

COMP 3111

SOFTWARE ENGINEERING

MODELING SOFTWARE SYSTEMS USING UML

LEARNING OBJECTIVES

1. Understand **what is the UML** and **how the UML can be used to model software systems**.
2. Appreciate that **the UML is a modeling language** and not a software development methodology.
3. Understand the **most basic components of UML class diagrams**: **class**, **association** and **generalization**.

MODELING SOFTWARE SYSTEMS USING UML:

OUTLINE

UML and Object-oriented Modeling

- Overview of the UML
- Object-oriented Modeling

Class

- Attribute
- Operation

Association

- Multiplicity
- Aggregation and Composition

Generalization

- Inheritance
- Coverage

Constraints

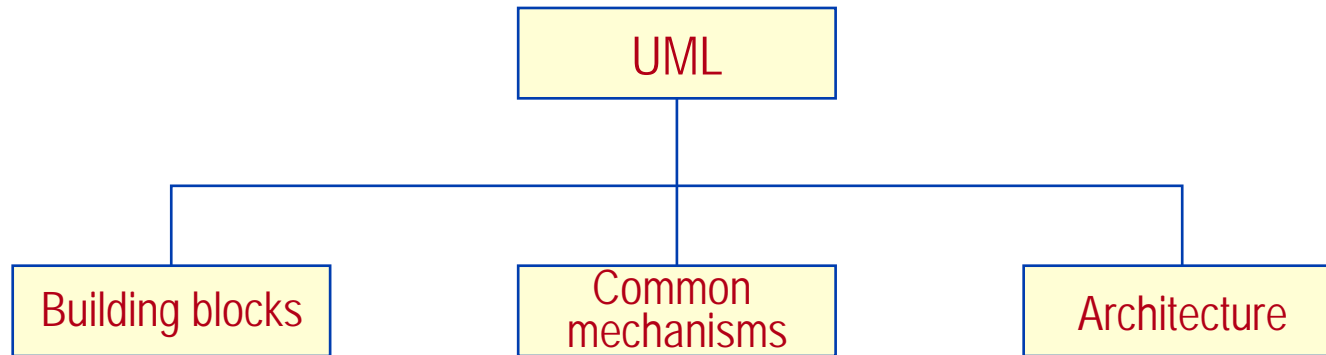
WHAT IS THE UML*?

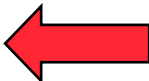
- General purpose *visual modeling language* for systems.
- *Incorporates current best practices* in OO modeling techniques.
- *Software development methodology/process neutral*.
- *Industry standard OO modeling language for modeling systems* (but can also be used for non-OO systems).

Basic Premise of the UML

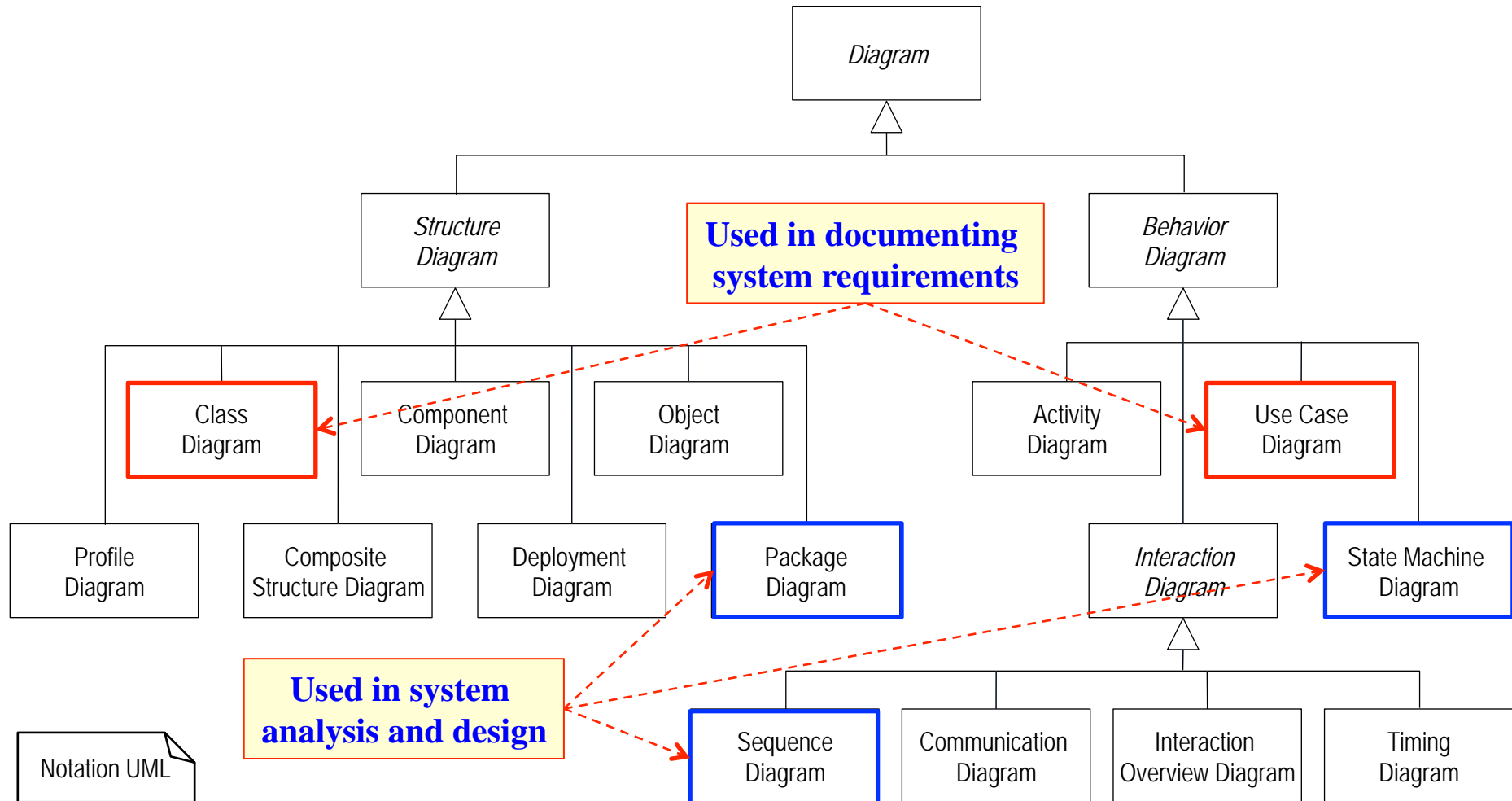
A software system can be modeled as
a collection of collaborating objects.

UML STRUCTURE

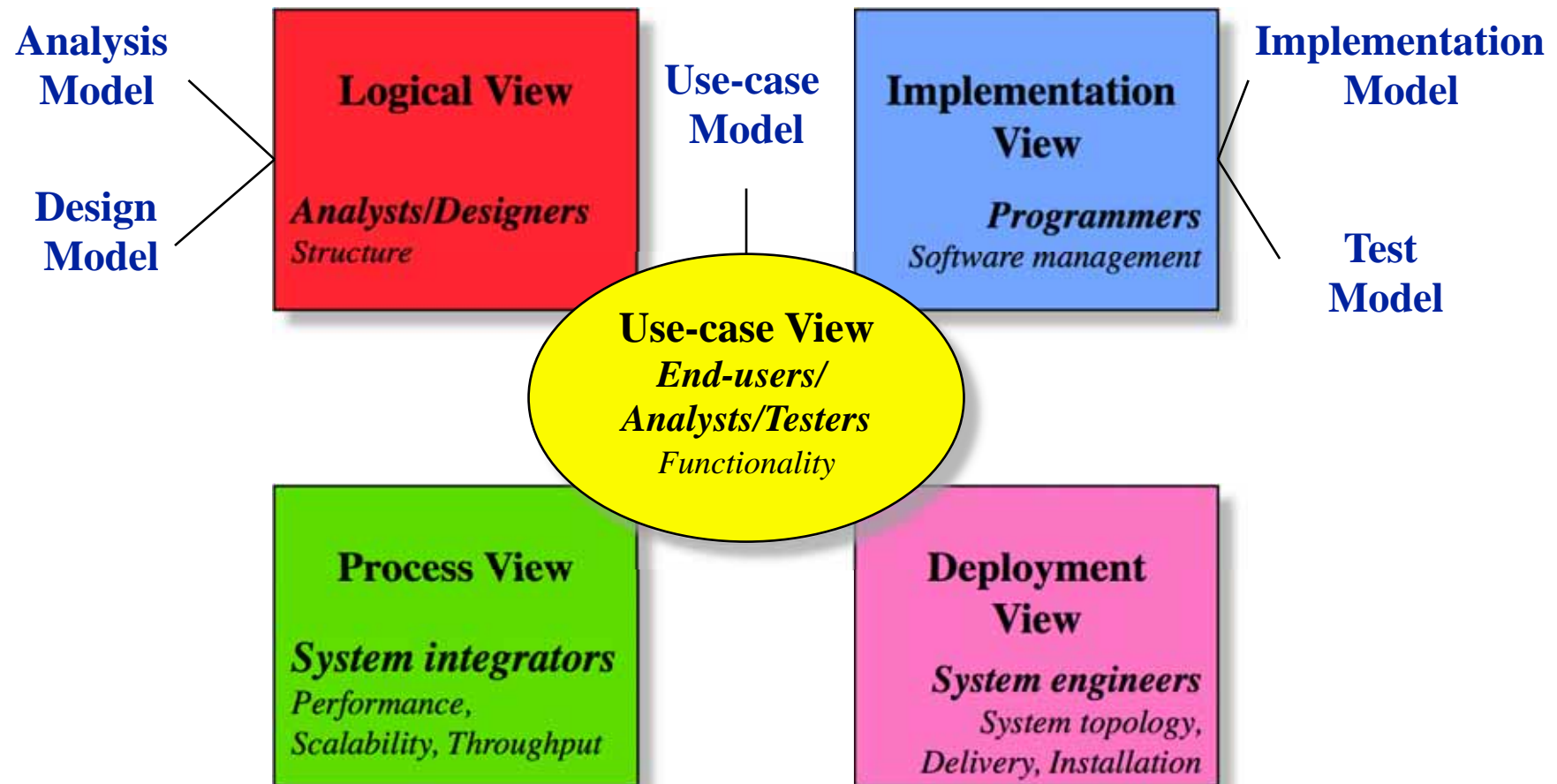


- **Building blocks** 
 - things
 - relationships
 - diagrams
- **Common mechanisms**
 - specifications
 - adornments
 - common divisions
 - extensibility mechanisms
- **Architecture**
 - use-case view
 - logical view
 - implementation view
 - process view
 - deployment view

UML DIAGRAMS



UML ARCHITECTURE



UML STRUCTURE (cont'd)

- **Building blocks** - basic UML modeling elements consisting of:
 - **things** are the modeling elements – classes, interfaces, use case, interactions, packages, notes. The modeling elements can represent structural, behavioural, grouping or annotational things.
 - **relationships** specify how two or more things are semantically related.
 - **diagrams** provide views into UML models *that show system structure or behaviour*. Diagrams show collections of things that visualize *what* the system will do or *how* it will do it.
- **Common mechanisms** - common ways of achieving specific goals:
 - **specifications**: textual descriptions of the *semantics of a modeling element*.
 - **adornments**: additional, visual details about modeling elements.
 - **common divisions**: particular *ways of thinking about the world* (e.g., classifier-instance; interface-implementation).
 - **extensibility mechanisms**: ways to *customize the modeling elements* for specific needs.

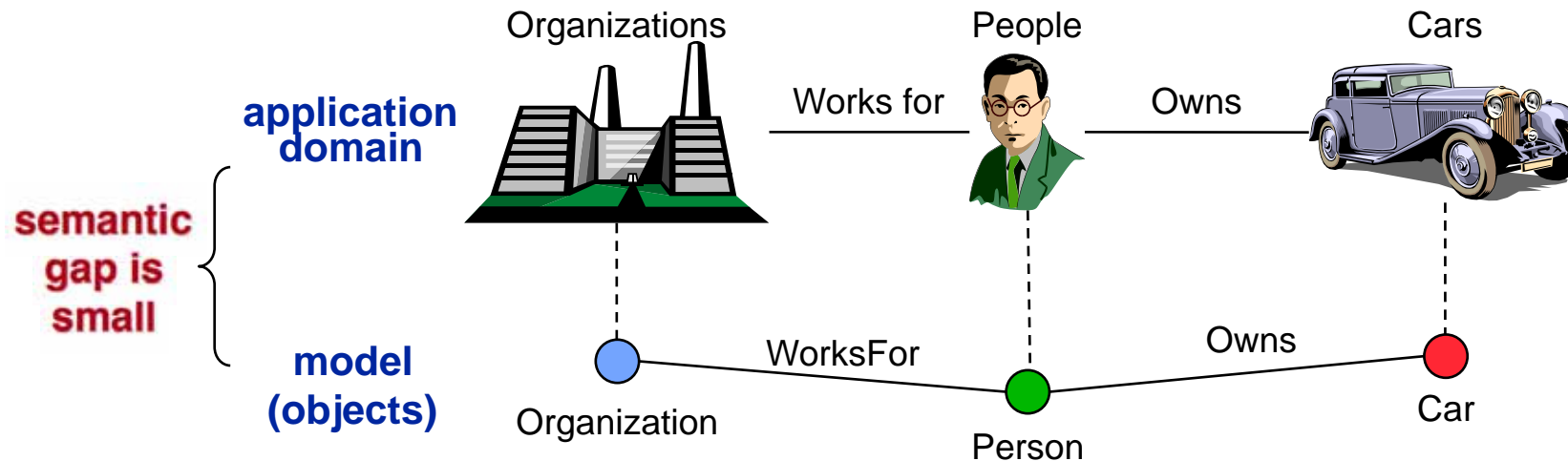
UML STRUCTURE (cont'd)

- **Architecture** - the **strategic aspects** of the **high-level structure** of the system that is captured by five views:
 - **use-case view** **captures the basic requirements for the system** and provides the basis for the construction of the other views.
 - of interest to end-users, analysts and testers
 - **logical view** **captures the vocabulary of the application domain** as a set of classes and objects and their interaction.
 - of interest to analysts and designers
 - **implementation view** **models the files and physical components** that make up the physical code base of the system (executable files, libraries of code, databases, etc.).
 - used by programmers
 - **process view** **deals with issues of concurrency** within the system by **modeling the executable threads and processes** in the system.
 - of interest to system integrators
 - **deployment view** **models the physical deployment of components** onto computational nodes such as computers and peripherals.
 - of interest to system engineers

WHY BUILD MODELS?

- Models **succinctly describe reality** (i.e., they _____).
 - They **show essential details** and **filter out non-essential details**.
- For software development, this allows us to **focus on the “big picture”**,
 - i.e., **programming-in-the-large**.
- Such a focus allows us to better **deal with the _____ of software development**,
 - i.e., with **human limitations in understanding** complex things.
- The result is **better understanding** of requirements, **cleaner designs**, and more **maintainable systems**.

WHY OBJECT-ORIENTED MODELING?



- ✎ Allows **direct representation of “things”** in an **application domain**.
- ✎ **Reduces the “semantic gap”** between the application domain and the model.
- ✎ Better represents **how people think about reality**.

An application domain is modeled as a collection of objects.

OO MODELING & LEVELS OF ABSTRACTION

Requirements level → We construct a _____ *model*.

- We do not consider any aspects of the implementation of objects.

 **Focus: identifying objects (concepts) in the application domain.**

Analysis & Design level → We construct a _____ *model*.

- We consider interfaces of objects (but no internal aspects).

 **Focus: how objects interact in the solution.**

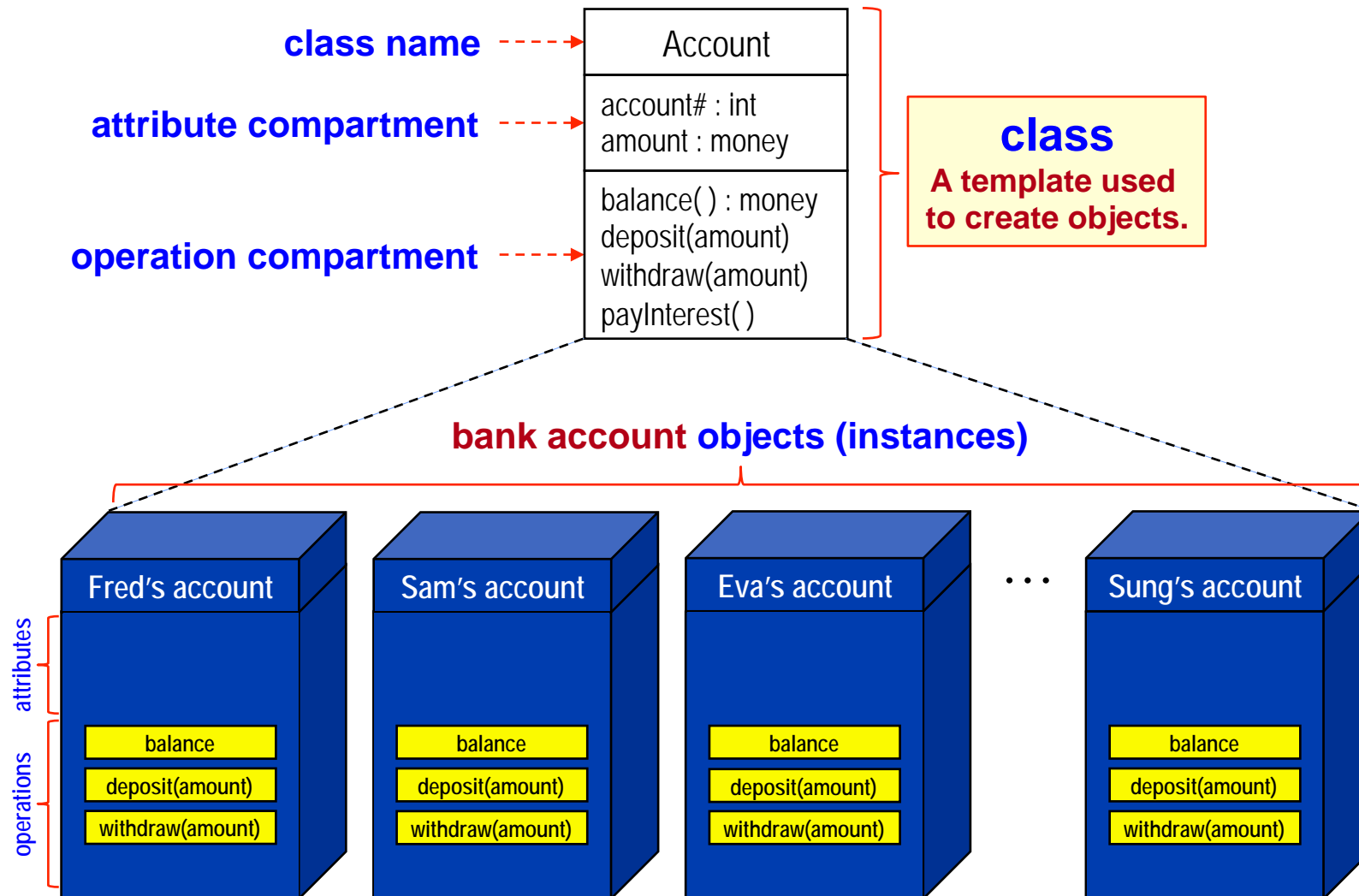
Implementation level → We implement the _____ *model*.

- We consider all details of objects (external and internal).

 **Focus: how to code objects.**

The same OO concepts can be used at all levels to build models.

CLASS



CLASS

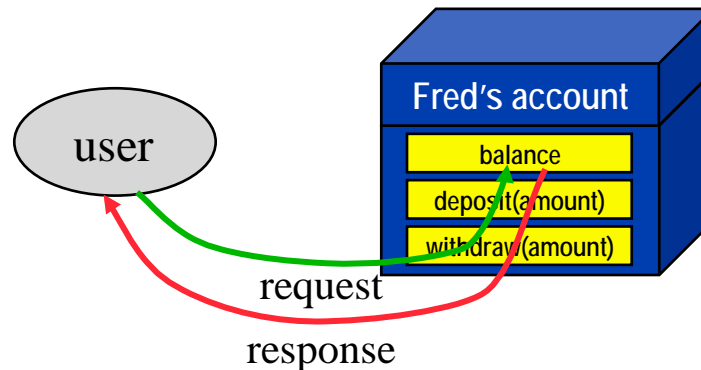
A **class** describes a **collection of objects** having common:

– semantics – attributes – operations – relationships

☞ A **class** is a **classifier**; an **object** is an **instance**.

- A class is a “**factory**” for creating objects.
- A good class should capture **one and only one abstraction**.
 - ☞ It should have **one major theme**.
- A class should be named using the **vocabulary** of the **application domain** (class names must be unique).
 - ☞ So that **it is meaningful and traceable** from the application domain to the model.

OBJECTS PROVIDE INFORMATION HIDING



- An object's interface **abstracts** and **encapsulates** the object providing _____.

✎ An object provides a **visible** (*public*) interface, but has a **hidden** (*private*) implementation.

The *information hiding* provided by objects promotes modular development.

CLASS: ATTRIBUTE

An **attribute** describes the **data values** held by objects in a class.

- Attribute properties:

- name: unique within a class, but not across classes.
- type: the domain of values – string, integer, money, etc.
- visibility: who can access the attribute's values.
public (+), private (–), protected (#), package (~)
- initial value [optional]: the attribute's initial value.
- multiplicity [optional]: the number of simultaneous values.
- changeability: whether the value can be changed.
unspecified (default) readOnly

For modeling,
name and type
should always
be specified.

Account
account# : int amount : money
balance() : money deposit(amount) withdraw(amount) payInterest()

CLASS: OPERATION

An **operation** describes a function or transformation that may be applied to or by objects in a class.

- Operation properties:

- operation signature

operation name
parameter names
result type

For modeling, all
should always be
specified.

- visibility

public (+), private (–), protected (#), package (~)

Account
account# : int amount : money
balance() : money deposit(amount) withdraw(amount) payInterest()

- An operation instance (its implementation) is called a **method**.

✎ An operation can have several methods that implement it (polymorphic operation).

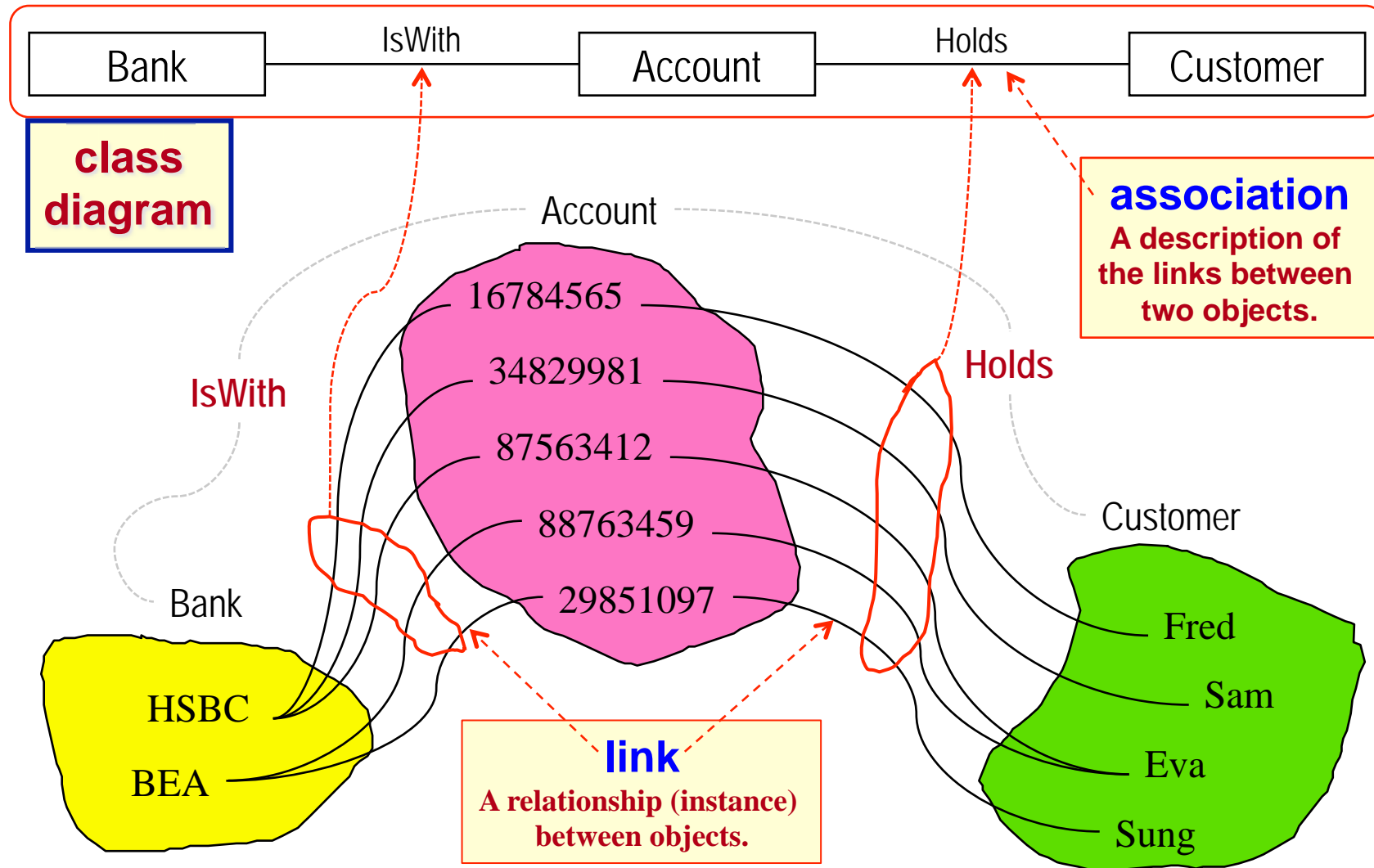
WHY CLASSES FOR MODELING SYSTEMS?

By abstracting a collection of objects and representing them as a class, *the complexity of developing a system is reduced* since it becomes easier to:

- **understand** the system → We need to understand only the classes, not the individual objects.
- **specify** the system → Classes provide a place to define and store common definitions only once.

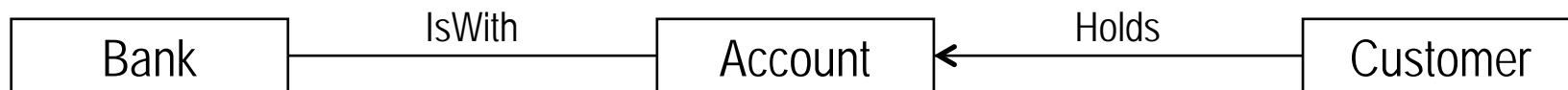
**Choosing appropriate classes is an
IMPORTANT DESIGN DECISION
that helps promote modular development.**

ASSOCIATION



ASSOCIATION

An *association* describes a collection of links with common semantics.



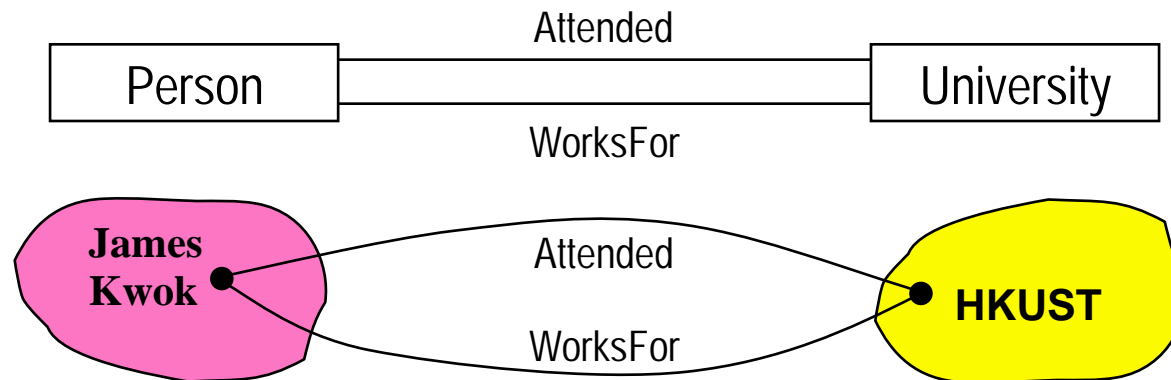
✎ An *association* is a *classifier*; a *link* is an *instance*.

✎ Conceptually, associations are inherently bi-directional.

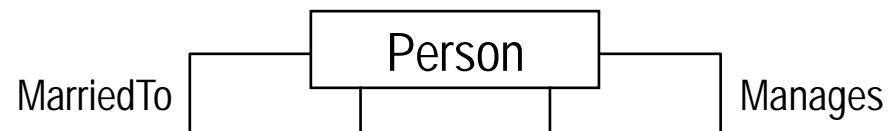
✎ Can show *navigability* of associations with an arrowhead.
(Implies that the source object has a reference to the target object.)

ASSOCIATIONS AND CLASSES

- Two different classes can be related by several associations.



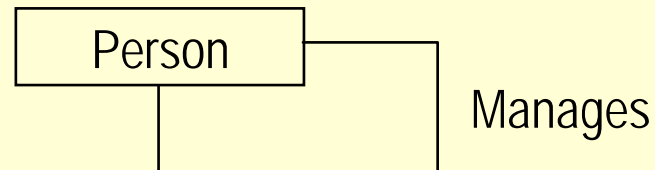
- The same class can be related by several associations.



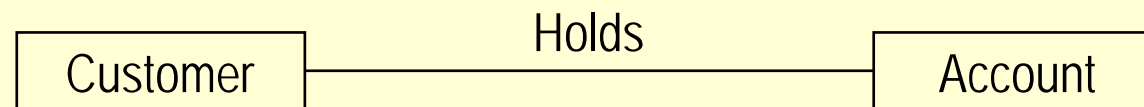
👉 The collection of class and association names **must be unique**.

ASSOCIATION: DEGREE

- **unary** (reflexive)
relates a class to itself

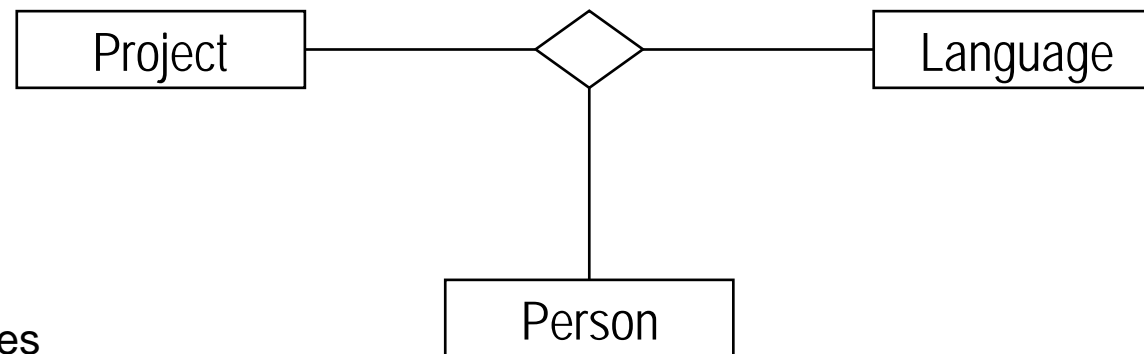


- **binary**
relates two classes



We will use only unary and binary associations in this course.

- **ternary**
relates three classes

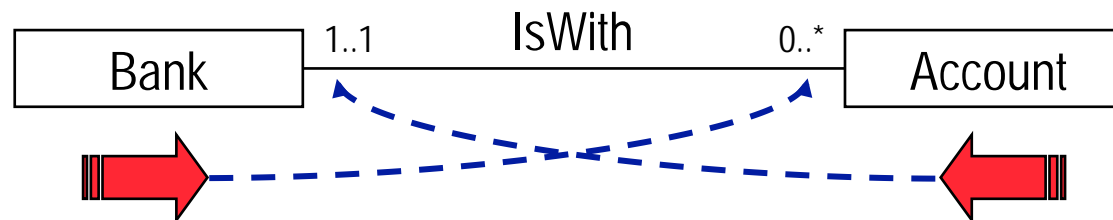


- **n-ary**
relates any number of classes

In practice, the majority of associations are binary!

ASSOCIATION: MULTIPLICITY

Multiplicity specifies a restriction on the number of objects in a class that may be related to an object in another class.



For **a given bank**, how many accounts can it have?

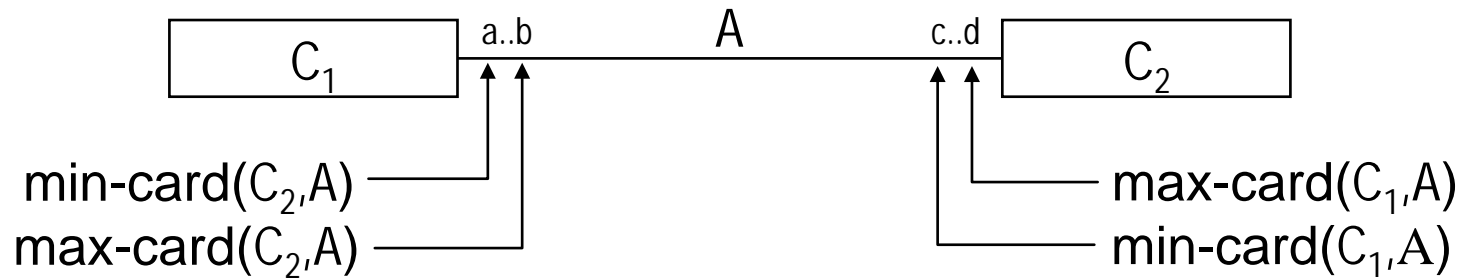
☞ A bank may have no accounts or it may have many accounts.

For **a given account**, how many banks can it be with?

☞ An account must be with exactly one bank.

Multiplicity is an *application domain constraint*!

ASSOCIATION: MULTIPLICITY (cont'd)



minimum cardinality (min-card)

$\min\text{-card}(C_1, A)$: the *minimum number of links* in which *each object* of C_1 can participate in association A

$\min\text{-card}(C_1, A) = 0 \rightarrow$ optional participation (*may not be related*)

$\min\text{-card}(C_1, A) > 0 \rightarrow$ mandatory participation (*must be related*)

maximum cardinality (max-card)

$\max\text{-card}(C_1, A)$: the *maximum number of links* in which *each object* of C_1 can participate in association A

ASSOCIATION: MULTIPLICITY (cont'd)



special cardinalities:

max-card = * → an unlimited upper bound (∞)

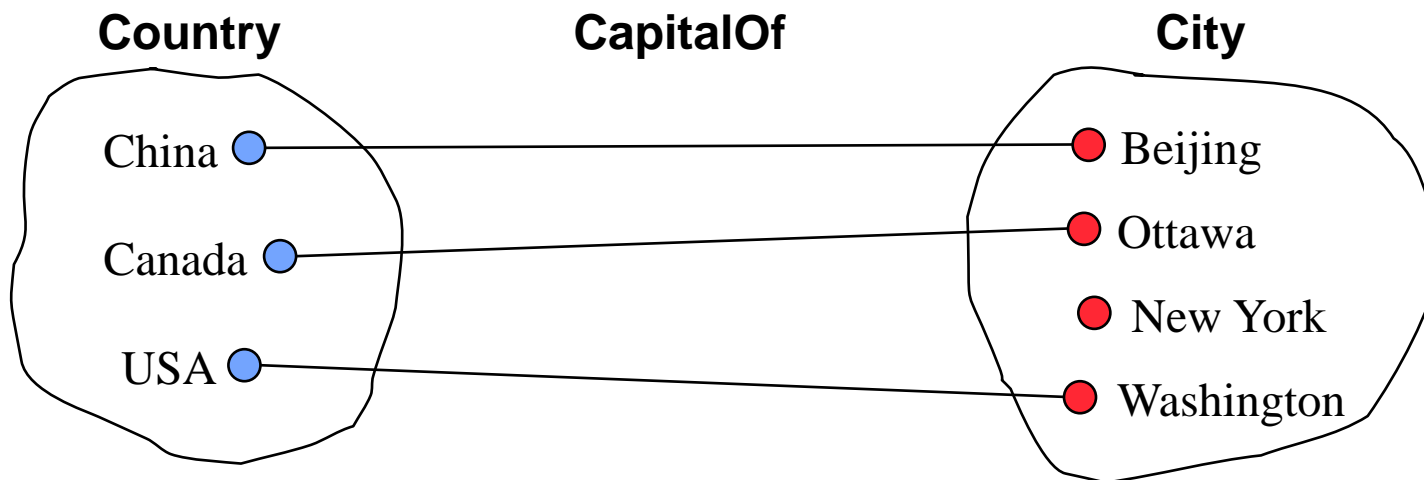
min-card = 1 and max-card = 1 → can use 1 by itself

min-card = 0 and max-card = * → can use * by itself

ASSOCIATION: MULTIPLICITY (cont'd)

$\text{max-card}(C_1, A) = 1$ and $\text{max-card}(C_2, A) = 1$

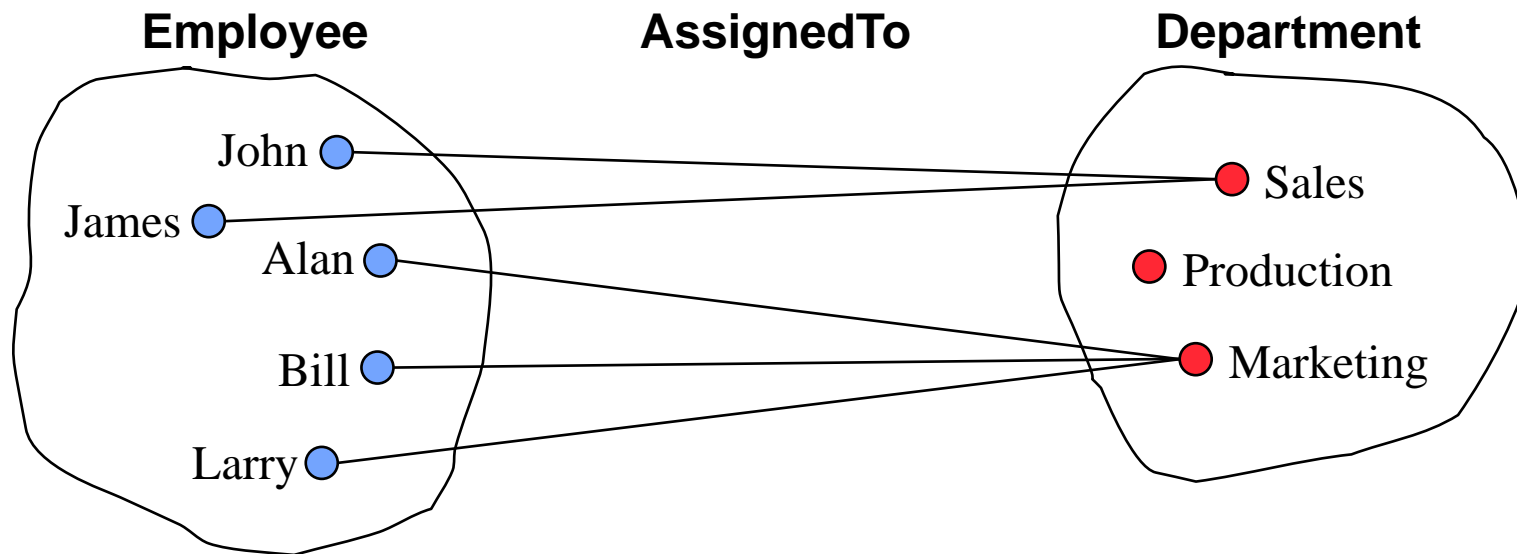
☞ one-to-one association (1:1)



ASSOCIATION: MULTIPLICITY (cont'd)

$\text{max-card}(C_1, A) = 1$ and $\text{max-card}(C_2, A) = *$

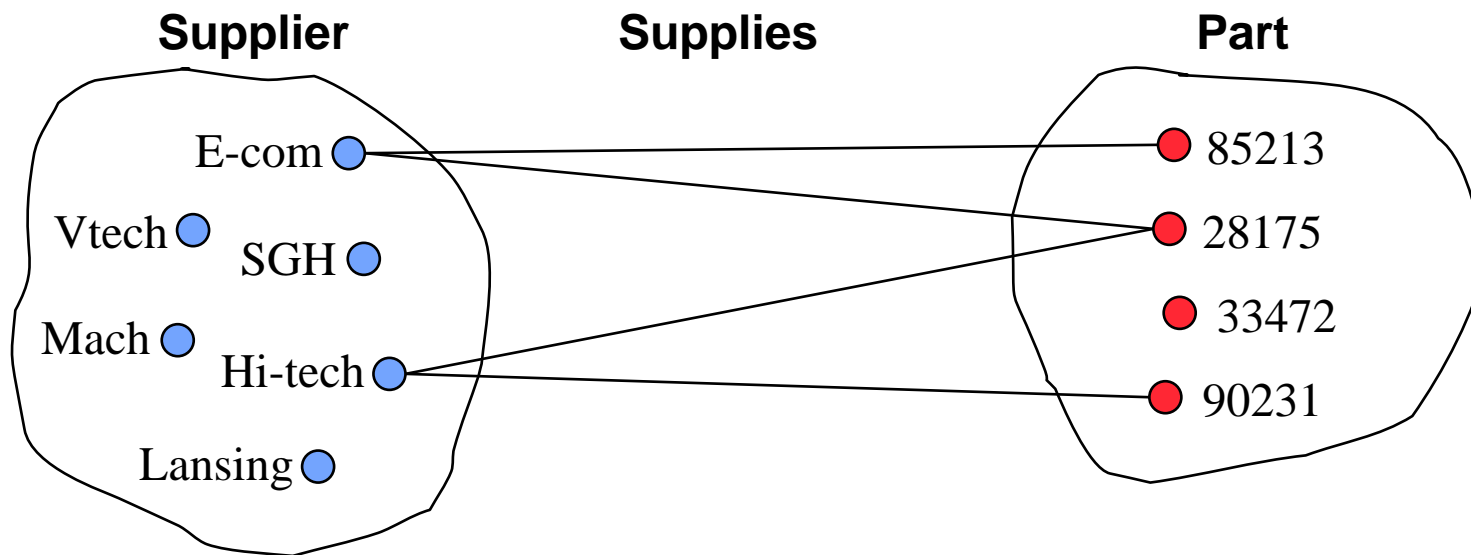
☞ one-to-many association (1:N)



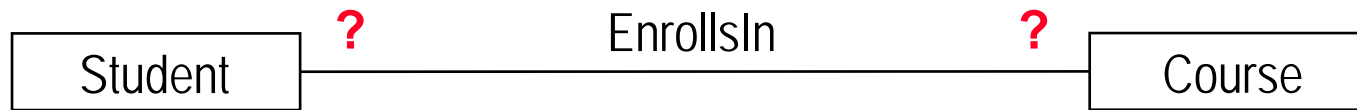
ASSOCIATION: MULTIPLICITY (cont'd)

$\text{max-card}(C_1, A) = *$ and $\text{max-card}(C_2, A) = *$

👉 many to many association (N:M)



QUESTION?



- A student must enroll in at least one course and can enroll in at most five courses
- A course must have at least ten students enrolled in it and cannot have more than forty-five students enrolled in it.



EXAMPLE: CAR OWNERSHIP AND LOANS

These classes are required to represent information about car ownership and car loans. However, they have some attributes that are **internal object identifiers (OIDs)** that are used to represent relationships and that **should not appear** in a class diagram. All such attributes conveniently have names ending in ID. 😊

Persons or companies may own cars. The car owner ID represents either the person or company that owns the car. A car may have only one owner (person or company). A car may have no loan or multiple loans. A loan is provided by a bank to a person or a company for the purchase of a car. Only the car owner may obtain a loan on the car. The car owner type and the loan customer type indicate whether the car owner/loan holder is a person or company.

Construct a class diagram in which all OIDs are replaced with UML associations. Show the most likely multiplicities for all associations and the final attributes for each class.

Note: *Your final class diagram should contain no OIDs.*

EXAMPLE: CAR OWNERSHIP AND LOANS

Person
name age address

Car
ownerID ownerType model year

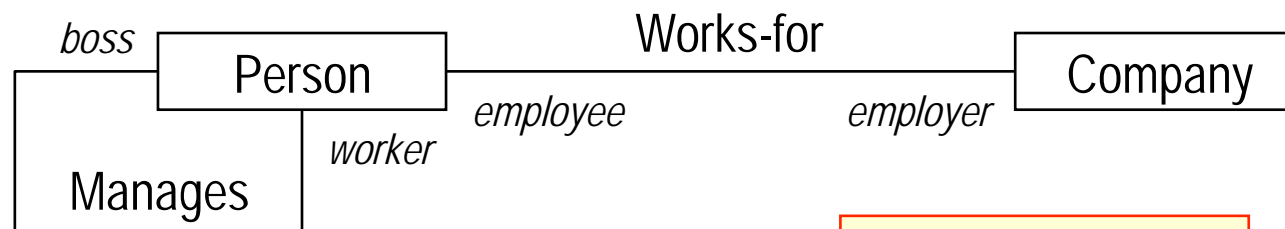
Company
name

Loan
vehilceID customerType customerID accountNumber bankID interestRate currentBalance

Bank
name

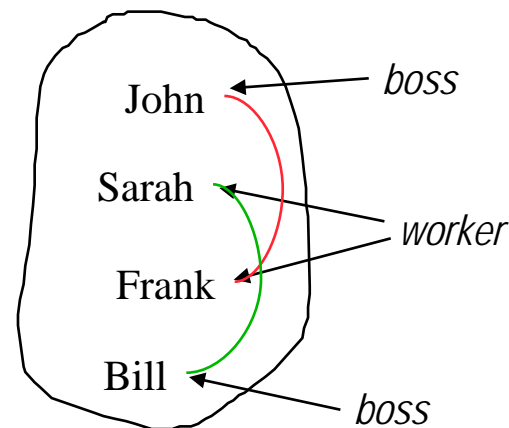
ASSOCIATION: ROLE

A **role** is one end of an association.



For unary and binary associations there are two roles.

Who manages who?

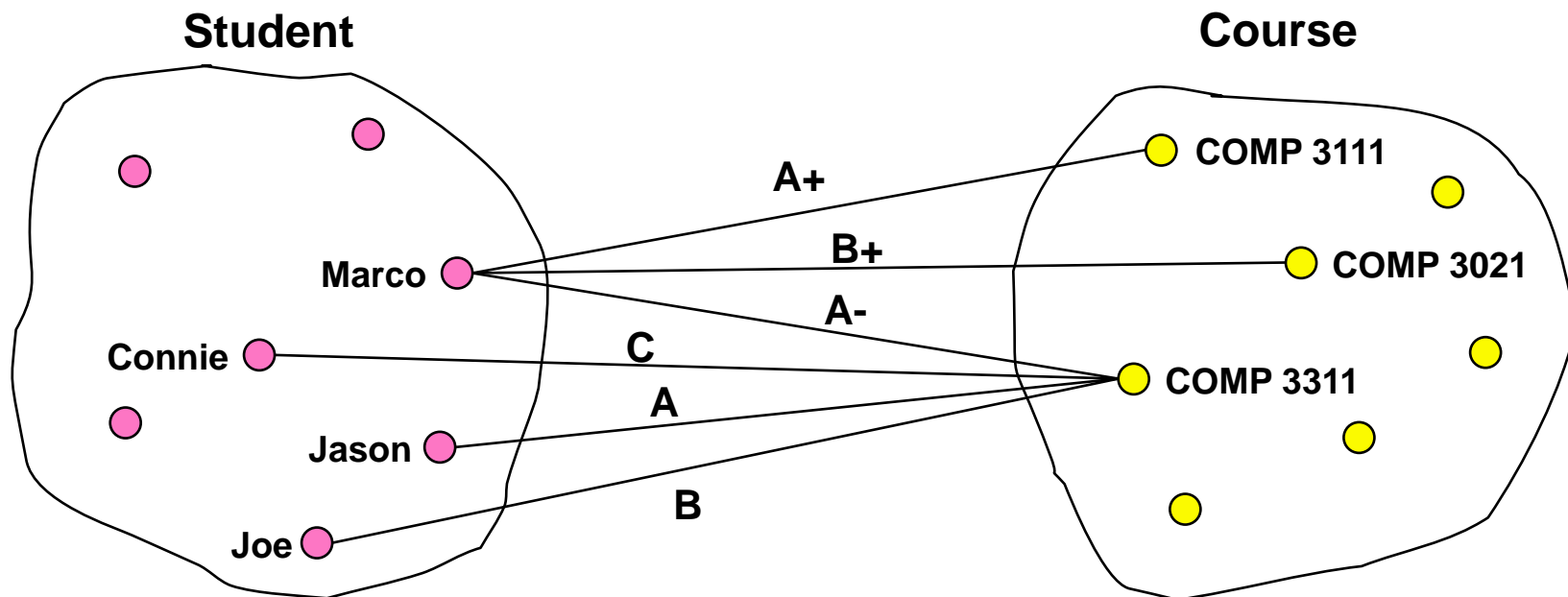


It is **necessary to use role names** when an association relates objects from the same class.

ASSOCIATION: ASSOCIATION CLASS



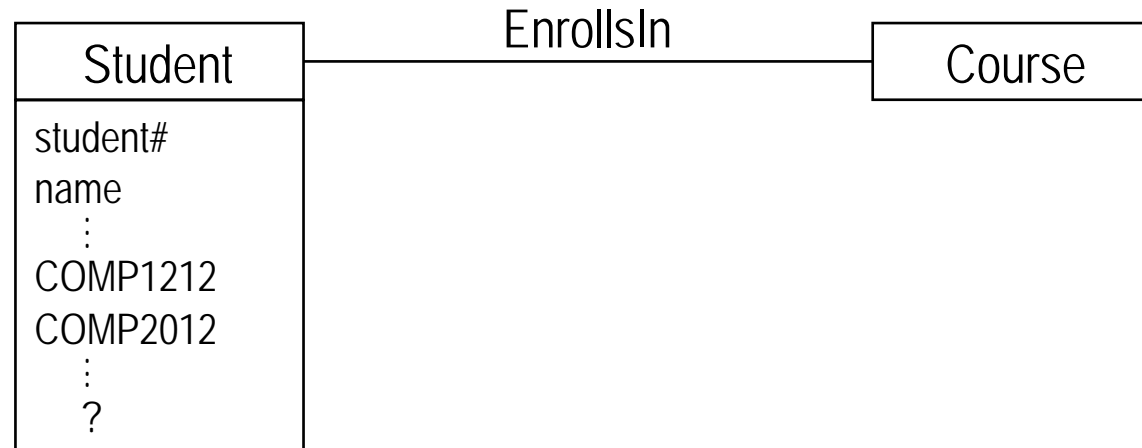
Where should we put an attribute like grade?



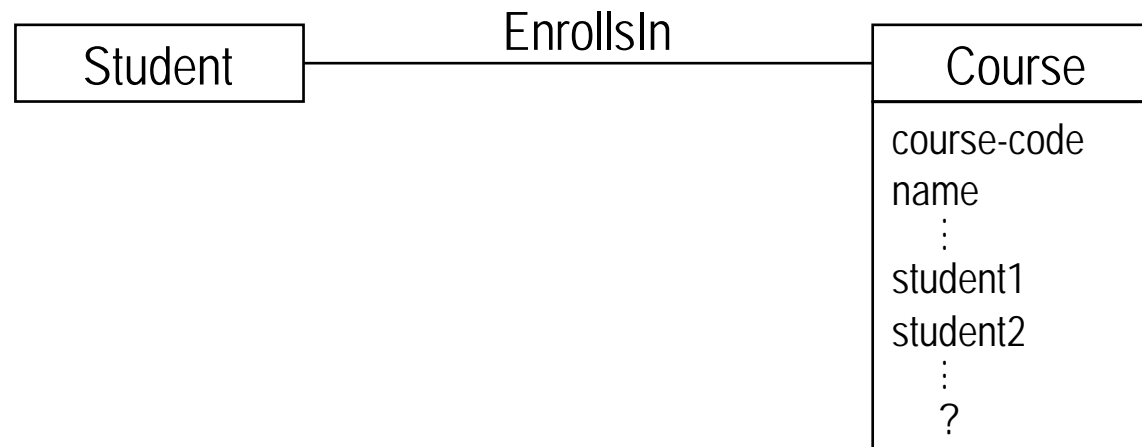
ASSOCIATION: ASSOCIATION CLASS (cont'd)

Representation 1: Use many attributes for grade.

(a) in Student



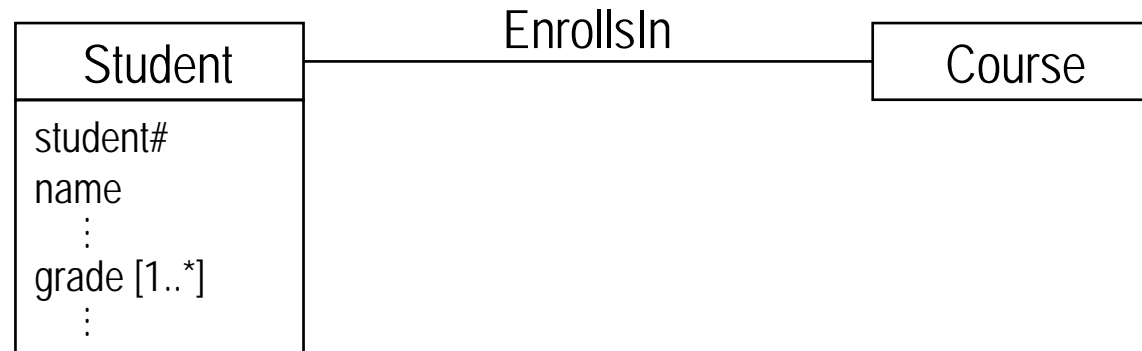
(b) in Course



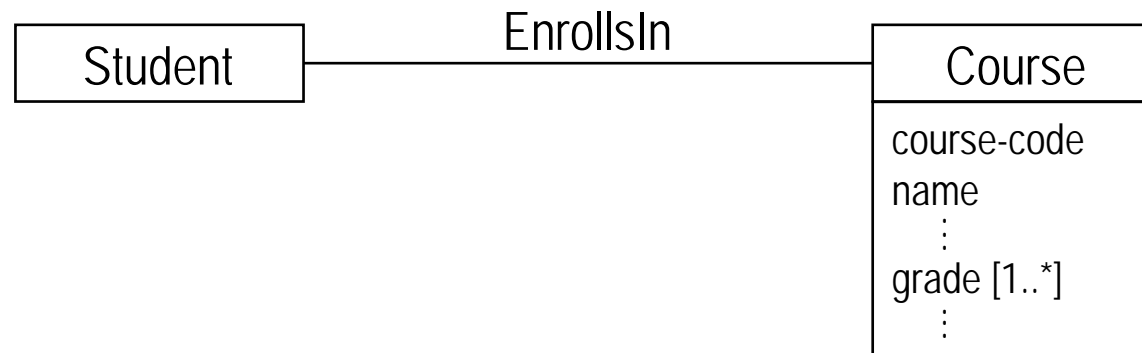
ASSOCIATION: ASSOCIATION CLASS (cont'd)

Representation 2: Use a multi-valued attribute for grade.

(a) in Student

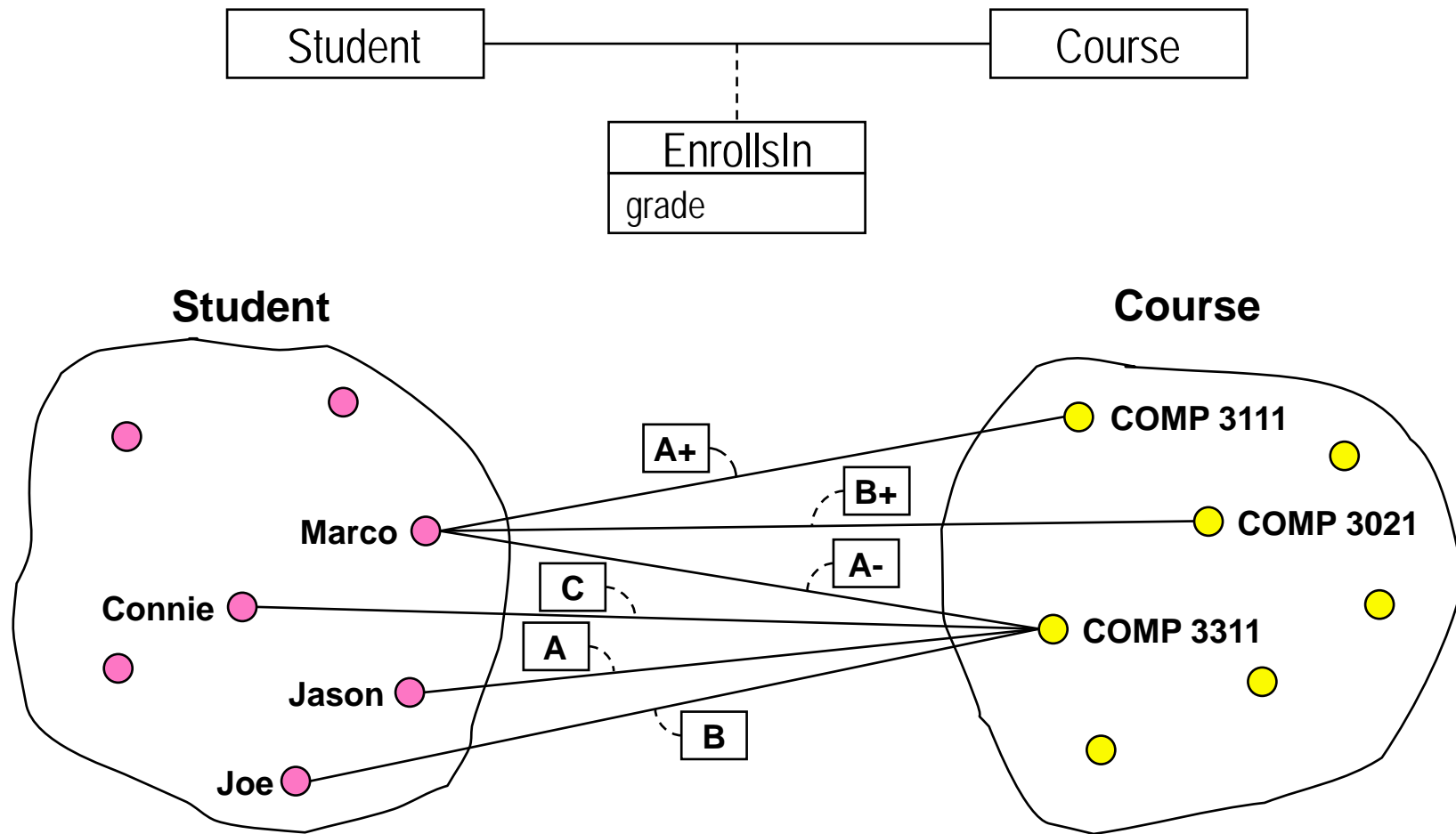


(b) in Course



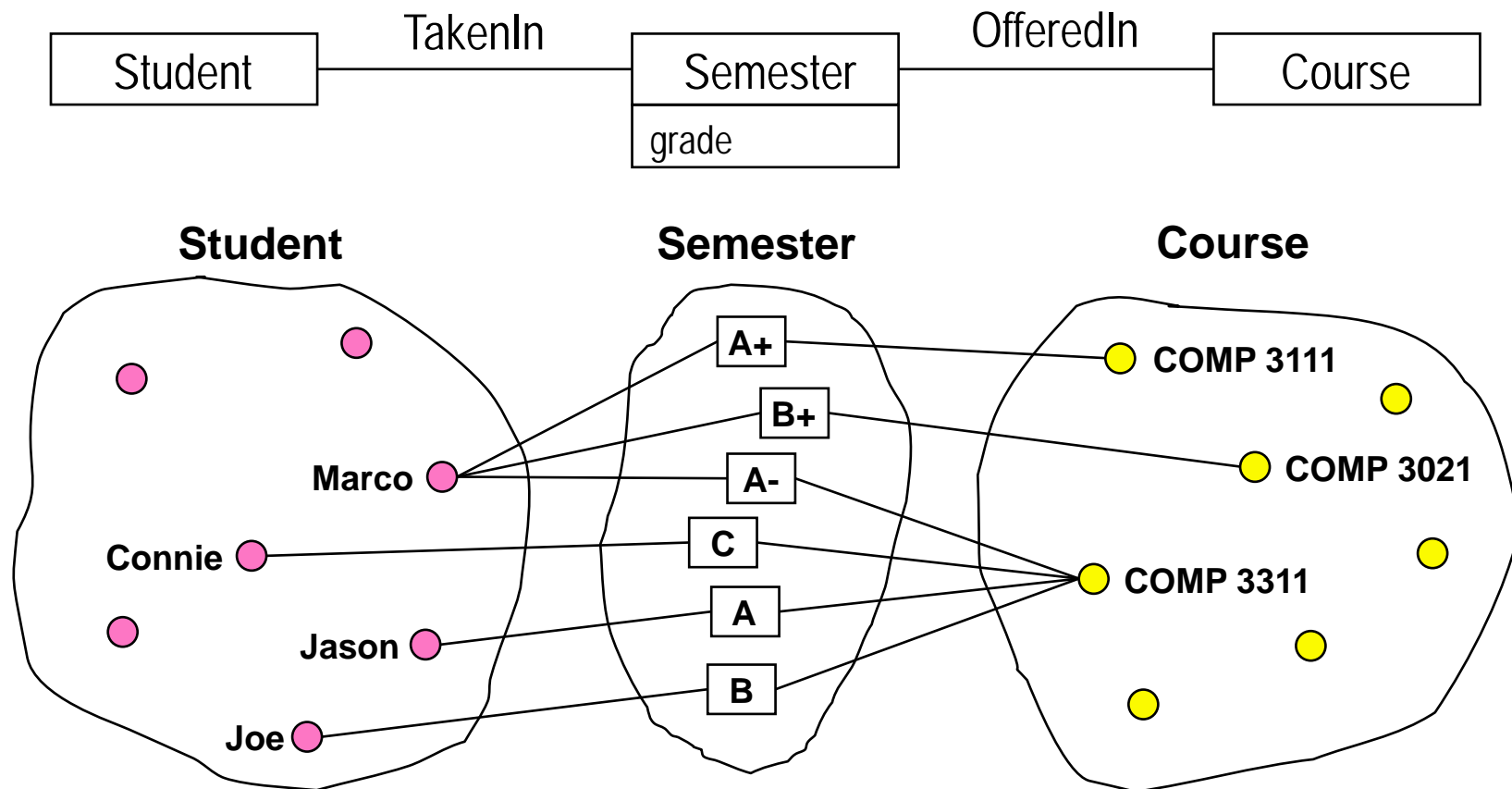
ASSOCIATION: ASSOCIATION CLASS (cont'd)

Representation 3: Use an **association class**.



ASSOCIATION: ASSOCIATION CLASS (cont'd)

Representation 4: Use a **separate class**.

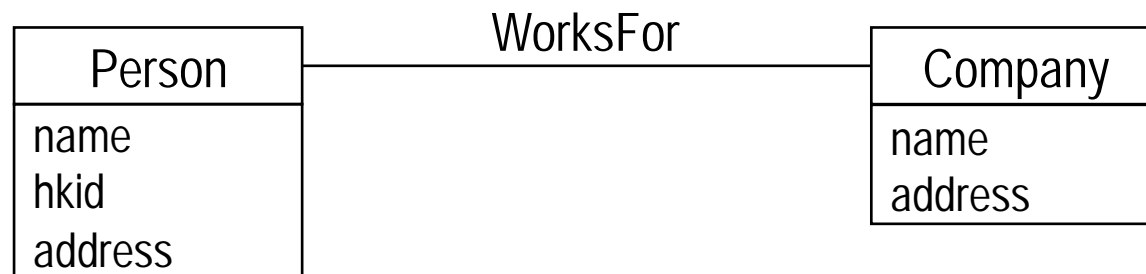


Note that each Semester object *must be related* to a Student and Course.

ASSOCIATION: ASSOCIATION CLASS (CONTD)

When to associate an attribute with an association?

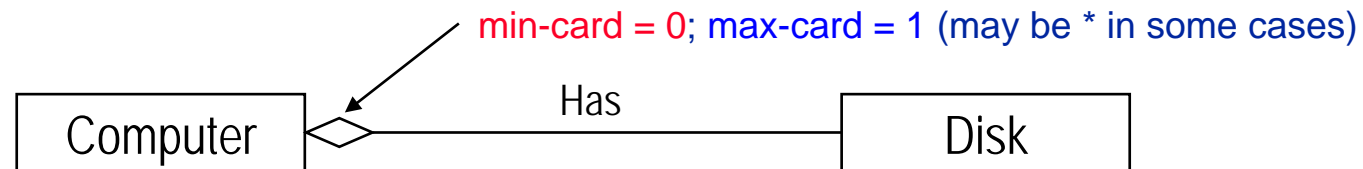
What about an attribute like salary ?



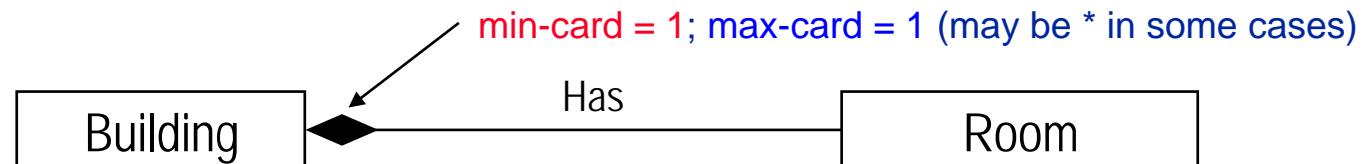
An association class is *most often* needed for
many to many associations!

AGGREGATION/COMPOSITION ASSOCIATION


- A special type of association in which there is a “**part-of**” **relationship** between one class and another class.
- ✎ A component **may exist independent** of the aggregate object of which it is a part → **aggregation**. [\diamond adornment]



- ✎ A component **may not exist independent** of the aggregate object of which it is a part → **composition**. [\blacklozenge adornment]



WHEN TO USE AGGREGATION/COMPOSITION?

- Would you use the phrase “**part of**” to describe the association or name it “**Has**”?
 **BUT BE CAREFUL!** Not all “Has” associations are aggregations.
- Is there an **intrinsic asymmetry** to the association where one object class is subordinate to the other(s)?
- Are operations on the **whole** automatically applied to the **part(s)**? → **composition**

The decision to use aggregation is a matter of **judgment**.
It is a **design decision**.

It is **not wrong** to use association rather than aggregation!
(In a real project, when in doubt, use association!)

GENERALIZATION

A *generalization* is a relationship between classes of the same kind.

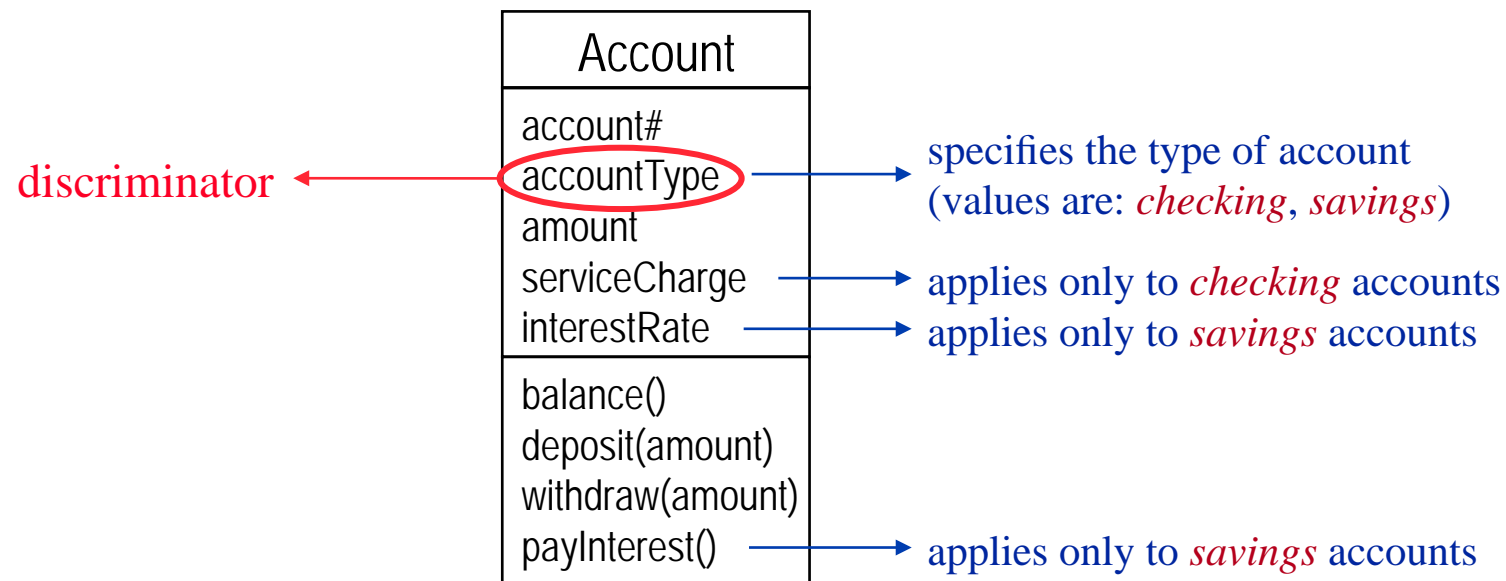
- *If it is meaningful to do so*, we classify classes according to the **similarity** of their **attributes**, **operations** and **associations**.

☞ Look for “**kind-of**” **statements** that are **true in an application domain**.

Goal: Simplicity of representation and modeling **clarity**.

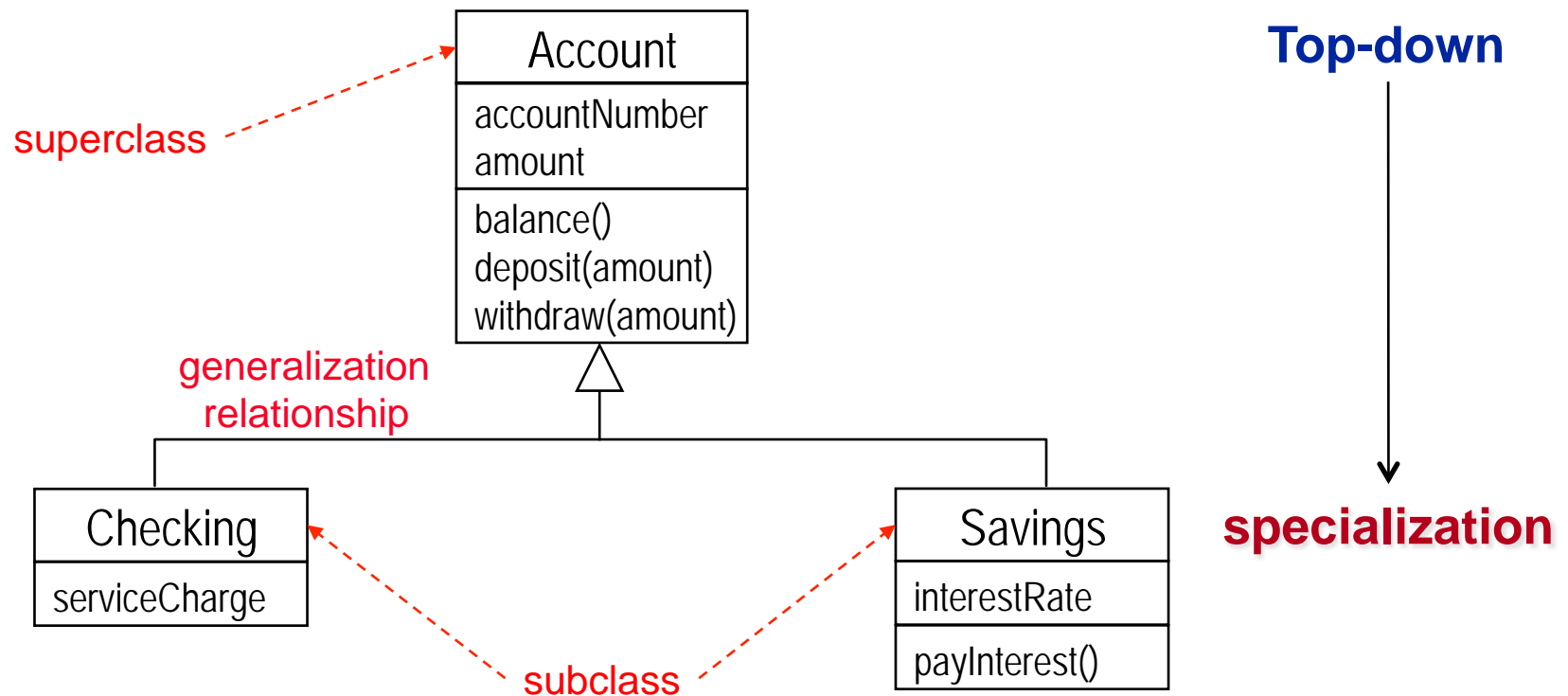
GENERALIZATION (cont'd)

discriminator: An **attribute of enumeration type** that indicates which property of a class is used to create a generalization relationship.



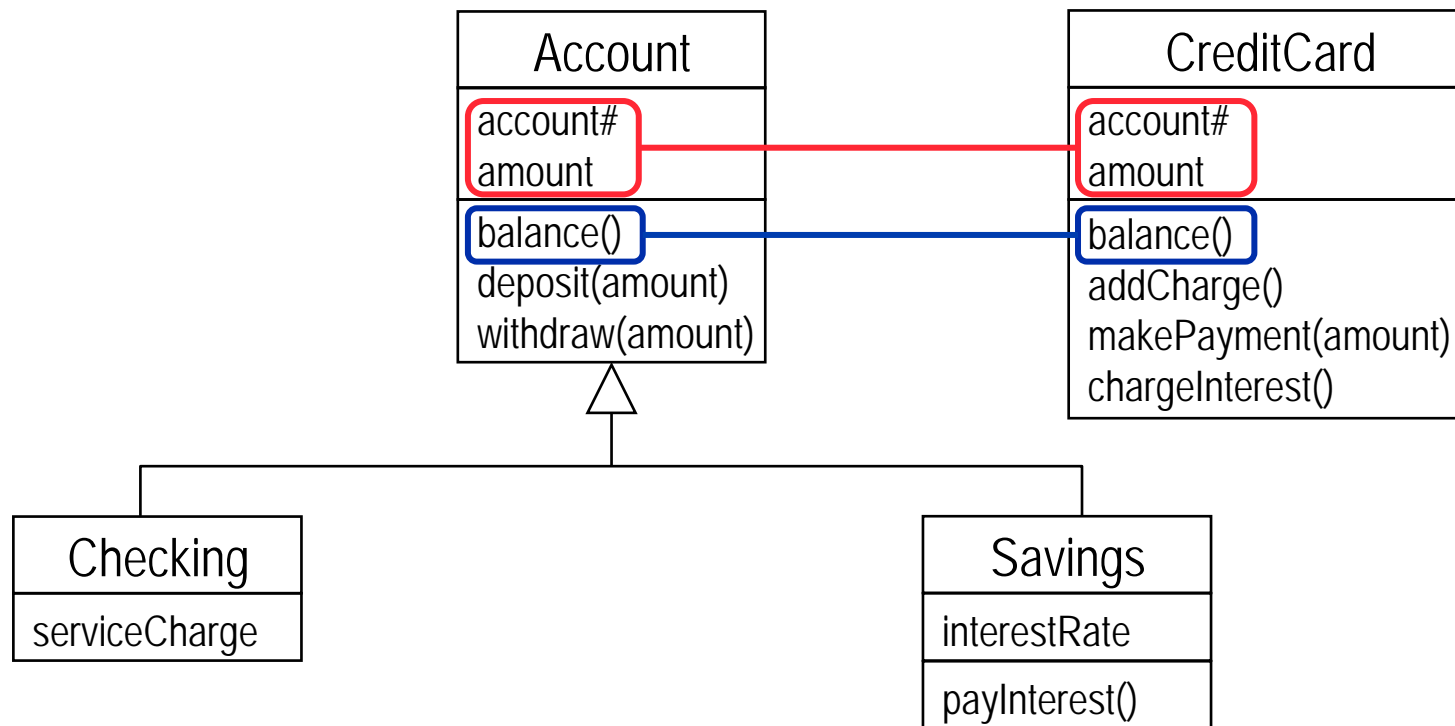
GENERALIZATION (CONT'D)

discriminator: An attribute of enumeration type that indicates which property of a class is used to create a generalization relationship.



GENERALIZATION (cont'd)

Can also be applied bottom-up

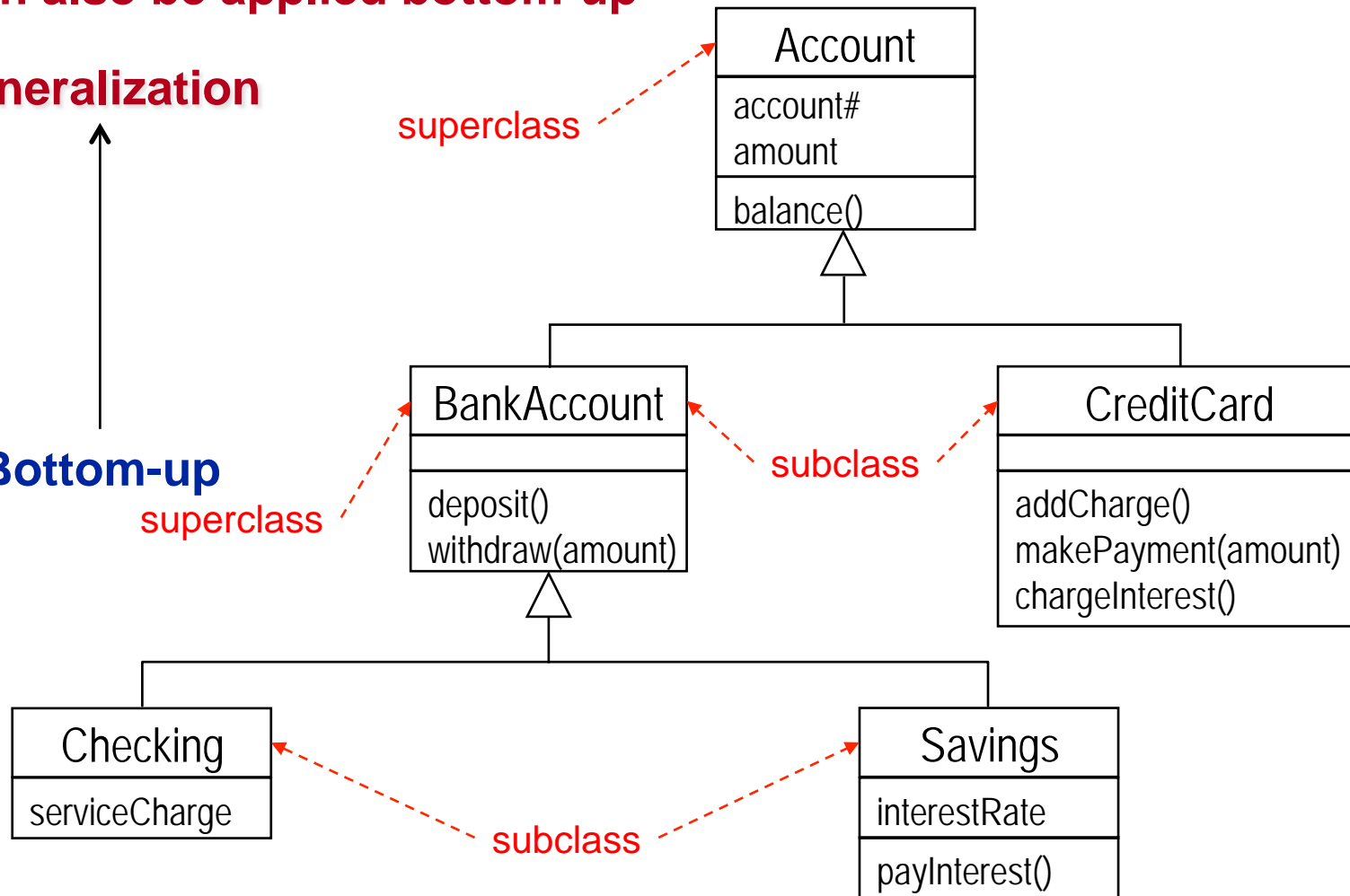


GENERALIZATION (CONTD)

Can also be applied bottom-up

generalization

Bottom-up



GENERALIZATION: INHERITANCE

Inheritance is the assumption of properties by a subclass from its superclasses.

- We place common attributes and operations in a superclass and inherit them to the subclass(es).

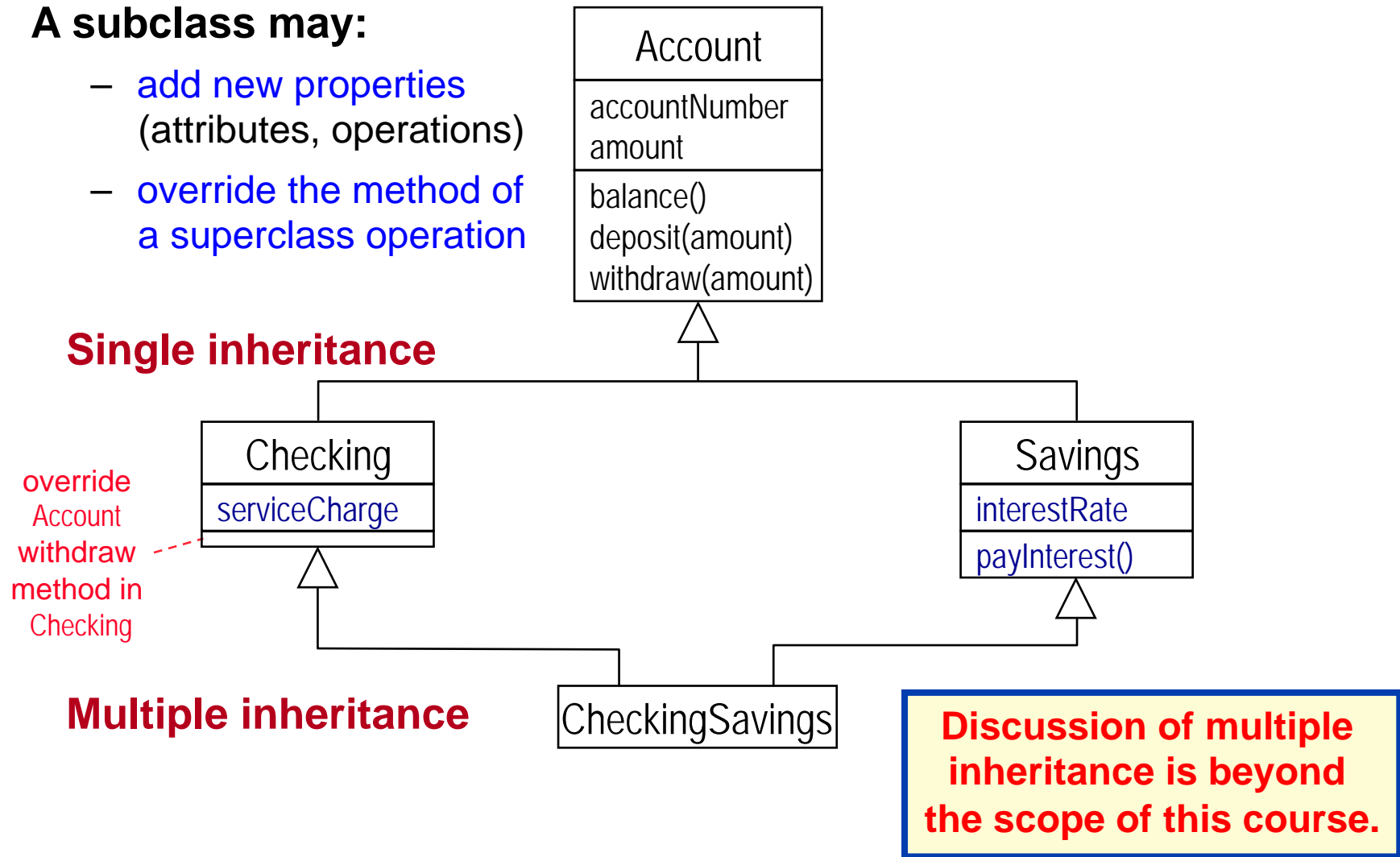
 **Attributes and operations are only defined in one place:**

- ✓ reduces redundancy of descriptions.
- ✓ promotes reusability of descriptions.
- ✓ simplifies modification of descriptions.

GENERALIZATION: INHERITANCE (cont'd)

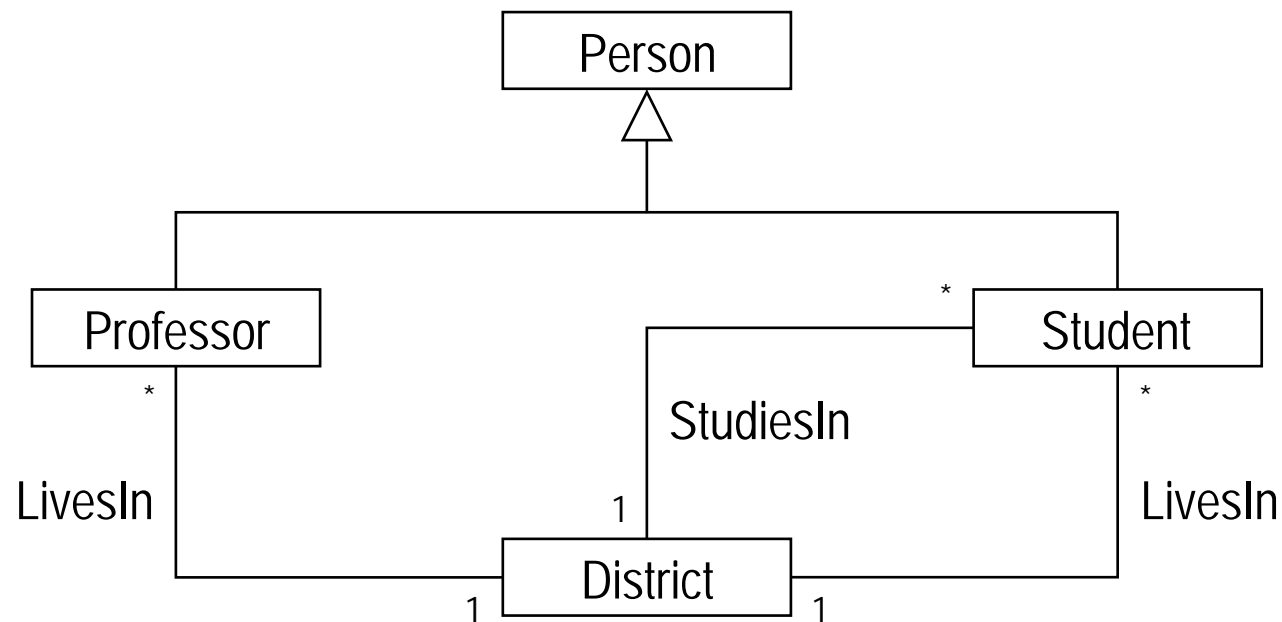
A subclass may:

- add new properties (attributes, operations)
- override the method of a superclass operation



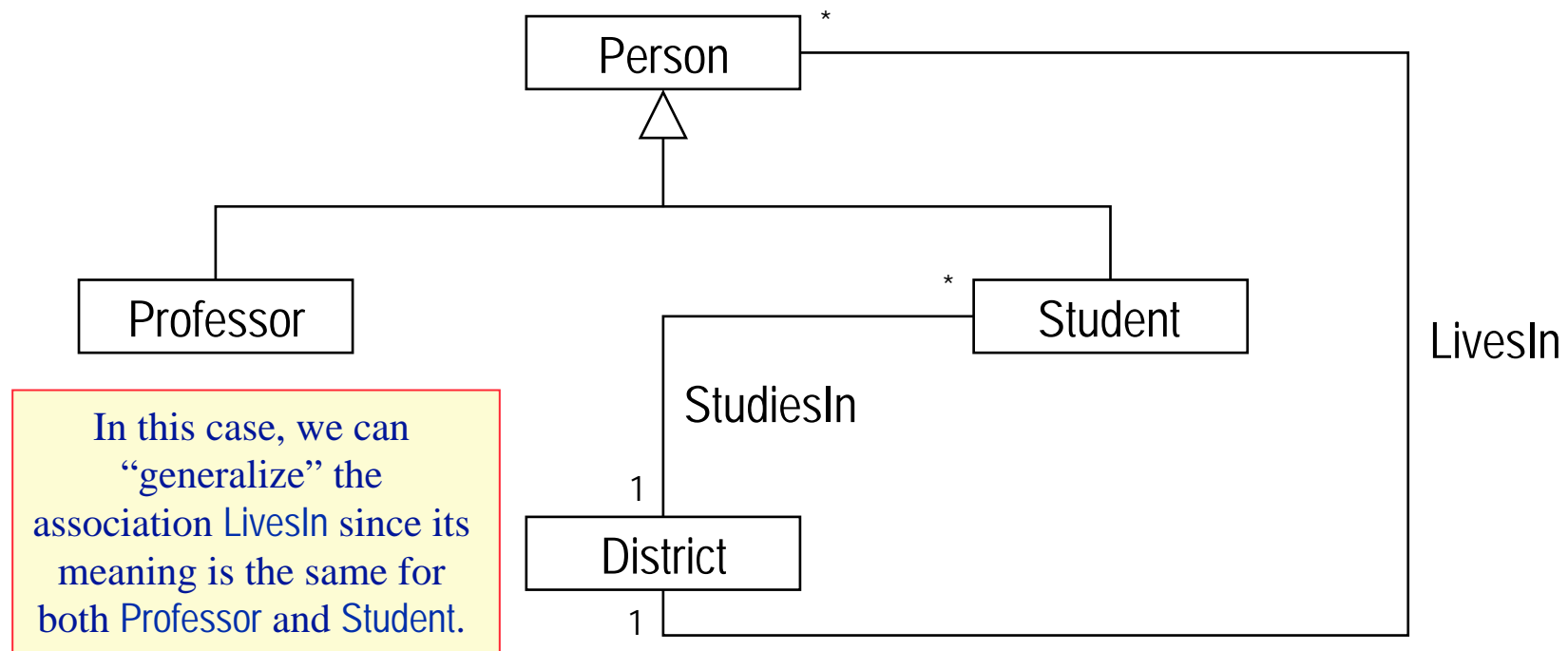
DOMAIN MODEL DETAIL: GENERALIZATION (cont'd)

👉 How to handle associations in which subclasses participate?



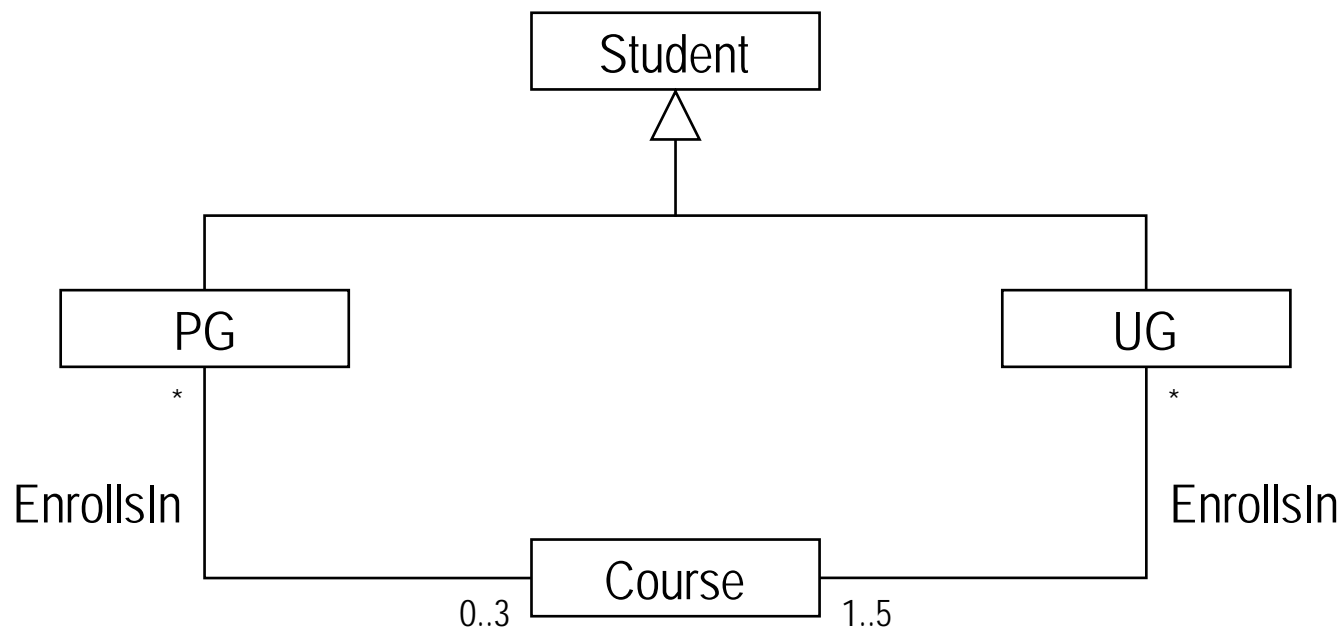
DOMAIN MODEL DETAIL: GENERALIZATION (cont'd)

👉 How to handle associations in which subclasses participate?



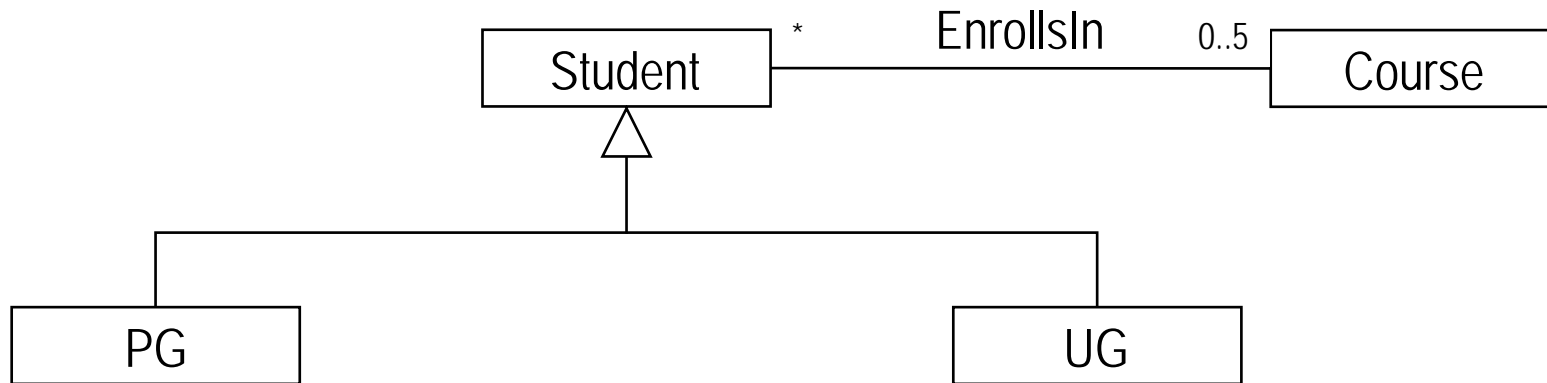
DOMAIN MODEL DETAIL: GENERALIZATION (cont'd)

✎ Sometimes it is necessary to “adjust” the multiplicity of an association.



DOMAIN MODEL DETAIL: GENERALIZATION (cont'd)

✎ Sometimes it is necessary to “adjust” the multiplicity of an association.

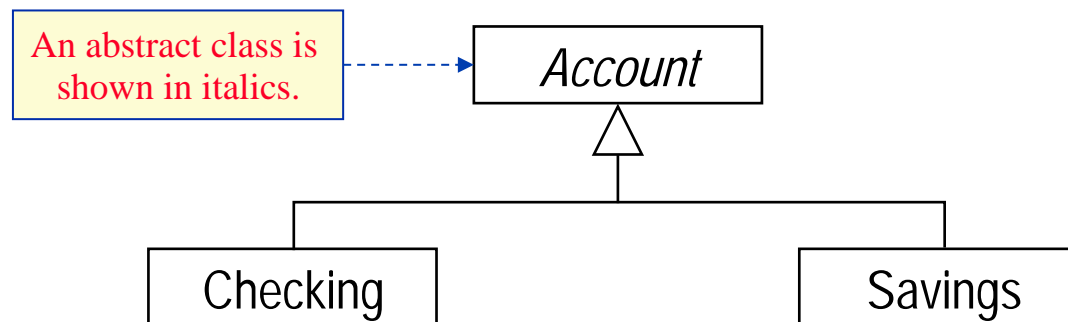


In this case, although we can “generalize” the **EnrollsIn** association, we need to use the *minimum* of the min-card and the *maximum* of max-card of both original **EnrollsIn** associations in order to preserve the semantics of the original class diagram.

GENERALIZATION: ABSTRACT CLASS

An *abstract class* is a class that has no direct instances.

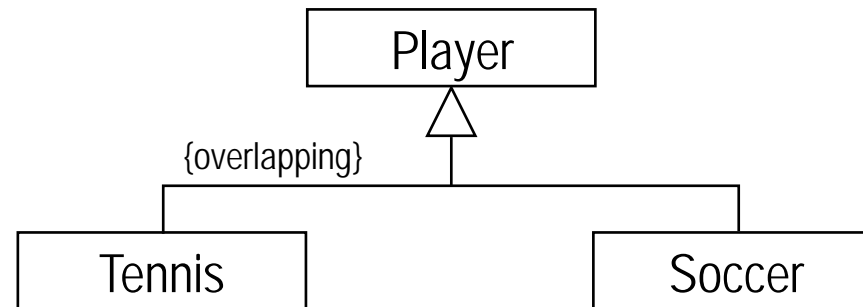
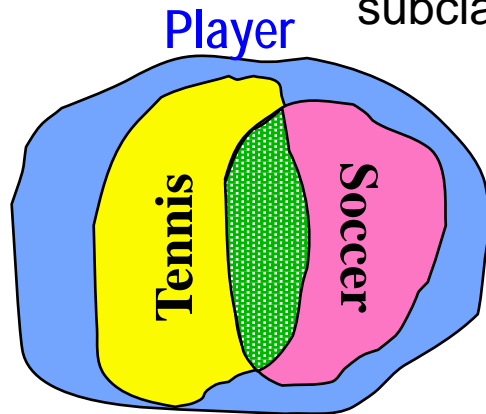
- An **abstract class** is used, for modeling purposes, as a *container for definitions*, but no instances of the class are of interest.



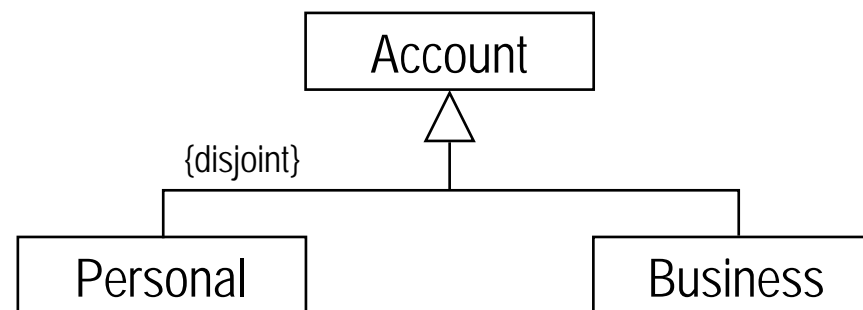
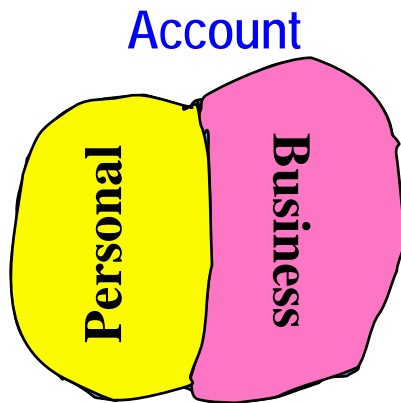
Note: Operations may also be abstract → no method specified.

GENERALIZATION: COVERAGE

overlapping - A superclass object can be a **member of more than one** subclass.

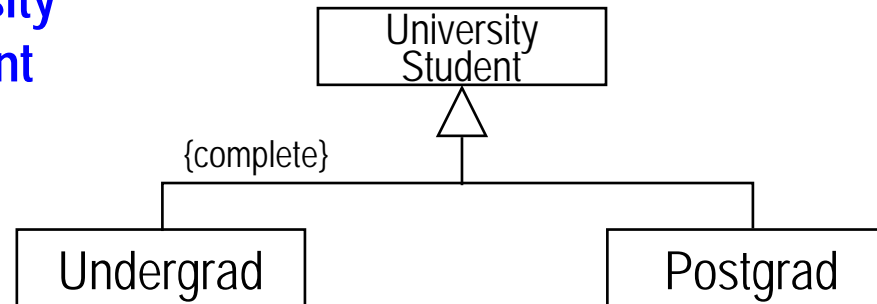
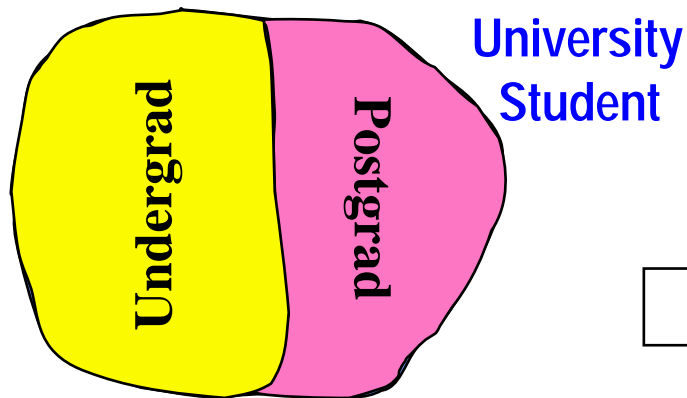


disjoint - A superclass object is a **member of at most one** subclass.

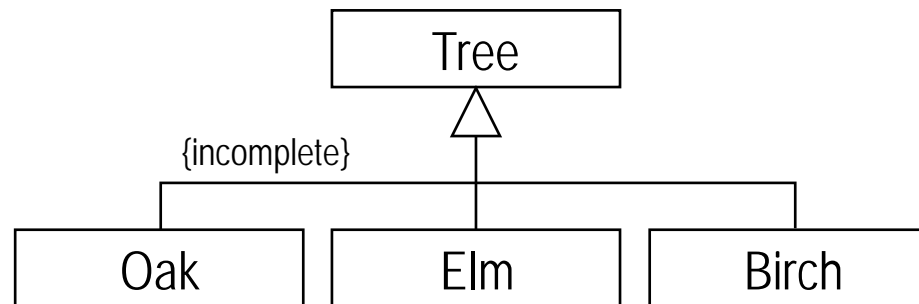
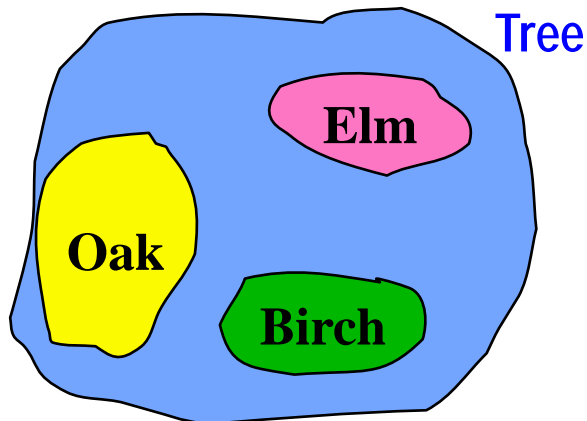


GENERALIZATION: COVERAGE (cont'd)

complete - All superclass objects **must be members** of some subclass.

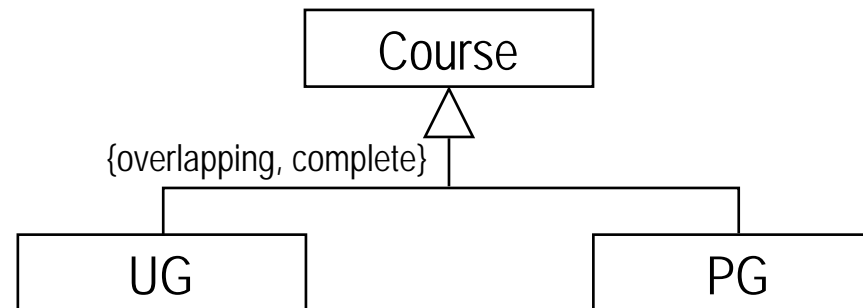
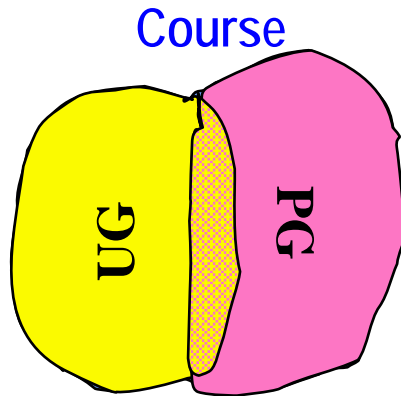


incomplete - Some superclass object is **not a member** of any subclass.

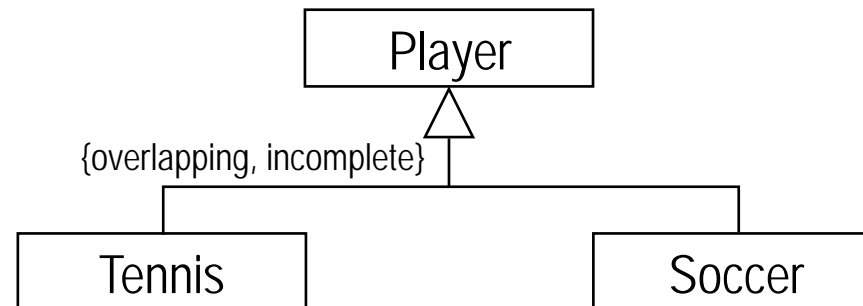
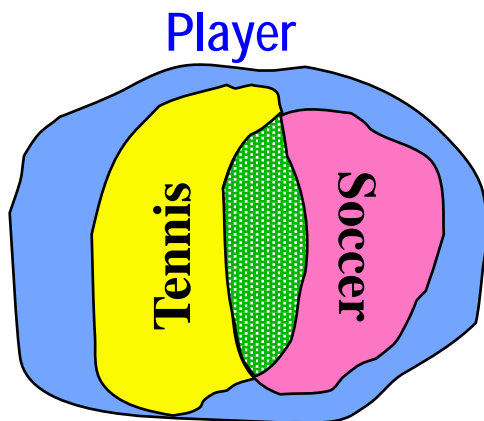


GENERALIZATION: COVERAGE TYPES

1. overlapping, complete

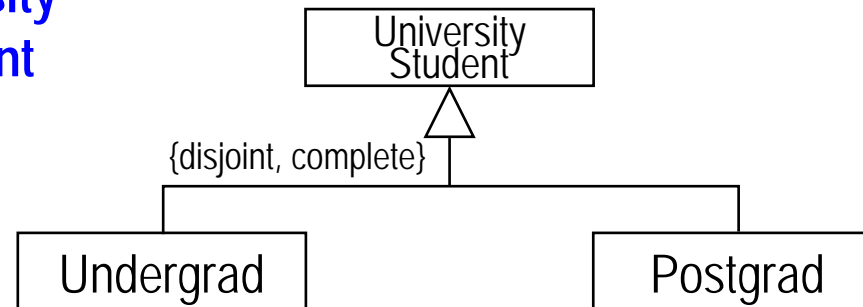
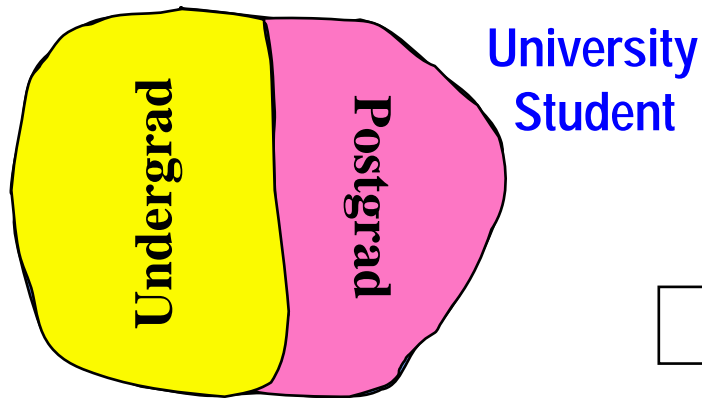


2. overlapping, incomplete

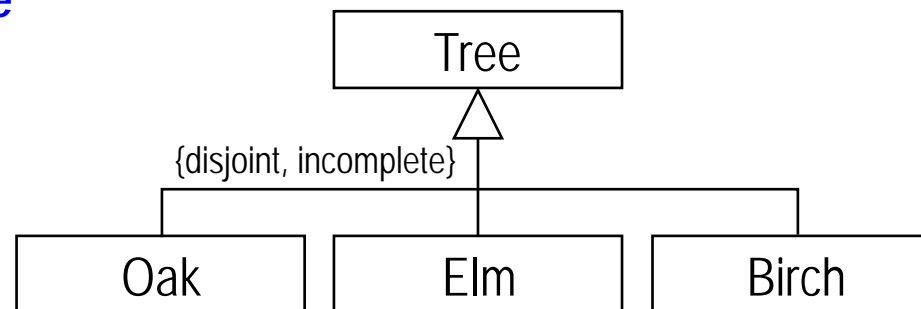
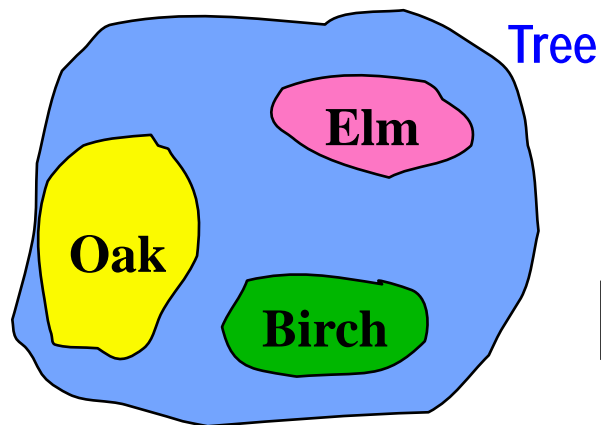


GENERALIZATION: COVERAGE TYPES (cont'd)

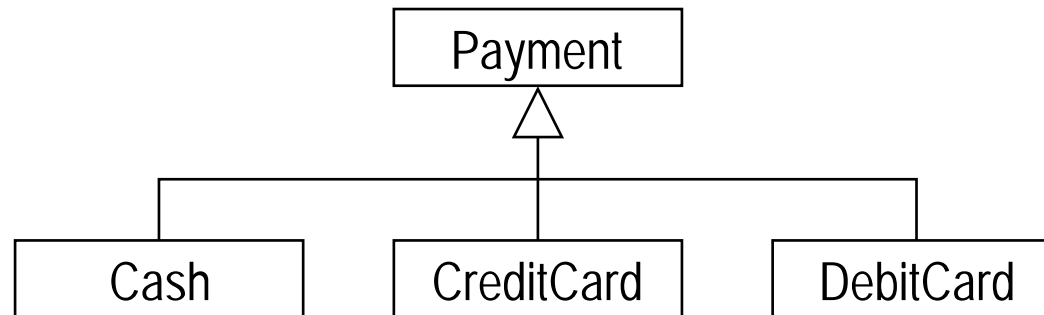
3. disjoint, complete



4. disjoint, incomplete



QUESTION?



The coverage of the generalization shown above is:

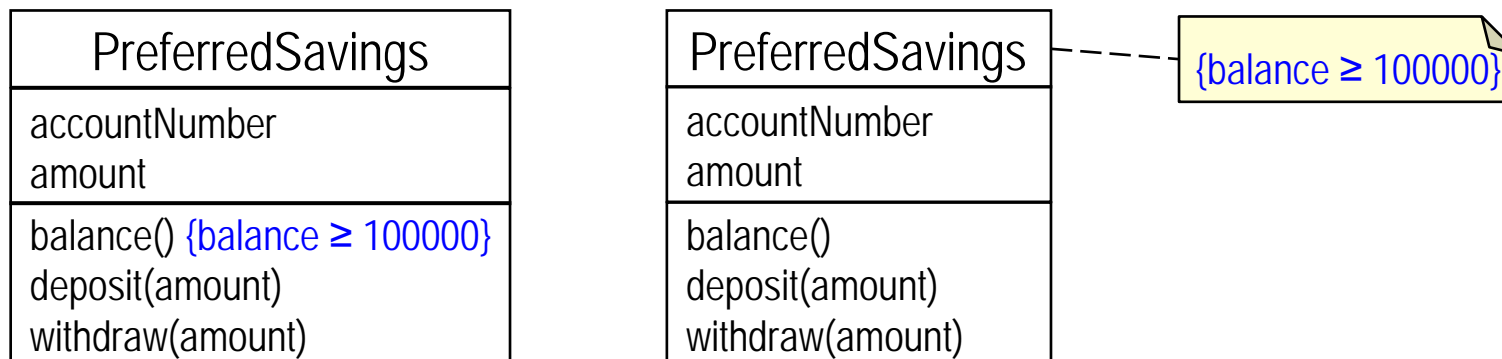


- overlapping, complete
- disjoint, complete
- overlapping, incomplete
- disjoint, incomplete

CONSTRAINTS

A *constraint* is an assertion about properties of model elements that must be satisfied by the system.

Example A preferential savings account whose balance always must be greater than \$100,000.

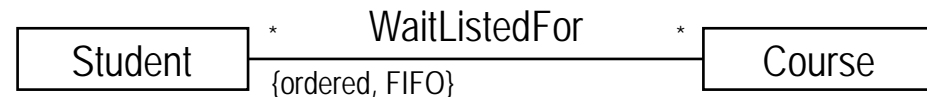


A constraint is a statement that can be tested (true/false) and should be enforced by the system implementation.

COMMON ASSOCIATION CONSTRAINTS

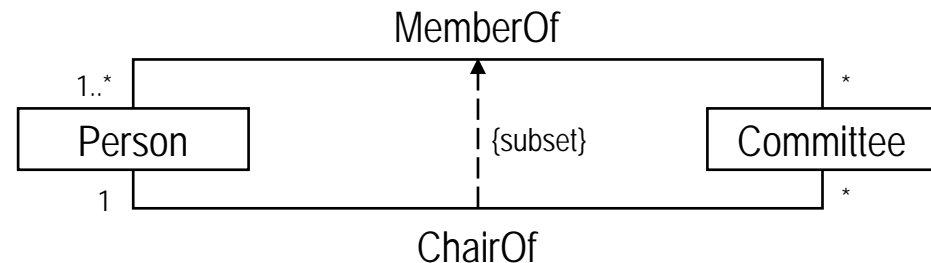
ordering

- States that the set of objects at one end of an association is in an explicit order (normally, they are unordered).



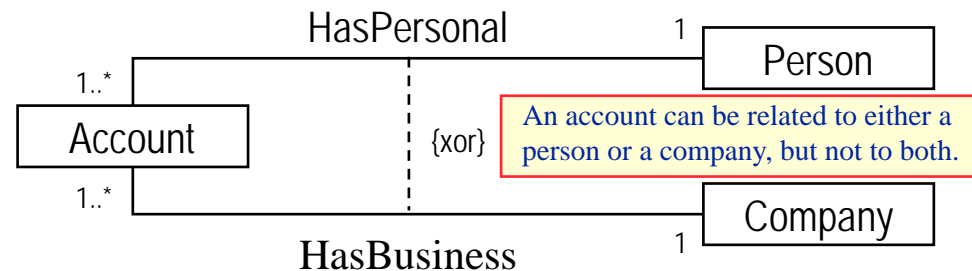
subset

- States that the set of links which are instances of one association must be a subset of those of another association.



xor

- States that *instances of the common class* can participate in only one of the constrained associations at any given time.



OBJECT CONSTRAINT LANGUAGE (OCL)

- UML provides a text-based formal constraint specification language called **Object Constraint Language (OCL)** (similar in form to C++ or Java).

Discussion of OCL is beyond the scope of this course.

- For more information on OCL, please see:

Object Constraint Language Specification, Version 2.0. available at <http://www.omg.org/technology/documents/formal/ocl.htm>.

Warmer, J. and Kleppe, A. *The Object Constraint Language Second Edition: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.

EXAMPLE: BANKING APPLICATION

The following concepts all represent classes, which are related according to the problem statement.

Bank	Deposit	Account	Savings
Customer	Transfer	CreditCard	Checking
Withdrawal	Transaction	Passbook	

Banks issue credit cards (e.g., Visa, MasterCard, etc.) to customers. Each credit card is issued by only one bank to only one customer. Customers hold accounts with banks. Each account is with only one bank and held by only one customer. An account may be a savings account or a checking account. Each savings account has a passbook, which shows the transactions made against the account. Checking accounts do not have passbooks. Customers make transactions against their accounts. A transaction can be a deposit, withdrawal or transfer.

Construct a class diagram, using only the eleven given classes.

STEREOTYPE

A *stereotype* is a new kind of modeling element, which is a subclass of an existing modeling element.

✎ Allows the object model to be **dynamically extended**.

Example: We can define different kinds of classes that are useful for modeling.



PenTracker



OrderForm



Account

✎ Special **icons** can be used for each stereotype that **improve modeling clarity**.

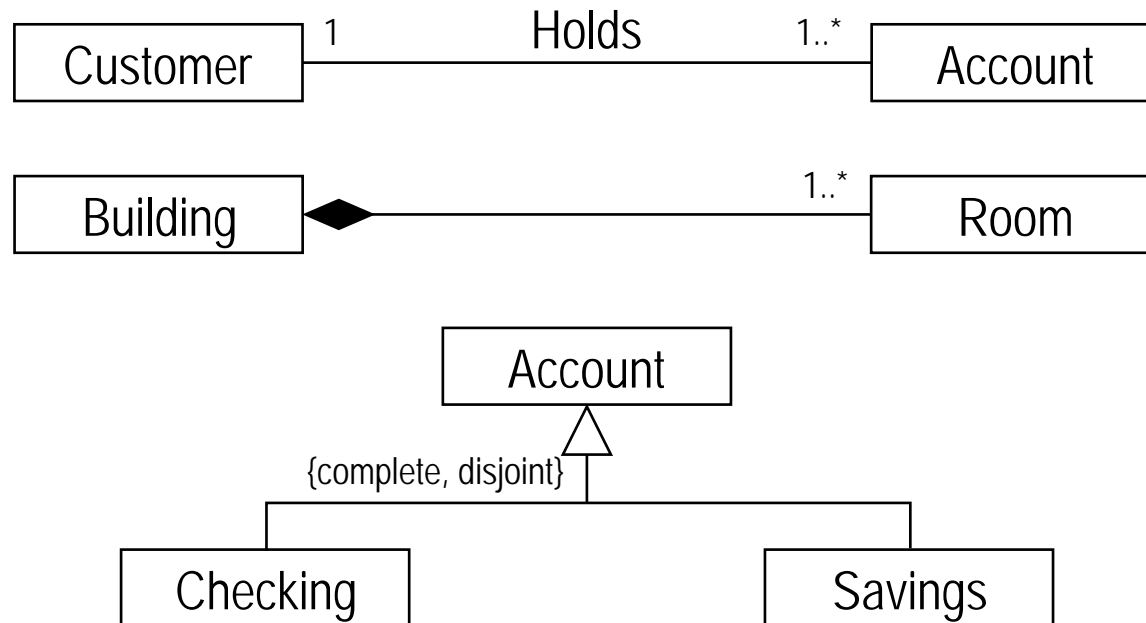
MODELING SOFTWARE SYSTEMS USING UML: SUMMARY

Class name
attribute compartment
operation compartment

Modeling Elements

<u>classifier</u>	<u>instance</u>
class	object
association	link

+ **Constraints**



MODELING SOFTWARE SYSTEMS USING UML: SUMMARY

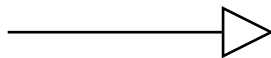
Relationships

association



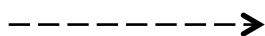
describes links among object instances (only relationship that relates object instances)

generalization



relates a more general class (superclass) to a more specific kind of the general class (subclass)

dependency



relates classes whose behaviour or implementation affect other classes

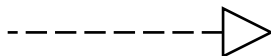
flow

relates two versions of an object at successive times

usage

shows that one class requires another class for its correct functioning

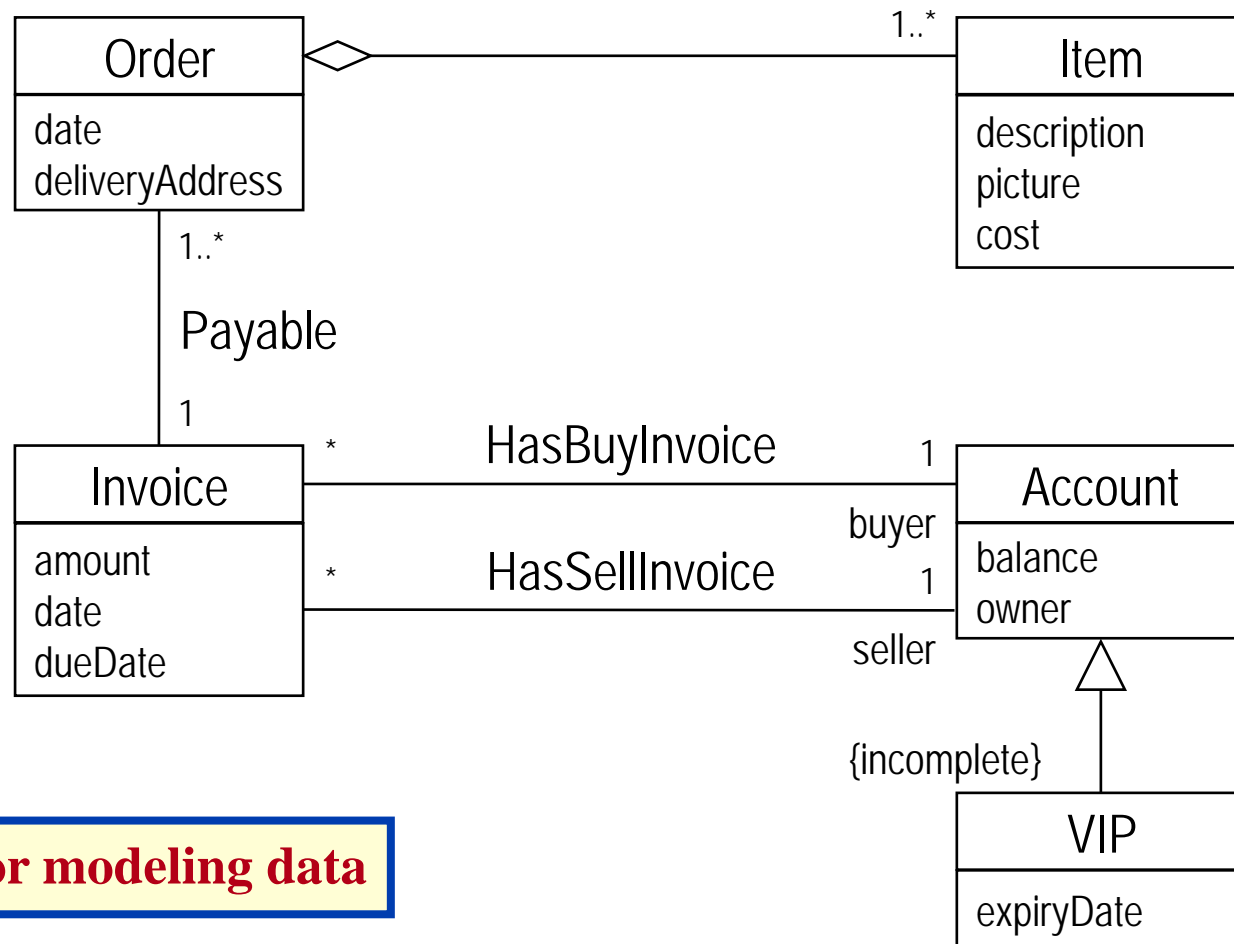
realization



relates a specification to its implementation (e.g., an interface to the classes that implement it)

MODELING SOFTWARE SYSTEMS USING UML: SUMMARY

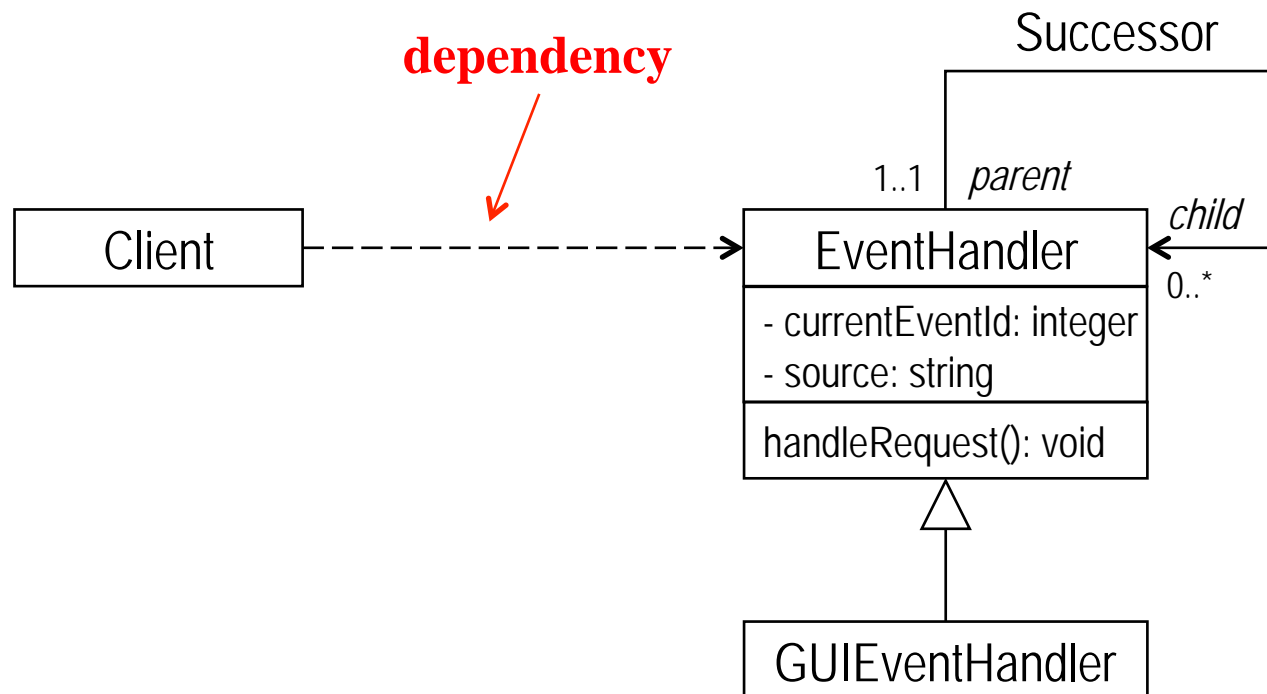
Static Modeling Example



For modeling data

MODELING SOFTWARE SYSTEMS USING UML: SUMMARY

Dynamic Modeling Example



For modeling programs