CS211 Lecture: Relationships Between Classes: Dependency, Inheritance, and Realization

last revised July 21, 2003

Objectives:

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

Materials:

1. Handout of class diagram for ATM Example (handed out previously)

I. Introduction

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.

3. In drawing a class diagram, we can depict *all* kinds of relationships - even those that are actually relationships between individual objects.

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.

II. Generalization

A. 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.

B. 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 isa A".

C. 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 <u>not</u> to further classify movies into VHS and DVD, because, though the rental rules may differ, 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.

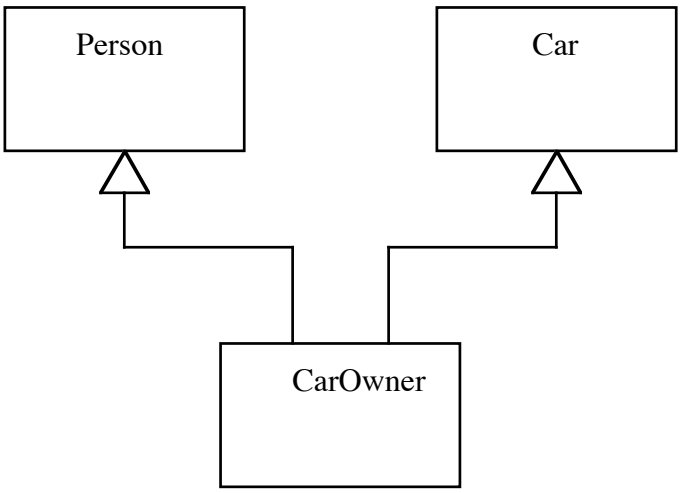
(a) This is most likely to happen in cases where the base class has been designed from the beginning to facilitate extension. (I.e. we usually consider extending classes whose initial designer created them with the intention that they be extended. For example, the Robot class was designed this way.)

(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.

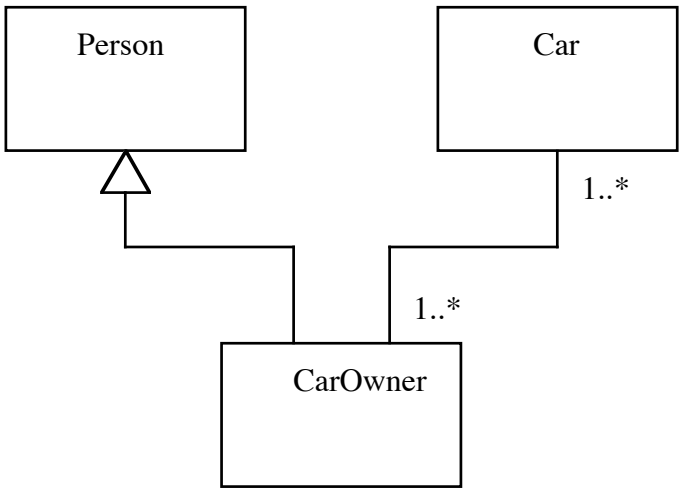
D. A danger particularly with both software inheritance and variation inheritance (but less so with model inheritance) is letting apparent convenience lead to misuse of inheritance. For example, Meyer cites a well-known software engineering text that develops the following scenario, using multiple inheritance:



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.)

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

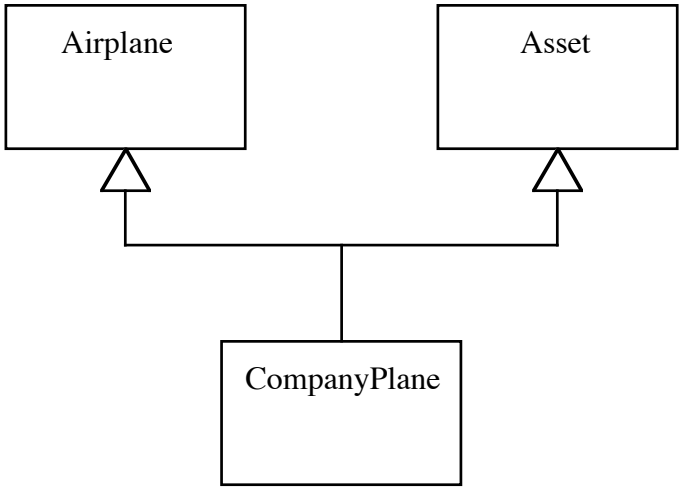
1. The class being extended may be either abstract or concrete.

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.

III. 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:

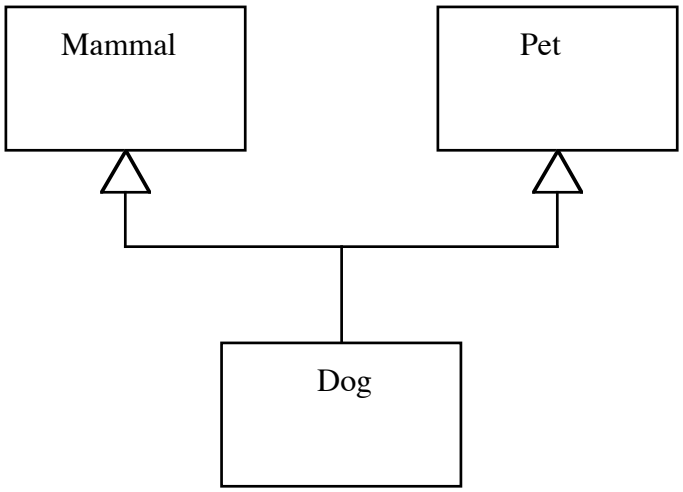


a) An airplane that is owned by a corporation (a company plane) is, at the same time, both an airplane and a company asset (in terms of bookkeeping)

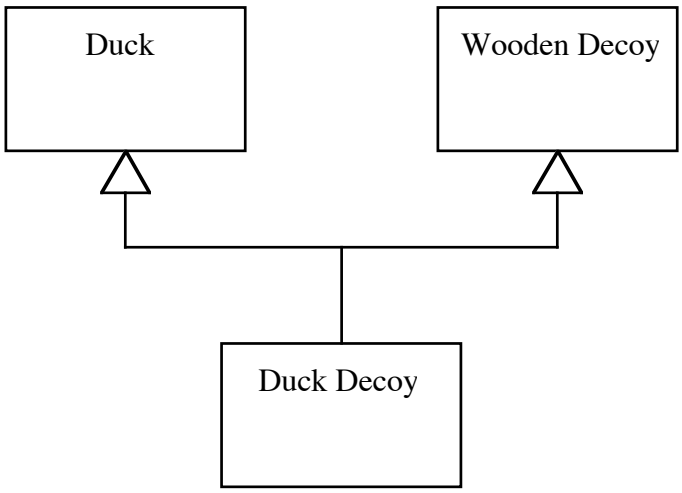
b) As an airplane, it has properties like manufacturer, model, range, capacity, runway length needed, etc.

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

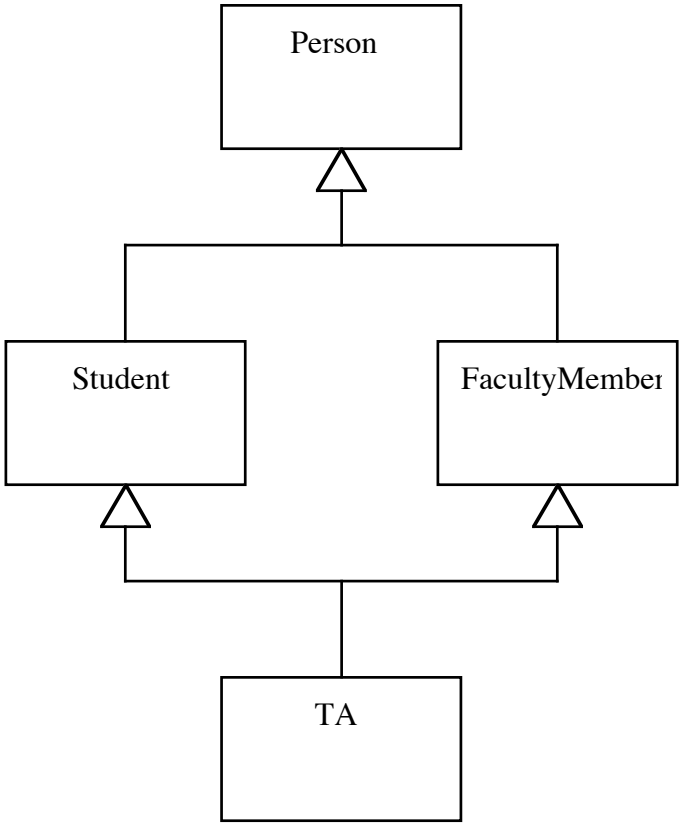
   CompanyPlane p;

   what would p.rate mean?

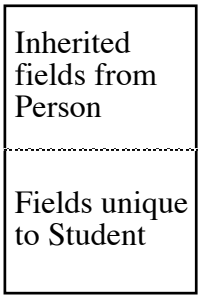
   (Arguably, this might not happen in this particular case - but it could. If it did, we could avoid it by changing the name of the field in one of the base classes - if we had access to the source, and if we could then change all the uses of the old name in other software that used this class - a nontrivial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

2. Repeated inheritance.

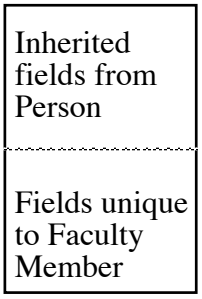
   Example: Consider the following situation, which could arise if multiple inheritance is used. (Perhaps in a research university) - and how the objects in question would need to be laid out in memory.



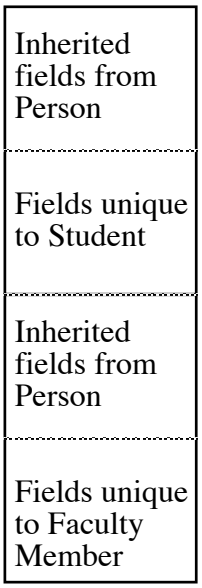
a) Student



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)

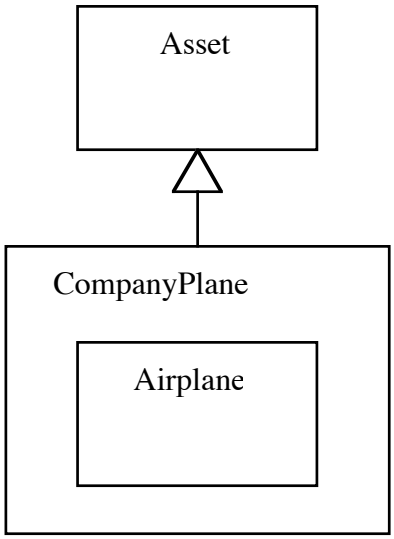
```
class ___________ extends Frame
   implements ActionListener, WindowEventListener
{
   ...
```

c) We'll discuss realization of interfaces shortly.

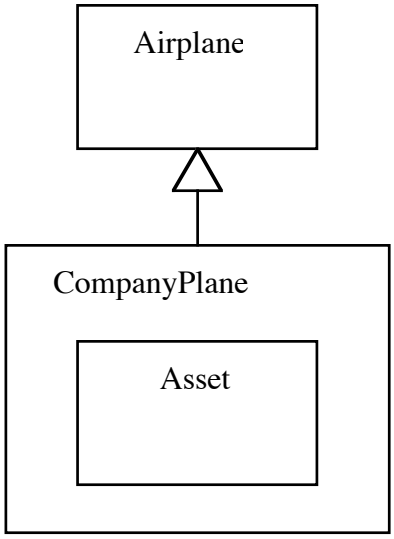
2. We can use *containment*.

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();
   }
...
```

IV. 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.

V. 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 <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 HANDOUT