CPS122 Lecture: Class Diagrams in UML

Last revised February 16, 2017

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

Materials:

1. Handout of class diagram for ATM Example
2. Handout of class/object diagram symbols
3. Projectable of simple class diagram showing pets and masters
4. Handout for major league baseball problem and skeleton for diagram
5. Projectable solutions to MLB problem and corresponding code
6. Handout of college class diagram problem

I. **Introduction**

   A. An earlier 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. The point of constructing a class diagram is that it forces us to <u>think</u> about certain key issues, and then to represent our thinking in a pictorial way that guides further design.

      2. I will demonstrate using astah to create class diagrams - but you can use another tool (or hand drawing) if you prefer.

B. First, though, a preliminary note. UML actually has two similar kinds of diagrams: class diagrams, in which the boxes stand for classes, and object diagrams, in which the boxes stand for individual objects.

1. We will see some examples of object diagrams later.

2. For now, in the diagrams we will be working with, the boxes stand for classes.

3. There are a couple of differences in the way the diagrams are drawn that serve to make this distinction clear. We will deal with these later

C. In the spirit of "seamless development" that characterizes OO, the initial development of a class diagram is an analysis task, which is refined as part of design.

1. Quick check question a

   I prefer to use a slightly different way to categorize these

   a) 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.

      (1) These may include one or more GUI components

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

   b) 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.

      For today, though, we will focus on the classes that are typically discovered early in the process.

      The classes we are focusing on now are often called entity classes because they represent concrete or abstract things.

2. The book suggests two general approaches to discovering the classes that initially belong in the class diagram.

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

    Quick check question b

    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

D. The book suggests several broad categories of objects to look for in initially developing a class diagram

Quick-Check question d [we'll do c in a moment]

1. People

2. Organizations

3. Physical things

4. Conceptual things

    Examples from Wheels

    ASK

    Customers
    Bicycles
    The hiring of a bicycle

E. However, not everything identified as a possible object should actually be considered as such when developing a class diagram. The book gives a host of reasons for rejecting potential objects.

Quick check question e

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

1. The classes themselves

2. The attributes of each class

3. The operations of each class

4. Relationships between classes

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

Quick check question c

The chapter in the book focused on the first four of these. For now, we will limit our focus to just two: identifying classes, and the relationships between classes.

H. Complete quick check questions before moving on

1. Do questions f-h

2. Collect questions

II. More About Relationships

A. 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.).

B. In this series of lectures, we will discuss four kinds of relationships (three of which are exemplified in following diagram for the ATM system).

HANDOUT ATM Class diagram

1. *Association* - a relationship between <u>objects</u>.

   EXAMPLES FROM CLASS DIAGRAM

   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 be multiplicities at both ends.

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

   EXAMPLES FROM DIAGRAM

These two are the most common - something of the difference can be illustrated by the following simple class diagram:

PROJECT
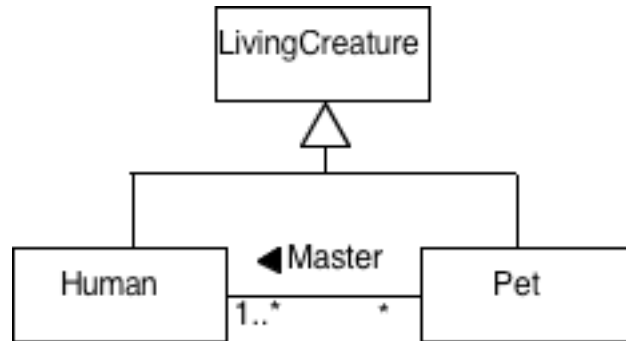
3. *Dependency* - a relationship between <u>classes</u>. In a UML diagram, this is represented by a dashed line with an arrowhead on one end.

    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 CLASS DIAGRAM - WILL DISCUSS BELOW

C. Everything we will discuss in this series of lectures is summarized in a handout.

HANDOUT Diagram Symbols

III. **Decorations on Associations**

   A. In the simplest case, an association may simply be drawn as line. But often, the line has one or more decorations *or* adornments that provide further information about the association. [Note: for clarity, as we talk about each type of decoration 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:
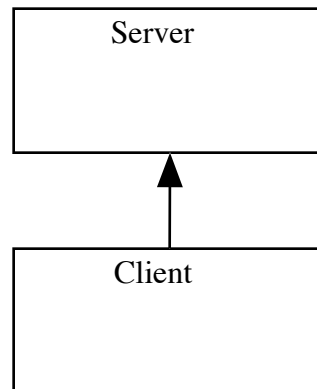
           The arrow says that the Client must know about the Server, but the Server does not need to know about the Client (except briefly, during the time it is responding to a message received from the Client.)

         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.

d) If you are using astah to produce class diagrams, you will notice that it only there are two tools for creating associations - one of which allows you to specify navigability, while the other does not.

DEMO with astah

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: relationship between a country and its capital city.
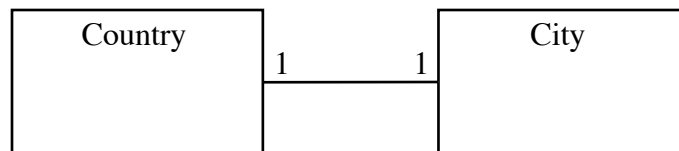
b) One-to-many: Example: the relationship between a book and the individual chapters that are part of it.
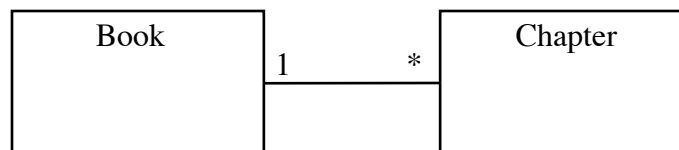
c) Many-to-many: Example: students and courses
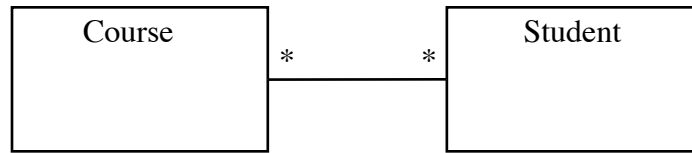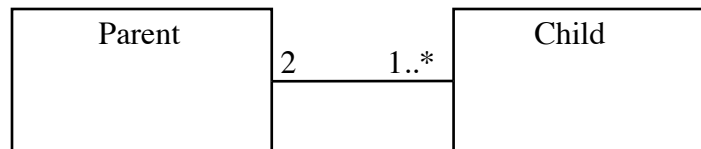
```
+----------------+                +----------------+
|    Course      |                |    Student     |
|                | *          *   |                |
|                |----------------|                |
|                |                |                |
+----------------+                +----------------+
```
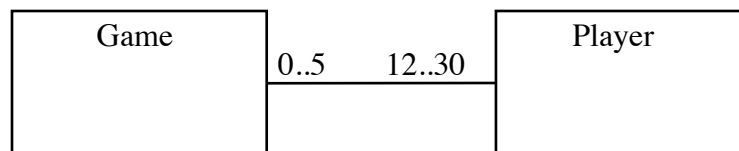
d) Often, the multiplicities will be expressed as *ranges*, rather than as simple values

(1) Example: a person has exactly two birth parents. A parent has at least one child (else he or she is not a parent!), but can have any number:

```
+----------------+                +----------------+
|    Parent      |                |    Child       |
|                | 2        1..*  |                |
|                |----------------|                |
|                |                |                |
+----------------+                +----------------+
```

(2) 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.
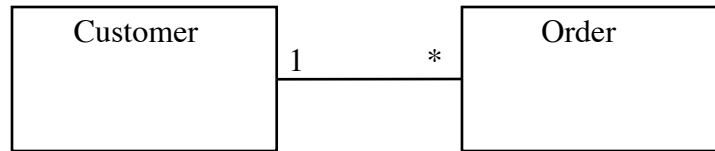
```
+----------------+                +----------------+
|    Game        |                |    Player      |
|                | 0..5    12..30 |                |
|                |----------------|                |
|                |                |                |
+----------------+                +----------------+
```

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

(3) The symbol * we have previously used means "0 or more" - hence it is equivalent to 0..*
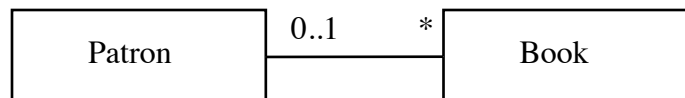
e) If the lower limit on the multiplicity of a certain relationship is 0, we say that the relationship is *optional*. If the lower limit is greater than 0, we say that the relationship is *mandatory*. Note that the same relationship may be optional in one direction, and mandatory in the other.

(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:

```
┌─────────────────┐                    ┌─────────────────┐
│   Customer      │                    │   Order         │
│                 │ 1              *    │                 │
│                 │────────────────────│                 │
│                 │                    │                 │
└─────────────────┘                    └─────────────────┘
```

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. (Note that while we allow a patron to have any number of books out, a book can only be checked out to one patron at a time.)
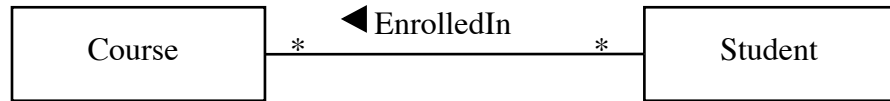
```
                     0..1          *
┌─────────────────┐                    ┌─────────────────┐
│                 │────────────────────│                 │
│   Patron        │                    │   Book          │
│                 │                    │                 │
└─────────────────┘                    └─────────────────┘
```

(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".

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.
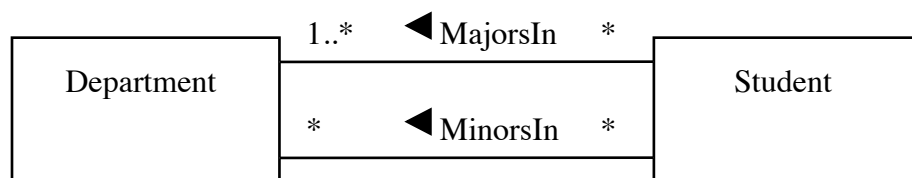
   a) *EXAMPLE:*

   ```
   ┌──────────────┐       ◄ EnrolledIn        ┌──────────────┐
   │    Course    │ *────────────────────* │   Student    │
   └──────────────┘                          └──────────────┘
   ```

   (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.
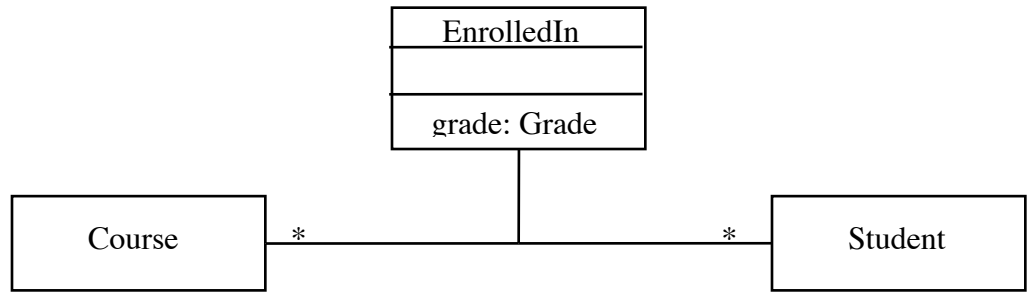
   EXAMPLE

   ```
   ┌──────────────┐  1..*   ◄ MajorsIn    *  ┌──────────────┐
   │              │─────────────────────────│              │
   │  Department  │                          │   Student    │
   │              │   *     ◄ MinorsIn    *  │              │
   └──────────────┘─────────────────────────└──────────────┘
   ```

   (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.)

```
                    ┌─────────────────┐
                    │   EnrolledIn    │
                    ├─────────────────┤
                    │                 │
                    ├─────────────────┤
                    │  grade: Grade   │
                    └─────────────────┘
                             │
┌───────────────┐           │           ┌───────────────┐
│    Course     │ *─────────┴─────────* │    Student    │
└───────────────┘                       └───────────────┘
```
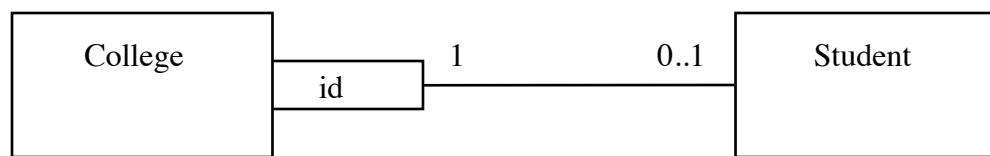
(Note the use of the three sets of lines in the box representing the association class, to make it crystal clear that this is a class.)

(Recall the quick check question about a class like Hire)

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.
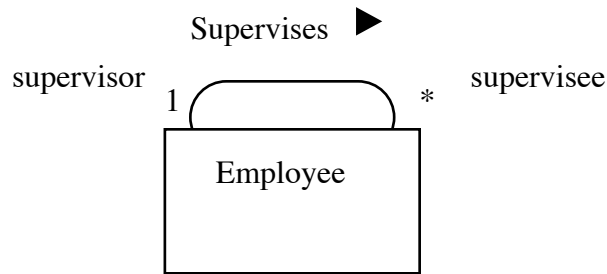
    EXAMPLE:

    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:

```
┌───────────────┬──────┐                      ┌───────────────┐
│   College     │      │                      │    Student    │
│               │  id  │ 1            0..1     │               │
│               └──────┤──────────────────────│               │
└───────────────┘      │                      └───────────────┘
```

    (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:

Supervises ▶

supervisor   supervisee
                1 ⌒⌒⌒⌒⌒⌒ *
               ┌─────────────┐
               │ Employee   │
               │   │
               └─────────────┘

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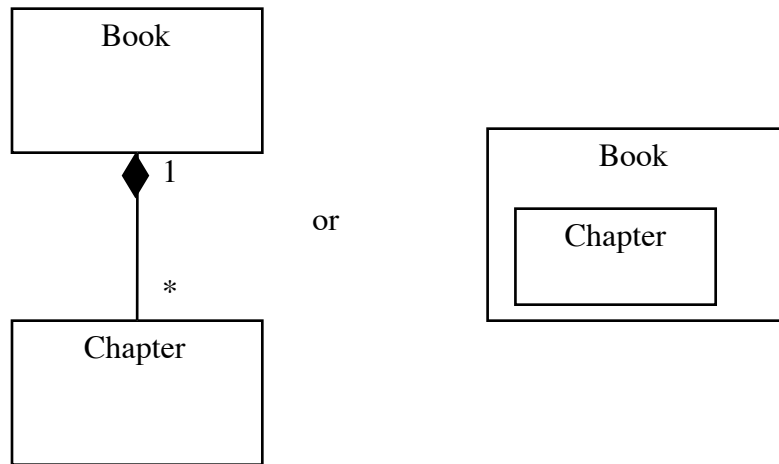
The latter representation might be particularly appropriate if the Chapter objects are accessible to the outside world only by *going through* a Book object - i.e. if they don't enter into any relationships with outside objects on their own.

B. Associations (including aggregation and composition) 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.)

C. *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.

D. Extended Example:

   Do Major League Baseball example in groups of 4. At each step, let groups work on then combine results and put on board.

   1. Class identification - what classes are needed?

   2. Pass out skeleton and fill in associations, then discuss solutions

      a) What associations are needed

      b) Where are aggregation or composition needed?

      c) Rationale for choice between aggregation & composition in each case.

   3. Project and discuss resultant code

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: A Bank might have a special kind of savings account that offers a higher interest rate in exchange for a high minimum balance. If the remaining properties of such an account are the same as those of other savings account, it might be desirable to specialize the class `SavingsAccount` to produce a class `HighBalanceSavingsAccount`.

D. We have already discussed the meaning and mechanics of inheritance 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 isa 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 <u>may</u> 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 <u>levels</u> 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.

EXAMPLE: In the library problem, the items the library checks out can be categorized as book and DVD. These probably have enough distinctions to warrant two classes inheriting from a common "Item" base class, because the information we need to store about each is different, and their behaviors are a bit different

(1) Books: store call number, title, author. When checked out, a book can be renewed.

(2) DVD: store call number, description, lead actor. Cannot be renewed.

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) 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.

c) 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) 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.

(2) 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. Frameworks are often 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:
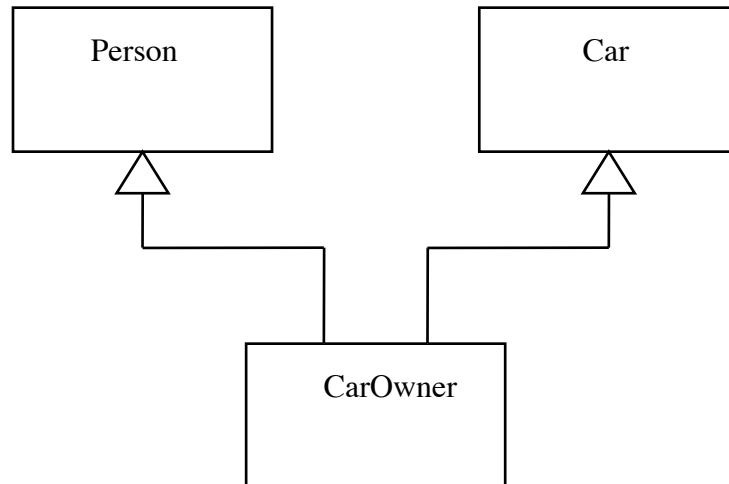
Clearly, having CarOwner inherit from Person makes sense - a car owner is a person - but making CarOwner inherit from Car is another story! The justification is that Car has attributes like registration number and excise taxes due that legitimately apply to a CarOwner as well - but we don't want to saddle 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:
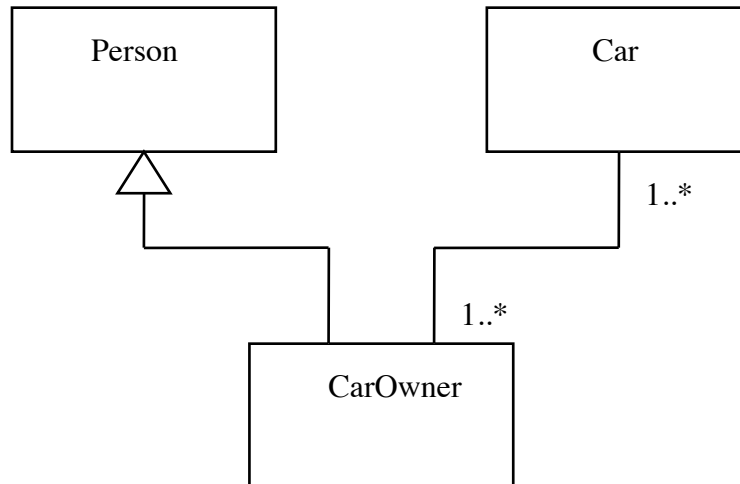
(By the way, note that doing it this way lets us allow for the possibility that an owner might have several cars, and that a car might have joint owners.)

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. **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.

    There are a couple of clear examples of this we will see later 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 will consider shortly, 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.

VI. **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 <u>interface</u> 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 <u>not</u> 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 CLASS DIAGRAM HANDOUT*

VII. **Do college class diagram exercise in teams of 4**