UML (Unified Modeling Language)

2 Class Diagrams: The Essentials

Software Institute of
Nanjing University

UML tools

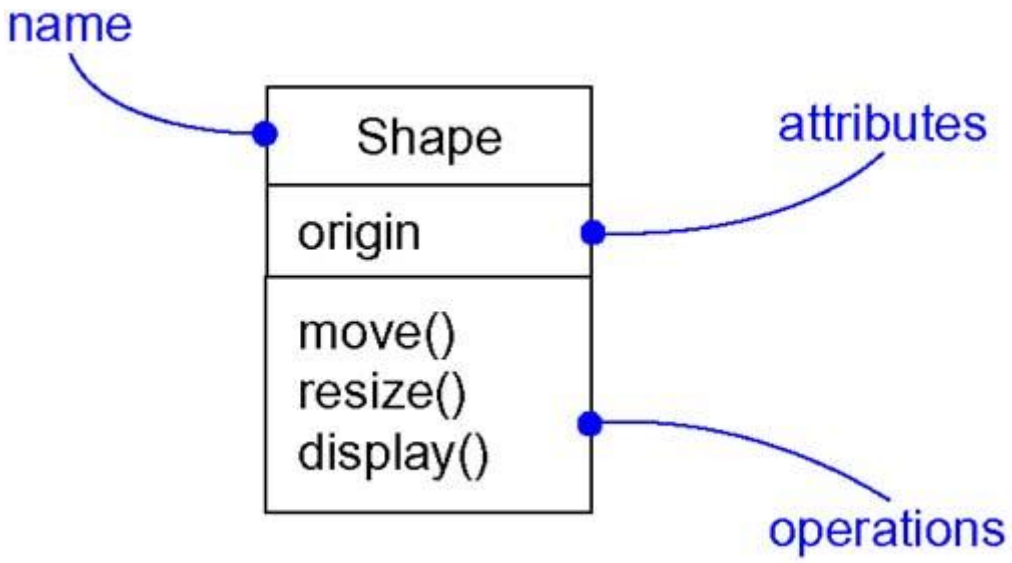
- Rational Rose www.rational.com
- Together www.togethersoft.com
- Microsoft Visio www.Microsoft.com
- Visual UML www.visualobjectmodelers.com
- Model Bridge www.metaintegration.net
- QuickUML www.excelsoftware.com

1. Classes

- A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces.

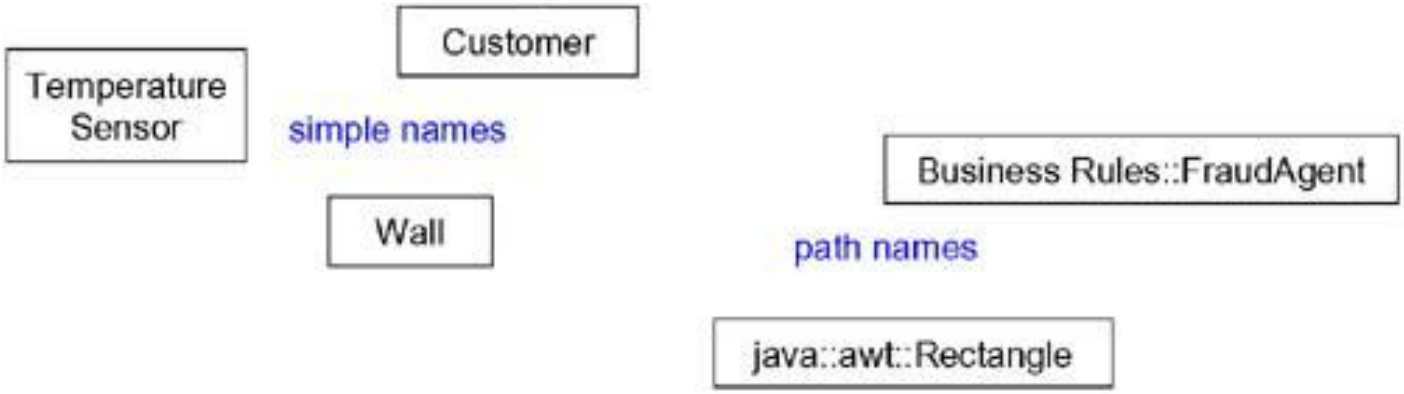
2. Classes in UML

- graphical representation
  - Three compartment : name, attributes, and operations



2.1 Names

- Every class must have a name that distinguishes it from other classes.
  - simple names
  - path names



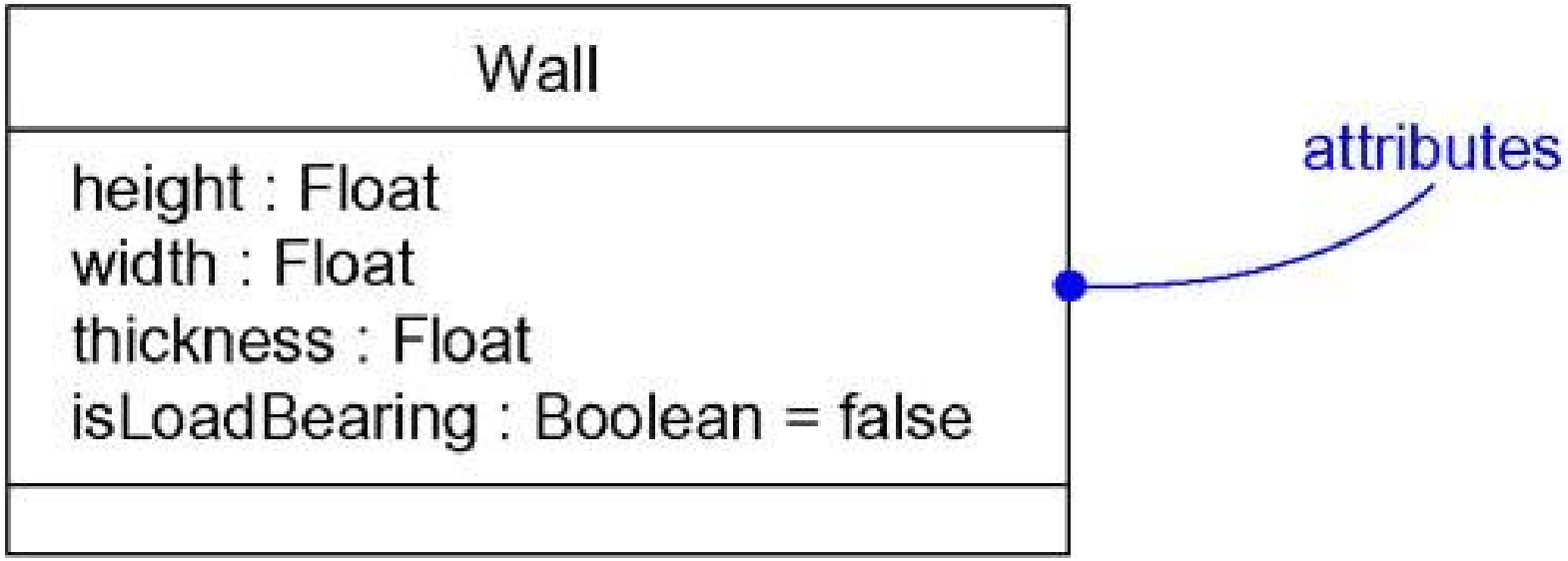
2.2 Attributes

- An *attribute* is a named property of a class that describes a range of values that instances of the property may hold.

2.2 Attributes (cont')

- Graphically, attributes are listed in a compartment just below the class name.
  - May be drawn showing only their names
  - Further specify an attribute by stating its class and possibly a default initial value

2.2 Attributes (cont')

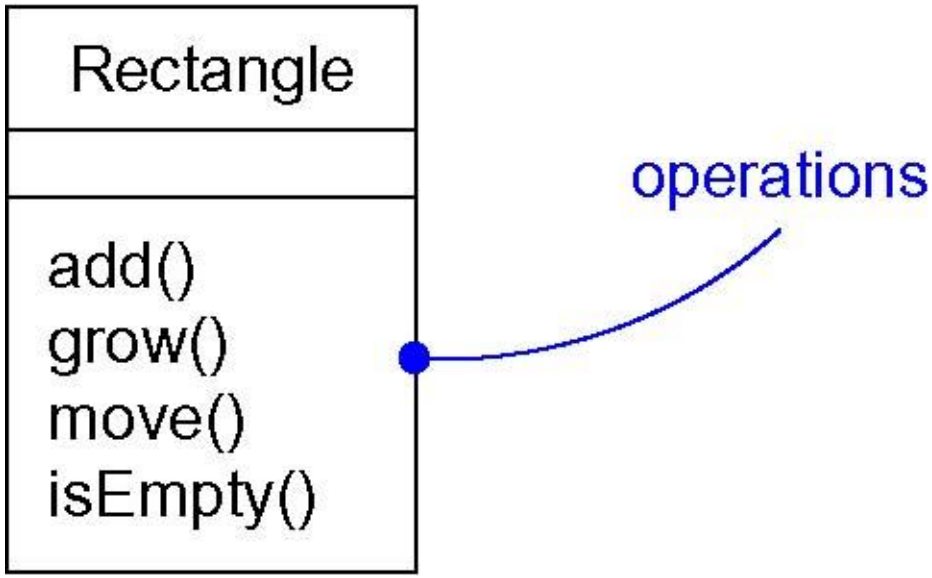


2.3 Operations

- An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior.
- Often (but not always), invoking an operation on an object changes the object's data or state.

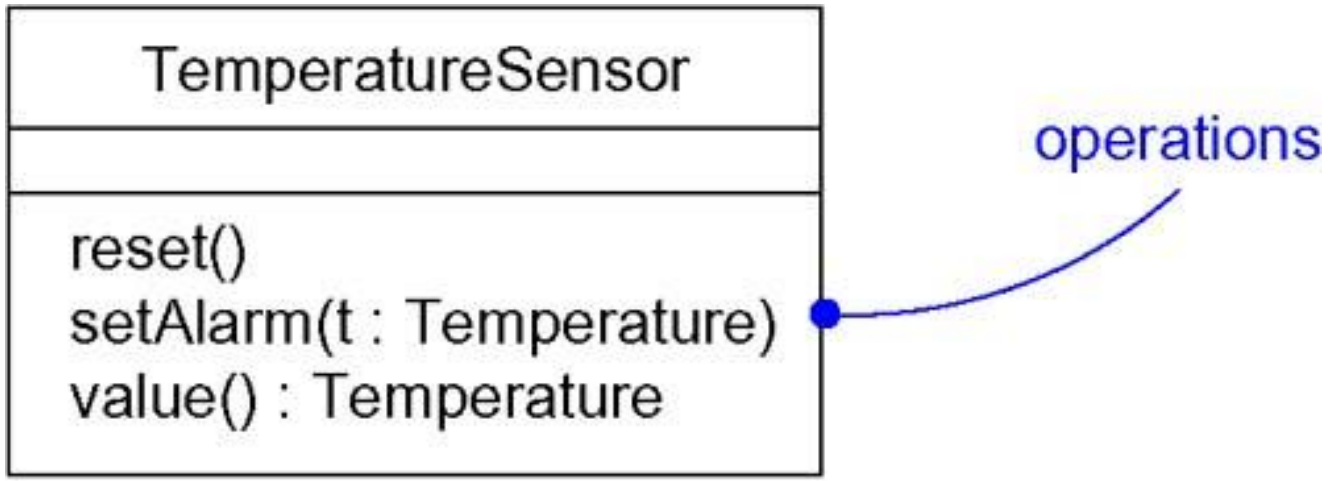
2.3 Operations (cont')

- Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names, as



2.3 Operations (cont')

- You can specify an operation by stating its signature, covering the name, type, and default value of all parameters and (in the case of functions) a return type, as



2.4 Organizing Attributes & Operations

- Don't have to show every attribute and every operation at once.
  - can't (there are too many of them to put in one figure)
  - shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view)

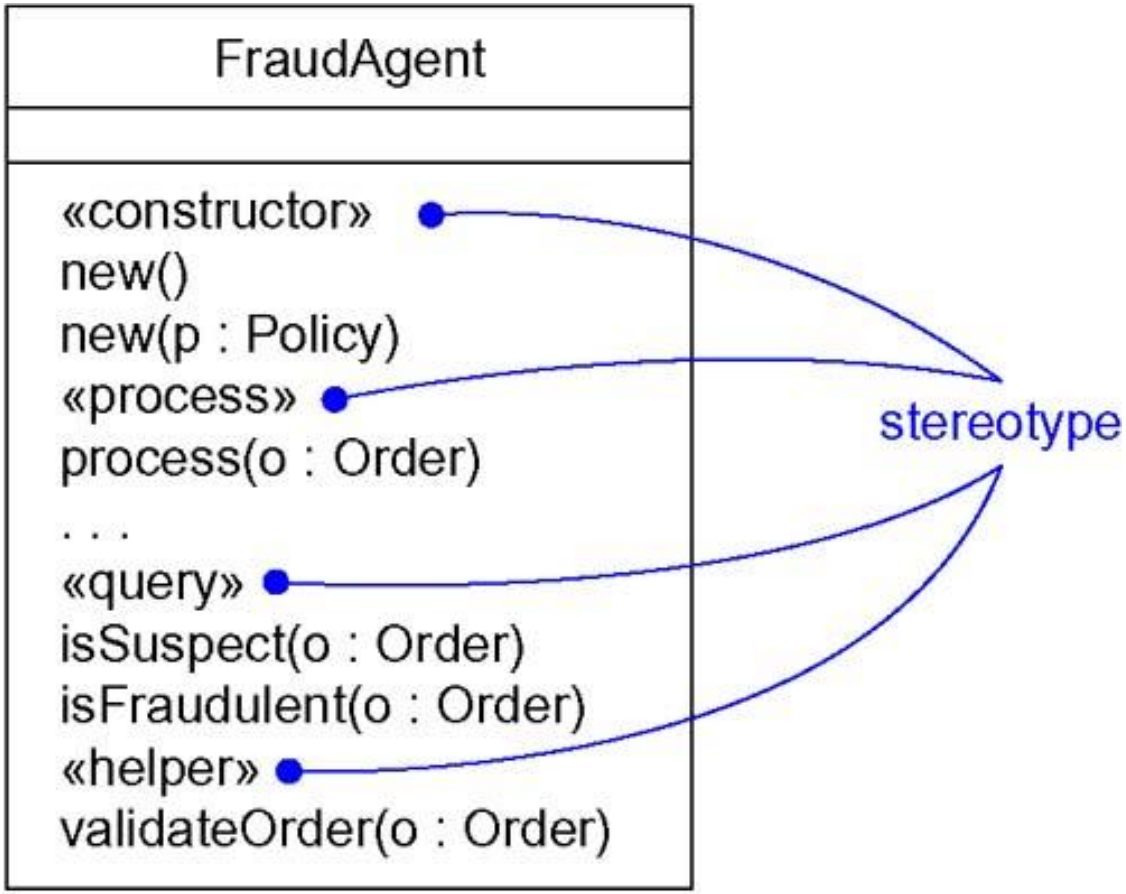
2.4 Organizing A & O (cont’)

- *elide a class* : meaning that you can choose to show only some or none of a class's attributes and operations.
- You can explicitly specify that there are more attributes or properties than shown by ending each list with an ellipsis ("...").

2.4 Organizing A & O (cont’)

- To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes, as ... (see the next page)

2.4 Organizing A & O (cont')

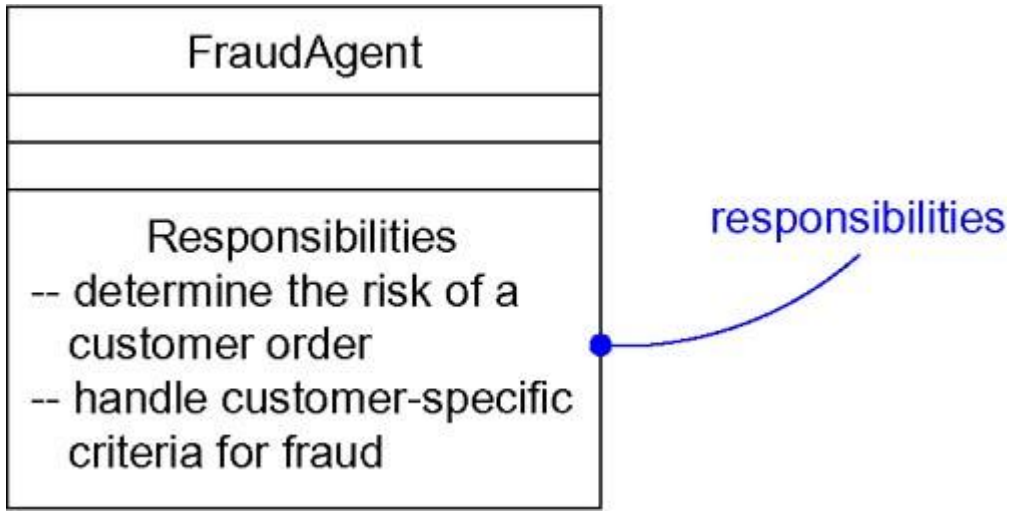


2.5 Responsibility

- A *responsibility* is a contract or an obligation of a class.
- A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful.

2.5 Responsibility (cont')

- Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon, as ... (or in a note)



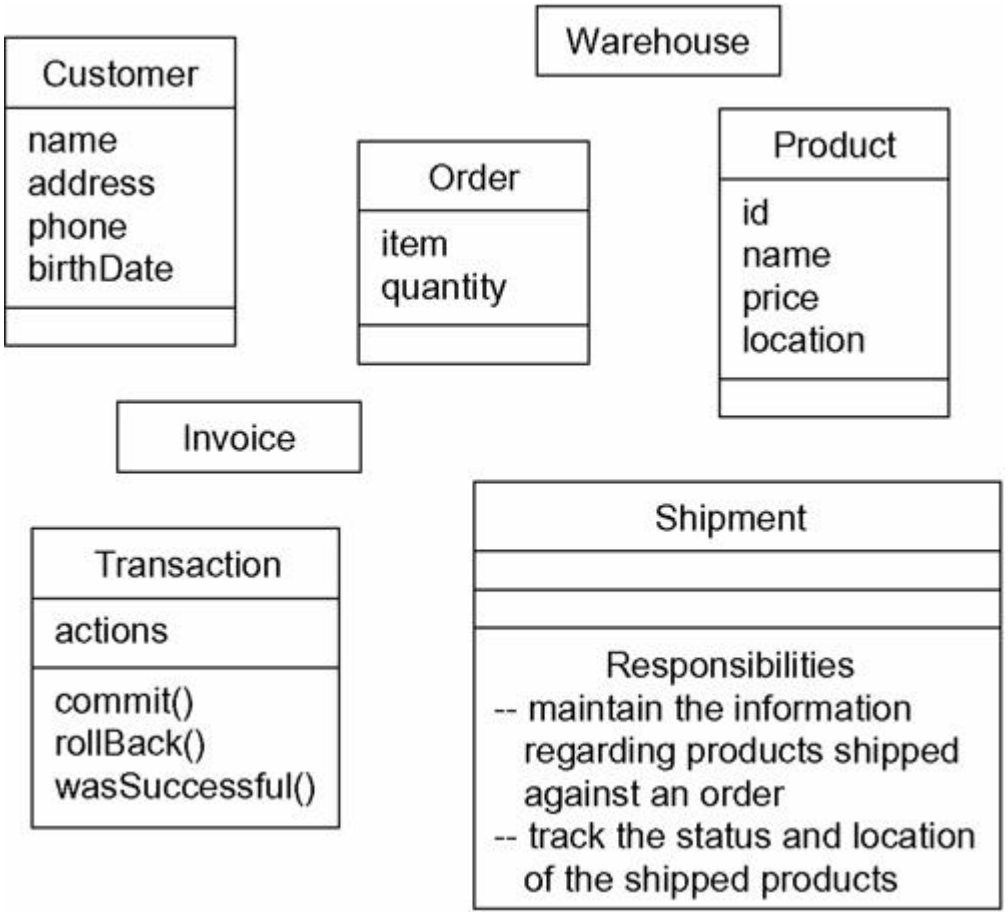
3. Common Modeling Techniques

- Modeling the Vocabulary of a System
- Modeling the Distribution of Responsibilities in a System
- Modeling Nonsoftware Things
- Modeling Primitive Types

3.1 Modeling the Vocabulary

- Identify those things that users or implementers use to describe the problem or solution.
- For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

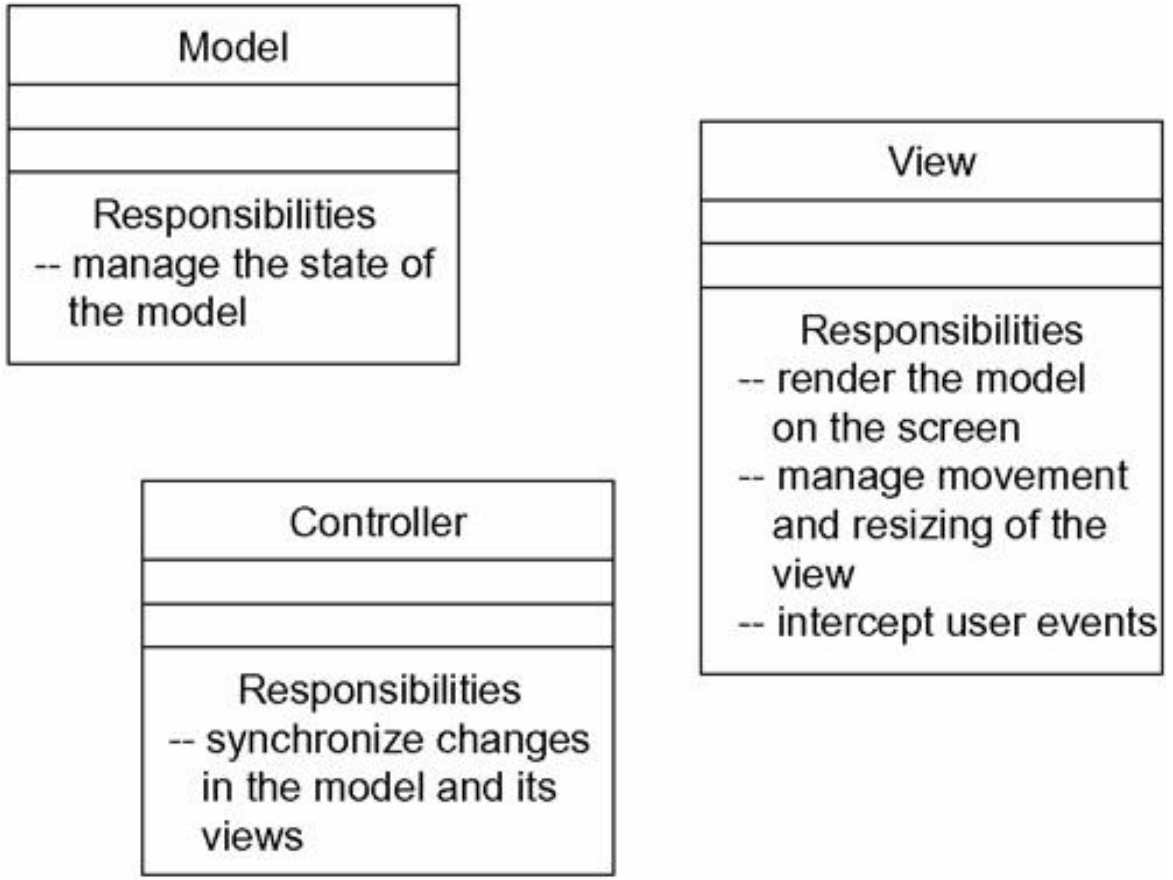
3.1 Modeling the Vocabulary (cont')



3.2 Modeling the Distribution of Responsibilities in a system

- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole
- Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities.

3.2 Modeling the Distribution of Responsibilities in a system (cont')



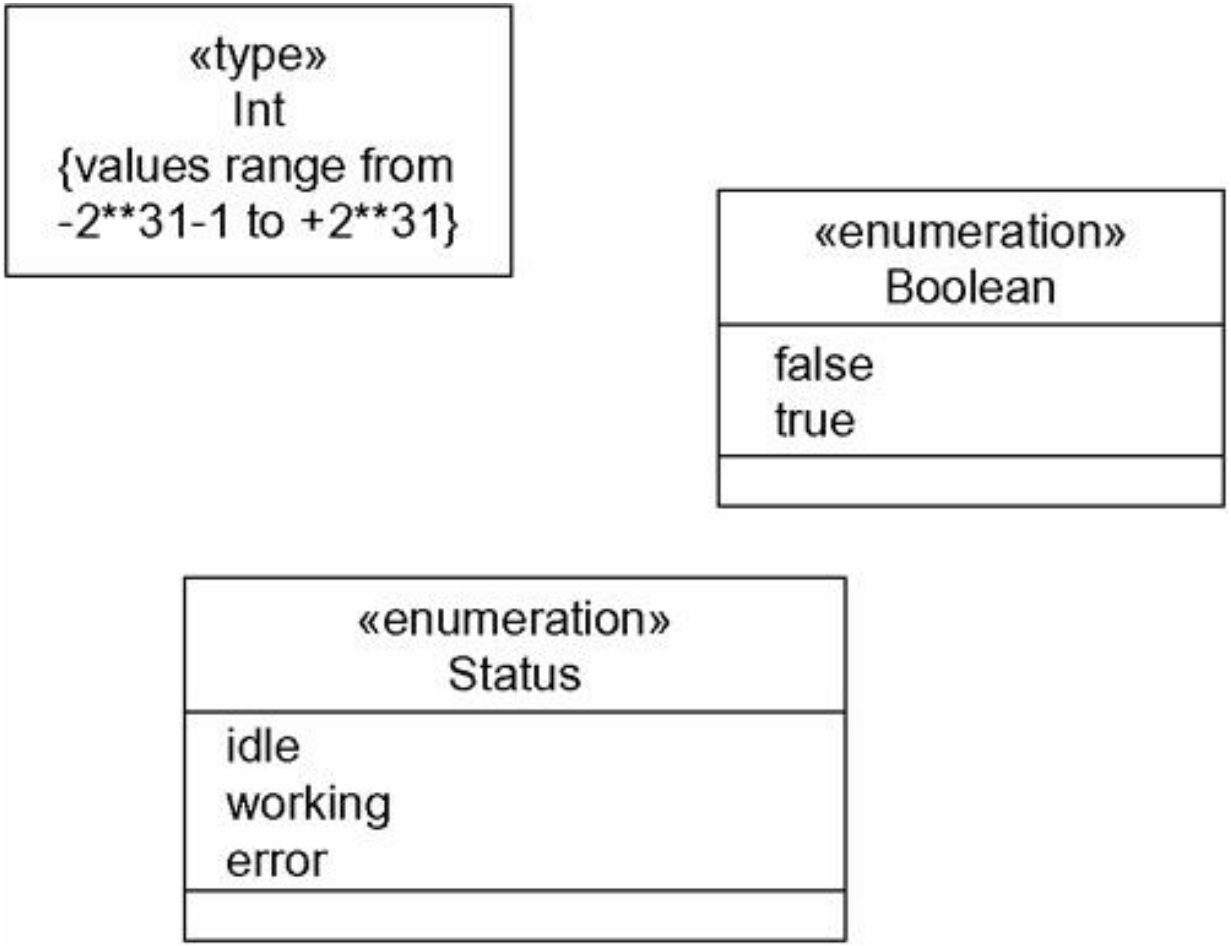
3.3 Modeling Nonsoftware Things

- Model the thing you are abstracting as a class.
- Using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node.

3.4 Modeling Primitive Types

- Model the thing you are abstracting as a type or an enumeration, which is rendered using class notation with the appropriate stereotype.
- If you need to specify the range of values associated with this type, use constraints.

3.4 Modeling Primitive Types (cont')



4. Relationships

- Very few of classes stand alone. Instead, most of them collaborate with others in a number of ways.
  - not only must identify the things that form the vocabulary of the system
  - Must also model how these things stand in relation to one another

4. Relationships (cont')

- In object-oriented modeling, there are three kinds of relationships that are especially important
  - *dependencies*, which represent using relationships among classes;
  - *generalizations*, which link generalized classes to their specializations;
  - *associations*, which represent structural relationships among objects.
- Each of these relationships provides a different way of combining your abstractions

4.2 dependency

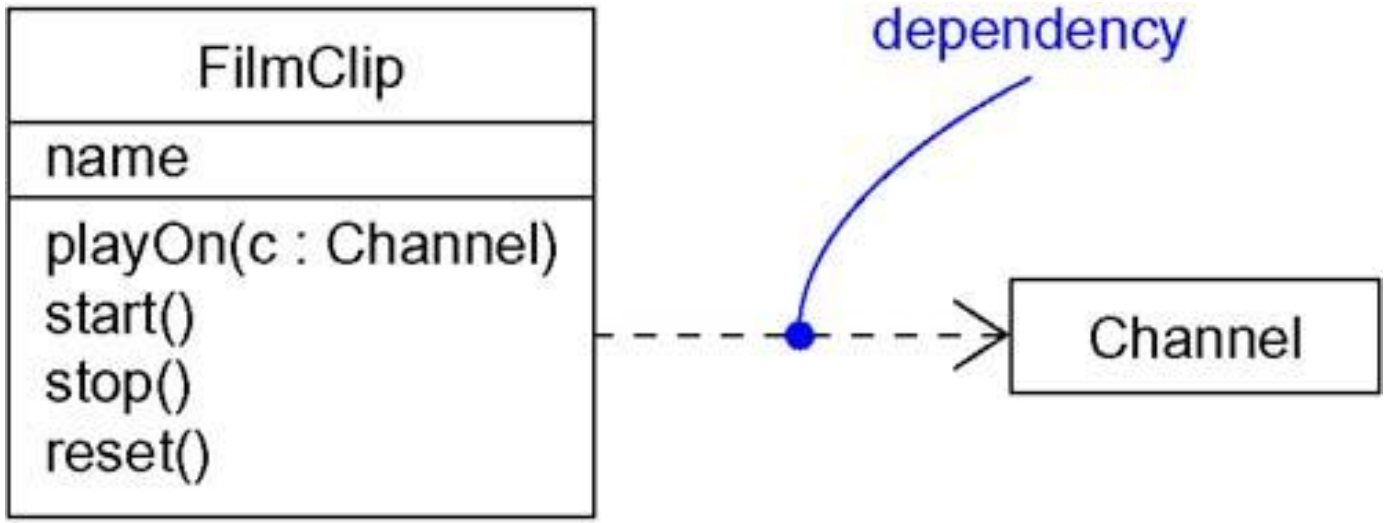
- A *dependency* is a using relationship that states that a change in specification of one thing (for example, class Event) may affect another thing that uses it (for example, class Window), but not necessarily the reverse.

4.2 dependency (cont’)

- Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Use dependencies when you want to show one thing using another.

4.2 dependency (cont')

- Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation



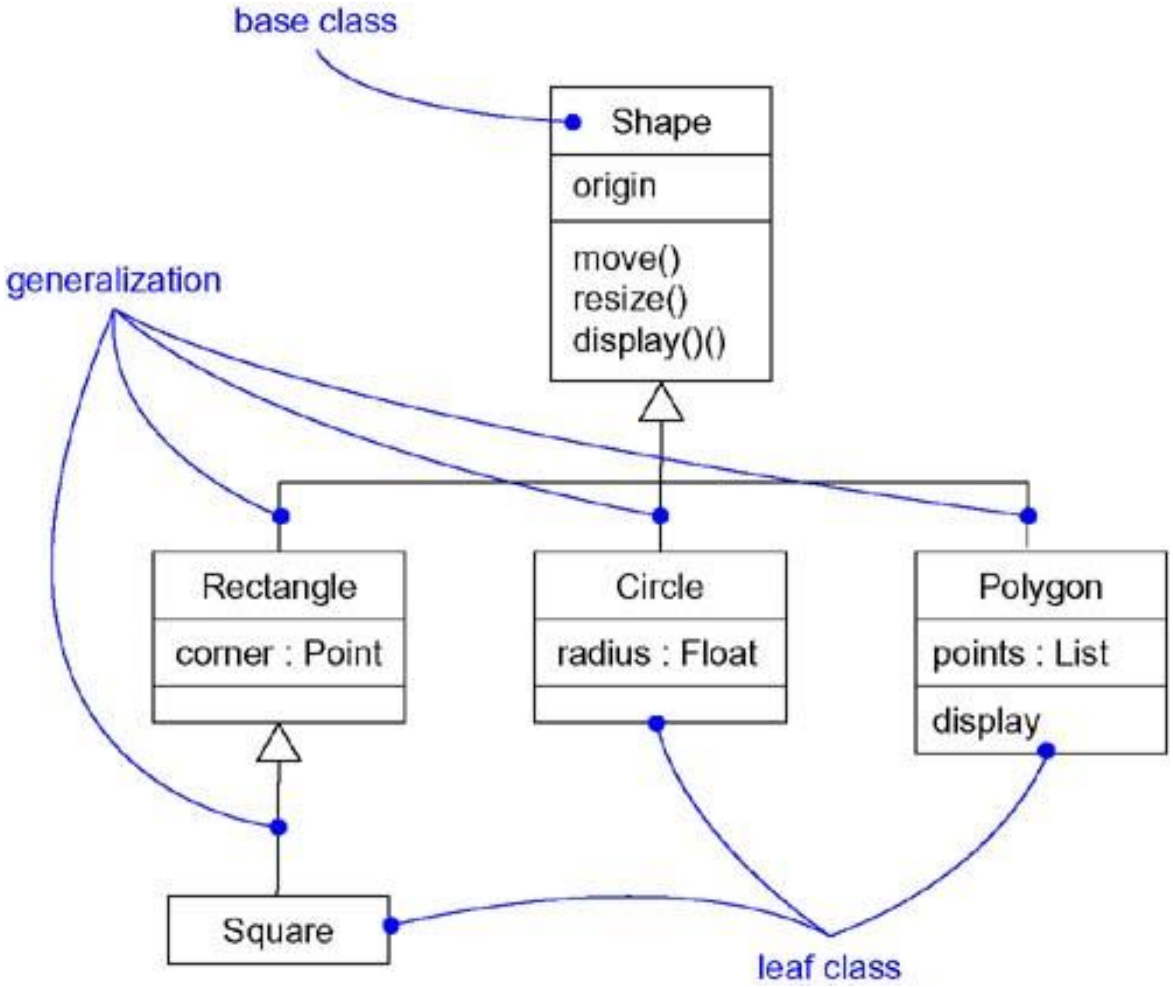
4.3 generalization

- A *generalization* is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing (called the subclass or child).
- Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window).

4.3 generalization (cont')

- Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent, as...
- A class may have zero, one, or more parents.
  - root class or base class： A class that has no parents and one or more children
  - leaf class： A class that has no children
  - single inheritance： A class that has exactly one parent
  - multiple inheritance： A class with more than one parent

4.3 generalization (cont')



4.4 association

- An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another.
- Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa.

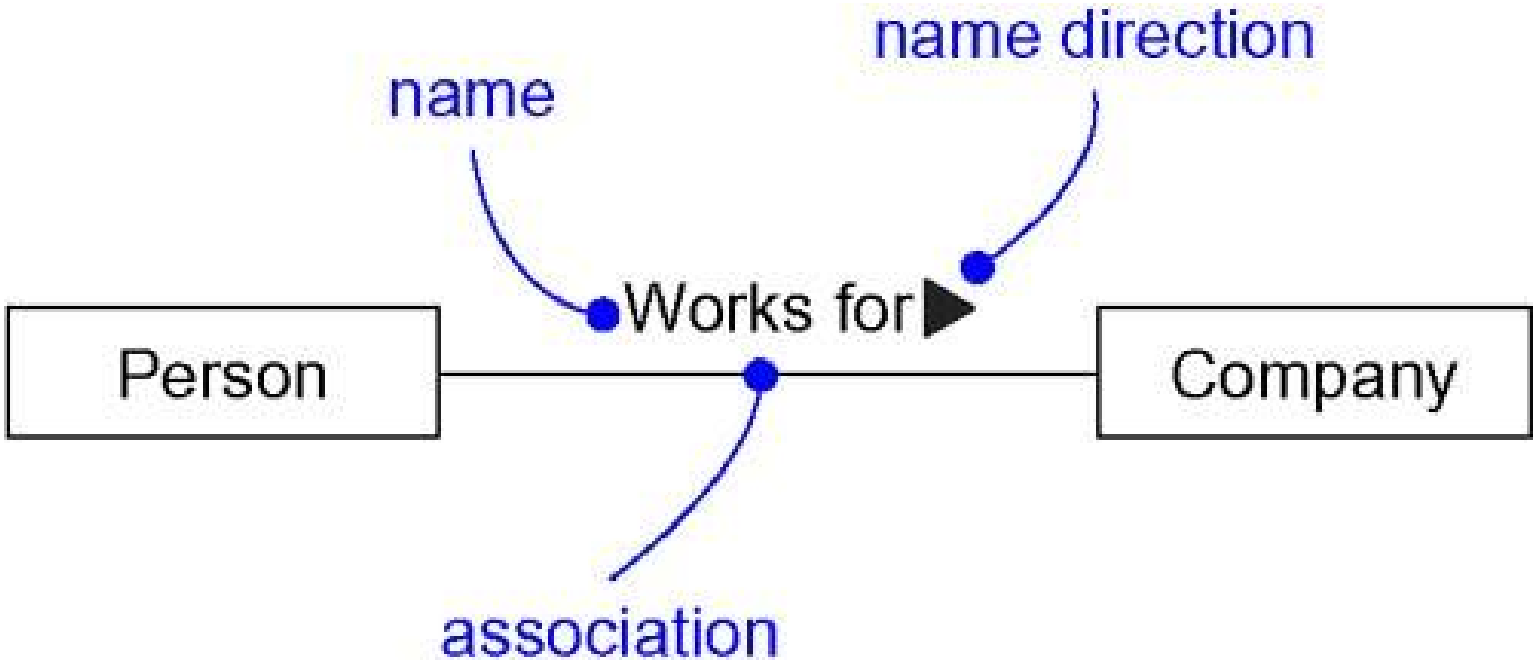
4.4 association (cont')

- Graphically, an association is rendered as a solid line connecting the same or different classes. Use associations when you want to show structural relationships.
- Beyond this basic form, there are four adornments that apply to associations.
  - Name / Role / Mulitplicity / Aggregation

4.4.1 name

- An association can have a name, and you use that name to describe the nature of the relationship.
- You can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name, as ...

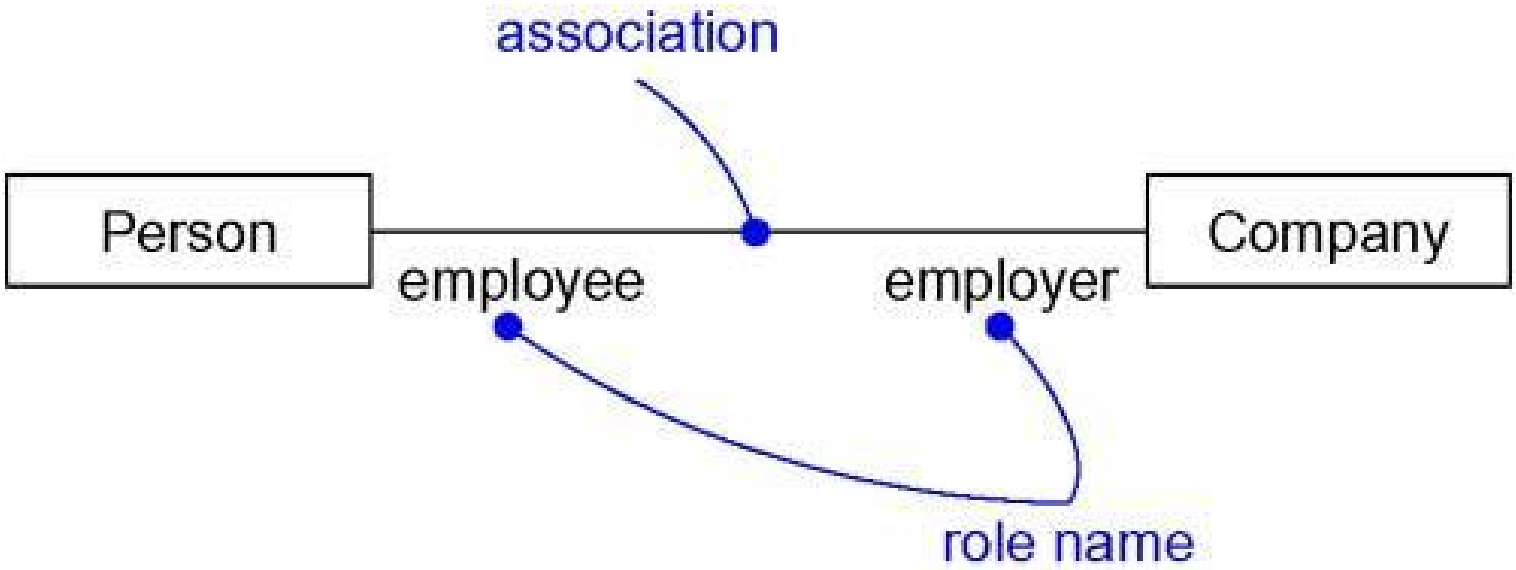
4.4.1 name (cont')



4.4.2 role

- When a class participates in an association, it has a specific role that it plays in that relationship;
- A role is just the face the class at the near end of the association presents to the class at the other end of the association.
- You can explicitly name the role a class plays in an association.

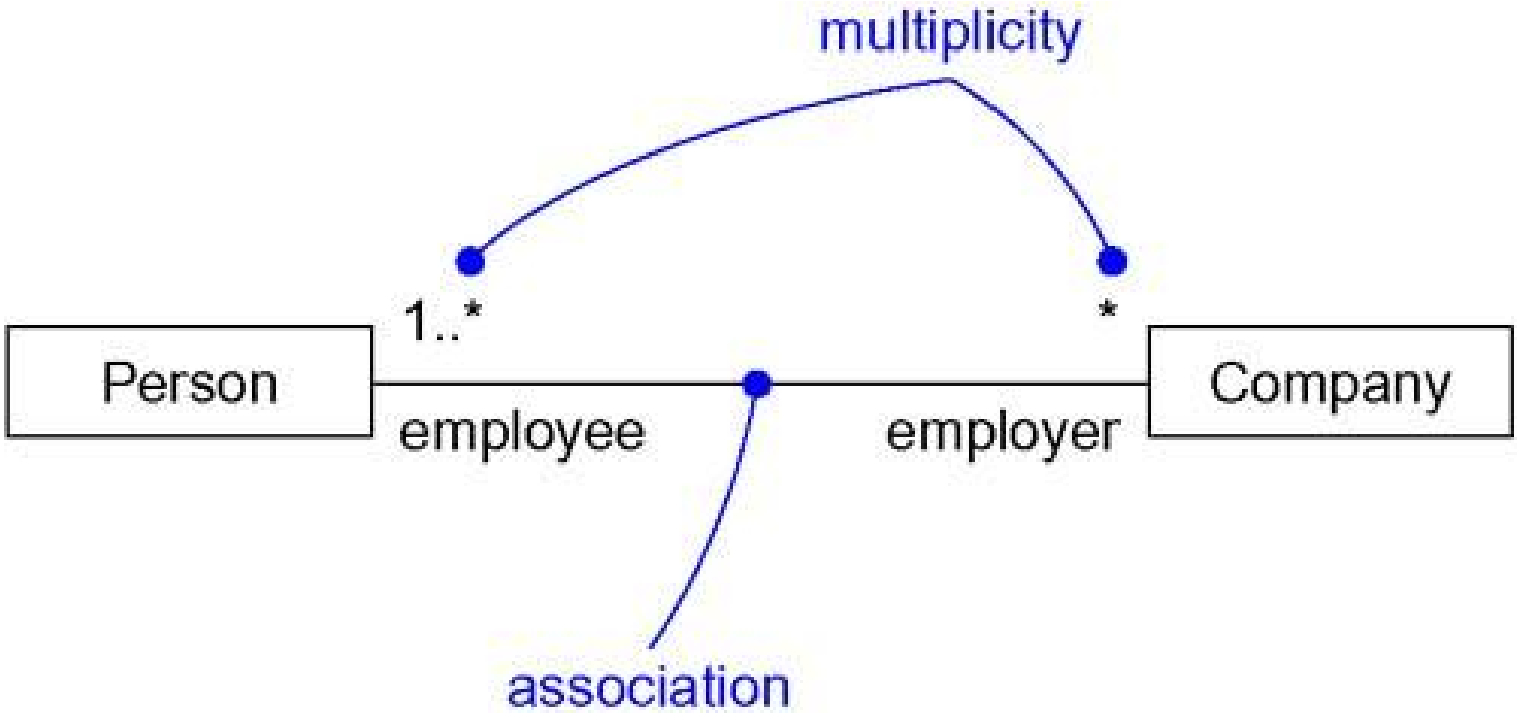
4.4.2 role (cont')



4.4.3 Multiplicity

- The multiplicity of a property is an indication of how many objects may fill the property.
- It is written as an expression that evaluates to a range of values or an explicit value as ...

4.4.3 Multiplicity (cont')



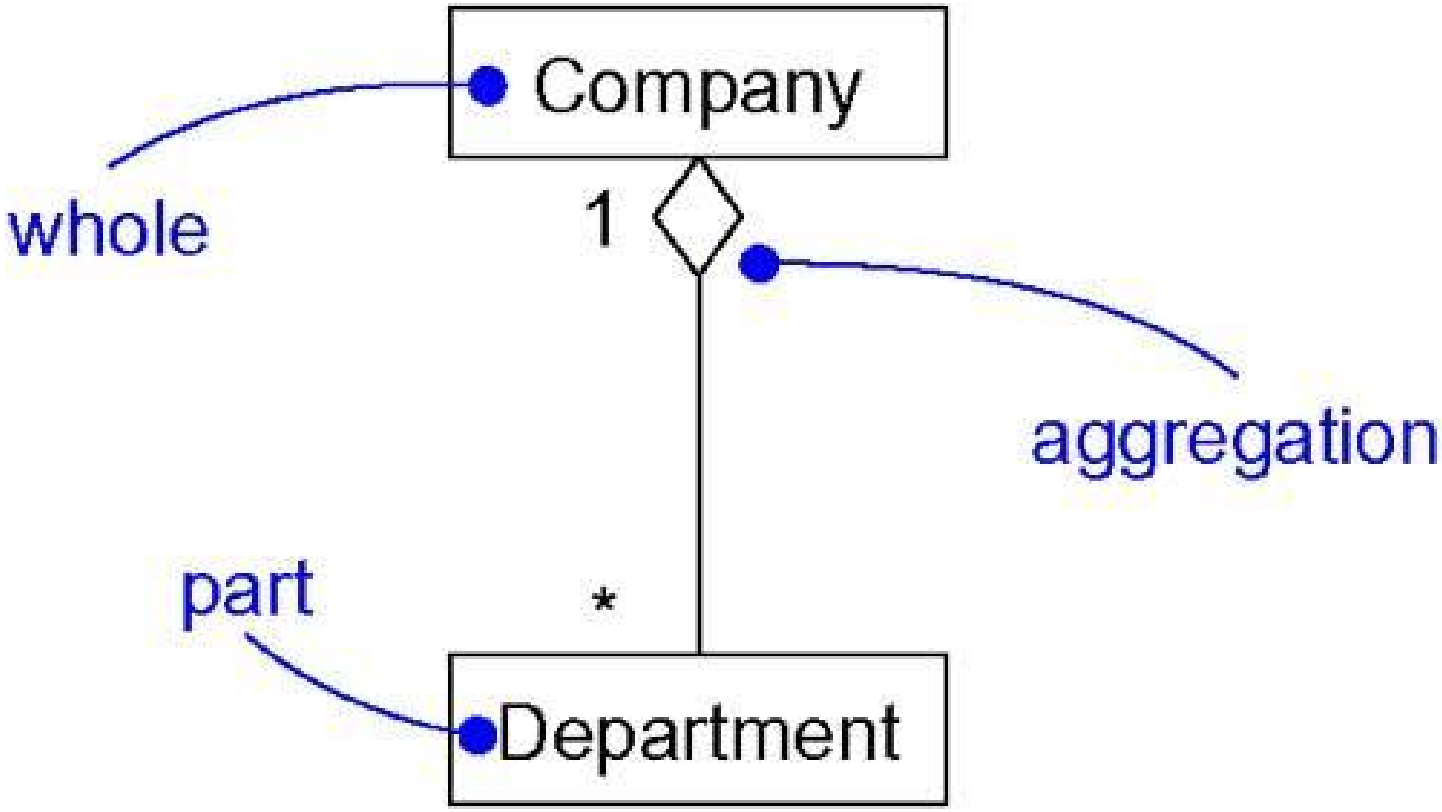
4.4.3 Multiplicity (cont')

- The most common multiplicities
  - 1 (An order must have exactly one customer.)
  - 0..1 (A corporate customer may or may not have a single sales rep.)
  - * (A customer need not place an Order and there is no upper limit to the number of Orders a Customer may place—zero or more orders.)
- More generally, multiplicities are defined with a lower bound and an upper bound, such as 2..4

4.4.4 aggregation

- Aggregation is “Whole/part” relationship
  - one class represents a larger thing (the "whole"),
  - It consists of smaller things (the "parts").
- It represents a "has-a" relationship
  - meaning that an object of the whole has objects of the part.
- It is specified by adorning a plain association with an open diamond at the whole end, as...

4.4.4 aggregation



5. Common Modeling Techniques

- Modeling Simple Dependencies
- Modeling Single Inheritance
- Modeling Structural Relationships

5.1 Modeling Simple Dependencies

- The most common kind of dependency relationship is the connection between a class that only uses another class as a parameter to an operation.
- To model this using relationship,
  - Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.

5.2 Modeling Single Inheritance

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these elements.
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.

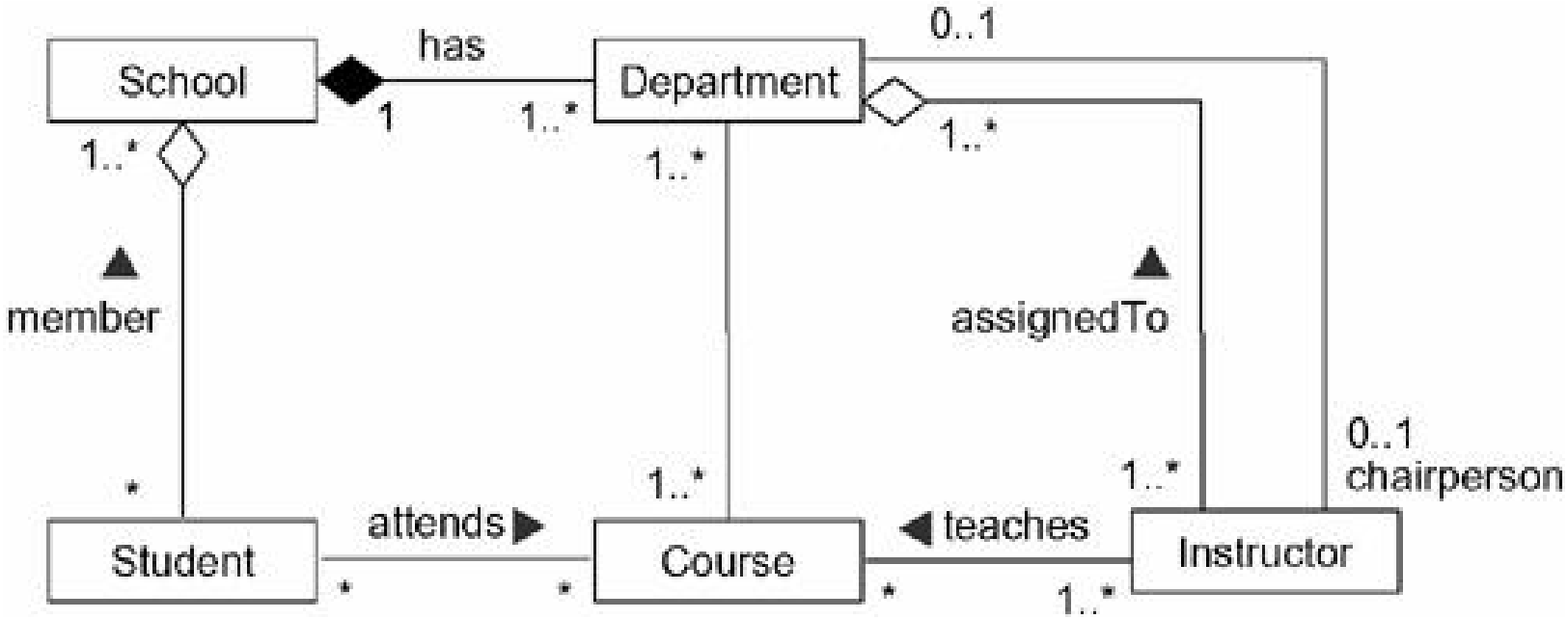
5.3 Modeling Structural Relationships

- For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of associations.
- For each pair of classes, if objects of one class need to interact with objects of the other class other than as parameters to an operation, specify an association between the two. This is more of a behavior-driven view of associations.

5.3 Modeling Structural Relationships (cont')

- For each of these associations, specify a multiplicity (especially when the multiplicity is not *, which is the default), as well as role names (especially if it helps to explain the model).
- If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole.

5.3 Modeling Structural Relationships (cont')



6. Hints and Tips

- When you model relationships in the UML,
  - Use dependencies only when the relationship you are modeling is not structural.
  - Use generalization only when you have an "is-a-kind-of" relationship; multiple inheritance can often be replaced with aggregation.
  - Beware of introducing cyclical generalization relationships.

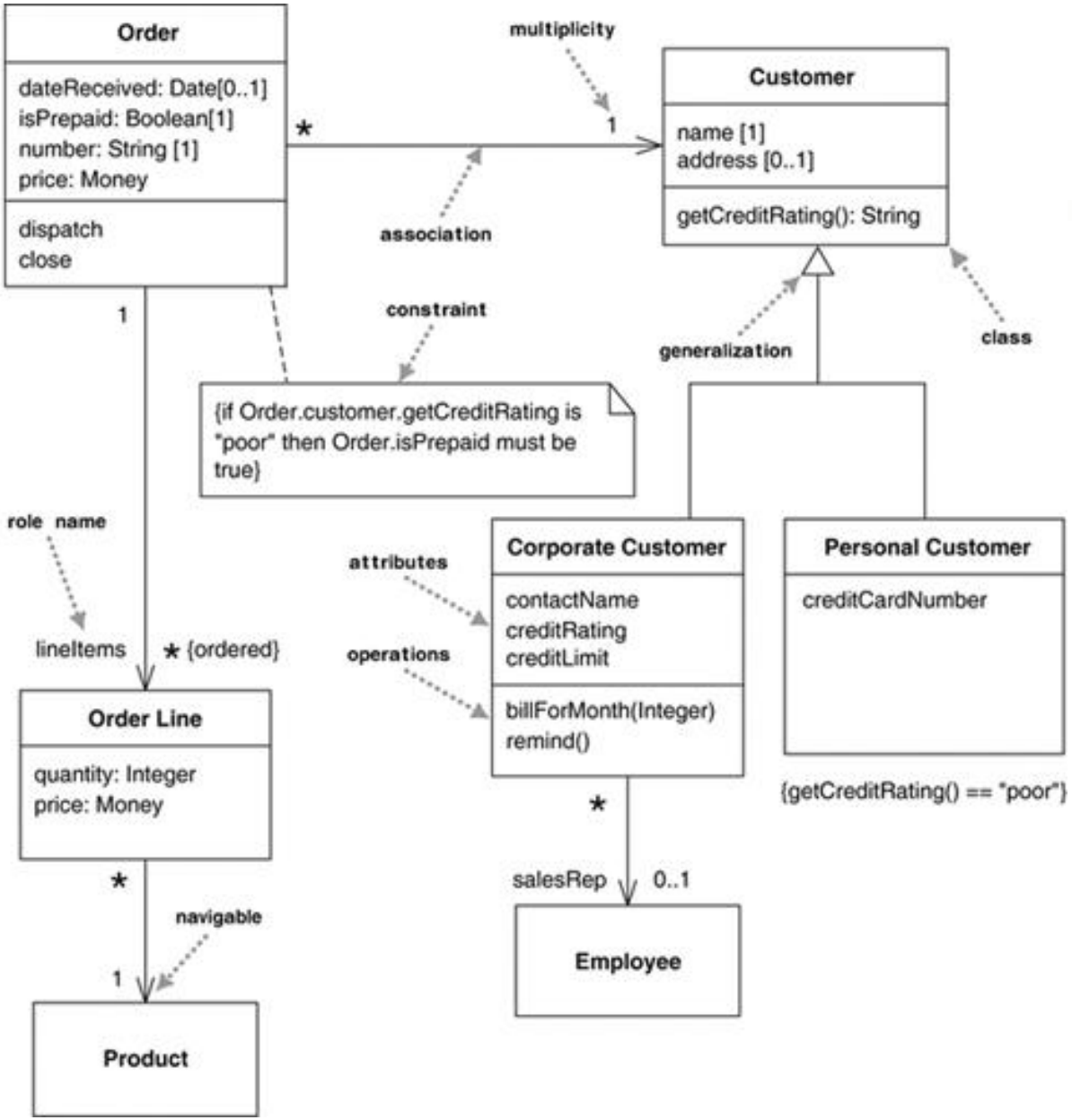
6. Hints and Tips (cont')

- Keep your generalization relationships generally balanced; inheritance latices should not be too deep (more than five levels or so should be questioned) nor too wide (instead, look for the possibility of intermediate abstract classes).
- Use associations primarily where there are structural relationships among objects.

6. Hints and Tips

- When you draw a relationship in the UML,
  - Use either rectilinear or oblique lines consistently.
  - Avoid lines that cross.
  - Show only those relationships that are necessary to understand a particular grouping of things. Superfluous relationships (especially redundant associations) should be elided.

ex.


Order
dateReceived: Date[0..1]
isPrepaid: Boolean[1]
number: String [1]
price: Money
dispatch
close
multiplicity
Customer
name [1]
address [0..1]
getCreditRating(): String
association
constraint
generalization
class
{if Order.customer.getCreditRating is "poor" then Order.isPrepaid must be true}
role name
lineItems
{ordered}
attributes
operations
Corporate Customer
contactName
creditRating
creditLimit
billForMonth(Integer)
remind()
Personal Customer
creditCardNumber
{getCreditRating() == "poor"}
Order Line
quantity: Integer
price: Money
navigable
salesRep
0..1
Employee
Product