ADVANCED CLASS MODELING

Advanced object and class concepts

- **<u>Enumerations:</u>**
- **<u>Multiplicity:</u>**
- ***<u>Scope:</u>***
- **<u>Visibility:</u>**

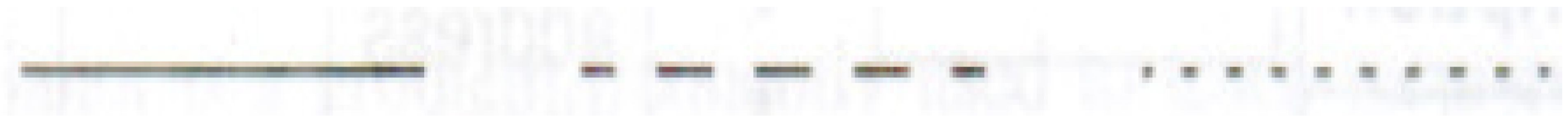
Advanced Object and Class Concepts:

- *A data type* is a description of values. Data types include
- Numbers,
- Strings, and
- Enumerations.

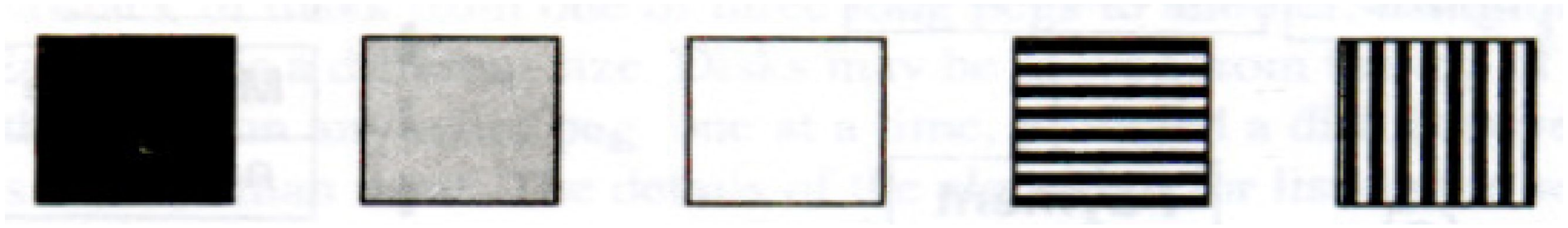
Enumerations:

- *An enumeration* is a *data type* that has a *finite set of values*.
- E.g. **Attributes:**

1. *accessPermission can be* an enumeration with possible values that include *read* and *read-write.*
2. *PenType* is an enumeration that includes *solid, dashed,* and *dotted.*



- *3.TwoDimensional.fillType* can be an enumeration that includes *solid, grey, none, horizontal lines,* and *vertical lines.*



- To define a new simple data type, called enumeration type, we need 2 things:
  - A name for the data type
  - A set of values for the data type

- enum {FALSE, TRUE};
- enum rank {TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE};
- enum colors {BLACK, BLUE, GREEN, CYAN, RED};
- The values are written in all caps because they are constants

Use of enumerations:

- Enumerations often occur and hence they must be carefully noted during modeling. They are important to *users.*
- Enumerations are also significant for an *implementation*: Possible values may be displayed with a pick list and the data must be restricted to the legitimate values.

Enumeration & Generalization:

Generalization shall not be used to capture the values of an enumerated attribute.

Enumeration is merely a list of values;

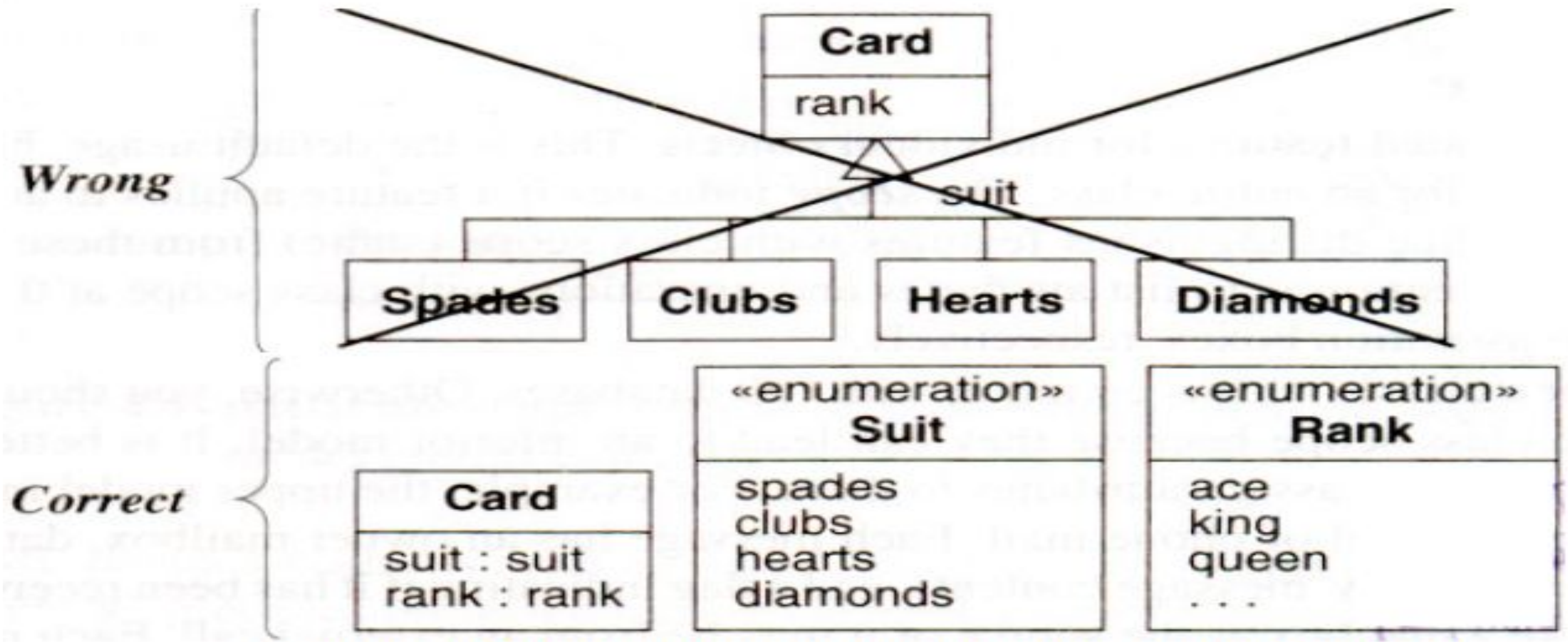
Generalization is a means for structuring the description of objects.

Generalization is introduced only when at least one subclass has significant attributes, operations, or associations that do not apply to the super-class.

Example:

- Generalization for *Card* should not be used because most games do not differentiate the behavior of spades, clubs, hearts, and diamonds.

Modeling enumerations.



Generalization must not be used capture the values of an enumeration

Enumeration in UML:

- Enumeration is a data type. It can be declared by listing the keyword *enumeration* in *guillemots* («») above the enumeration name in the top section of a box. The second section lists the enumeration values.

Multiplicity:

- Multiplicity is a constraint on the cardinality of a set. Multiplicity applies not only to *associations* but also to ***attributes***. It is often helpful to specify multiplicity for an attribute, especially for ***database applications***.

Multiplicity for an attribute :

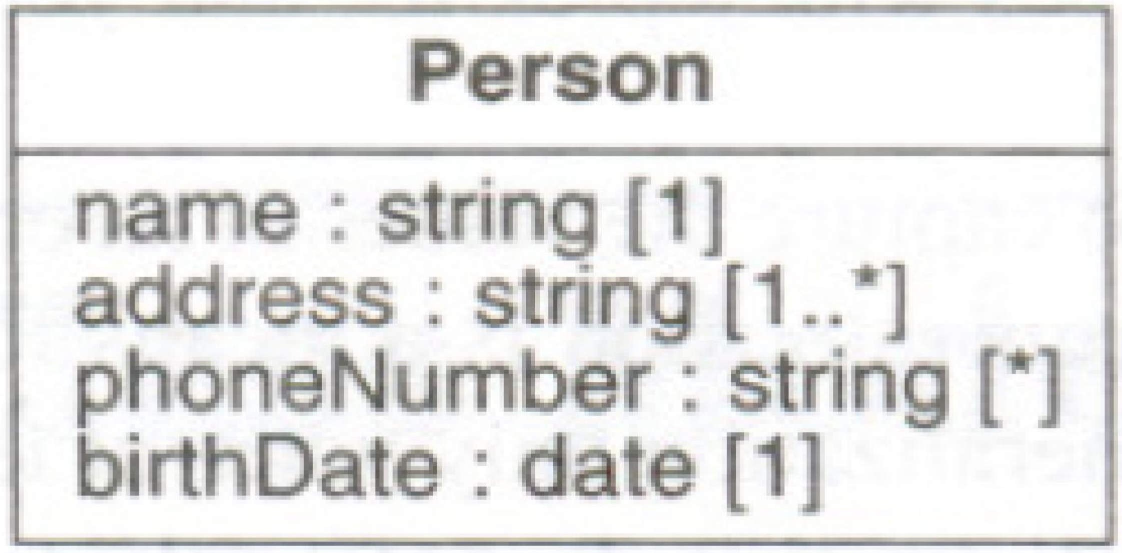
Specifies the number of possible values for each instantiation of an attribute. The most common specifications are a mandatory *single value* [1], an optional *single value [0..1]*, and *many* [*****].

Multiplicity specifies whether an attribute is mandatory or optional (in database terminology whether an attribute can be null). Multiplicity also indicates if an attribute is *single valued* or can be a *collection*.

If not specified, an attribute is assumed to be a ***mandatory single value*** **([1]).**

Example:

A person has one name, one or more addresses, zero or more phone numbers, and one birth-date.



Scope:

- The *scope* indicates if a *feature* applies to an *object* or a *class*.
- An underline distinguishes features with class scope (*static*) from those with object scope.
- *Convention* is to list attributes and operations with class scope **at the top of the attribute and operation boxes,** respectively.

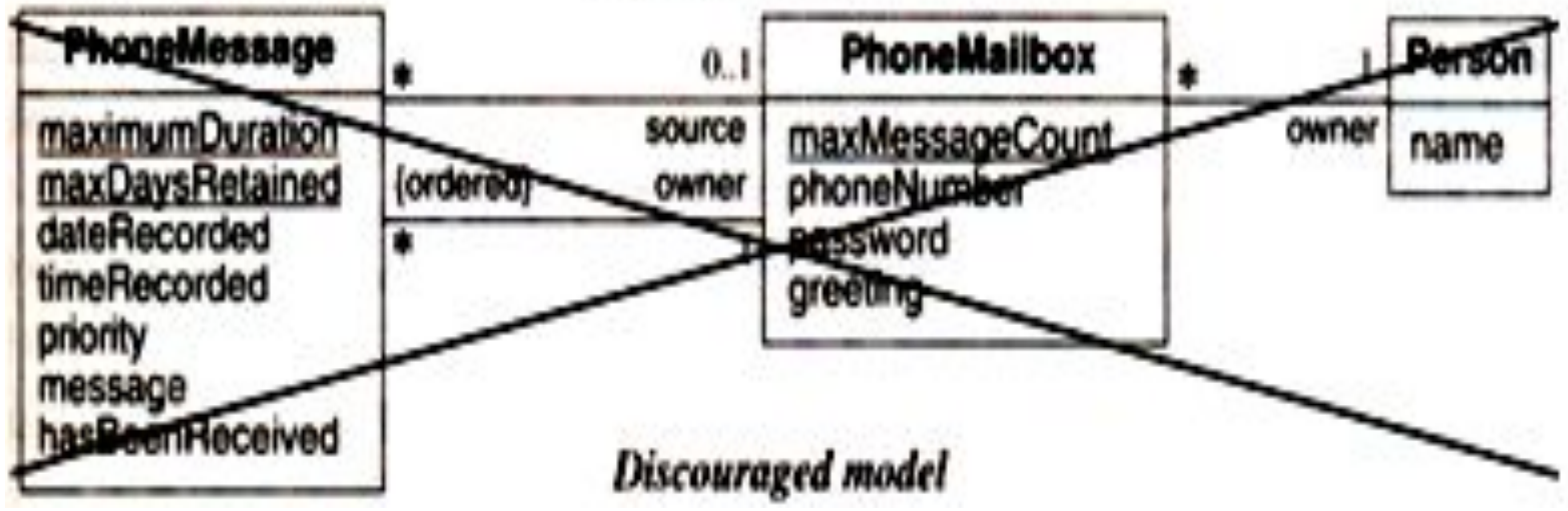
Attributes with class scope:

- It is acceptable to use an attribute with class scope to hold the *extent* of a class (the set of objects for a class)-this is common with OO databases.
- Otherwise, ***attributes with class scope*** *must be avoided* because they can lead to an inferior model. It is better to model groups explicitly and assign attributes to them.

Example:

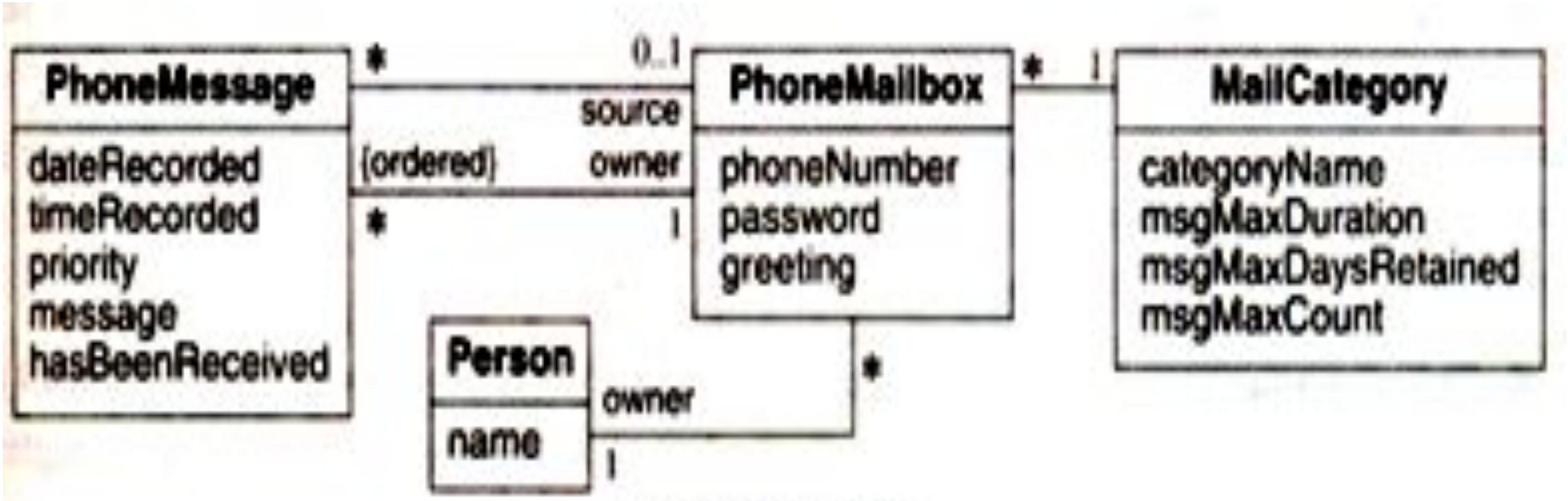
- Figure shows a simple model of phone mail.
- Each message has an owner mailbox, date recorded; time recorded, priority, message contents, and a flag indicating if it has been received. A message may have a mailbox as the source or it may be from an external call. Each mailbox has a phone number, password, and recorded greeting. For the *PhoneMessage* class, one can store the maximum duration for a message and the maximum days a message will be retained. For the *PhoneMailbox* class, one can store the maximum number of messages that can be stored.

Attribute scope



Discouraged model

- The model shown is inferior, however, because the maximum duration, maximum days retained, and maximum message count have a single value for the entire phone mail system.
- In the following figure, these limits can vary for different kinds of users, yielding a more flexible and extensible phone mail system.



Preferred model

Instead of assigning attributes to classes, model groups explicitly.

Operations with class scope:

- *In contrast to attributes,* it is acceptable to define operations of *class* scope.
- The most common use of class-scoped operations is to create new instances of a class.
- Sometimes it is convenient to define class-scoped operations to provide summary data.
- The use of class-scoped operations for distributed applications must be carefully dealt with.

Visibility:

Refers to the ability of a method to reference a feature from another class and has the possible values of ***public, protected, private***, and ***package***.

The precise meaning depends on the programming language.

Any method can freely access *public* features.

Only methods of the containing class and its descendants via inheritance can access *protected* features. (Protected features also have package accessibility in Java.)

Only methods of the containing class can access *private* features.

Methods of classes defined in the same package as the target class can access *package* features.

UML Notation:

- *UML denotes* visibility with a prefix. The character "+" precedes public features. The character "#" precedes protected features. The character "-" precedes private features. And the character “~" precedes package features.
- The lack of a prefix reveals no information about visibility.

Issues to be considered when choosing visibility:

1. **Comprehension**
2. **Extensibility**
3. **Context**

Comprehension:

- **All public features must be understood** to understand the capabilities of a class. In contrast, private, protected, and package features can be ignored-they are merely an implementation convenience.

Extensibility:

- Many classes can depend on public methods, so it can be highly disruptive to change their signature (number of arguments, types of arguments, type of return value). Since fewer classes depend on private, protected, and package methods, there is more latitude to change them.

Context:

- Private, protected, and package methods may rely on preconditions or state information created by other methods in the class. Applied out of context, a private method may calculate incorrect results or cause the object to fail.

Association Ends:

- As the name implies, an *association end* is an end of an association. A binary association has two ends, a ternary association has three ends, and so forth.

Additional properties of association ends:

(In unit-1 the properties covered are: **Association end name, Multiplicity, Ordering, Bags and sequences, Qualification)**

- **Aggregation**
- **Changeability**
- **Navigability**
- **Visibility**
- **Multiplicity**
- **Qualification**
- **Ordering**
- **Bags and sequences**

- **Aggregation**:The association end may be an aggregate or constituent part. Only a binary association can be an aggregation; one association end must be an aggregate and the other must be a **constituent**.
- **Changeability**: It specifies the update status of an association end. The possibilities are *changeable* (can be updated) and *readonly* (can only be initialized).

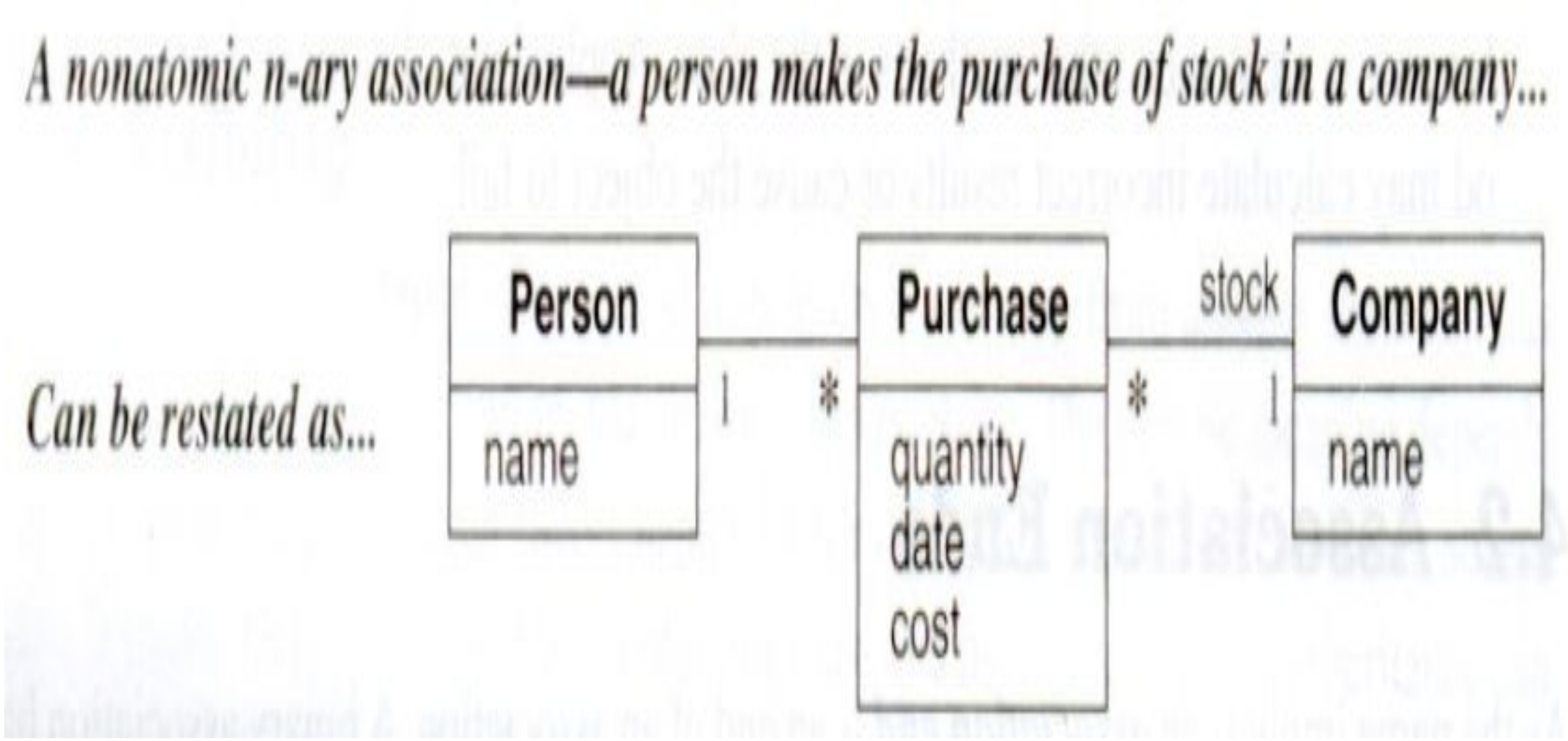
- **Navigability**: Conceptually, an association may be traversed in either direction. However, an implementation may support only one direction. The UML shows navigability with an arrowhead on the association end attached to the target class. Arrowheads may be attached to zero, one, or both ends of an association.
- **Visibility:** Association ends may be *public, protected, private,* or *package*.

N-ary Associations:

- Whereas binary associations are associations between two classes, *n-ary associations* are associations among three or more classes.
- Most of n-ary associations can be *decomposed* into binary associations, with possible qualifiers and attributes. Hence **they must possibly be avoided**.

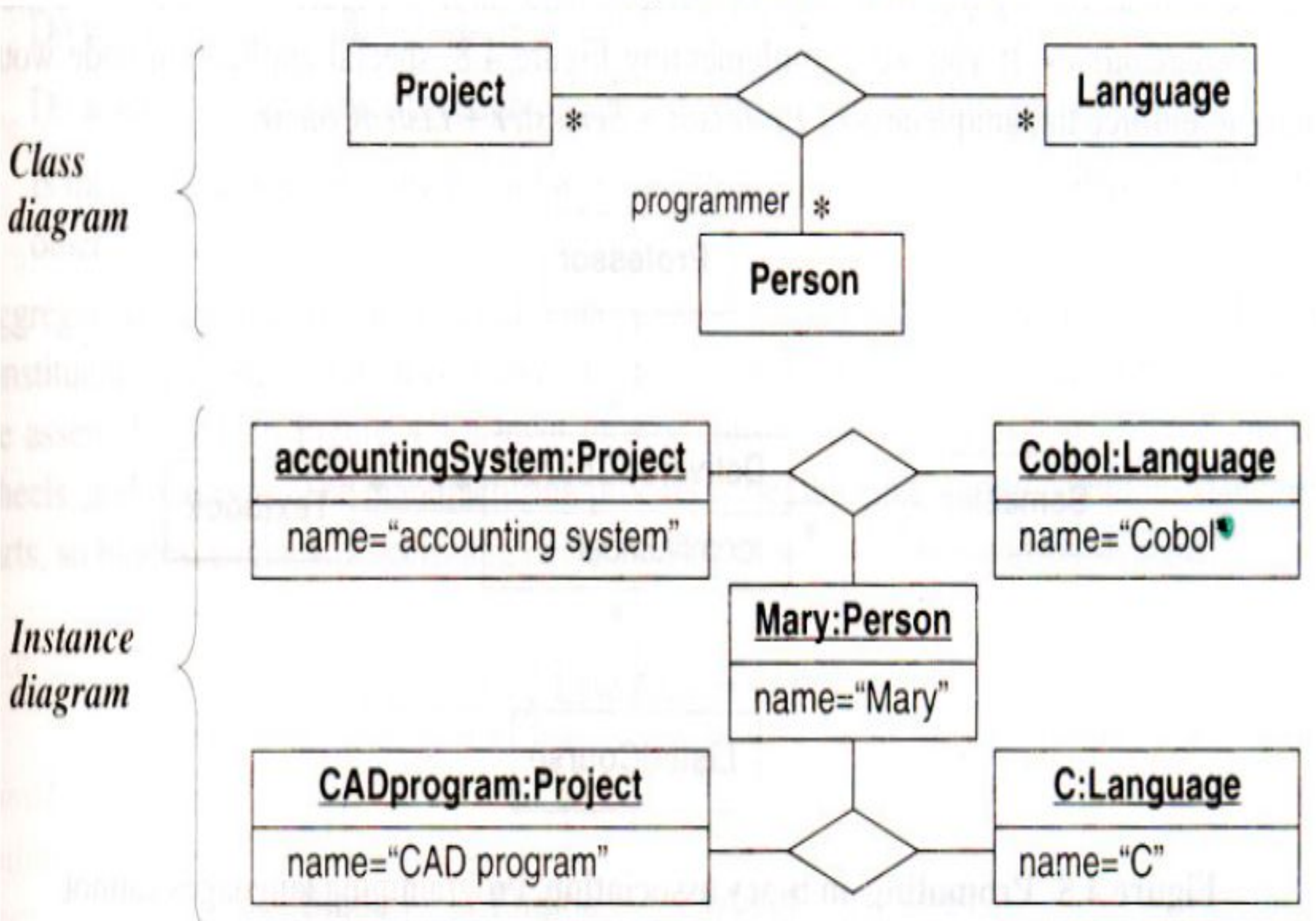
Example:

- Figure shows an association that at first glance might seem to be an n-ary but can readily be restated as binary associations.


A nonatomic n-ary association—a person makes the purchase of stock in a company...
Can be restated as...
Person
name
1
*
Purchase
quantity
date
cost
*
1
stock
Company
name

Example: n-ary (ternary) association

Ternary association and links.

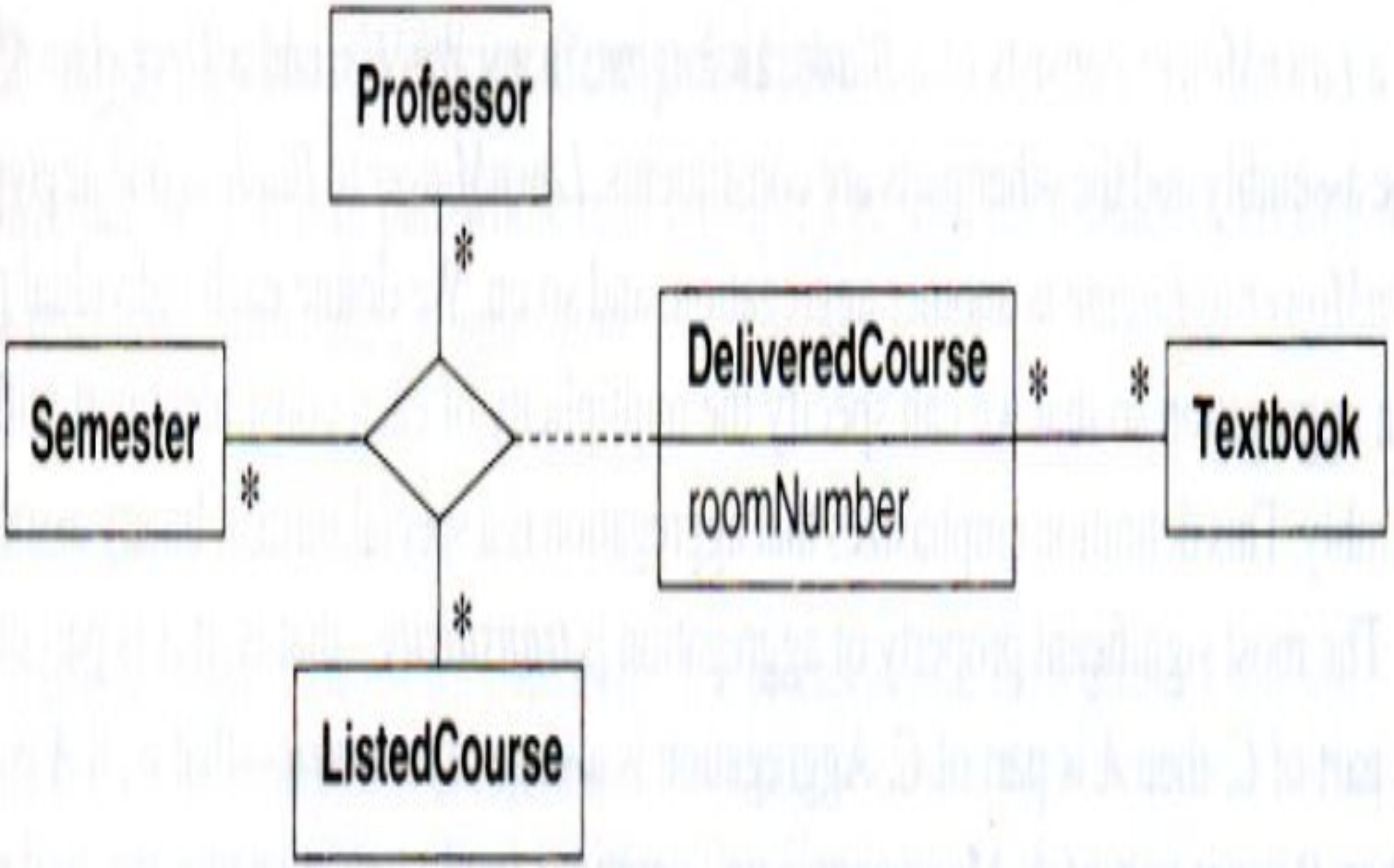


An n-ary association can have association end names, just like a binary association.

The UML symbol **for n-ary associations:**

- A **diamond** with lines connecting to related classes.
- If the association has a name, it is written in *italics* next to the diamond.
- An n-ary association can have a name for each end just like a binary association. End names are necessary if a class participates in an n-ary association more than once.
- n-ary associations **can not be traversed** from one end to another as with binary associations, so end names do not represent pseudo attributes of the participating classes.
- The OCL [Warmer-99] does not define notation for traversing n-ary associations.

Another ternary association.

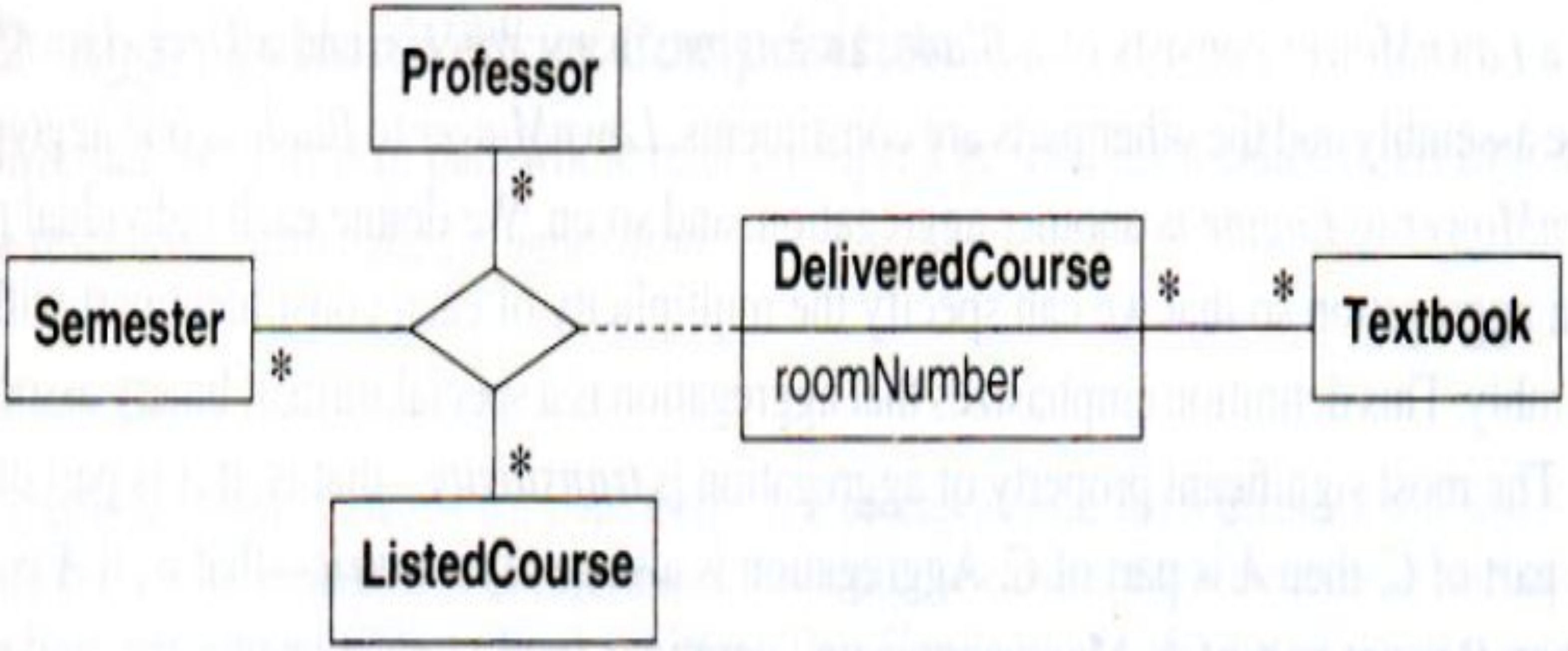


N-ary associations are full-fledged associations and can have association classes.

Note:

- When an n-ary association is promoted to a class, **the meaning of a model is changed**. An n-ary association enforces that there is at most one link for each combination-for each combination of *Professor, Semester,* and *ListedCourse* in the following **non-promoted** figure, there is one *DeliveredCourse.*

Another ternary association



N-ary associations are full-fledged associations and can have association classes.

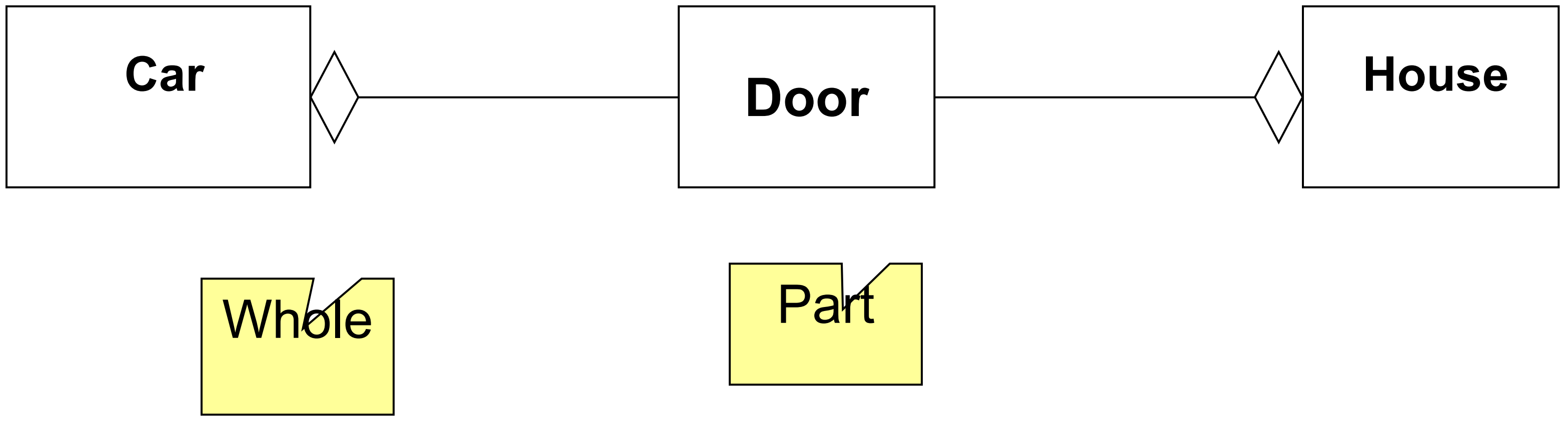
Aggregation:

- *Aggregation* is a strong form of association in which an aggregate object is ***made of constituent parts***. Constituents are ***part of*** the aggregate. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.
- Aggregation *relates* an assembly class to *one* constituent part class.
- An assembly with many kinds of constituent parts corresponds to many aggregations.

- The most significant property of aggregation is ***transitivity***-*i.e.* if *A* is part of Band *B* is part of C, then *A* is part of C.
- Aggregation is also ***anti-symmetric***-*that* is, if *A* is part of *B,* then *B* is not part of *A.*
- Many aggregate operations imply ***transitive closure**** and operate on both ***direct*** and ***indirect parts***.

Aggregation(conti..)

- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
  - Models a “is a part-part of” relationship.
- The **aggregation association** represents the part-whole relation between classes.
  - Denoted by a solid diamond and lines
  - Diamond attaches to the aggregate (whole) while lines attach to the parts



Aggregation:

- 1) Aggregation Versus Association
- 2) Aggregation Versus Composition
- 3) Propagation of Operation

Aggregation Versus Association:

- Aggregation is a special form of association, **not an independent concept.** Aggregation **adds semantic connotations**. If two objects are tightly bound by a ***part-whole*** relationship, it is an aggregation.
- **If the two objects are usually considered as independent, even though they may often be linked, it is an association**

E.g.

- *LawnMower* consists of a *Blade,* an *Engine,* many *Wheels,* and a *Deck. LawnMower* is the assembly and the other parts are constituents. *LawnMower* to *Blade* is one aggregation, *LawnMower* to *Engine* is another aggregation, and so on. Each individual pairing is an aggregation so that multiplicity of each constituent part can be specified within the assembly. This definition emphasizes that aggregation is a ***special form of binary association*****.**

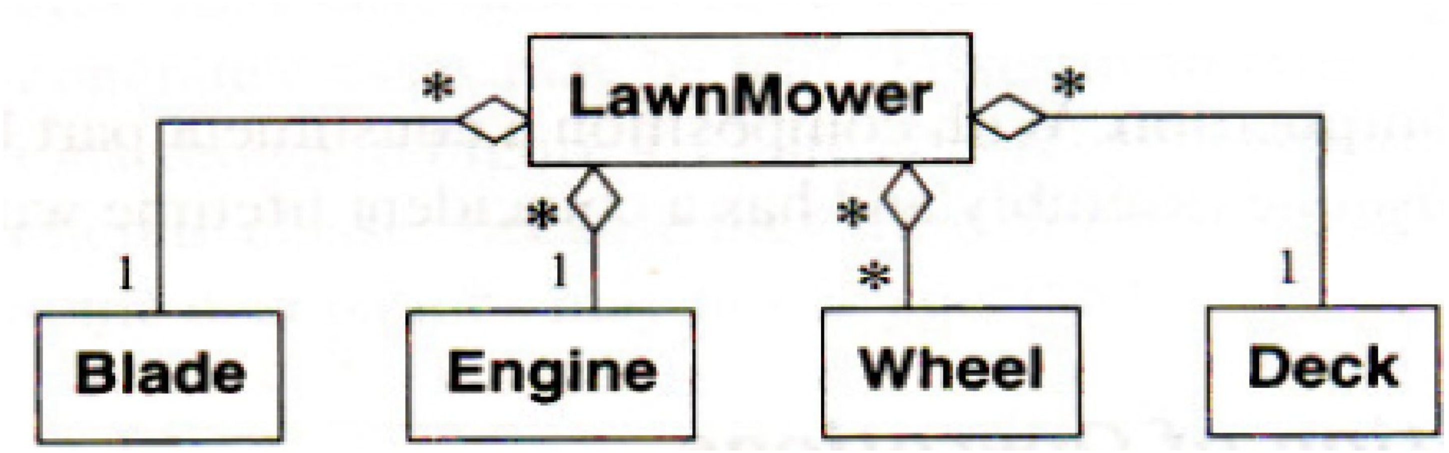
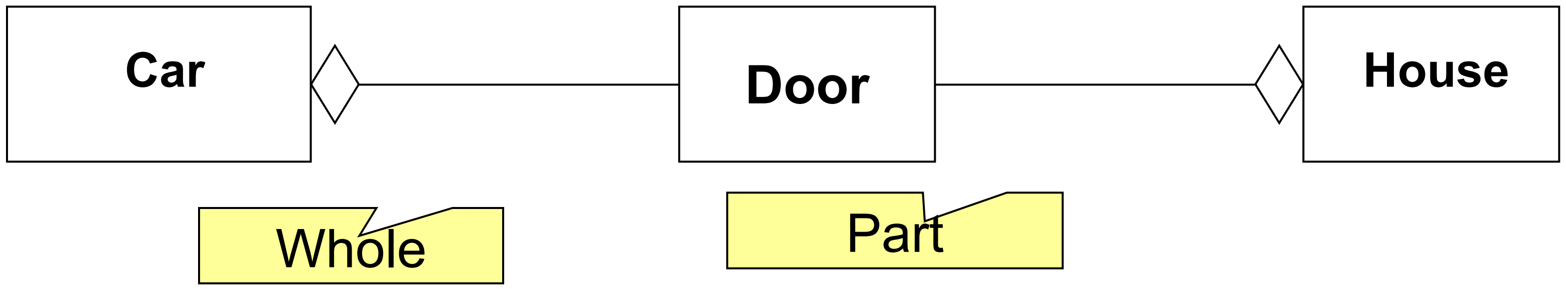


Figure 4.9: Aggregation is a kind of association in which an aggregate object is made of constituent parts

Some tests include:

- Would one use the phrase ***part of***?
- Do some ***operations*** on the whole ***automatically apply to its parts***?
- Do some attribute values ***propagate*** from the whole to all or some parts?
- Is there an ***intrinsic asymmetry*** to the association, where one class is subordinate to the other?

Aggregation (cont.)



- Aggregation tests:
  - Is the phrase “part of” used to describe the relationship?
    - A door is “part of” a car
  - Are some operations on the whole automatically applied to its parts?
    - Move the car, move the door.
  - Are some attribute values propagated from the whole to all or some of its parts?
    - The car is blue, therefore the door is blue.
    - A door **is** part of a car. A car **is not** part of a door.

- Aggregation is **drawn like ass**ociation, except a ***small diamond*** indicates the ***assembly end***.
- The decision to use aggregation is a *matter of judgment* and can be *arbitrary*. Often it is not obvious if an association should be modeled as an aggregation. To a large extent this kind of *uncertainty is typical of modeling; modeling requires seasoned judgment and there are few hard and fast rules*. But careful and consistent judgment ensures no problems with imprecise distinction between aggregation and ordinary association.

Aggregation Versus Composition:

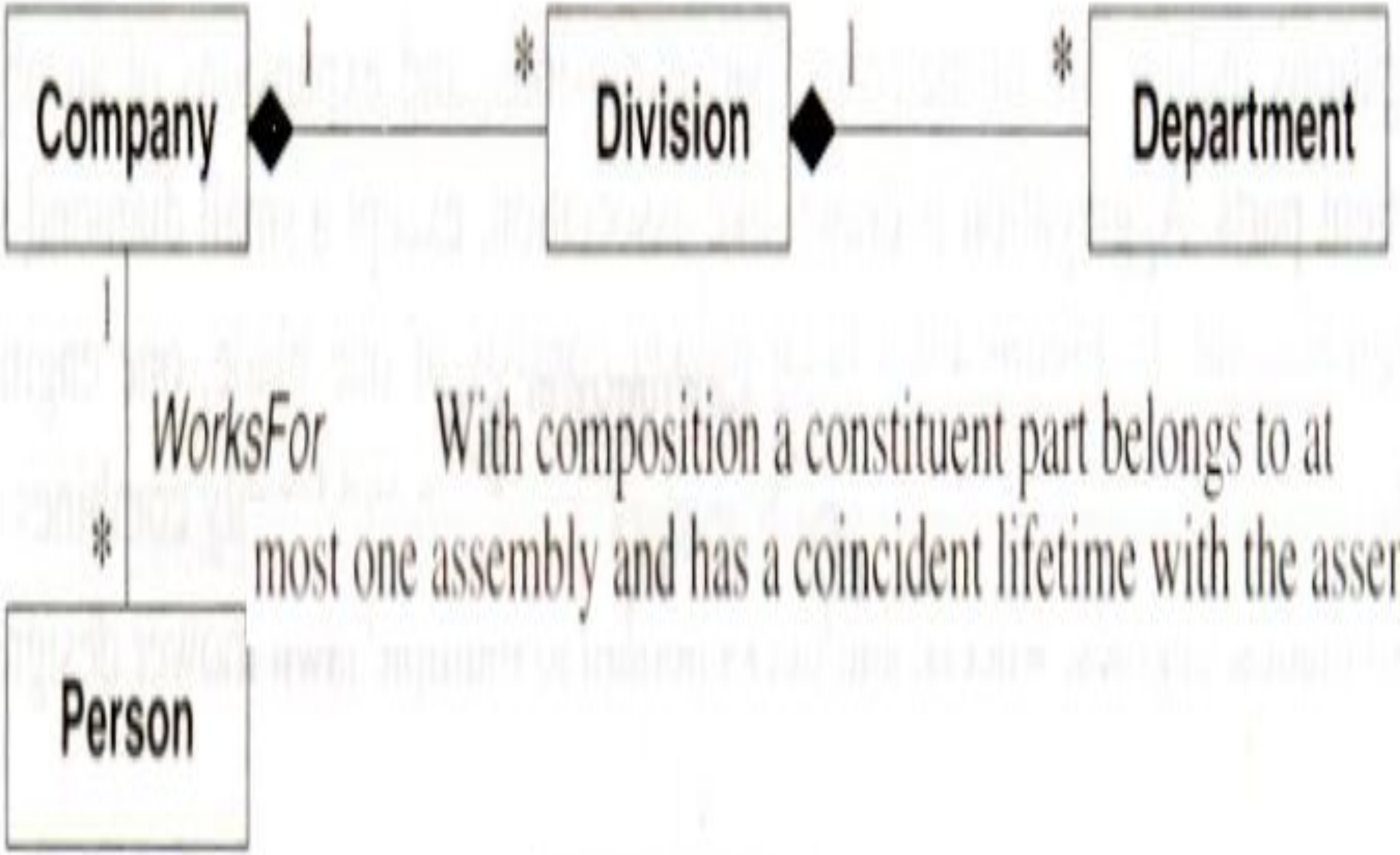
- The UML has **two forms of part-whole relationships**:

 a general form called *aggregation and*

 a more restrictive form called ***composition.***

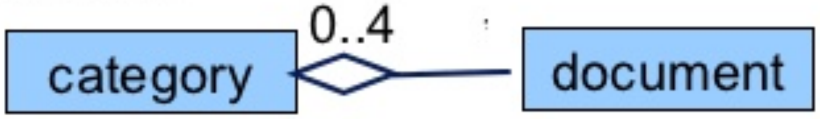
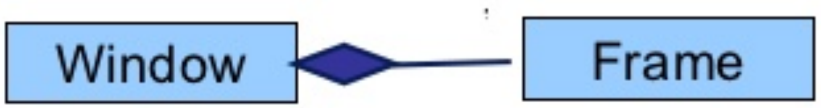
Composition is a form of aggregation with **two additional constraints**. A constituent part can belong to at most one assembly. Furthermore, once a constituent part has been assigned an assembly, it has a ***coincident lifetime*** with the assembly. Thus composition implies **ownership** of the parts by the whole. **This can be convenient for programming**: Deletion of an assembly object triggers deletion of all constituent objects via composition. The notation for composition is a small ***solid diamond*** next to the assembly class (vs. a small *hollow diamond* for the general form of aggregation).

Composition.



With composition a constituent part belongs to at most one assembly and has a coincident lifetime with the assembly.

Composition vs. Aggregation

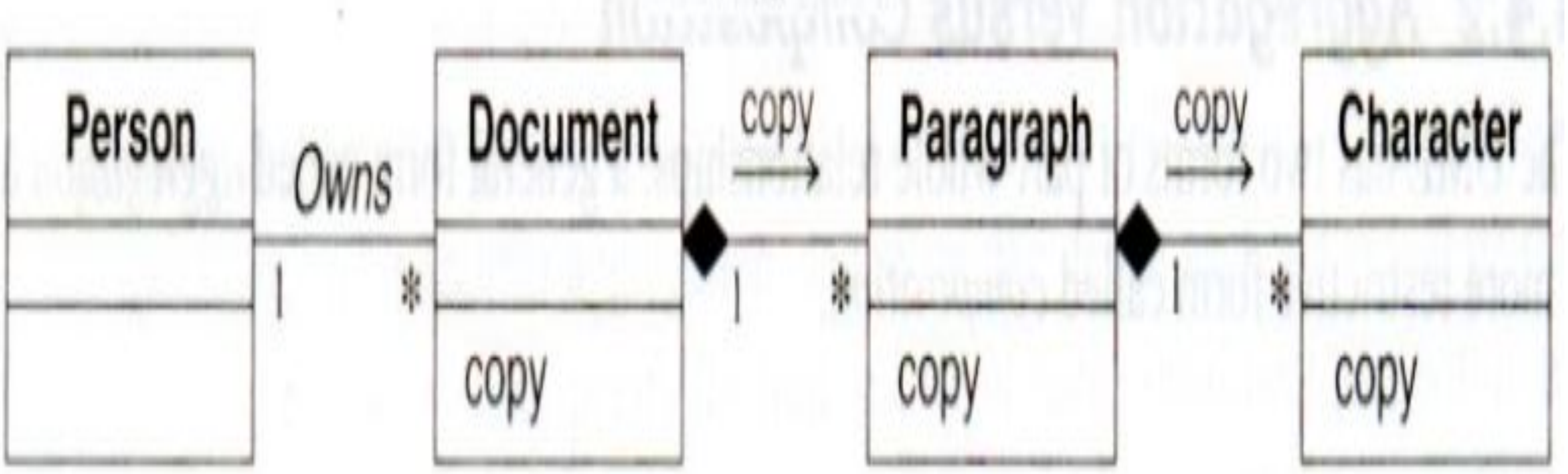
Aggregation	Composition
Part can be shared by several wholes 	Part is always a part of a single whole 
Parts can live independently (i.e., whole cardinality can be 0..*)	Parts exist only as part of the whole. When the wall is destroyed, they are destroyed
Whole is not solely responsible for the object	Whole is responsible and should create/destroy the objects

Propagation of Operations:

- *Propagation* (also called ***triggering)*** is the automatic application of an operation to a network of objects when the operation is applied to some starting object [Rumbaugh-88].
- E.g. ***Moving*** an aggregate moves its parts; the move operation propagates to the parts. Propagation of operations to parts is often a good indicator of aggregation.

- **E.g**. A person owns multiple documents. Each document consists of paragraphs that, in turn, consist of characters. The copy operation propagates from documents to paragraphs to characters.

Propagation.



Operations can be propagated across aggregations and compositions

- **Copying** a paragraph copies all the characters in it. The operation **does not propagate in the reverse direction**; a paragraph can be copied without copying the whole document. Similarly, copying a document copies the owner link but does not spawn a copy of the person who is owner.

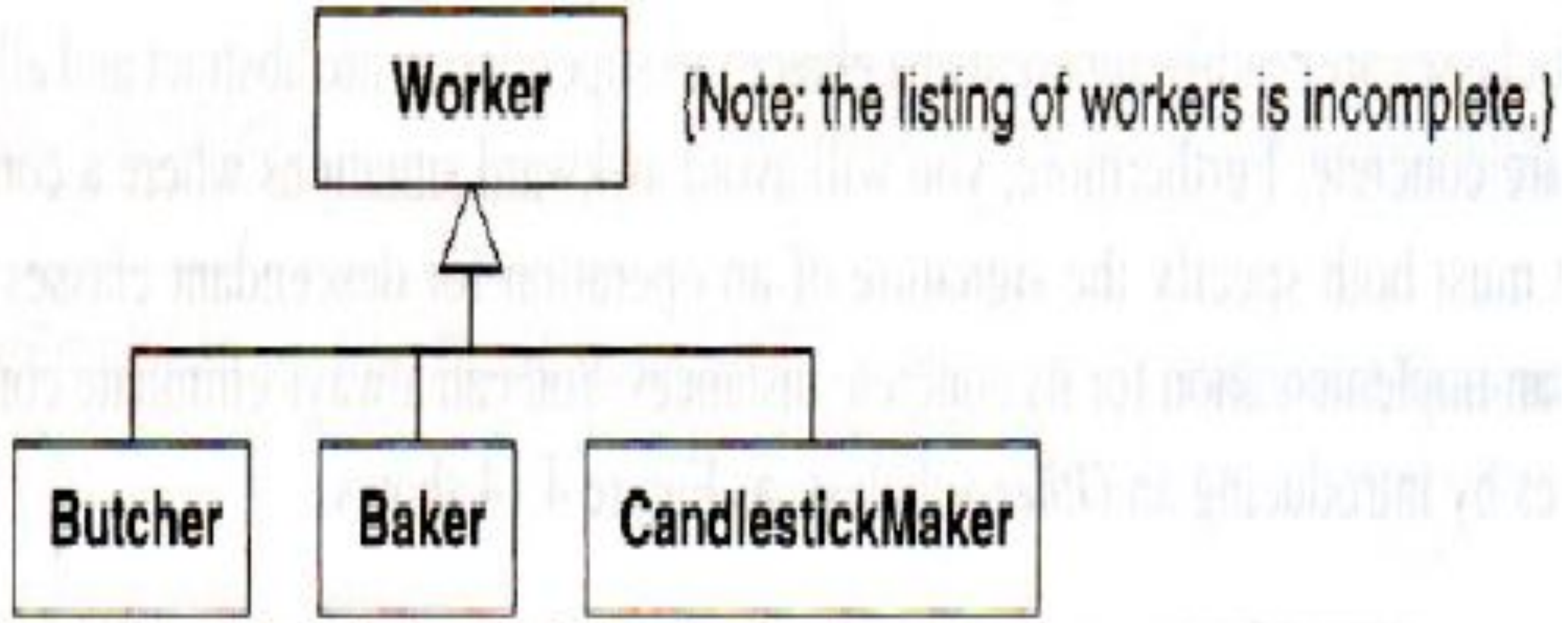
Abstract Classes:

- An *abstract class* is a class that has *no direct instances* but whose *descendant classes* have *direct instances*.
- A *concrete class* is a class that is *instantiable*; that is, it can have direct instances. A concrete class may have abstract subclasses (but they, in turn, must have concrete descendants). Only concrete classes may be *leaf classes* in an inheritance tree.

All the occupations shown below are concrete classes.

- *Butcher;Baker;* and *CandlestickMaker* are concrete classes because they have direct instances. *Worker* also is a concrete class because some occupations may not be specified.

Concrete classes

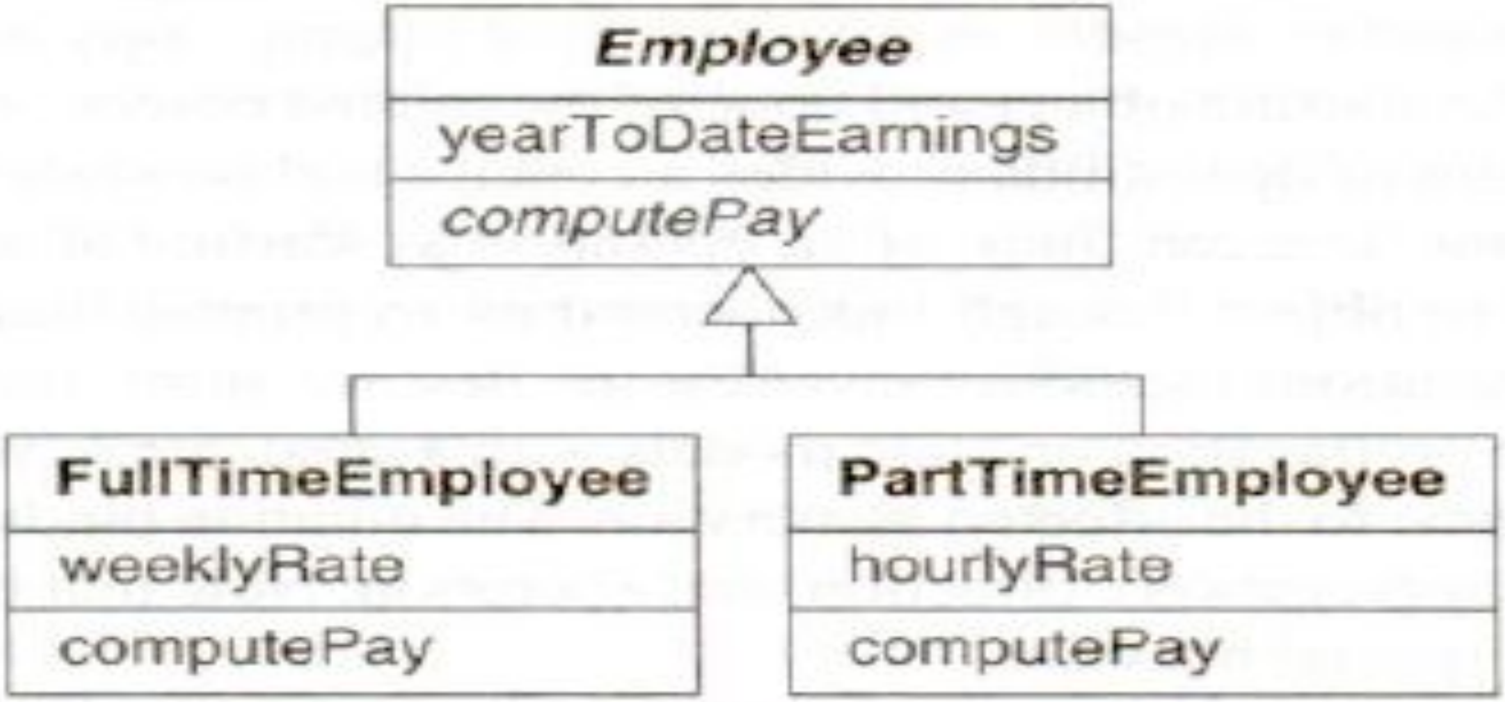


A concrete class is instantiable; that is, it can have direct instances.

Class *Employee* below is an example of an abstract class.

- All employees must be either full-time or part-time. *FullTimeEmployee* and *PartTimeEmployee* are concrete classes because they can be directly instantiated.

Abstract class and abstract operation.



An abstract class is a class that has no direct instances

UML notation :

- Abstract class name is listed in an ***italic*** font. Or the keyword ***{abstract}*** may be placed below or after the name.
- Abstract classes can be used to define methods that can be inherited by subclasses.
- Alternatively, an abstract class can define the signature for an **operation** without supplying a corresponding **method**. This operation is called an ***abstract operation.*** (Recall that an operation specifies the form of a function or procedure; a method is the actual implementation.)

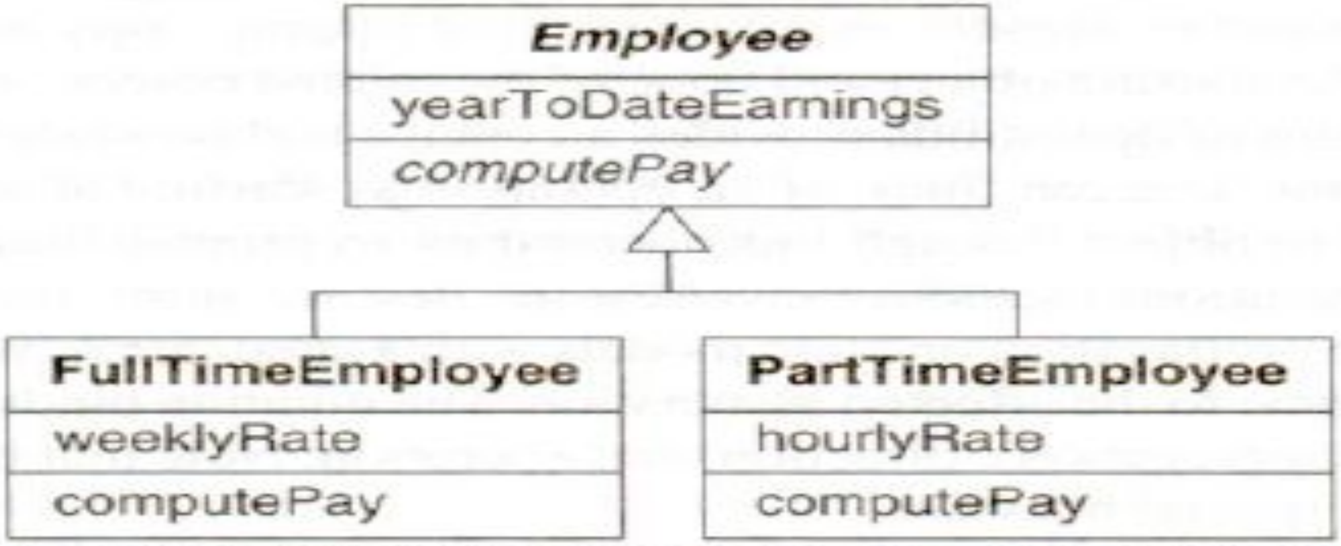
Abstract operation

- An abstract operation defines the signature of an operation for which each *concrete subclass* must provide its *own* **implementation.** A concrete class may not contain abstract operations, because objects of the concrete class would have *undefined operations*.

Figure below shows an ***abstract operation***.

- An abstract operation is designated by ***italics*** or the keyword ***{abstract}.*** *ComputePay* is an abstract operation of class *Employee;* its signature but not its implementation is defined. Each subclass must supply a method for this operation.

Abstract class and abstract operation.

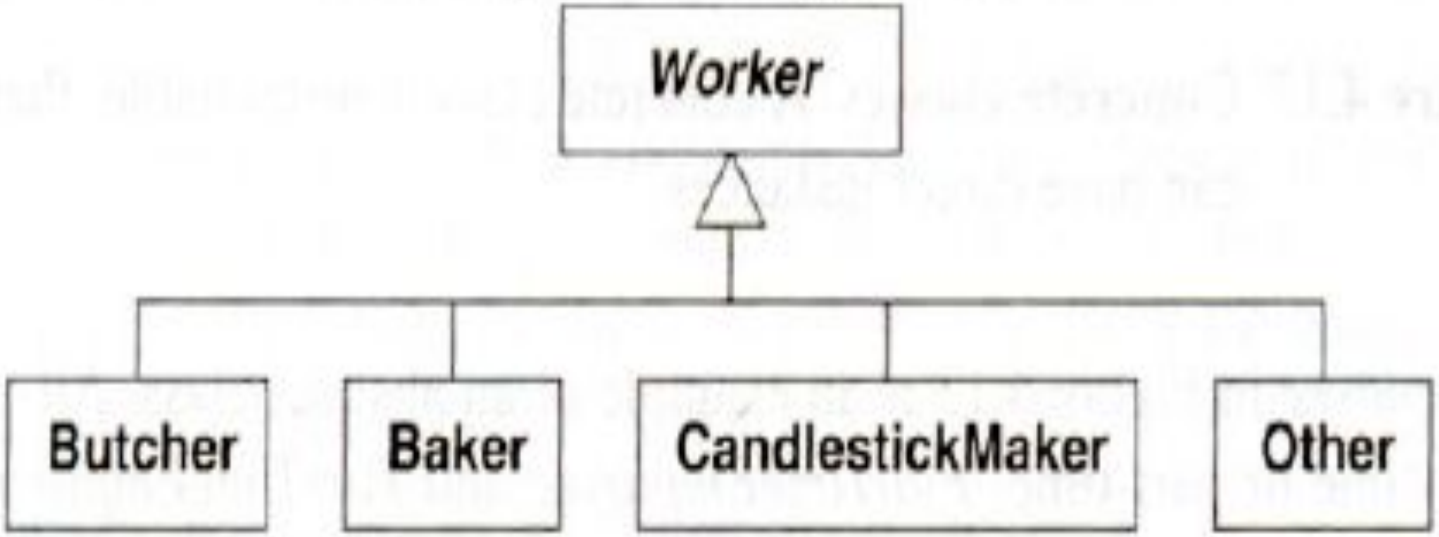


An abstract class is a class that has no direct instances

- Note that the ***abstract nature of a class is always provisional***, depending on the point of view. A concrete class can always be refined into subclasses, making it abstract. Conversely, an abstract class may become concrete in an application in which the difference among its subclasses is unimportant.
- As a matter of style, it is a good idea to *avoid concrete* superclasses. Then, abstract and concrete classes are readily apparent at a glance; all superclasses are abstract and all leaf subclasses are concrete.

- Furthermore, a concrete superclass specifying both the signature of an operation for descendant classes and also provide an implementation for its concrete instances is an awkward situations, which should be avoided. **Concrete superclasses can be eliminated** by introducing an *Other* subclass, as below.

Avoiding concrete superclasses.



always eliminate concrete superclasses by introducing an *Other* subclass.

Multiple Inheritance:

- *Multiple inheritance* permits a class to have ***more than one superclass*** and to inherit features from ***all parents***. Then information from two or more sources can be mixed.
- This is a more complicated form of generalization than ***single inheritance***, which restricts the class ***hierarchy to a tree***. The advantage of multiple inheritance is **greater power in specifying classes and an increased opportunity for reuse**. **The disadvantage is a *loss of conceptual and implementation simplicity*.**

- The term *multiple inheritance* is used **somewhat imprecisely** to *mean* either
- the ***conceptual relationship*** between classes or the ***language mechanism that implements that relationship***.
- Whenever possible, *generalization* (the *conceptual relationship*) and *inheritance* (the *language mechanism*) must be distinguished, but the term "*multiple inheritance*" is more *widely used* than the term "multiple generalization."

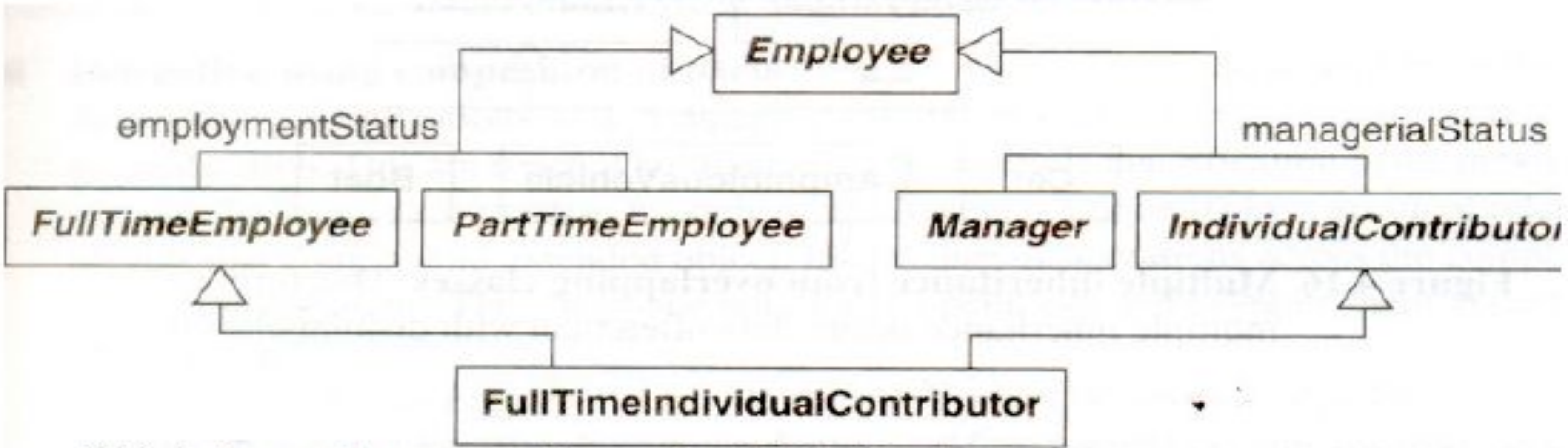
Kinds of Multiple Inheritance:

- **Multiple Inheritance From sets of disjoint classes**:
- **M I With overlapping classes**:

From sets of disjoint classes:

- Each subclass inherits from one class in each set.
- In Figure, *FullTimeIndividualContributor* is both *FullTimeEmployee* and *IndividualContributor* and combines their features. *FullTimeEmployee* and *PartTimeEmployee* are disjoint; each employee must belong to exactly one of these. Similarly, *Manager* and *IndividualContributor* are also disjoint and each employee must be one or the other. The model does not show it, but we could define three additional combinations: *FullTimeManager, PartTimeIndividualContributor,* and *PartTimeManager.*

Multiple inheritance from disjoint classes.



This is the most common form of multiple inheritance.

- Each generalization should cover a single aspect. Multiple generalizations must be used if a class can be ***refined on several distinct and independent aspects***.
- E.g. In previous figure, class*Employee* independently specializes on employment status and managerial status. Consequently the model has two separate generalization sets.

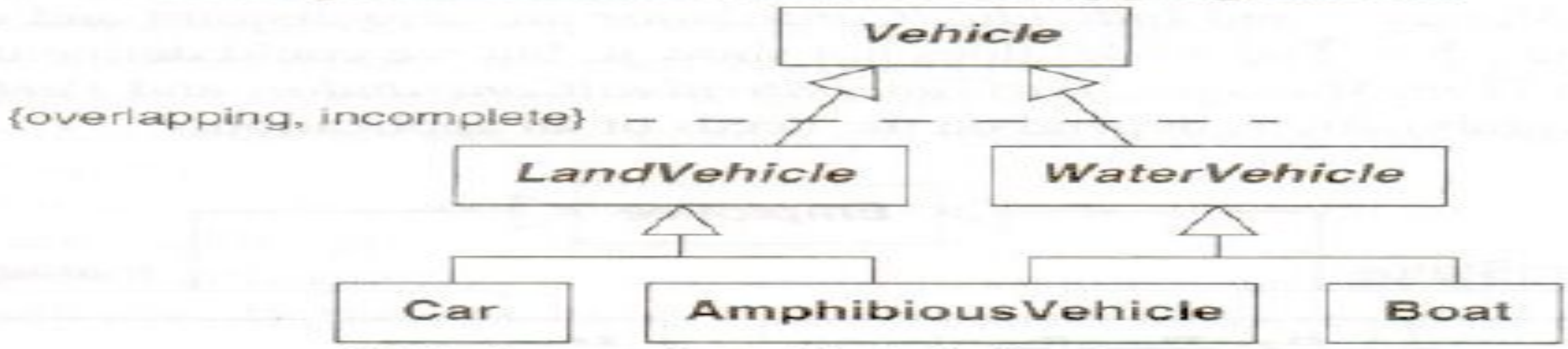
- A subclass inherits a feature from the same ancestor class found along more than one path ***only once***; it is the same feature. E.g. in the previous figure, *FullTimeIndividualContributor* inherits *Employee* features along two paths, via *employmentStatus* and *managerialStatus.* However, each *FullTimeIndividualContributor* has only a single copy of *Employee* features.

- ***Conflicts*** among parallel definitions create ambiguities that ***implementations must resolve***. In practice, such conflicts should be avoided or explicitly resolved in models, even if a particular language provides a priority rule for resolving conflicts.
- E.g. Suppose that *FullTimeEmployee* and *IndividualContributor* both have an attribute called *name. FullTimeEmployee.name* could refer to the person's full name while *IndividualContributor.name* might refer to the person's title. In principle, there is no obvious way to resolve such clashes. The best solution is to try to avoid them by restating the attributes as *FullTimeEmployee.personName* and *IndividualContributor.title.*

With overlapping classes:

- In figure, *AmphibiousVehicle* is both *LandVehicle* and *WaterVehicle. LandVehicle* and *WaterVehicle* overlap, because **some vehicles travel *on* both land and water.** The UML uses a constraint (Section – Constraints) to indicate an overlapping generalization set; the notation is a dotted line cutting across the affected generalizations with keywords in braces. In this example, ***overlapping*** **means that an individual vehicle may belong to more than *one* of the subclasses. *Incomplete* means that all possible subclasses of vehicle have not been explicitly named.**

Multiple inheritance from overlapping classes.



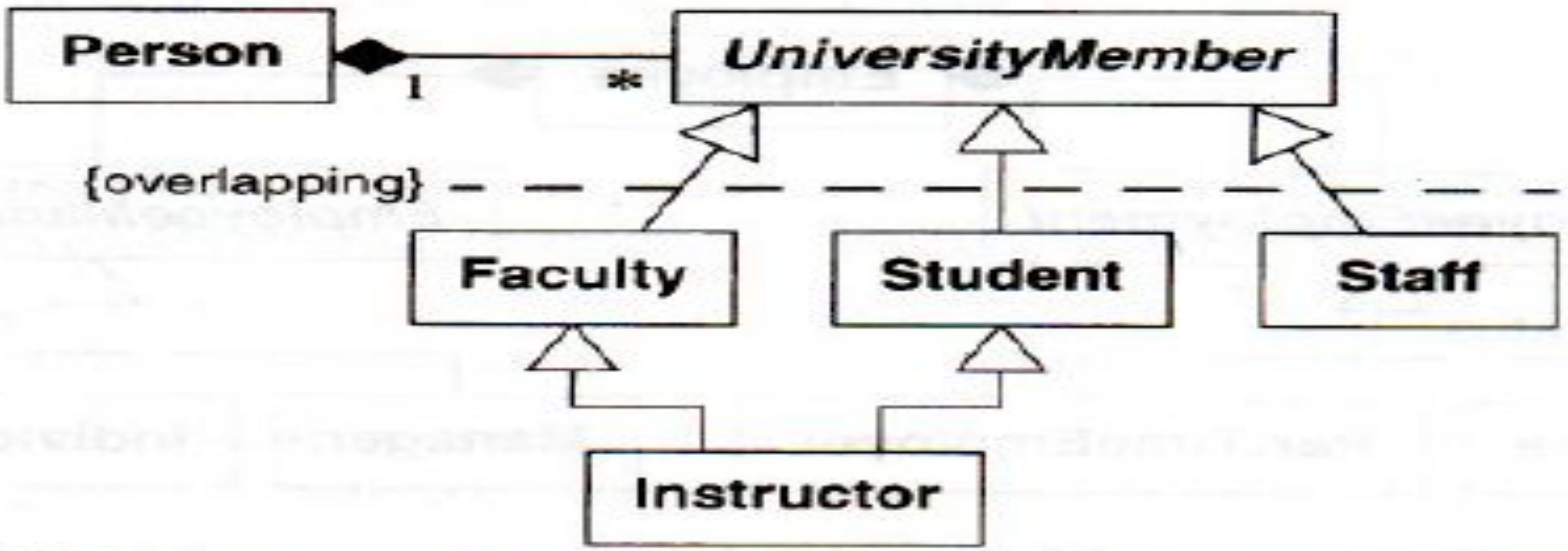
This form of multiple inheritance occurs less often than with disjoint classes

Multiple Classification:

- An instance of a class is inherently an instance of all ancestors of the class. E.g. an instructor could be both faculty and student. But what about a Harvard Professor taking classes at MIT? There is no class to describe the combination (it would be artificial to make one). This is an example of multiple classification in which ***one instance happens to participate in two overlapping classes***.

- The *UML permits* multiple classification, but most *OO languages handle it poorly.*
- As figure shows, the best approach using conventional languages is to treat Person as an object composed of multiple UniversityMember objects. This ***workaround*** replaces inheritance with ***delegation*** (discussed in the next section). This is not totally satisfactory, because there is a ***loss of identity*** between the separate roles, but the alternatives involve radical changes in many programming languages [McAllester-86].

Workaround for multiple classification.



OO languages do not handle this well, so you must use a workaround.

Workarounds

- Dealing with lack of multiple inheritance is really an implementation issue, but early restructuring of a model is often the easiest way to work around its absence.
- Two of the approaches make use of delegation, which is an implementation mechanism by which an object forwards an operation to another object for execution

- **Delegation using composition of parts**. You can recast a superclass with multiple independent generalizations as a composition in which each constituent part replaces a generalization.

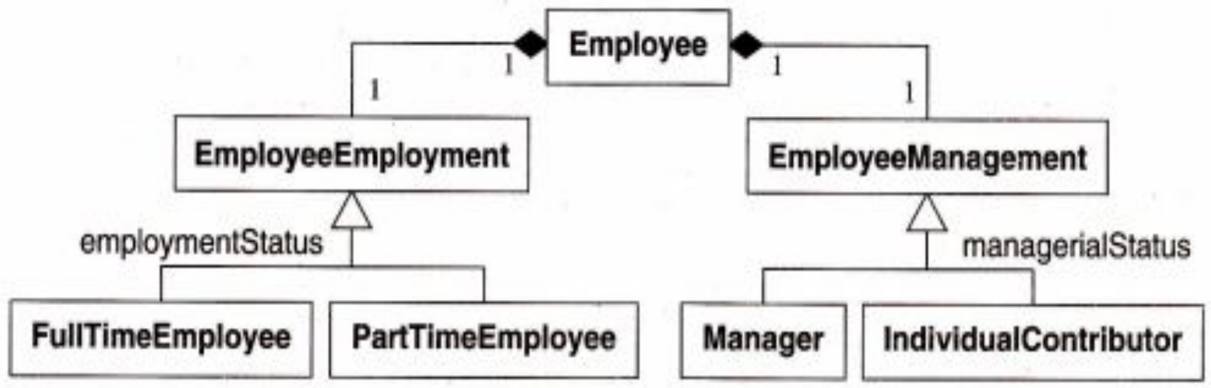


Figure 4.18 Workaround for multiple inheritance—delegation

- **Inherit the most important class and delegate the rest.** Figure 4.19 preserves identity and inheritance across the most important generalization. You degrade the remaining generalizations to composition and delegate their operations as in the previous alternative.

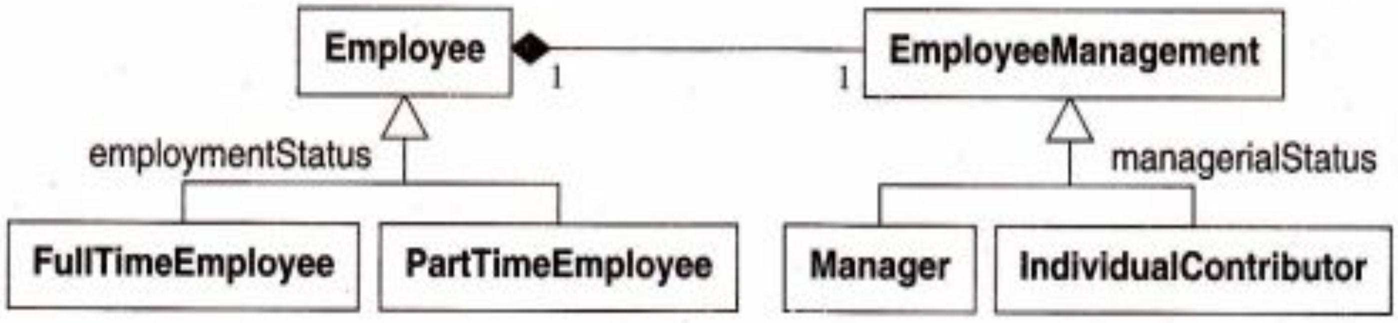


Figure 4.19 Workaround for multiple inheritance—inheritance and delegation

- **Nested generalization Factor on one generalization** first, then the other. This approach multiplies out all possible combinations. For example, in Figure 4.20 under FullTimeEmployee and PartTimeEmployee, add two subclasses for managers and individual contributors. This preserves inheritance but duplicates declarations and code and violates the spirit of OO programming.

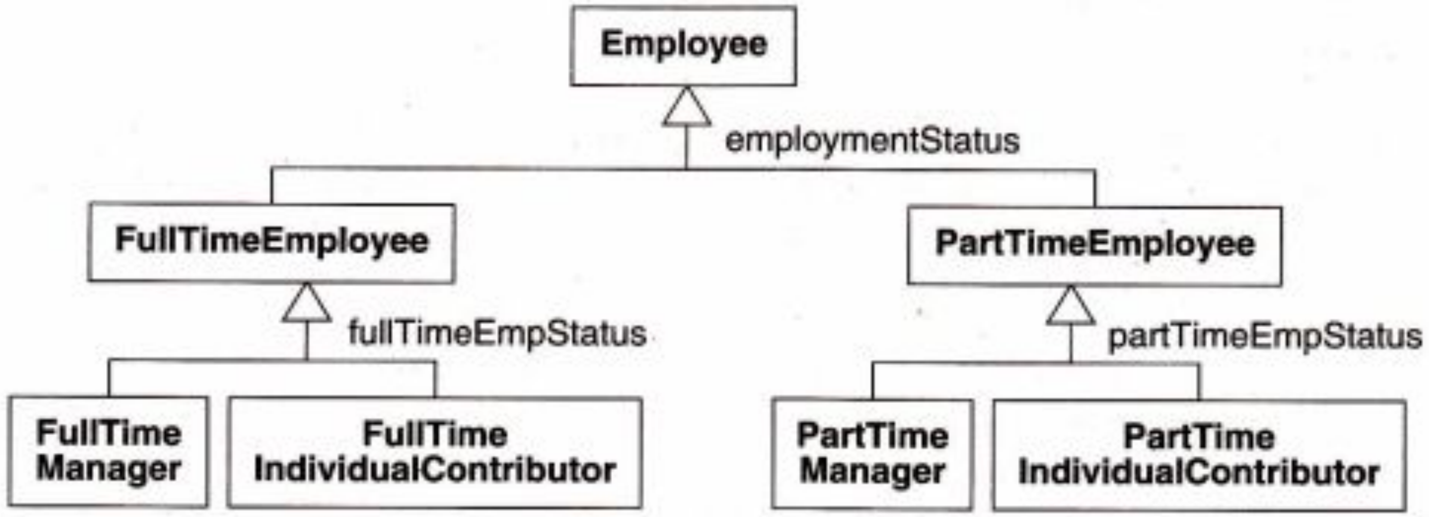


Figure 4.20 Workaround for multiple inheritance—nested generalization

- There are several issues to consider when selecting the best workaround.
- **Superclasses of equal importance**: If a subclass has several superclasses, all of equal importance, it may be best to use delegation (Figure 4.18) and preserve symmetry in the model.
- **Dominant superclass.** If one superclass clearly dominates and the others are less important, preserve inheritance through this path. figure 4.19 or Figure 4.20).

- **Few subclasses.** If the number of combinations is small, consider nested generalization (Figure 4.20).If the number of combinations is large, avoid it.
- **Sequencing generalization sets.** If you use nested generalization (Figure 4.20),factor on the most important criterion first, the next most important second, and so forth.
- **Large quantities of code.** Try to avoid nested generalization (Figure 4.20) if you must duplicate large quantities of code.

- **Identity.** Consider the importance of maintaining strict identity. Only nested generalization (Figure 4.20) preserves this.

Metadata

- Metadata is data that describes other data.
- For example, a class definition is metadata. Models are inherently metadata, since they describe the things being modelled (rather than being the things).
- Many real-world applications have metadata, such as parts catalogs, blueprints, and dictionaries.
- Computer-language implementations also use metadata heavily

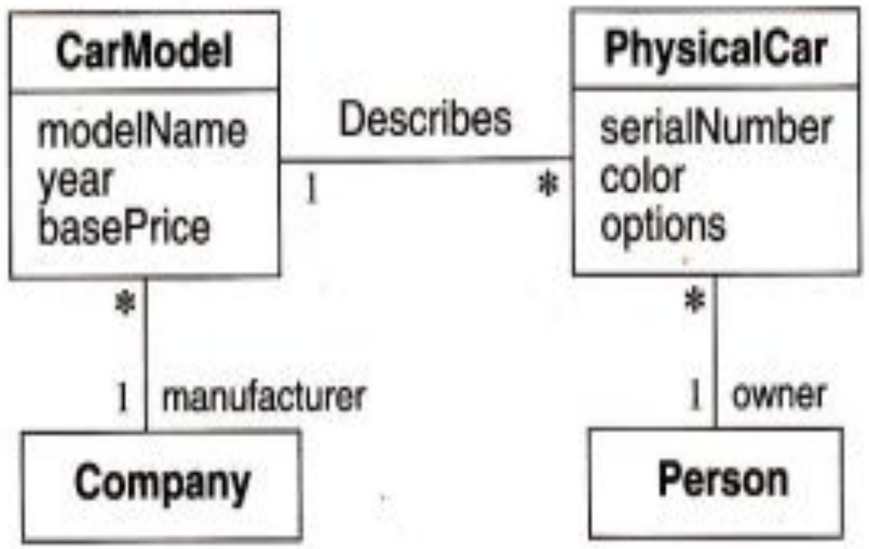


Figure 4.21 Example of metadata. Metadata often arises in applications.

Reification

- Reification is the promotion of something that is not an object into an object.
- Reification is a helpful technique for meta applications because it lets you shift the level of abstraction.
- On occasion it is useful to promote attributes, methods, constraints, and control information into objects so you can describe and manipulate them as data.

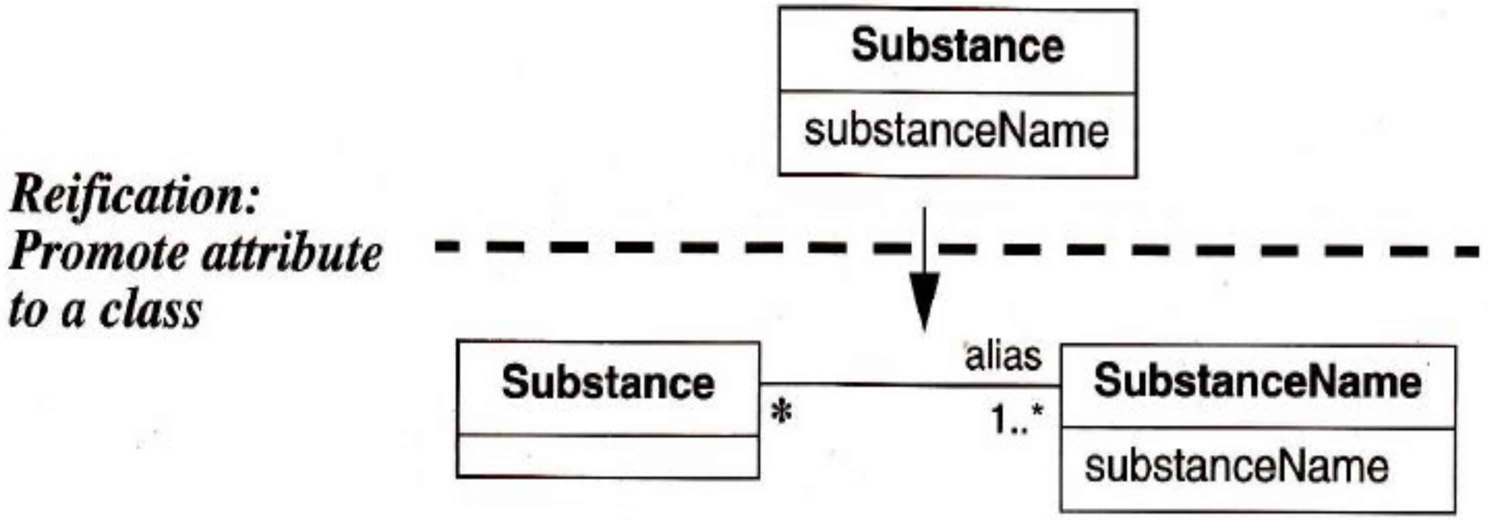


Figure 4.22 Reification. Reification is the promotion of something that is not an object into an object and can be helpful for meta applications.

Constraints

- A constraint is a boolean condition involving model elements, such as objects, classes, attributes, links, associations, and generalization sets.
- A constraint restricts the values that elements can assume.

Constraints on Objects

- Figure 4.23 shows several examples of constraints.
- No employee's salary can exceed the salary of the employee's boss (a constraint between two things at the same time).
- No window can have an aspect ratio (length/width) of less than 0.8 or greater than 1.5 (a constraint between attributes of a single object).
- The priority of a job may not increase (constraint on the same object over time).
- You may place simple constraints in class models.

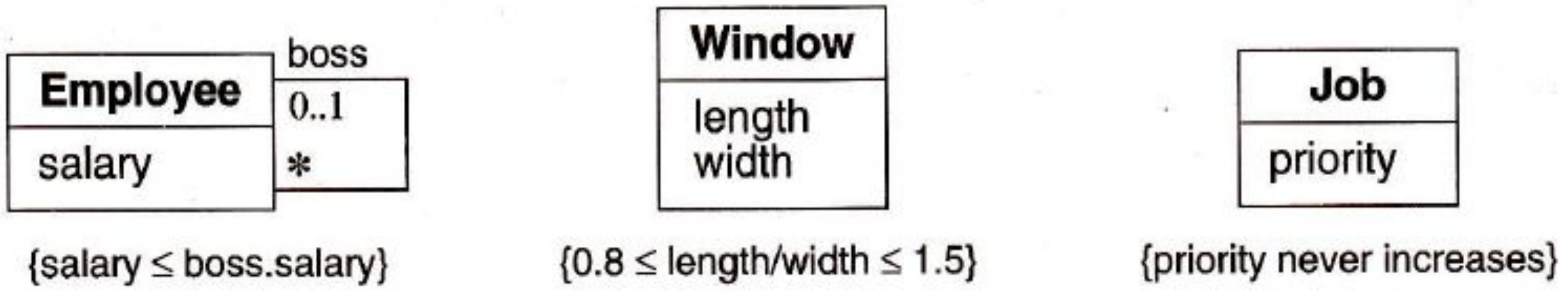


Figure 4.23 Constraints on objects. The structure of a model expresses many constraints, but sometimes it is helpful to add explicit constraints.

Constraints on generalization Sets

- class models capture many constraints through their very structure.
- With single inheritance the subclasses are mutually exclusive.
- Furthermore, each instance of an abstract superclass corresponds to exactly one subclass instance.
- Each instance of a concrete superclass corresponds to at most one subclass instance.

The UML defines the following keywords for generalization sets.

- **Disjoint:** The subclasses are mutually exclusive. Each object belongs to exactly one of the subclasses.
- **Overlapping:** The subclasses can share some objects. An object may belong to more than one subclass.
- **Complete**: The generalization lists all the possible subclasses.
- **Incomplete**: The generalization may be missing some subclasses.

Constraints on Links

- Multiplicity is a constraint on the cardinality of a set. Multiplicity for an association restricts the number of objects related to a given object.
- Multiplicity for an attribute specifies the number of values that are possible for each instantiation of an attribute
- Qualification also constrains an association. A qualifier attribute does not merely describe the links of an association but is also significant in resolving the "many" objects at an association end.

- An association class implies a constraint. An association class is a class in every right; for example, it can have attributes an operations, participate in associations, and participate in generalizations.
- But an association class has a constraint that an ordinary class does not; it derives identity from instances of the related classes.
- An ordinary association presumes no particular order on the objects of a "many" end.
- The constraint {ordered} indicates that the elements of a "many" association end have an explicit order that must be preserved.

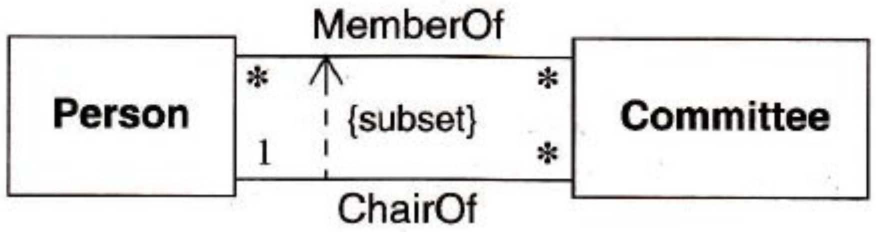


Figure 4.24 Subset constraint between associations.

Derived data

- Classes, attributes and associations may be derived from others.
- A derived element is a function of one or more elements, which in turn may be derived.
- The notation for a derived element is a slash (/) in front of the element name.
- The constraint that determines the derivation should also be shown.

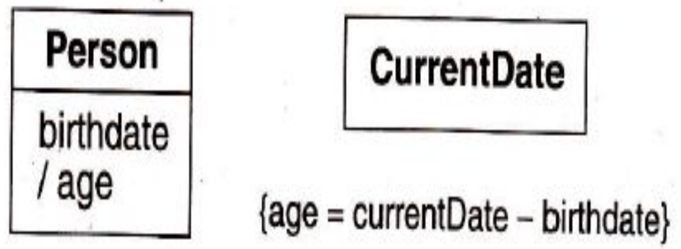


Figure 4.25 Derived attribute. A derived attribute is a function of one or more elements.

Packages

- A package is a group of elements (classes, association, generalization, and lesser packages) with a common theme.
- A package partitions a model making it easier to understand and manage. Large applications may require several tiers of packages.
- UML Notation for package is a box with a tab.

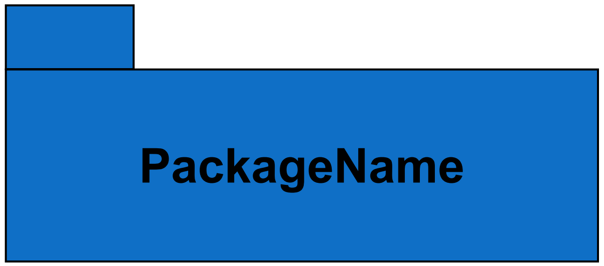
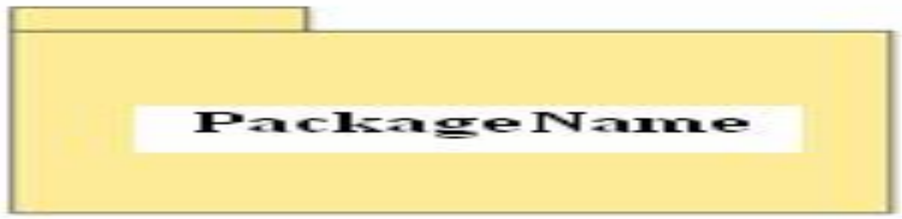
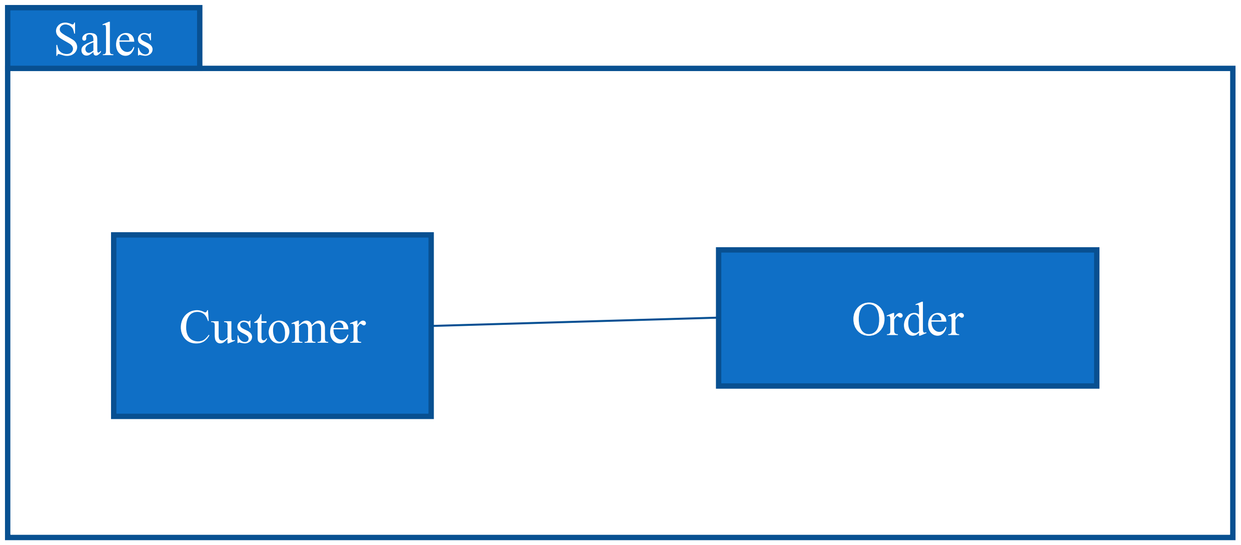


Figure 4.27 Notation for a package. Packages let you organize large models so that persons can more readily understand them'


PackageName


Sales
Customer
Order

- We can offer the following tips for devising packages.
- **Carefully delineate each package's scope:** The precise boundaries of a package are a matter of judgment. Like other aspects of modeling, defining the scope of a package requires planning and organization.
- Make sure that class and association names are unique within package, and use consistent names across packages as much as possible.
- **Define each class in a single package:** The defining package should show the class name, attributes, and operations. Other packages that refer to a class can use a class icon, a box that contains only the class name.
- This convention makes it easier to read class models, because a class is prominent in its defining package. Readers are not distracted by definitions that may be inconsistent or misled by forgetting a prior class definition. This convention also makes it easier to develop packages concurrently.

- **Make packages cohesive**: Associations and generalizations should normally appear in a single package, but classes can appear in multiple packages, helping to bind them.
- Try to limit appearances of classes in multiple packages.
- Typically no more than 2O-30% of classes should appear in multiple packages.