2. Analysis. The goal of this is to *understand* the problem. The key question is "What?".

3. Design. The goal of this is to develop the *overall structure* of a solution to the problem in terms of individual, buildable components and their relationships to one another. The key question is "How?".

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

5. Installation / Maintenance / Retirement

   All of these must be done in a context of commitment to Quality Assurance - *ensuring* that the individual components and the system as a whole do what they are supposed to do (which may involve identifying their shortcomings and fixing them.)

C. Last class dealt with initial identification of the key classes comprising a system - an analysis task. At this point, we begin to construct a class diagram, which continues to be refined as system development proceeds.

   1. Ultimately, the complete class diagram for a system will contain three general types of classes:

      a) During analysis, the classes discovered will typically be entity classes which represent "things" in the domain that the system must work with to fulfill its requirements.

         (1) The book suggests two general approaches to discovering these classes

            (a) One can consider what objects are involved in realizing a given use case.

               Quick check question b (p. 118)

               When all the objects appearing in each collaboration are combined, the result will be an overall class diagram for the system. (Note: there will typically be objects that appear in more than one collaboration)

            (b) One can seek to develop a model of the general domain

            (c) Either approach should result in the same overall model

         (2) The book suggests several broad categories to look for

            Quick-Check question d (p 119)

            (a) People

            (b) Organizations

            (c) Physical things

            (d) Conceptual things

Examples from Wheels

ASK

Customers
Bicycles
The hiring of a bicycle

b) As one moves into the design stage, one typically begins dealing with the other types of classes

Quick check question a (p 118)

I prefer to use a slightly different way to categorize these

(1) Boundary classes whose objects serve as means by which actors interact with the system - i.e. conceptually they sit on the boundary drawn during use case analysis.

(a) These may include one or more GUIs

(b) These may include interfaces to other systems via a network

(2) Controller classes whose objects are responsible for controlling the operation of the system. Typically each use case will be assigned to a controller object - though one controller may be responsible for multiple use cases.

2. Ultimately, the class diagram will contain quite a bit of information

a) The classes themselves

b) The attributes of each class

c) The operations of each class

d) Relationships between classes

3. The book suggests an overall process for developing a class diagram

Quick check question c (p. 118)

D. For the next few sessions, we want to look at the UML Class Diagram, which can represent a lot of information. In particular, we will focus on various relationships between classes.

1. What these mean

2. How they are represented

E. 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. (Indeed, the class diagram is the more frequently used type of diagram in UML in general.).

F. 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) first, and class relationships (the next three) later.

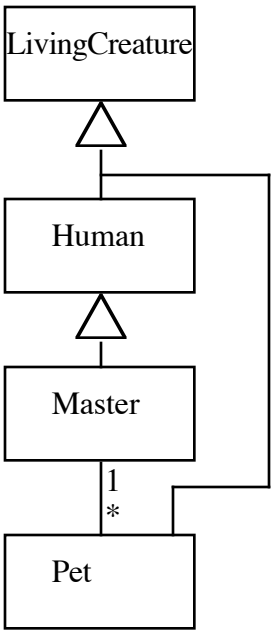
HANDOUT: Diagram symbols

1. *Association* - a relationship between objects.

EXAMPLES FROM CLASS DIAGRAM; SYMBOLS HANDOUT

a) In a class diagram, this kind of relationship is represented by a solid line, possibly with a plain arrow head on one end. There can multiplicities at both ends.

When there is an association between two classes, it means that an object belonging to one class can be related in this way to an object belonging to another class.



b) There are two special kinds of associations, which we have already looked at briefly, and will say more about later

(1) Aggregation - an association representing a whole-part relationship

(2) Composition - a strong form of aggregation

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; SYMBOLS HANDOUT

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; SYMBOLS HANDOUT

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 CLASS DIAGRAM - WILL DISCUSS BELOW; SYMBOLS HANDOUT

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.

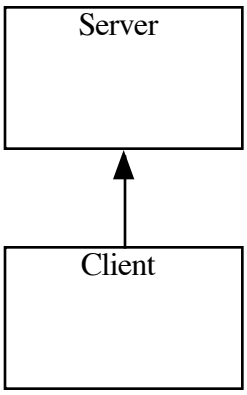
B. Technically, the fact that an object of class A can be associated with an object of class B is called an *association* and the corresponding connection between a specific object of class A and a specific object of class B is called a *link*. That is, an association is conceptually a *set* of links.

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. [Note: for clarity, as we talk about each type of adornment we will omit others that might otherwise belong in the diagram]

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:

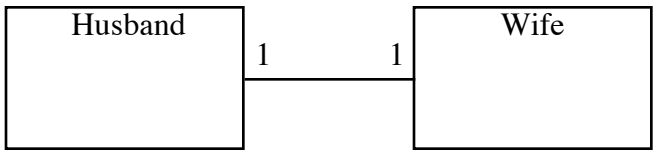


The arrow says that the Client must know about the Server, but the Server does not need to know about the Client (except briefly, during the time it is responding to a message received from the Client.)

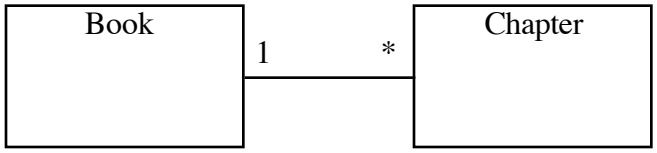
c) Why would we want to identify an association as being unidirectional where this is appropriate is? The presence of an association in the class diagram implies that the implementation will need to maintain information about this association. Keeping information about a bidirectional association means 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:

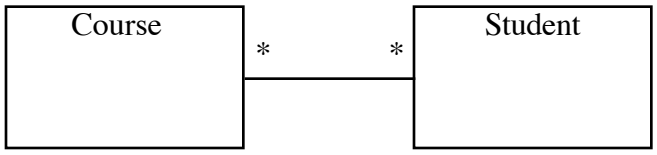
   a) One-to-one. Example: marriage (at least as intended!)



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

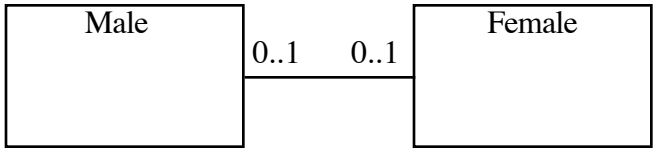


   c) Many-to-many: Example: students and courses

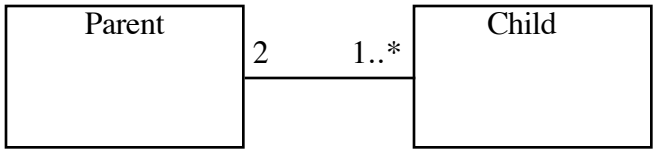


   d) Often, the multiplicities will be expressed as *ranges*, rather than as simple values

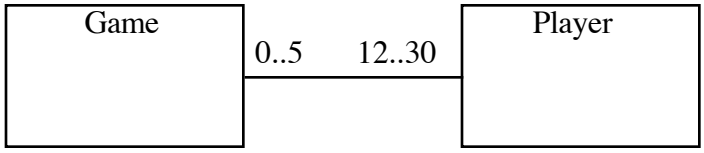
      (1) Example: the marriage relationship above was shown as 1 to 1 between the classes Man and Woman. If it were shown as a association between class Male and Female, the multiplicities would need to be expressed as ranges. (One cannot be a Husband or Wife without being married, but one can be a Male or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



(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.

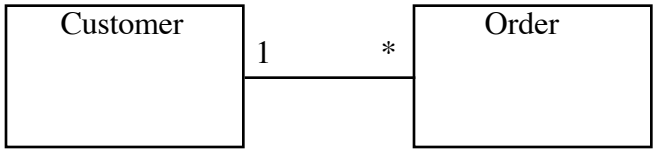


(This one's a bit contrived to illustrate a point, I admit :-).

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

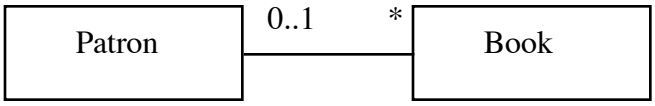
e) If the lower limit on the multiplicity of a certain relationship is 0, we say that the relationship is *optional*. If the lower limit is greater than 0, we say that the relationship is *mandatory*. Note that the same relationship may be optional in one direction, and mandatory in the other.

(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 use the notation "n" instead of * in a range - so * (= 0..*) is written as "0..n" and 1..* is written as "1..n".

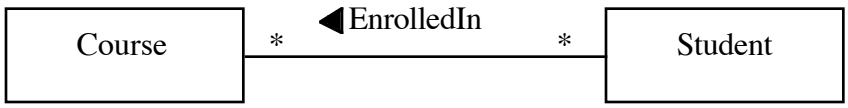
f) Note: Often in the literature the term "cardinality" is used for what we have called "multiplicity". The authors of the UML reference point out that - technically - cardinality refers to the properties of a *particular* instance of an association, while multiplicity refers to the overall properties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

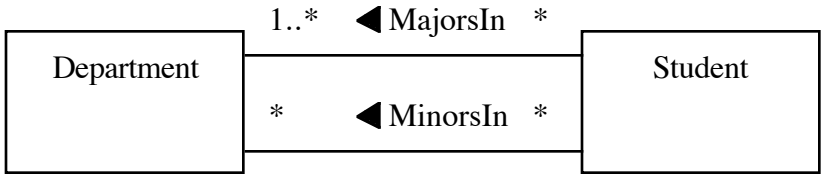
a) EXAMPLE:



(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.

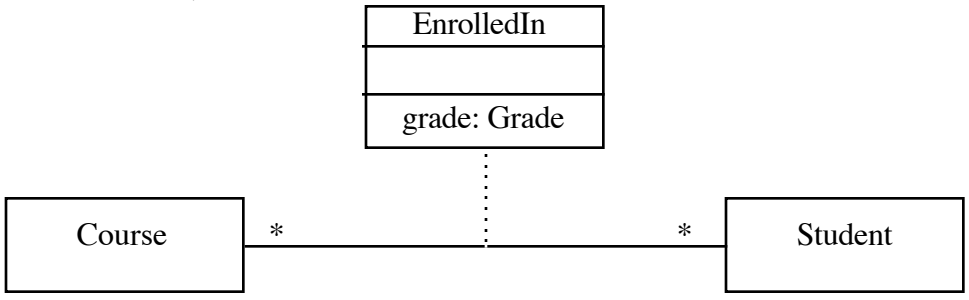
EXAMPLE



(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.)

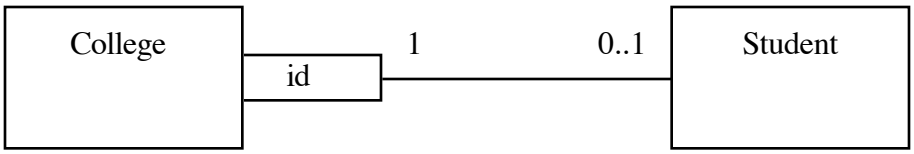


(Note the use of the three sets of lines in the box representing the association class, to make it crystal clear that this is a class.)

4. Qualified Association: Sometimes, a given object can be associated with many objects of some other class, but there is some qualifier such that, for any given value of the qualifier, the object is associated with at most one other object.

   EXAMPLE:

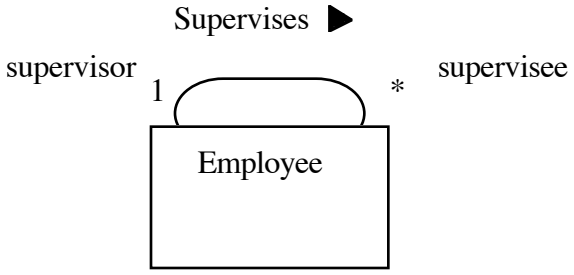
   A college is associated with many students; but for any given student id, there is at most one associated student (or possibly none). We say that the association between the college and students is a qualified association, with student id as the qualifier. This can be depicted as follows:



   (Note how the effect of the qualification is to reduce the multiplicity from 1 : n to 1 : 0..1 - for any given id value, there is at most one matching student)

5. 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.

6. Aggregation/Composition: 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.

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.)

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

c) 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.

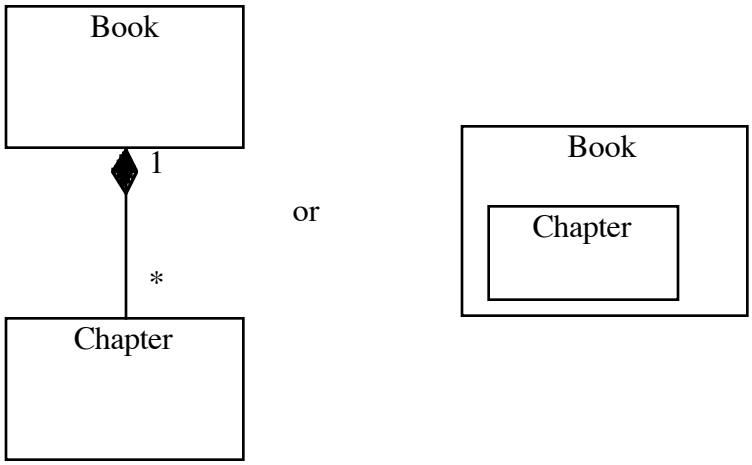
(d) Of the two examples we have considered:

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.

d) 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.

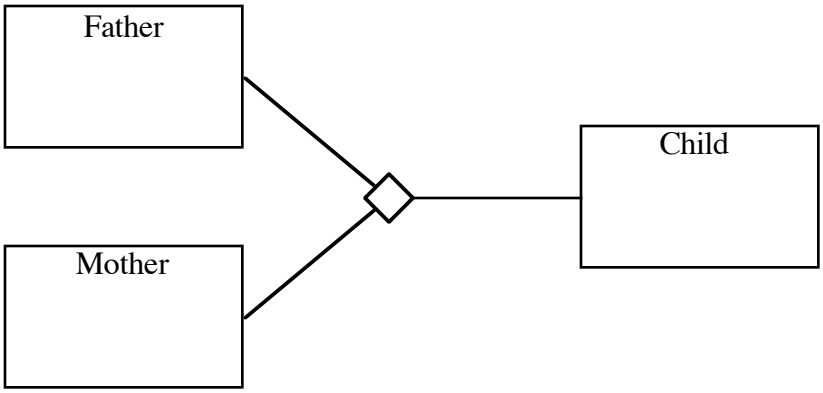
EXAMPLE: Consider the relationship between chapters of a book and the book itself. Clearly, each chapter is a part of one and only one book, and its existence is directly tied to the book of which it is a part. Thus, the association between a book and its chapters is a composition. *Either* of the following 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): [Any multi-way association can be converted to use only binary associations; in practice, this is almost always done]



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:

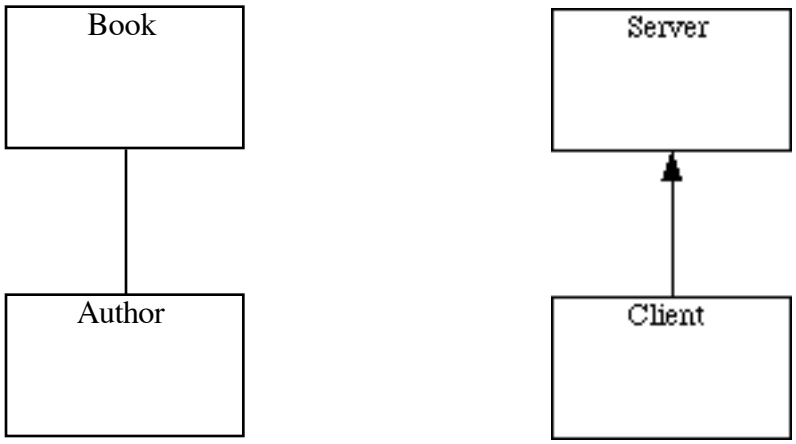
1. For each different association that relates objects of a given class to other objects, there will be a field in each object containing a link to the appropriate object(s).

   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.

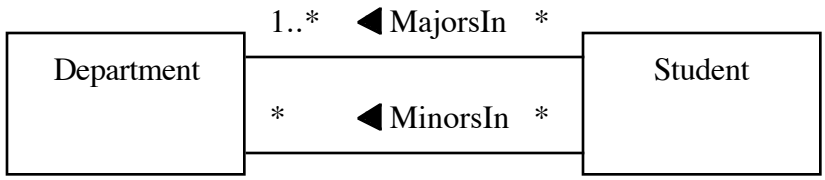
   (This reduction in information that needs to be maintained is why we consider the possibility of unidirectional associations.)

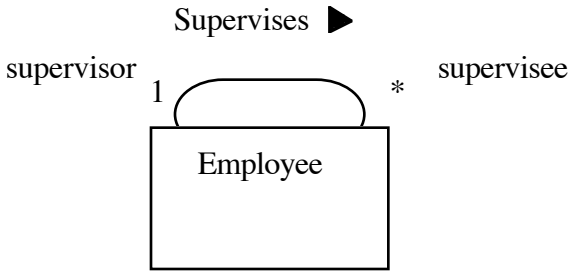
2. Frequently, the field name will be derived from the name of the association, or from the role names, if such are present. If not, the name will often come from the name of the class at the other end.

   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 not 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. Or the fields might be named majorsIn/minorsIn in the Student and still probably majors and minors in the Course.

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.) Or the fields might be named supervises and supervisedBy.

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).

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. (Actually, Java Maps are not technically Collections due to some implementation issues, but it is common to speak of them as "small c" collections.

(1) Maps are often used for qualified associations, with the qualifier serving as the key, and the associated object the value

EXAMPLE: The qualified association between a college and its students could be represented by a map (stored in the college object) consisting of pairs where the student id is the key and the corresponding student object is the value.

(2) 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?".

(3) 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.

(4) 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) In the case of maps, the Iterator is actually obtained from either its set of keys or its set of values.

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 obtained from the key set will access the elements of the map in key order.

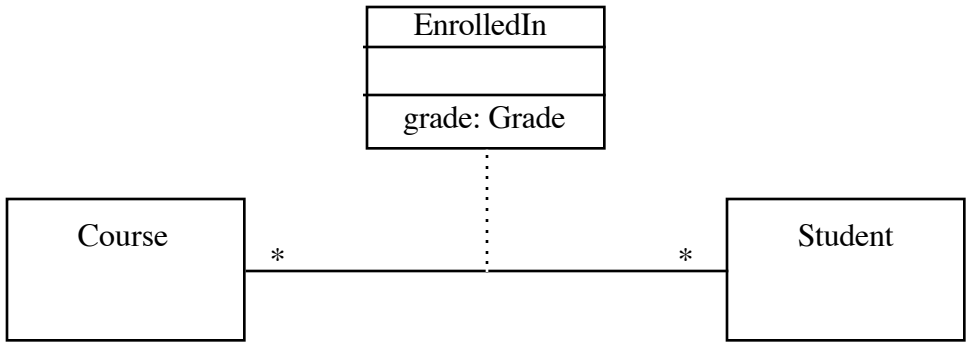
4. We will devote a lab to working with Java Collections

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. Now, let's think about the various associations in the Video Store problem and how they might be represented by Java collections: For now, let's restrict ourselves to a subset of the requirements, assuming only movies (not games) are being rented. It will also prove helpful to assume that there is a singleton object (perhaps of a class called Store) which represents the Store and "owns" all the other objects.

1. First though, we need to consider an interesting (and actually quite tricky) issue that arises in connection with rentable items.

   a) Typically, a video store owns multiple copies of popular movies.

      (1) Presumably, we need a separate object for each copy, since each can be rented to a different customer, be due on a different day, etc..

      (2) At the same time, we want to associate all kinds of information with a copy - its title, its actors, its director ... etc - but storing all this information multiple times (once for each copy) is problematic

         (a) Wasteful of space

         (b) Makes extra work when a new copy comes in

         (c) Suppose we needed to correct a piece of information - e.g. maybe we had recorded a wrong actor. We would need to make this change in <u>each</u> copy.

      (3) Moreover, when we accept a reservation for a particular movie, we don't want to associate the reservation with a <u>specific</u> copy - we want to associate it with the movie itself. Why?

         ASK

   b) There is a very problematic way to handle a case like this:

      (1) Create a separate <u>class</u> for each title. Thus, we would have a class Shrek3, a class BourneUltimatum ...

         (a) The various items of information we would want to record about a movie could be represented as <u>static</u> fields of the appropriate class - e.g. we would have something like

```
class BourneUltimatum {
```

```
static String leadActor="Matt Damon";
...
```

(b) A copy would be represented by an object of the appropriate class. Thus, if we had 10 copies of The Bourne Ulitimatum, each would be represented by an object of class BourneUltimatum - 10 in all.

(2) Why is this very problematic?

ASK

(a) A serious problem is that we have to create a new class every time a new movie comes out. This means that the software would have to be revised and recompiled about once a week!

(b) A similar problem is that any change to information stored about a movie would necessitate revising and recompiling the software. While this may seem relatively improbable in this case, it could be a serious problem in other, similar cases.

c) A much better way is to make use of a design pattern, which is, in essence, a good solution to a commonly-occurring tricky problem. (We will talk about design patterns later in the course.) In this case, we want to use a pattern known as the Abstraction-Occurrence pattern.

(1) In essence, what we want to do is to use two classes to represent movies.

(a) One class - which we might call something like `Movie` - represents the abstraction.

i) There would be, then one object of class `Movie` for Shrek 3, another for the Bourne Ultimatum ....

ii) This one object holds all the information that pertains to all copies of the movie - its actors, director ...

iii) When a new movie comes out, we create a new object of this class to represent it.

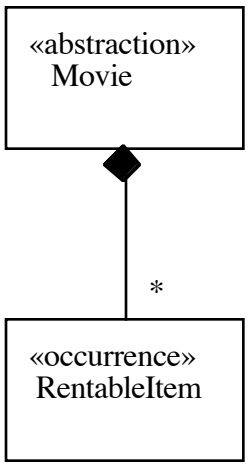
(b) A second class - which we might call `RentableItem` - represents the occurrence.

i) There would be one object of this class for each physical copy of a movie that we own - thus, if we had 10 copies of the Bourne Ultimatum, there would be 10 objects of class `RentableItem` representing them.

ii) Each object of class `RentableItem` would be associated with the object of class `Movie` to which it pertains.

(c) If the store owned 10,000 DVD's in all, representing 3000 different movies, there would, in total, be 10,000 objects of class `RentableItem`, and 3000 objects of class `Movie`.

(d) We might show this in UML as follows:



i) Composition is appropriate here, because,

(1) At least as far as the store is concerned, a movie is composed of its copies

(2) Each copy is associated with one and only one movie

(3) A copy of one movie can never be associated with a different movie

ii) A multiplicity of * is appropriate on the RentableItem end, because a given Movie can have an arbitrary number of copies. (We might even have 0 for a short time, if our only copy is lost and we're waiting for a reorder to come in)

iii) Bidirectional navigability is appropriate, because me want to be able to answer both of the following kinds of question:

(1) What copies of "The Bourne Ultimatum" do we own?

(2) What is the title of the Movie that rentable item 1234 is a copy of?

2. Given this, we want to have an association between the store and the movies it owns. Clearly this is 1..*.

   a) What navigability is appropriate here?

      ASK

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

      ASK

      Map, keyed on something like title of the movie. The Store object would have a declaration like

      ```
      Map < Movie > movies;
      ```

      But nothing would be needed in the Movie object, since we have no need to navigate from a Movie to the (one and only) Store

3. Again, we need an association between the store and the rentable items it owns as well. Clearly, this is also 1..* When a customer wants to rent an item, the ID number on the item is used to find the appropriate object to associate with the customer.

   a) What navigability is appropriate here?

      ASK

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

      ASK

Map, keyed on the item's ID. The Store object would have a declaration like

```
Map < RentableItem > items;
```

But nothing would be needed in the object, since we have no need to navigate from an item to the (one and only) Store

4. 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.

   a) What navigability is appropriate here?

      ASK

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

      ASK

      Map, keyed on the customer's ID. The Store object would have a declaration like

      ```
      Map < Customer > customers;
      ```

      But nothing would be needed in the Customer object, since we have no need to navigate from a Customer to the (one and only) Store

5. We have already said that we need an association between a customer and the specific items the customer has out.

   a) What is the multiplicity here?

      0..1 : * - an item is either out to 1 customer, or it's on the shelf; but a customer can have any number of items out

   b) In this case, we probably want bidirectional navigability, so we can answer questions like

      (1) What items does customer 1234 have out?

      (2) What customer rented item 5678 (that was just returned)

c) Observe that the act of checking out an item involves 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

6. We also need an association between a customer and the movies he/she has reservations for.

a) What is the multiplicity here?

ASK

It must be * .. * - any number of customers can have a reservation for a given movie, and a customer can have reservations for multiple movies.

b) In this case, we probably want bidirectional navigability, so we can answer questions like

(1) Who has reservations for "The Bourne Ultimatum"?

(2) What reservations does customer 1234 have?

c) Presumably, we want to keep track of reservations on a first-come first-served basis. This being the case, what sort of collection is needed in a Movie object to keep track of the customers who have a reservation for the Movie?

ASK

7. What would we need to do to this structure to also handle games?

ASK

a) Create a new base class (Title?) with subclasses Movie and Game.

b) Copies and reservations are now associated with a Title - the mechanics are the same for either Movies or Games. That's all we need to do!

IV. Generalization

A. We saw earlier 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.

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

B. We have been studying associations, which are relationships between objects. We now turn to the study of relationships between classes, of which UML class diagrams recognize three.

C. Probably the most prominent sort of relationship between classes is inheritance, which UML calls "Generalization".

1. Generalization relationships are denoted in UML by using a solid line with a triangle on the base class end.

   NOTE IN HANDOUT

2. Actually, as noted in the book, inheritance can arise in two closely related ways:

   a) Generalization: a base class is created that embodies the common characteristics of a number of similar subclasses.
   We may discover an opportunity for generalization during design when we notice that two or more classes have a number of characteristics in common, which can be put into a common base class so that they don't have to be duplicated in each class.

   EXAMPLE: Suppose we are developing a system for maintaining course registration information, and create classes "Student" and "Professor". As we develop these classes, we realize they have a lot in common (name, address, phone number, date of birth, etc.) and so create a generalized class Person that each inherits from.

   b) Specialization: a class is created that is similar to its base class, but with certain special characteristics.

   We may discover an opportunity for specialization during design when we notice that a class we need to create is very similar to an existing class, with a few variations. Rather than starting from class, we reuse the existing class by inheriting from it and only implementing the things which are different.

EXAMPLE: We did this from the very beginning of our work with Karel J. Robot last semester. The various kinds of robot classes we created were created by specializing the class Robot - or in some cases by specializing one of its specializations.

D. We have already discussed the meaning and mechanics inheritance both in CS112 and in this course. Our focus now will be on using inheritance as part of the *design* process. When do we use it, and how?

1. Inheritance can be a very powerful and useful tool, saving a great deal of redundant effort.

2. Unfortunately, inheritance can be - and often is - misused. So we will want to consider both how *to* use inheritance and how *not to* use it.

3. A cardinal rule for using inheritance well is *the rule of substitution*.

   ASK

   If a class B inherits from a class A, then it must be legitimate to use a B anywhere an A is expected. That is, it must be legitimately possible to say "a B <u>isa</u> A".

E. Actually, there are a variety of reasons for using inheritance in the design of a software system - including some not so good ones! One writer, Bertrand Meyer, has written a classic article in which he identified twelve! Some of the uses identified in Meyer's article are fairly sophisticated. I will draw on his work here, but in simplified form. Broadly speaking, Meyer classifies places where inheritance can be used as:

1. *Model inheritance* - when the inheritance structure in the software mirrors a hierarchical classification structure in the reality being modeled by the software.

a) One key feature of human knowledge is that many fields of learning have classification systems:

(1) The taxonomic system of biology

(2) The Dewey Decimal and Library of Congress systems used in libraries.

(3) Other examples?

ASK

b) When the reality we are working with has such a natural hierarchy, we may want to reflect that hierarchy in our software. However, Meyer warns about what he calls "taxomania" - the tendency to go overboard with classification hierarchies in software. In particular, there is a danger of creating too many levels in a hierarchy, without enough distinctions between classes at a level.

c) In general, we want to reflect a natural hierarchy in our software if the different objects we are working with fall into classes that have enough significant differences in attributes and behavior to make classification worthwhile.

(1) EXAMPLE: In the video store problem, the items the store rents can be categorized as movie tapes and game cartridges. These probably have enough distinctions to warrant two classes inheriting from a common "RentableItem" base class, because the information we need to store about each is quite different:

(a) Movies: studio, actor(s), genre, rating, running time

(b) Games: system made for, rating (using a very different sort of rating scale from that for movies)

(2) EXAMPLE: If the store rents both VHS tapes and DVD's, we may not to further classify movies into VHS and DVD, because the kind of information we keep about each is the same.

2. A second broad type of inheritance is what Meyer calls *software inheritance*. Here, the inheritance structure reflects a hierarchy that does not exist in the reality being modeled, but is useful nonetheless in the software.

a) Actually, as it turns out, what Meyer calls software inheritance shows up in UML models in two places - here, and under realization. We'll discuss the latter case later.

b) The usages we made of inheritance when working with Karel J. Robot really fall into this category. For example, at one point we created the class RightTurnerRobot by extending Robot. It is, however, unlikely that you would find separate catalog listings for these two types of robot - rather, we created this hierarchy to make software development easier.

c) One common motivation for this sort of inheritance is to facilitate *polymorphism*. Suppose we want to create a collection class whose

elements are to be various sorts of objects - e.g. perhaps a home inventory that lists the different items found in our home (useful information in case of a fire or theft.) In order to place these different items in the same polymorphic container, they would need to all derive from a common base class, which is the class of things the collection actually stores. (E.g. in this case, we might create a class HomeAsset and make things like furniture, books, artwork, electronic equipment etc. inherit from it.)

NOTE: In this case, the common base class will most likely be abstract.

EXAMPLE: The Transaction class hierarchy in the ATM system can be regarded as an example of this. The class Session needs to be able to refer polymorphically to different types of Transaction, which are made subclasses of a common abstract base class.

d) Another motivation for using software inheritance is to reuse work already done. When we are designing a new class, it is worth asking the question "is there any already existing class that does most of what this class needs to do, which I can extend?"

(1) EXAMPLE: When we were working with Karel J. Robot in CS112, we used a basic Robot class that had certain primitive capabilities (move(), turnLeft(), etc.) which we could extend by adding new capabilities (e.g. turnAround(), turnRight(), etc.)

(2) However, we need to proceed cautiously when we do this, because this kind of inheritance can easily be abused. When extending an existing class to create a new class, we should ask questions like:

(a) Is the law of substitution satisfied?

If the law of substitution is not satisfied, then we are almost certainly abusing inheritance.

(b) Are we mostly adding new attributes and methods to the existing class, or changing existing methods to do something entirely different? In the latter case, we are likely abusing inheritance - extension means "adding to" an existing set of capabilities.

(c) Are all (or at least most) of the existing methods of the base class relevant to the new class? If not, it is again likely that we are abusing inheritance.

(3) Note that, in cases like this, we generally do not have to create the base class - instead, we use an existing class to help create a new one.

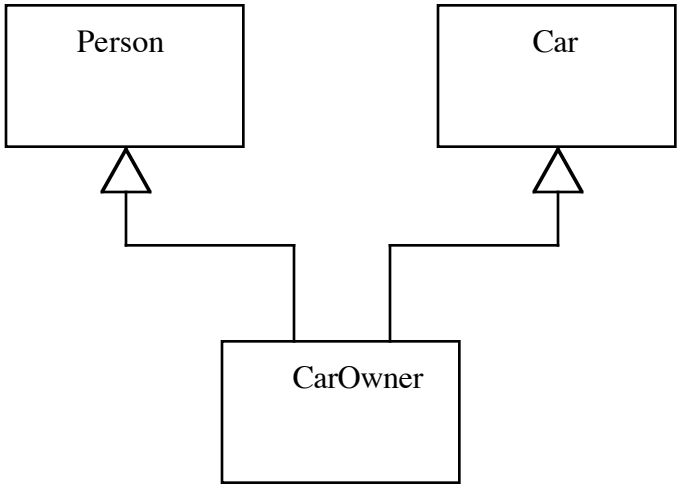
(a) This is most likely to happen in cases where the base class has been designed from the beginning to facilitate extension. (I.e. we usually consider extending classes whose initial designer created them with the intention that they be extended. For example, the Robot class was designed this way.)

(b) A related idea is that, where appropriate, we should try to design our classes in such a way as to facilitate later extension in other applications. This may mean making a class more general than in needs to be for a specific application, in order to facilitate later reuse.

3. A third broad type of inheritance Meyer identifies is called *variation inheritance*. Here, a class B inherits from a class A because it represents some sort of variation of A. Meyer describes this sort of inheritance this way: "Variation inheritance is applicable when an existing class A, describing a certain abstraction, is already useful by itself, but you discover the need to represent a similar though not identical abstraction, which essentially has the same features, but with different signatures or implementations." (p. 829)

   We will not discuss this type of inheritance further; its applications are a bit more sophisticated than what we're dealing with here.

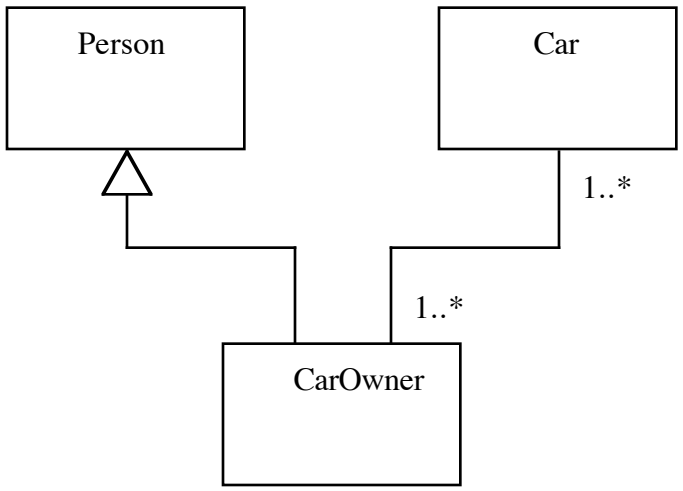
F. A danger particularly with both software inheritance and variation inheritance (but less so with model inheritance) is letting apparent convenience lead to misuse of inheritance. For example, Meyer cites a well-known software engineering text that develops the following scenario, using multiple inheritance:



Clearly, having CarOwner inherit from Person makes sense - a car owner is a person - but making CarOwner inherit from Car is another story! The justification is that Car has attributes like registration number and excise taxes due that legitimately apply to a CarOwner as well - but we don't want to saddle a CarOwner with having to have a carburetor, four tires, and brakes!

1. This example, and others like it, typically fail the fundamental law of substitution test. A CarOwner simply cannot be substituted for a car. (Try spending some time in a car wash!)

2. The mistake that is often made is confusing the "has a" relationship (association) with the "isa" relationship (inheritance). A correct way to represent the structure of the problem would be to use inheritance in one case, and association in the other:



(By the way, note that doing it this way lets us allow for the possibility that an owner might have several cars, and that a car might have joint owners.)

G. In Java, inheritance is specified by using the keyword *extends*.

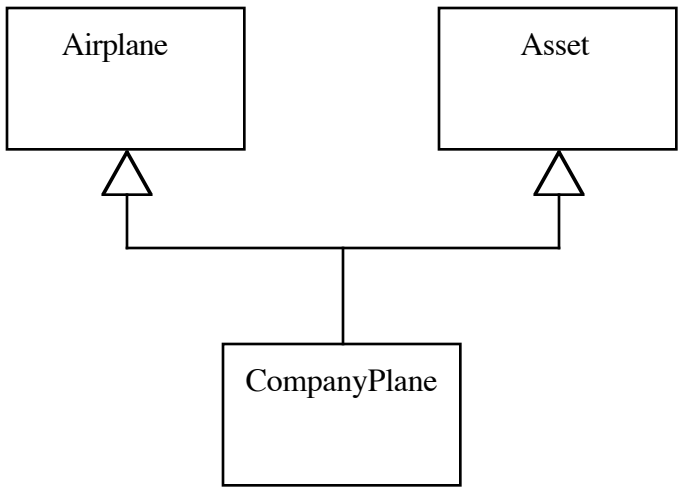
1. The class being extended may be either abstract or concrete.

2. As you know, Java allows a class to only extend one other class - i.e. it does not support multiple inheritance - something which many OO languages do support - but which introduces some interesting complexities we won't get into now.

V. Multiple Inheritance

A. Sometimes, it makes sense for a single class to generalize two (or more) bases classes. We call such a situation *multiple inheritance*.

1. The following example is given by Meyer:

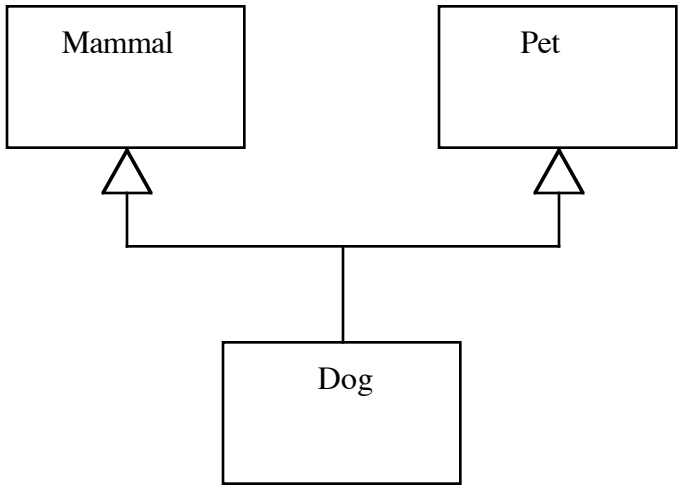


a) An airplane that is owned by a corporation (a company plane) is, at the same time, both an airplane and a company asset (in terms of bookkeeping)

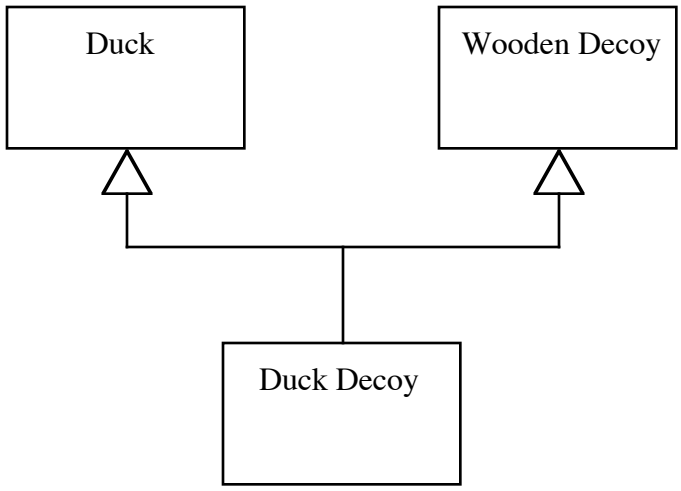
b) As an airplane, it has properties like manufacturer, model, range, capacity, runway length needed, etc.

c) As an asset, it has properties like cost, depreciation rate, current value, book value etc.

2. Here's another example:



3. However, multiple inheritance is easily misused. It is easy to create questionable (or obviously bad) examples. For example, the following is sometimes cited as an example of a place where multiple inheritance is useful, but is really a fairly bad example:



B. Multiple inheritance can give rise to some interesting problems. We will consider two - there are others.

1. Features with the same name in two different base classes.

   Example: The company plane example. Suppose that the class airplane had a field called rate (meaning speed), and the class asset had a field called rate (meaning depreciation rate.) If we declared

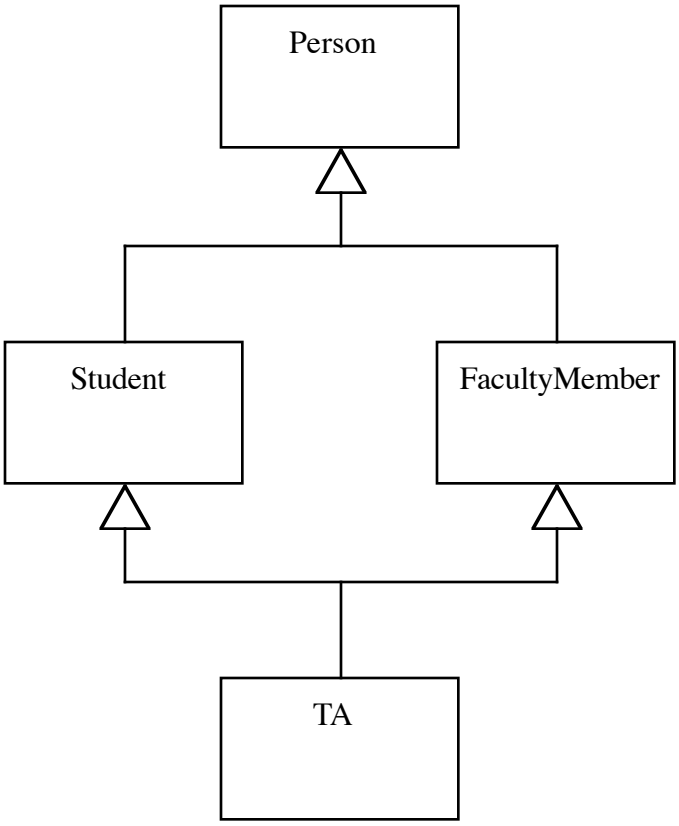
   CompanyPlane p;

   what would p.rate mean?

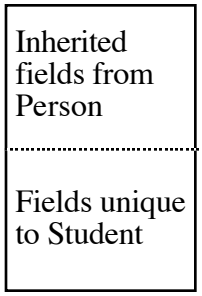
   (Arguably, this might not happen in this particular case - but it could. If it did, we could avoid it by changing the name of the field in one of the base classes - if we had access to the source, and if we could then change all the uses of the old name in other software that used this class - a nontrivial task.)

2. Repeated inheritance.

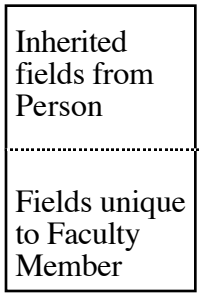
   Example: Consider the following situation, which could arise if multiple inheritance is used. (Perhaps in a research university) - and how the objects in question would need to be laid out in memory.



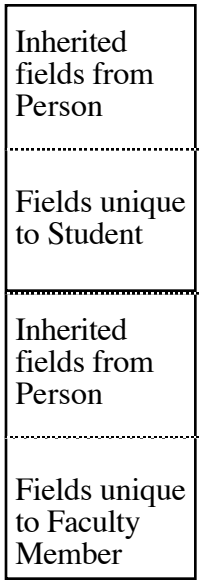
a) Student



b) FacultyMember



c) ∴ TA



Note that the straightforward layout of a TA object contains <u>two</u> copies of the Person fields - leading to all sorts of potential ambiguities.

C. Programming languages that support multiple inheritance have to deal with these complexities in some way.

EXAMPLE: C++

1. The possibility of having the same field name (or method name) occur in two different base classes is dealt with by allowing the use of a class name as a qualifier.

   e.g. Airplane::rate is the rate field inherited from class Airplane.

2. The possibility of repeated inheritance can be dealt with by something called a *virtual base class* - which we won't discuss! (Suffice it to say it's a tad messy!)

D. Java, as you know, does not support multiple inheritance. Since multiple inheritance is not often really needed, this is not a major issue. If it is needed, there are two ways to get the job done in Java:

1. If only the *interface* needs to be inherited, but not the *implementation*, then Java interfaces can be used.

a) A Java class can implement any number of interfaces

b) Example (one we've used more than once)

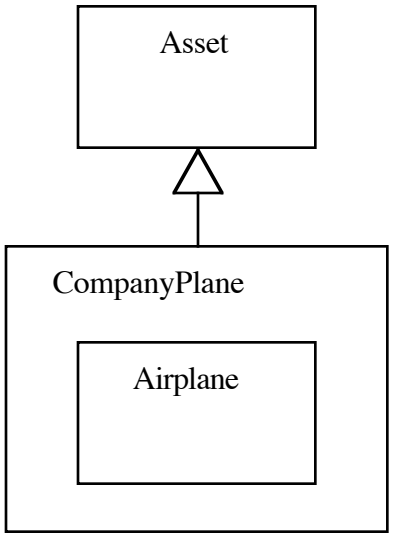
```
class ___________ extends Frame
   implements ActionListener, WindowEventListener
{
   ...
```

c) We'll discuss realization of interfaces shortly.

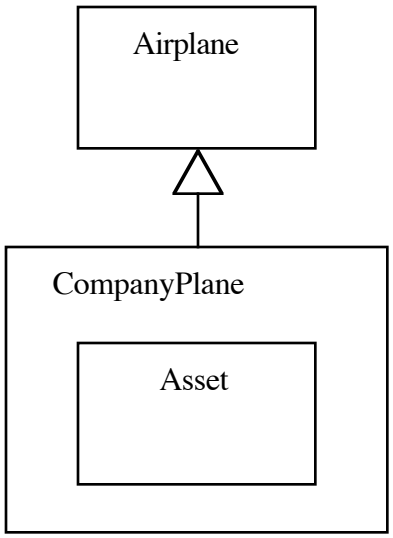
2. We can use *containment*.

Example: the CompanyPlane class in Java

a) implement as



(or)



b) Then use "forwarding" of methods - example (first case)

```
class CompanyPlane extends Asset
{
   Airplane myInnerPlane;

   public int getCapacity()
   {
         return myInnerPlane.getCapacity();
   }
...
```

VI. Realization

A. The next sort of relationship between classes we want to consider is called *realization* in UML.

1. In many ways, it is similar to inheritance - in fact, in some languages this relationship is represented the same way as ordinary inheritance.

2. Its uses a notation similar to that for generalization, except using a dashed, rather than solid line.

B. In ordinary inheritance, if B inherits from A, then B inherits both A's *interface (specification)* and A's *implementation*. Realization (or what is sometimes called interface inheritance) occurs when we want to specify that a class must provide certain behaviors, without specifying <u>how</u> these behaviors are provided.

We have seen a couple of examples of this in the Java libraries.

1. The ActionListener interface used with Buttons and MenuItems specifies that an ActionListener object must have a method with signature actionPerformed(ActionEvent), which is called when the Button is clicked or the MenuItem is chosen. However, different ActionListeners may do very different things.

2. In the Collections facility we considered earlier, List, Map, and Set are interfaces, which can be implemented in a variety of different ways. (In fact, each is implemented in at least two different ways in the Java library, and other implementations could be created by a user.)

C. The standard Java mechanism for realization is to have a class declare that it *implements* an *interface*. (Thus, both the realizing class and the interface it realizes are declared in a special way.)

1. Java actually provides another mechanisms that can be used for specifying an inheritable interface: an abstract class. However, when the realization relationship is intended, implementing an interface is the appropriate facility to use.

2. Sometimes, in Java, we will use the "implementing an interface" mechanism for inheritance as well as realization. This may be needed because Java does not support multiple inheritance. If we need multiple inheritance to model a particular reality, and one of the classes being inherited is there just for behavior, then implementing it as an interface may let us do what we need to do.

   NOTE: In this case, the UML relationship we are modeling is actually generalization, not realization.

VII. Dependency

A. The final kind of relationship between classes we will consider is *dependency*.

   1. Dependency is denoted in UML by a dashed line with an arrow head from the dependent class to the class it depends upon.

   2. We say that class A depends on class B if a change to class B's interface could necessitate a change to A. (I.e. A's implementation depends on facilities made available by B.)

B. Dependencies are of various kinds. We will consider only one: *usage dependencies* - where the dependent class *uses* the class it depends upon as part of its implementation.

C. A usage dependency relationship arises when one or more of the following holds:

   1. The dependent class has a method that takes an object of the class it depends on as a parameter, and uses that object in some way in implementing the method.

   2. The dependent class has a method that returns as its value an object of the class it depends on.

   3. The dependent class creates an object of the class it depends on, but only uses it within one method (doesn't keep a reference to it as an instance variable - if it did, we would have an association.)

4. In Java, usage dependencies typically show up in the signatures of methods - as the type of a parameter or a return value - but the object in question is not stored as an instance variable.

D. We take note of dependencies in a UML diagram because they serve to alert us to the fact that whenever we change a class, we need to make sure that we don't need to also change classes that depend upon it.

1. In particular, any time we use an object of a class A as a parameter or a return value of a method of class B, we generally create a dependency from B to A which we should take note of. (No dependency is created if the value is just "passed through" to some other class.)

2. Of course, any time we have an association between objects, we have a dependency between their classes - but we don't take separate note of this - association implies dependency.

3. Likewise, any time we have a generalization or realization relationship, we also have an implicit dependency, which again does not need to be noted separately.

4. We only take note of a dependency when it is present and the classes seem otherwise unrelated to each other.

E. GO OVER EXAMPLES ON HANDOUT