OO Modeling with UML

Programming through pictures

Key Questions

- Why visual modeling?
- What is round trip engineering?
- What is a class diagram?
- What is multiplicity?
- What is constraint?

Why modeling?

User.java

```
package hkust.cse.calendar.unit;

import java.io.Serializable;

public class User implements Serializable {

        private String mPassword;
        private String mID;


        // Getter of the user id
        public String ID() {
                return mID;
        }
```

ass 'User' which set

```
        // up the user id and password
        public User(String id, String pass) {
                mID = id;
                mPassword = pass;
        }

        // Another getter of the user id
        public String toString() {
                return ID();
        }

        // Getter of the user password
        public String Password() {
                return mPassword;
        }

        // Setter of the user password
        public void Password(String pass) {
                mPassword = pass;
        }
}
```

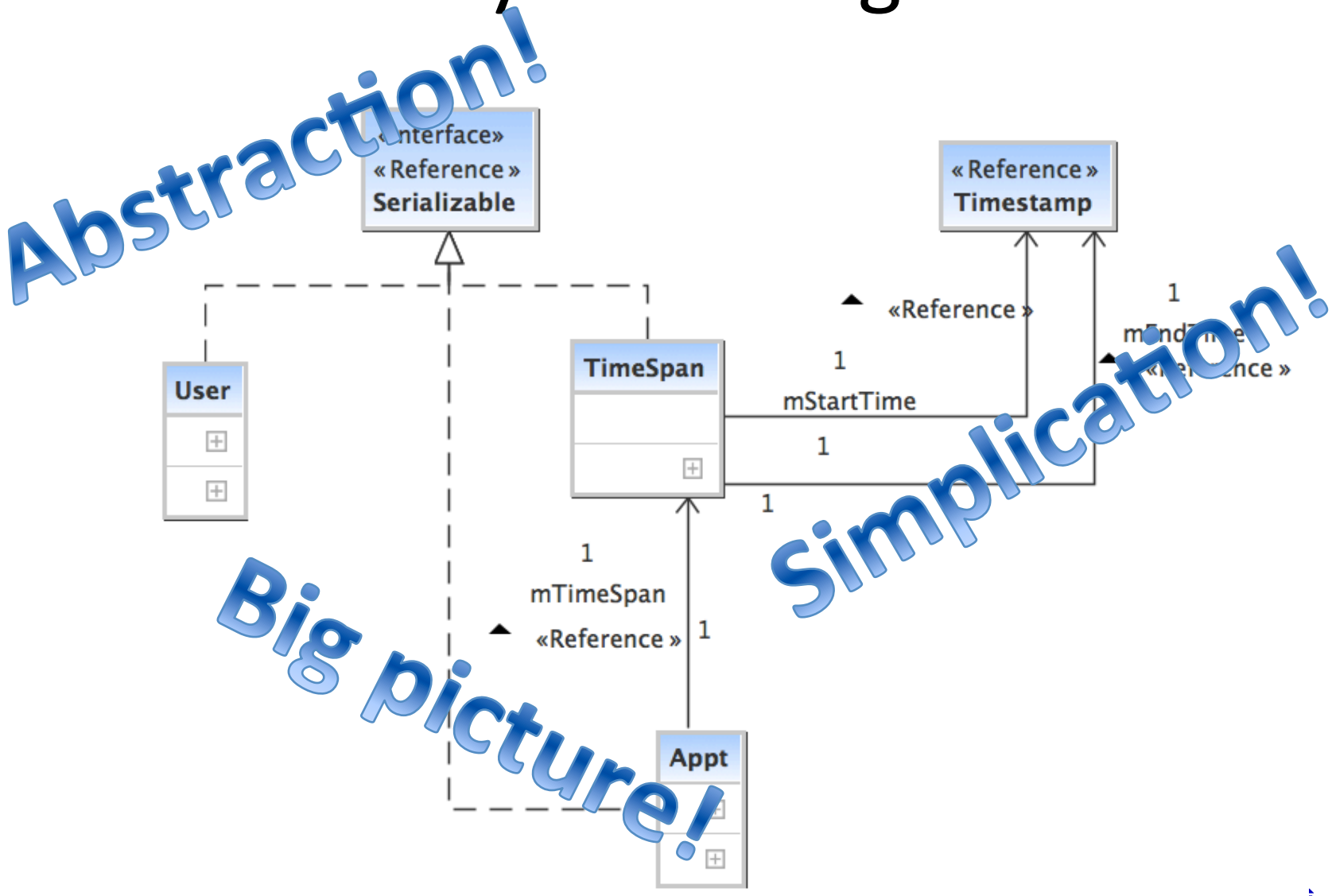
TimeSpan.java

```
package hkust.cse.calendar.unit;

import java.io.Serializable;
import java.sql.Timestamp;

/* This class represents the time span between two points of time */
public class TimeSpan implements Serializable {
        /* The starting time of the time span */
        private Timestamp mStartTime;
        /* The ending time of the time span */
        private Timestamp mEndTime;

        /* Create a new TimeSpan object with the specific starting time and ending time */
        public TimeSpan(Timestamp start, Timestamp end) {

                if (start.getYear() >= 1900) {
                        start.setYear(start.getYear() - 1900);
                }
                if (end.getYear() >= 1900) {
                        end.setYear(end.getYear() - 1900);
                }
                mStartTime = start;
                mEndTime = end;
        }

        /* Get the starting time */
        public Timestamp StartTime() {
                return mStartTime;
        }

        /* Check whether a time span overlaps with this time span */
        public boolean Overlap(TimeSpan d) {
                if (d.EndTime().before(mStartTime) || d.EndTime().equals(mStartTime))
                        return false;
                if (d.StartTime().equals(mEndTime) || mEndTime.before(d.StartTime()))
                        return false;
                return true;                    // Else, the time span overlaps with this time span

        }

        /* Calculate the length of the time span if the starting time and ending time are within the same day */
        public int TimeLength() {

                /* return -1 if the starting time and ending time are not in the same day */
                if (mStartTime.getYear() != mEndTime.getYear())
                        return -1;
                if (mStartTime.getMonth() != mEndTime.getMonth())
                        return -1;
                if (mStartTime.getDay() != mEndTime.getDay())
                        return -1;

                /* Calculate the number of minutes within the time span */
                int result = mStartTime.getHours() * 60 + mStartTime.getMinutes()
                                - mEndTime.getHours() * 60 - mEndTime.getMinutes();
                if (result < 0)
                        return -1;
                else
                        return result;
        }

        /* Set the starting time */
        public void StartTime(Timestamp s) {
                mStartTime = s;
        }

        /* Set the ending time */
        public void EndTime(Timestamp e) {
                mEndTime = e;
        }
}
```

Appt.java

```
package hkust.cse.calendar.unit;

import java.io.Serializable;
import java.util.LinkedList;

public class Appt implements Serializable {

        private TimeSpan mTimeSpan;
        private String mTitle;
        private String mInfo;
        private int mApptID;
        private int joinApptID;
        private boolean isjoint;
        private LinkedList<String> attend;
        private LinkedList<String> reject;
        private LinkedList<String> waiting;
        public Appt() {
mApptID = 0;
                mTimeSpan = null;
                mTitle = "Untitled";
                mInfo = "";
                isjoint = false;
                attend = new LinkedList<String>();
                reject = new LinkedList<String>();
                waiting = new LinkedList<String>();
                joinApptID = -1;
        }

        // Getter of the mTimeSpan
        public TimeSpan TimeSpan() {
                return mTimeSpan;
        }

                                ent title

                                rn mTitle;

        // Getter of appointment description
        public String getInfo() {
                return mInfo;
        }

        // Getter of the appointment id
        public int getID() {
                return mApptID;
        }

        // Getter of the join appointment id
        public int getJoinID(){
                return joinApptID;
        }

        public void setJoinID(int joinID){
                this.joinApptID = joinID;
        }
        // Getter of the attend LinkedList<String>
        public LinkedList<String> getAttendList(){
                return attend;
        }

        // Getter of the reject LinkedList<String>
        public LinkedList<String> getRejectList(){
                return reject;
        }

        // Getter of the waiting LinkedList<String>
        public LinkedList<String> getWaitingList(){
                return waiting;
        }

        public LinkedList<String> getAllPeople(){
                LinkedList<String> allList = new LinkedList<String>();
                allList.addAll(attend);
                allList.addAll(reject);
                allList.addAll(waiting);
                return allList;
        }

        public void addAttendant(String addID){
                if (attend == null)
                                attend = new LinkedList<String>();
                attend.add(addID);
        }

        public void addReject(String addID){
                if (reject == null)
                                reject = new LinkedList<String>();
                reject.add(addID);
        }
```


Why modeling?



Why modeling?

- **Why build models?**
  - Models succinctly describe reality (i.e., they abstract reality).
    - 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 *complexity* 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.

Demo

- Round-trip engineering with UML Lab

Unified Model Language (UML)

- The UML is a graphical language for
  - specifying
  - visualizing
  - constructing
  - documenting

  the artifacts of software systems
- Created by unifying the Booch, OMT, and Objectory modeling languages
- Specified and managed by OMG in November 1997 as UML 1.1
- Most recent revision is UML 2.4

UML 2.0: Complete specification

- **Structure diagrams**.
  - 1.Class diagram
  - 2. Composite structure diagram
  - 3. Component diagram
  - 4. Deployment diagram
  - 5. Object diagram
  - 6. Package diagram
  - 7. Use-case diagram
- **Behavior diagrams**
  - 8. State machine diagram
  - 9. Activity diagram
  - 10. Sequence diagram
  - 11. Communication diagram
  - 12. Interaction overview diagram
  - 13. Timing diagram

UML: Structural modeling

- Structural model: a view of an system that emphasizes the structure of the objects, including their classifiers, relationships, attributes and operations.
- Show the static structure of the model
  - the entities that exist (e.g., classes, interfaces, components, nodes)
  - internal structure
  - relationship to other entities
- Do not show
  - temporal information
- Kinds
  - static structural diagrams
    - class diagram
    - object diagram
  - implementation diagrams
    - component diagram
    - deployment diagram

Review of OOP

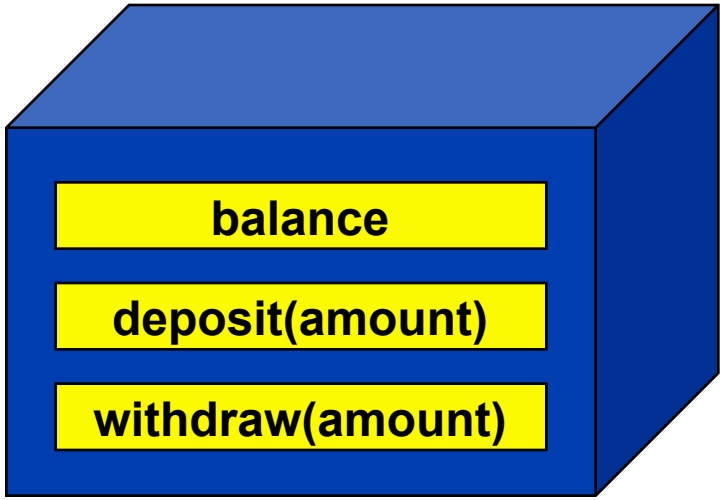
- Object-orientation
  - Classes and objects
  - Attributes and operations
  - Relationships between objects
- Three properties of OOP
  - Inheritance
  - Encapsulation
  - Polymorphism

Object: Instance

A physical or abstract thing with crisp boundaries and meaning which can respond to a request for an operation.

Fred's Bank Account
(a real-life object)

Our Bank Account object has three visible operations.

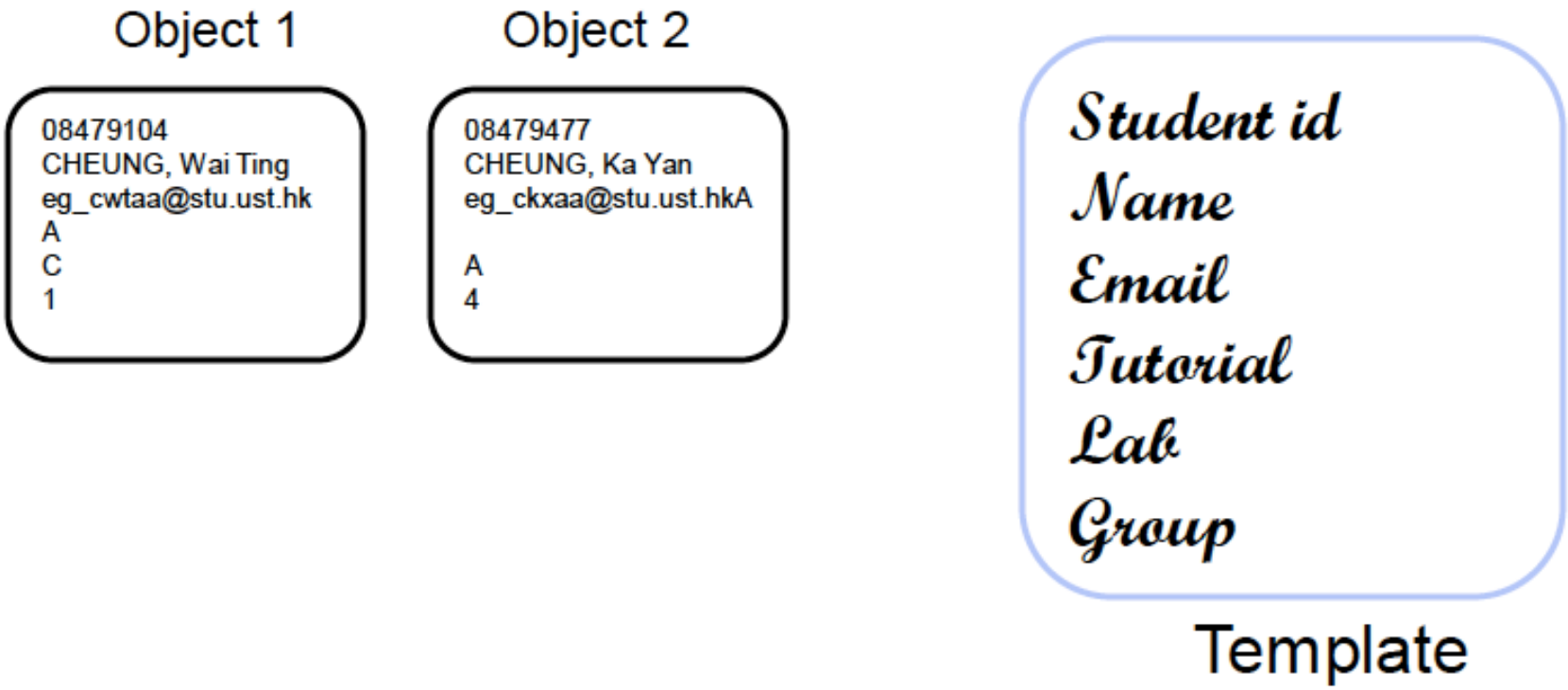


Object: Class

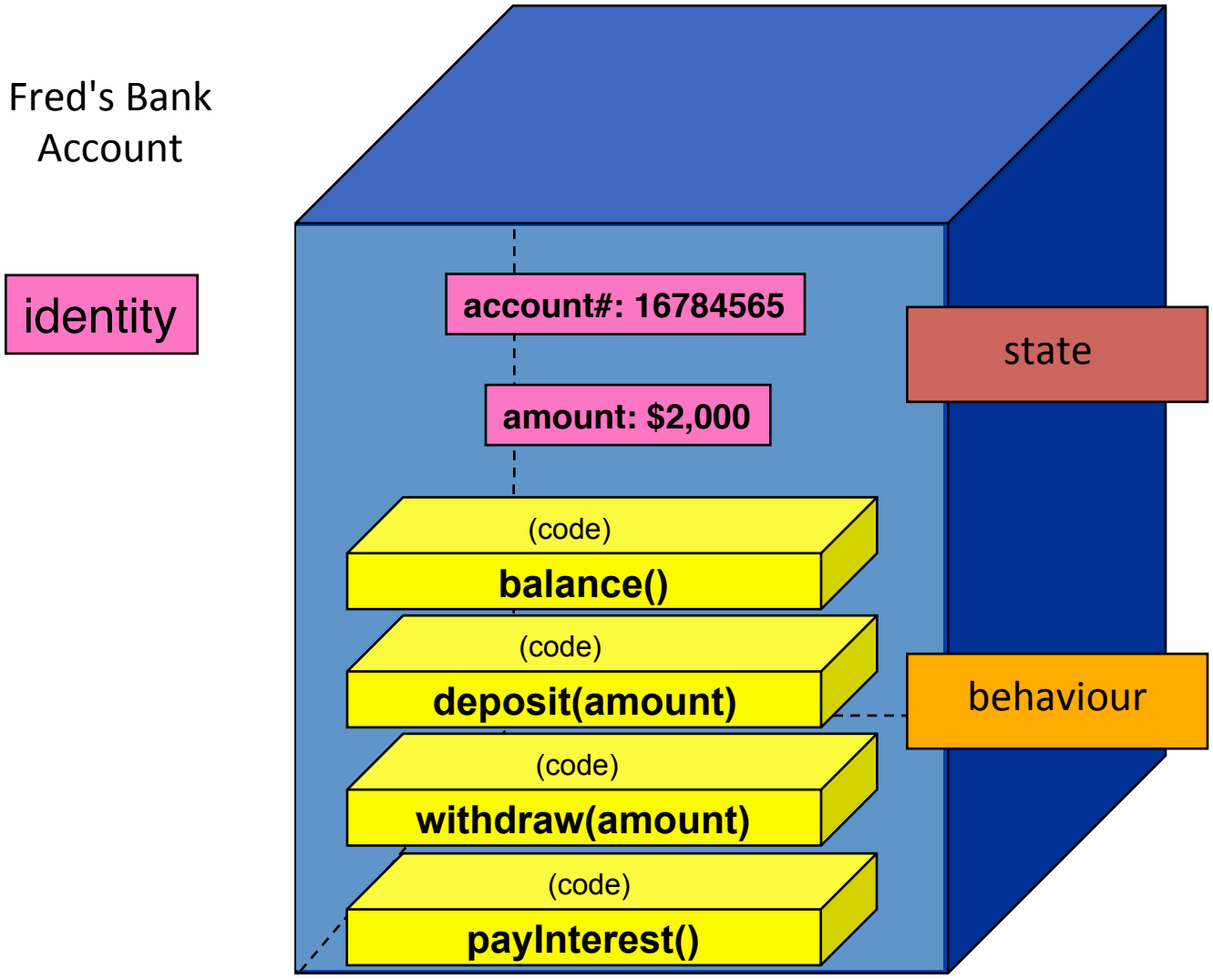
A *class* describes a group of objects having common:
– semantics – state – behaviour – relationships

- A class provides a template to create objects (i.e., it is a "factory" for objects).
  - ☞ In the UML a class is a classifier; an object is an instance.
- 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 problem domain.
  - ☞ So that it is meaningful and traceable from the real world to a model.

Object and Class



Object: Properties



Class: Attributes

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

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

- An attribute can be either a base attribute or a derived attribute
  - e.g., birthdate versus age

Class: Operation

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

Company → hire, fire, pay-dividends, …

Course → register, waive-prerequisite, …

- An operation is invoked through the object's interface and has the following properties:
  - operation signature: name of the operation (called the **selector**)
    names and types of the arguments
    type of the result value
  - visibility: public (+), private (–), protected (#), package (~)
- An operation is said to have side effects if its execution changes the state of an object (a query operation has no side effects).

Class: Method

A *method* is the implementation (code) of an operation.

- An operation is visible in the interface; the method is hidden.
  - ☞ In the UML an operation is a classifier; a method is an instance.
- polymorphic operation – An operation that can have several different methods.

class	operation
Account	**payInterest**
Savings	calculate interest on savings accounts
Checking	calculate interest on checking accounts

The calculations can be done differently.

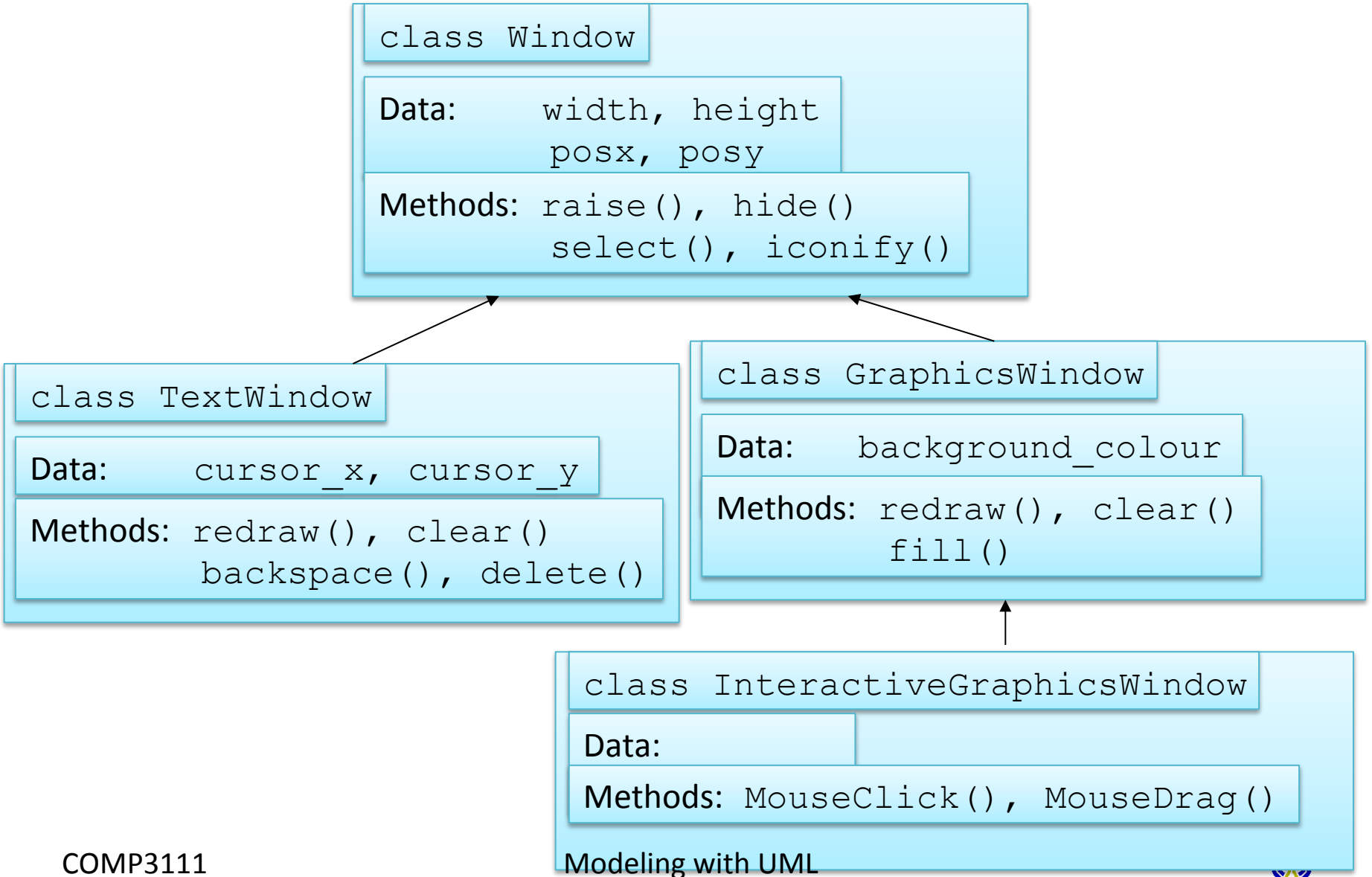
  - ☞ Useful for overriding methods in subclasses (see generalization).
- dynamic binding – Choosing the method to execute for an operation based on the object's class.

Inheritance

- Hierarchical relationships often arise between classes
- Object-oriented design supports this through *inheritance*
- An derived class is one that has the functionality of its "parent" class but with some extra data or methods
- Provides reuse and flexibility to evolve
- In C++

```
class A : public B {
…
};
```


Inheritance Example



Polymorphism

- *Polymorphism*, Greek for "many forms"
- One of the most powerful object-oriented concepts
- Ability to hide alternative implementations behind a common interface
- Ability of objects of different types to respond in different ways to a similar event
- Example
  - TextWindow and GraphicsWindow, redraw()

Polymorphism Implementation

- In C++ run-time polymorphism implemented via *virtual functions*

```
class Window {

…
  virtual void redraw();
};
```

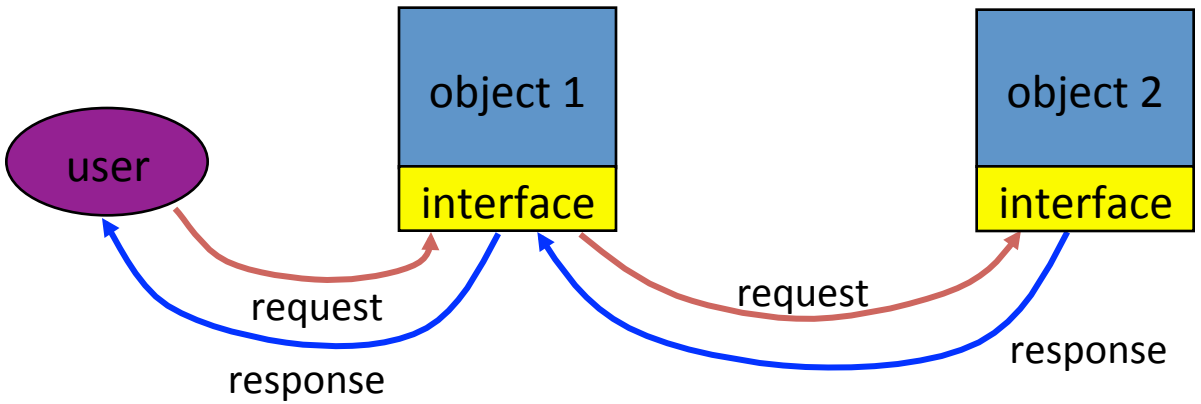

Polymorphism Example

- Class A is base class, B and C both inherit from A
- If the object is of type A then call A's func()
- If the object is of type B then call B's func()
- If the object is of type C then call C's func()

- Reusing code

```
abstract Class Sorter {
   Data sort(Data d);
}

void mytask (Sorter s){
    data = s.sort(data);
}

mytask(bubblesorter);
mytask(heapsorter);
mytask(mergesorter);
```


Encapsulation



- An object's interface encapsulates and abstracts an object thereby providing *information hiding*.
- An object provides a visible (*public*) interface, but has a hidden (*private*) implementation.

The *information hiding* provided by objects helps promote modular development.

Encapsulation Example

- Performing sorting

```
interface Sorter {
    Data sort(Data d);
}

void mytask (Sorter s){
     data = s.sort(data);
}

class BubbleSort implements Sorter{}
class HeapSort implements Sorter {};
class MergeSort implements Sorter {};

mytask(bubblesorter);
mytask(heapsorter);
mytask(mergesorter);
```

Interface

"mytask" only knows the input and the output. The implementation is not known or "encapsulated".

Let’s see how these concepts are expressed as graphic models

ATTRIBUTES: UML TEXTUAL NOTATION

«stereotype» visibility name [multiplicity]: typeExpression = initialValue
{propertyString}

propertyString → a comma separated list of properties or constraints

☞ **Only "name" is mandatory.**

Examples

+ size: area = (100,000) {readOnly}

name: string

telephone[0..2]: string

telephone[1, 3..4]: string

– salary: money {>0, <1,000,000}

You are not required to know the details of this syntax!

Which one is right?

- public field “age” of type integer
  1. public age: int;
  2. + int age;
  3. + age: int;
  4. None of the above

OPERATION: UML TEXTUAL NOTATION

«stereotype» visibility name (parameterList): returnType {propertyString}

parameterList → kind name: typeExpression = defaultValue

kind → in - pass by value
out - pass by reference (no input value; output value only)
inout - pass by reference (input and output value)

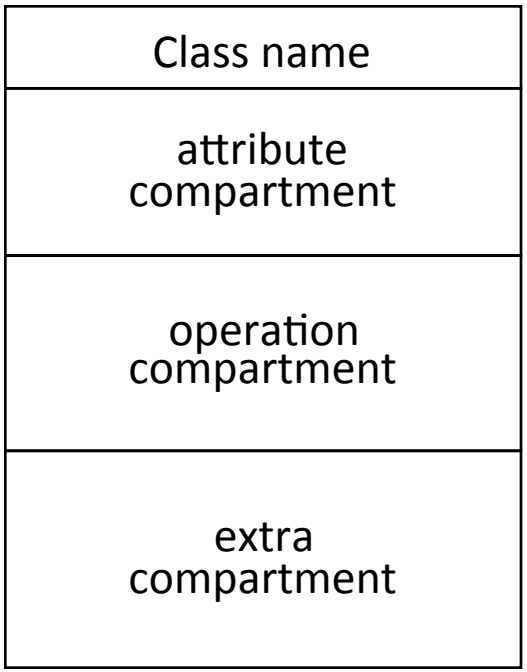
propertyString → a comma separated list of properties or constraints

isQuery → true or false (if true ® no side effects)
isPolymorphic → true or false
concurrency → sequential - callers must coordinate to ensure only one call to an object may execute at one time
guarded - multiple calls to an object may occur simultaneously, but only one is allowed to execute at a time; other calls are blocked
concurrent - multiple calls may occur simultaneously to an object and all execute concurrently

operation signature → name (parameterList): returnType (mandatory)

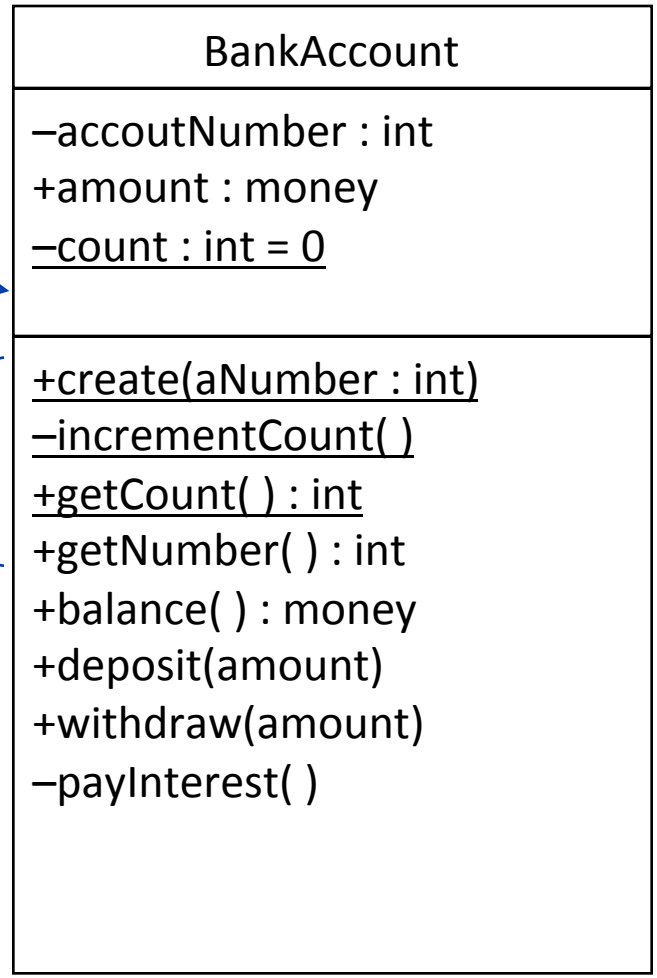
You are not required to know the details of this syntax!

CLASS: UML GRAPHICAL NOTATION



class attributes are underlined

class operations are underlined



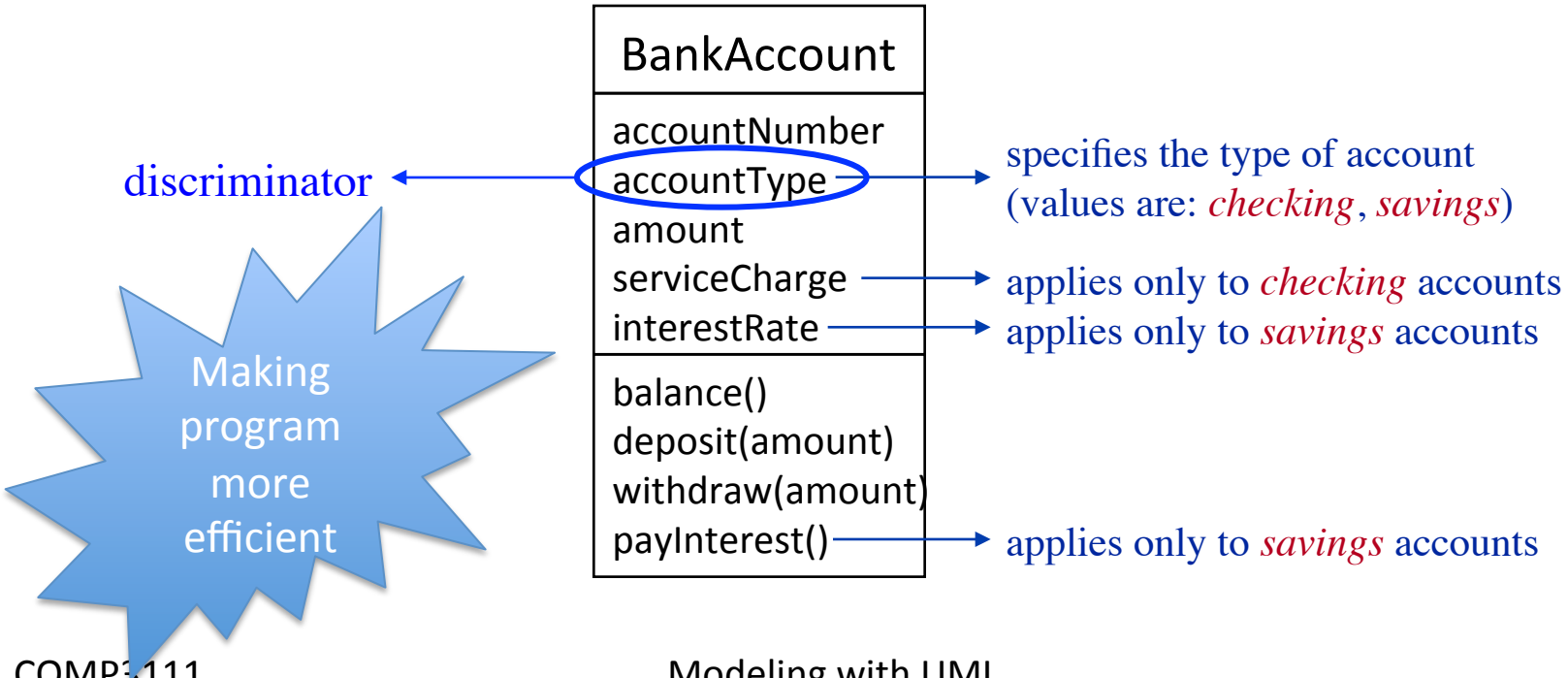
Visibility
+ public
– private
protected
~ package

The UML textual notation for attributes and operations is used within the compartments of a class.

Refactoring: Generalization

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

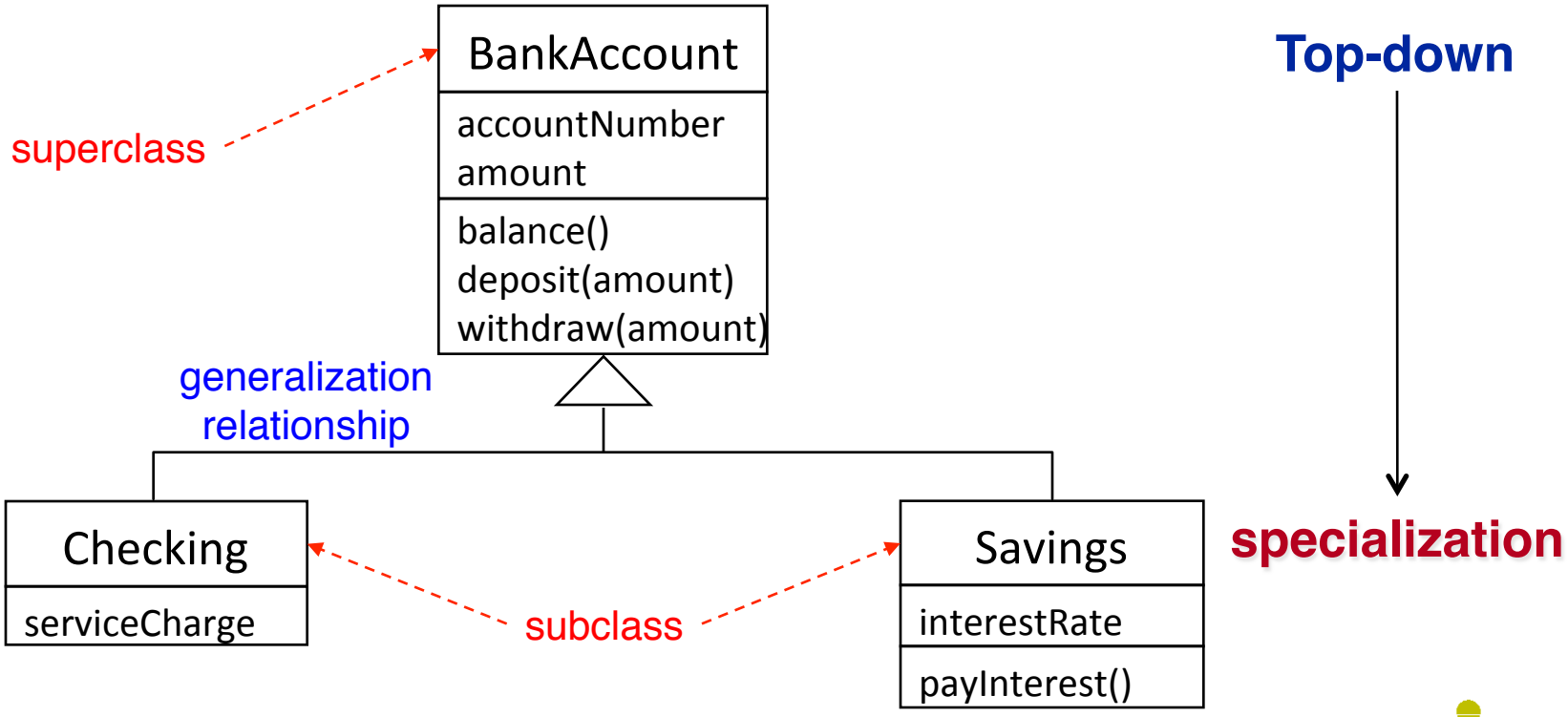
discriminator: An attribute of enumeration type that indicates which property of a class is used to create a generalization relationship.



Generalization (cont'd)

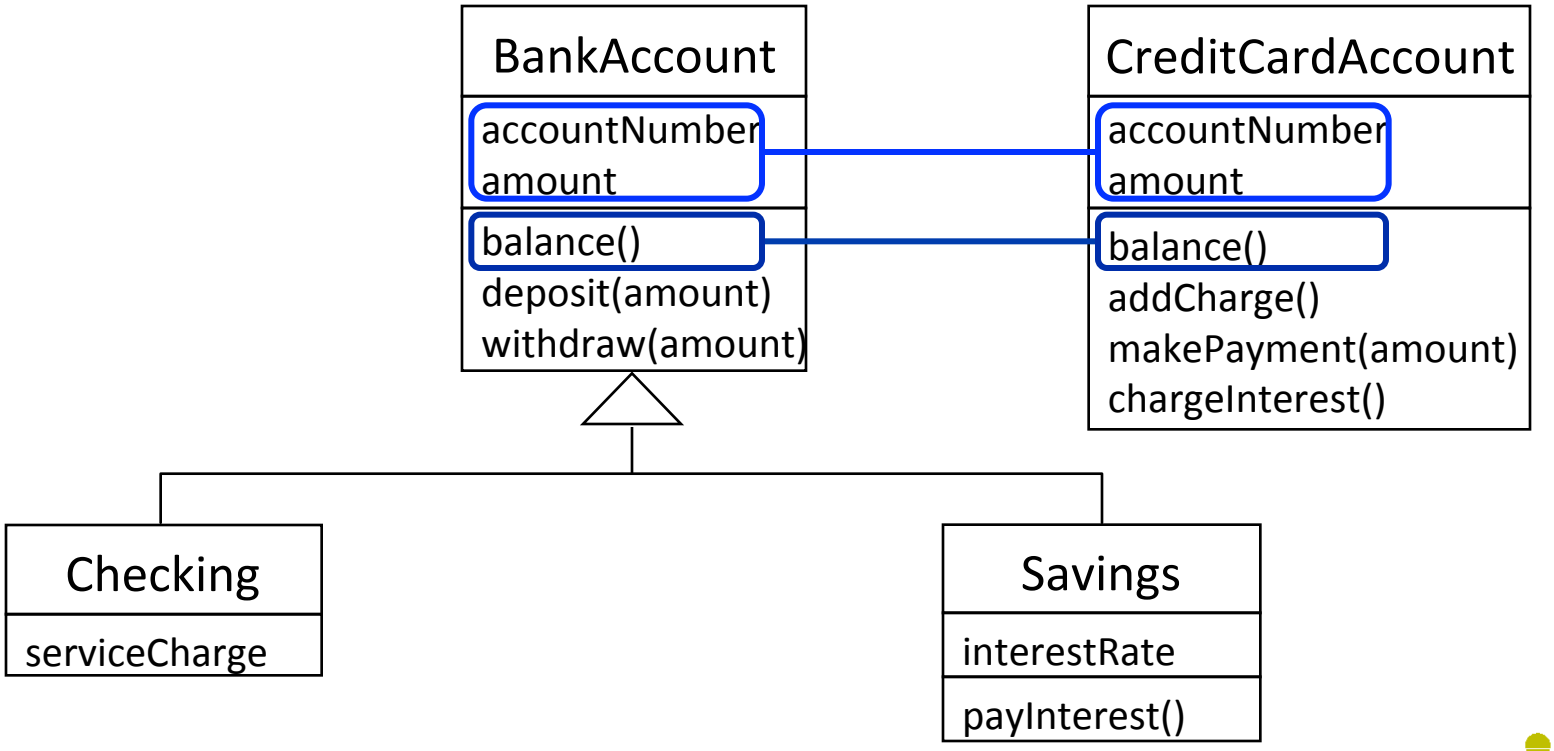
A *generalization* is a relationship between classes of the same kind.

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



Refactoring: Generalization

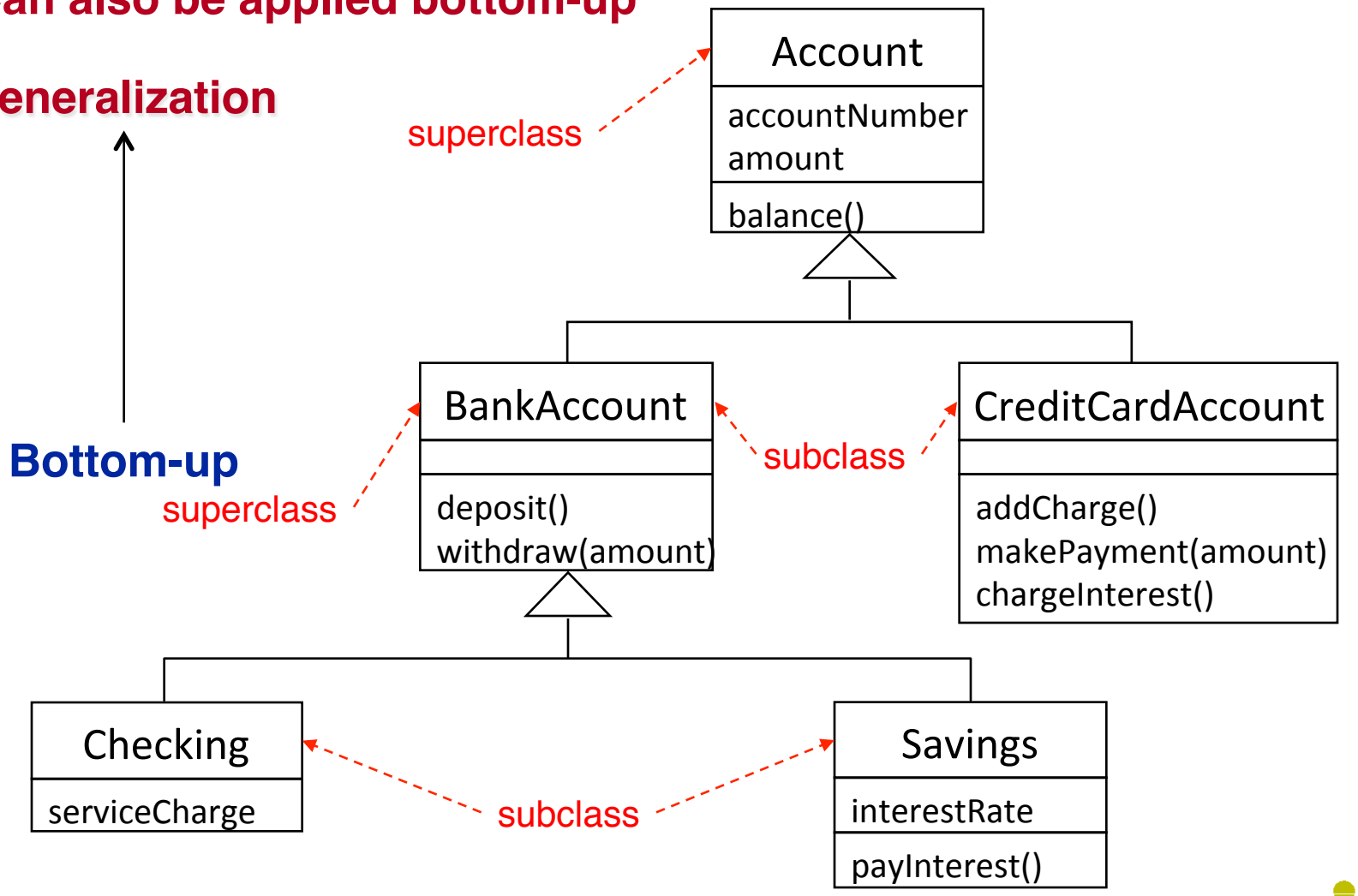
Can also be applied bottom-up



Generalization (cont'd)

Can also be applied bottom-up

generalization



Dual of generalization: Inheritance

- 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:**

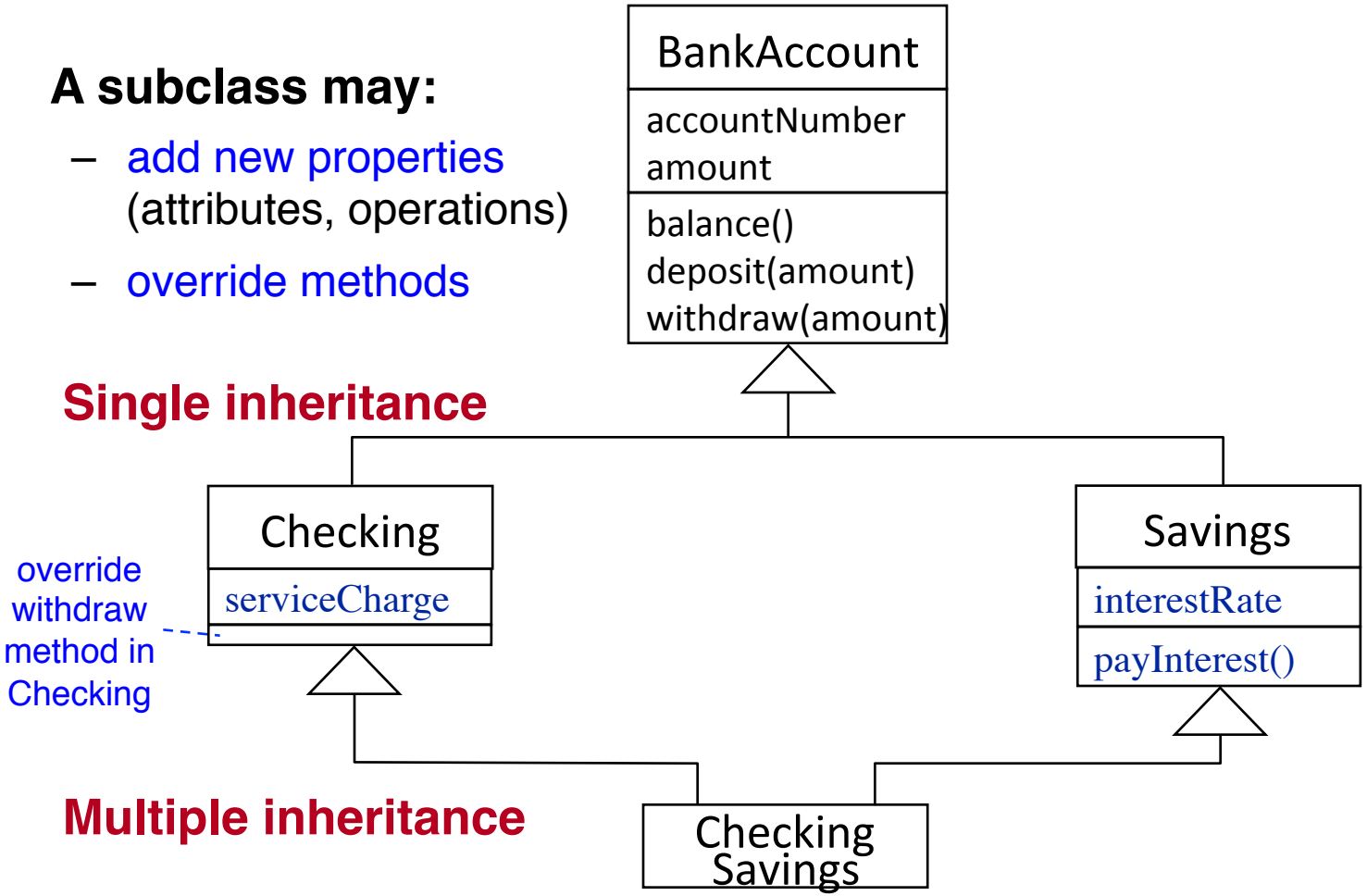
- ✓ reduces redundancy of descriptions.
- ✓ promotes reusability of descriptions.
- ✓ simplifies modification of descriptions.

☞ **An object of a superclass should be able to be substituted with an object of a subclass. (Liskov substitution principle)**

Inheritance (cont'd)

A subclass may:

- add new properties (attributes, operations)
- override methods

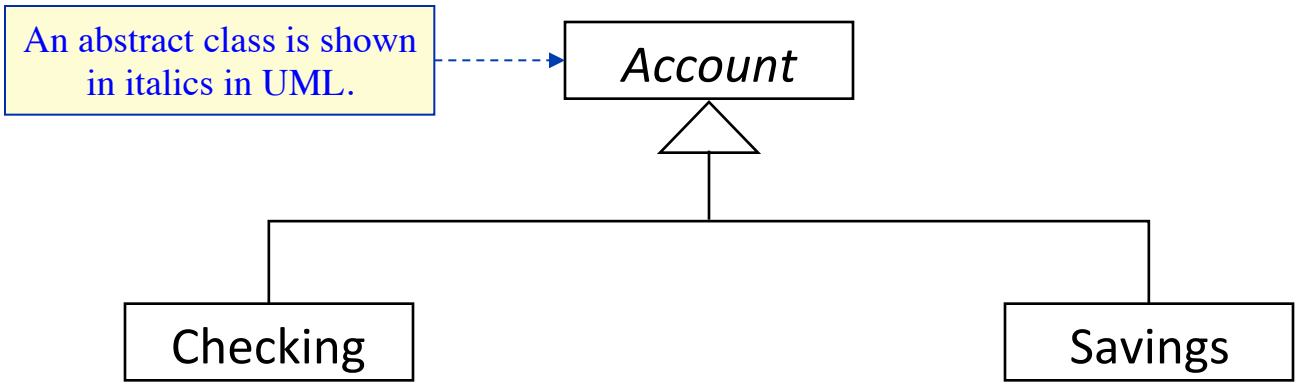


Discussion of multiple inheritance is beyond the scope of this course.

ABSTRACT CLASS

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

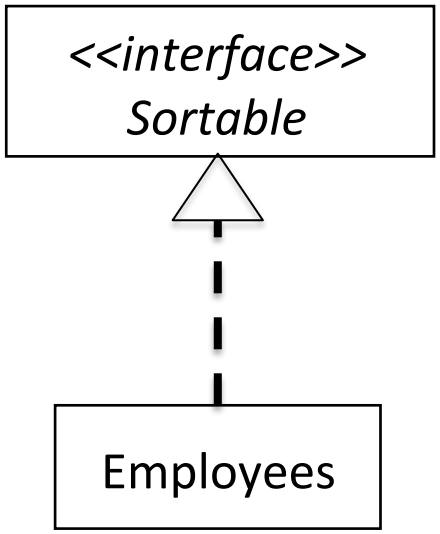
- An abstract class is used, for modeling purposes, as a *container for definitions*, but no instances of the class are of interest.

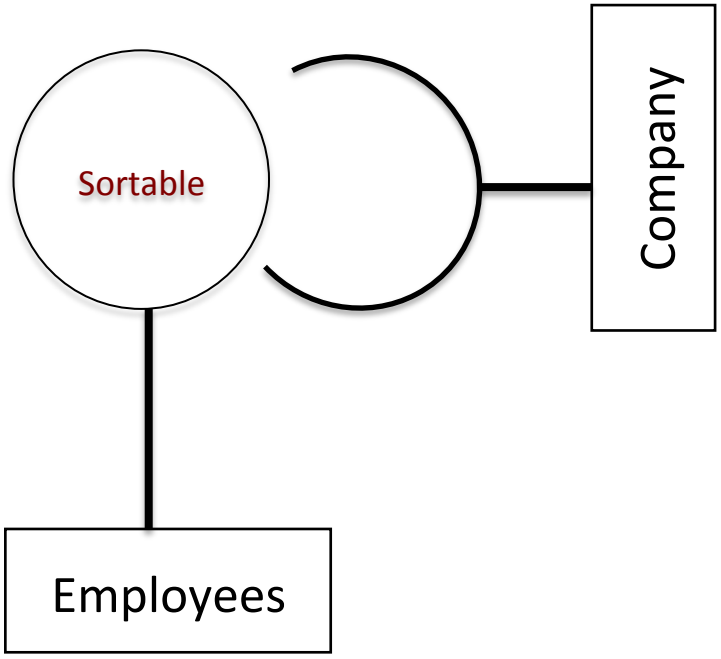


Note: Operations may also be abstract → no method specified.

Interface

An *interface class* is a class that has no object-specific data.



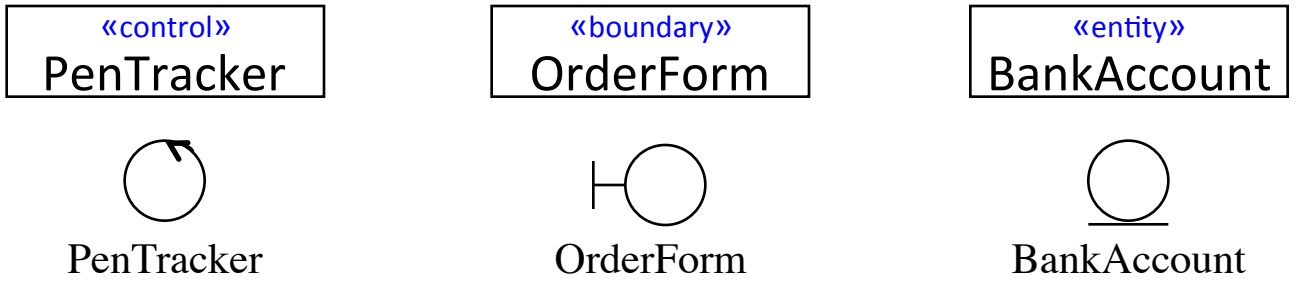


Stereotype

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.**

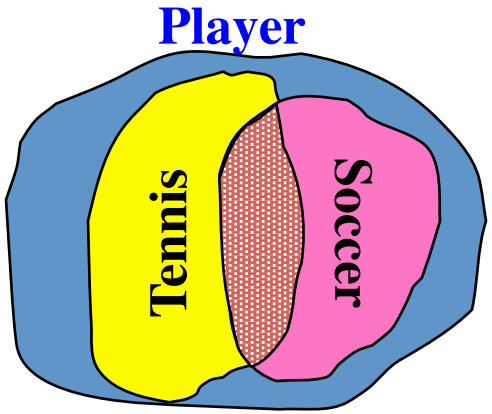
Example: We can define different kinds of classes that are useful for modeling.

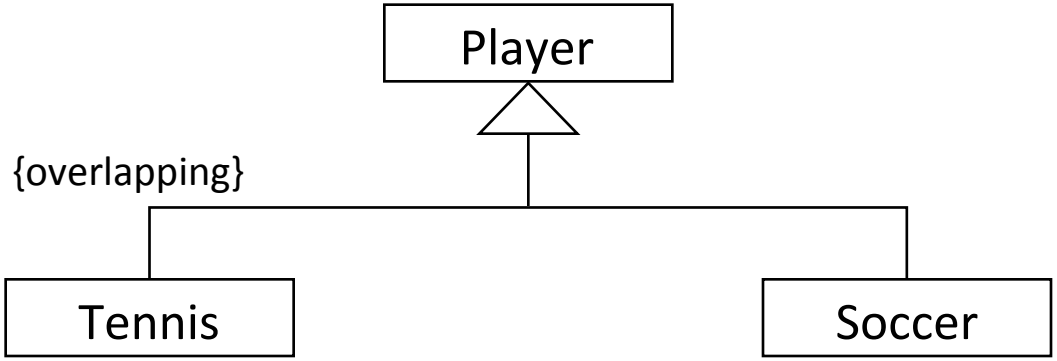


☞ **Special icons can be used for each stereotype that improve modeling clarity.**

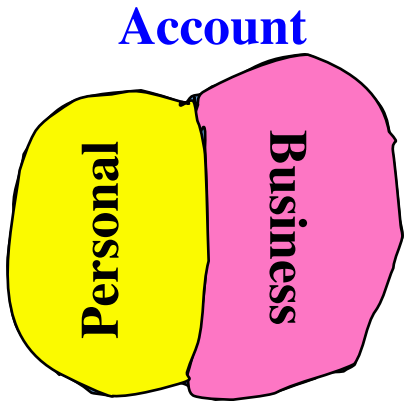
Generalization: Coverage

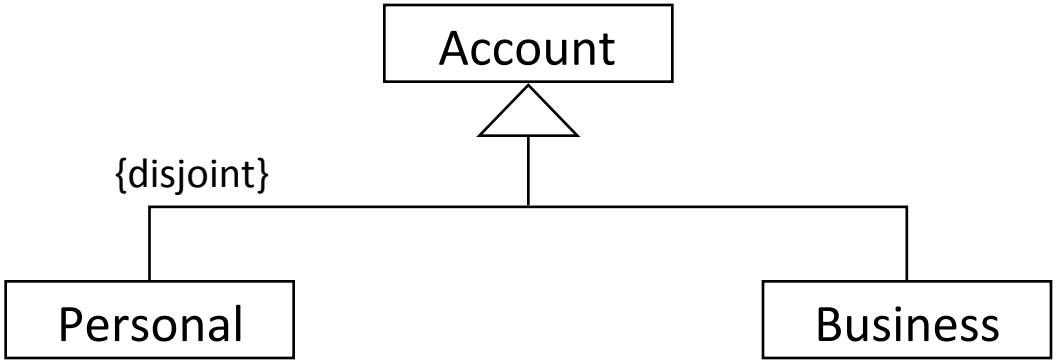
overlapping - A superclass object can be a member of more than one subclass.





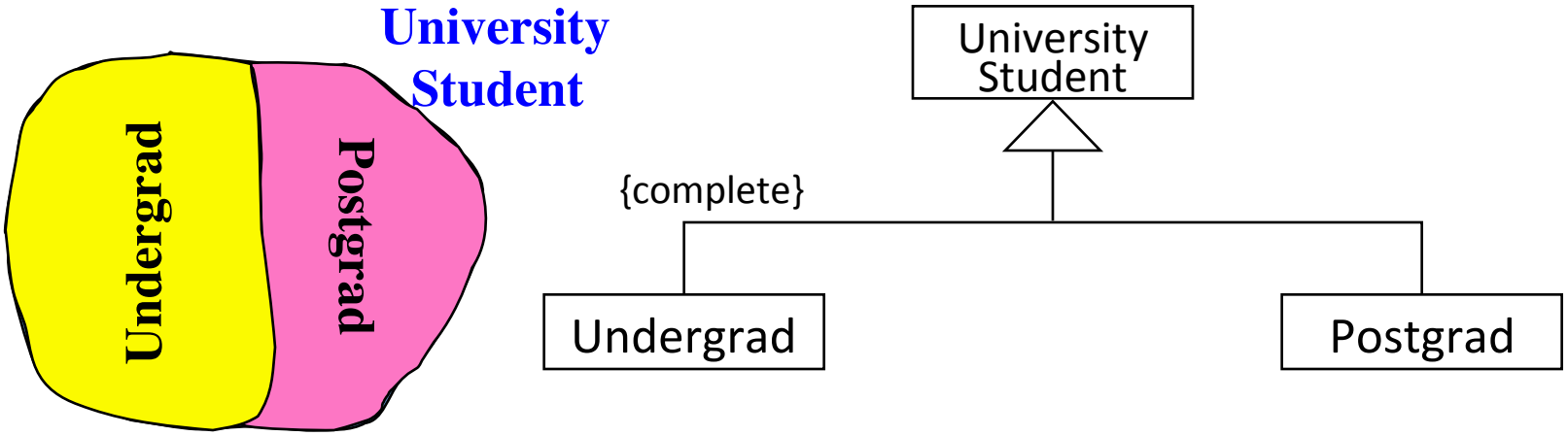
disjoint - A superclass object is a member of at most one subclass.



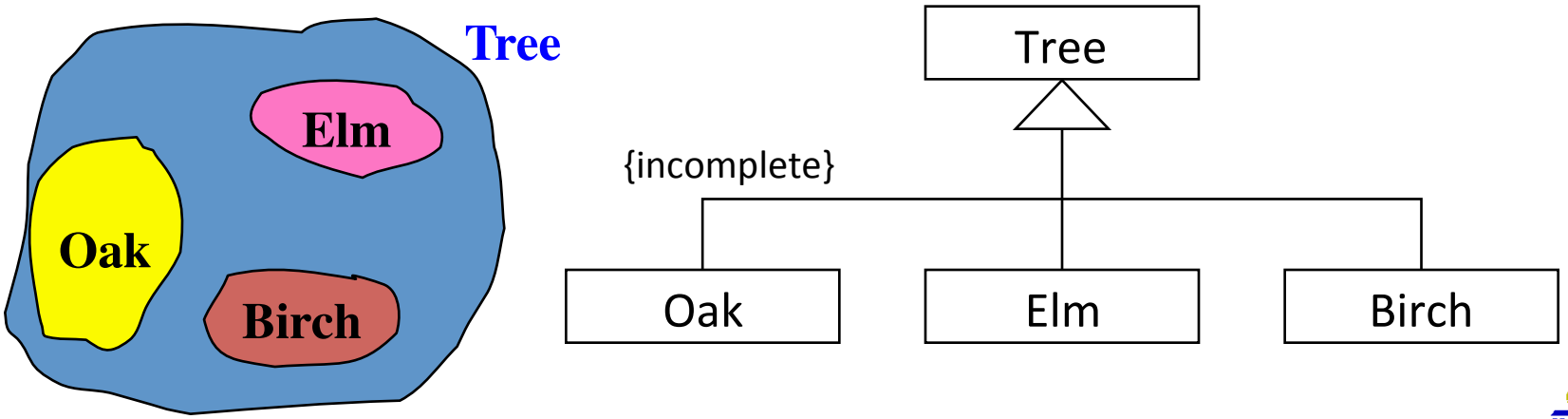


Coverage (cont'd)

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

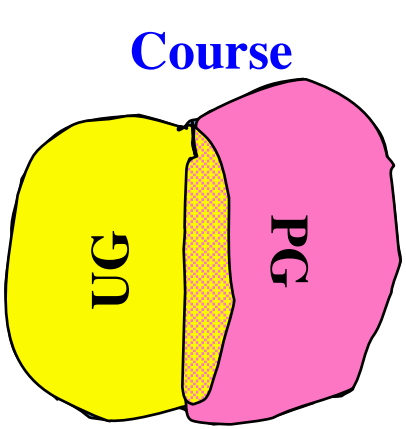


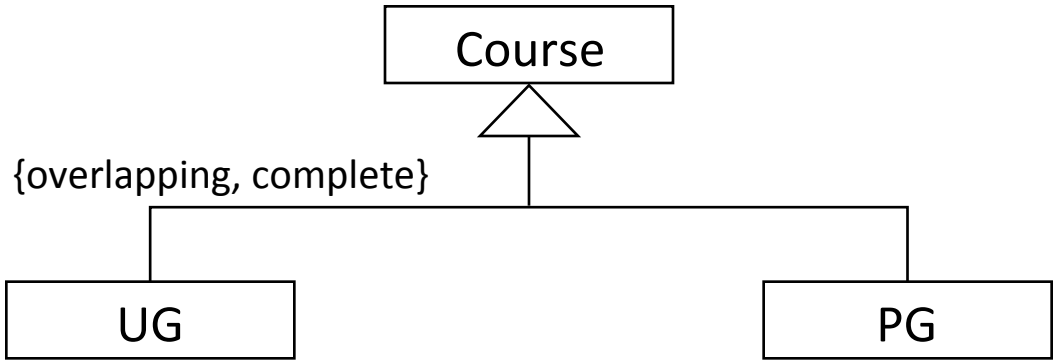
incomplete - Some superclass object is not a member of any subclass.



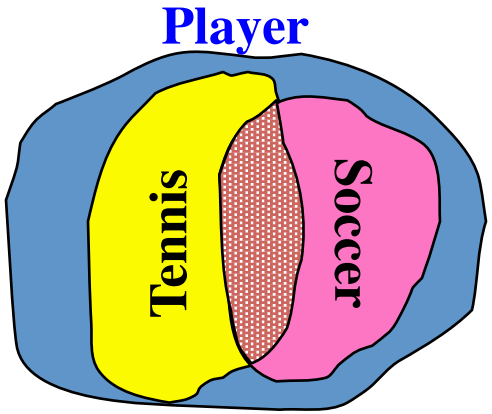
Possible coverage types

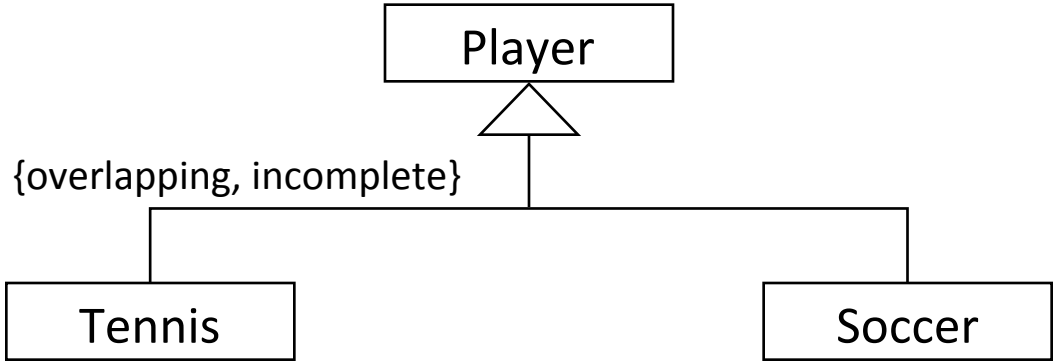
1. overlapping, complete





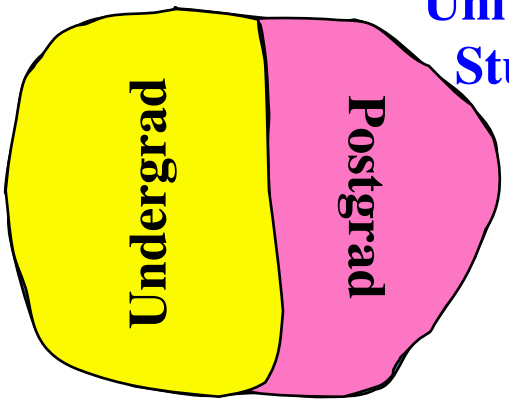
2. overlapping, incomplete

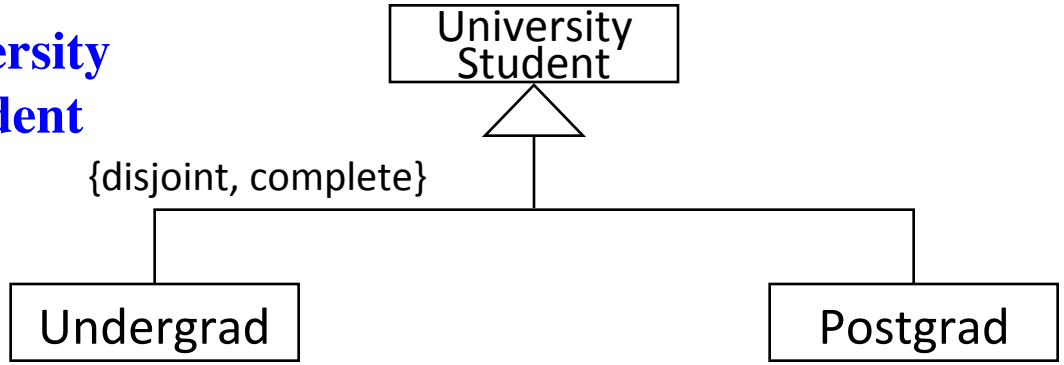




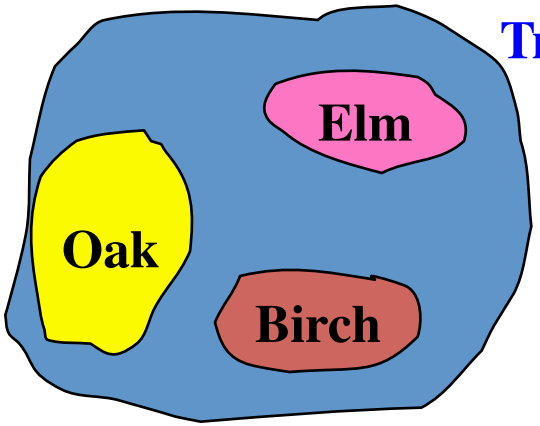
Possible coverage types (cont'd)

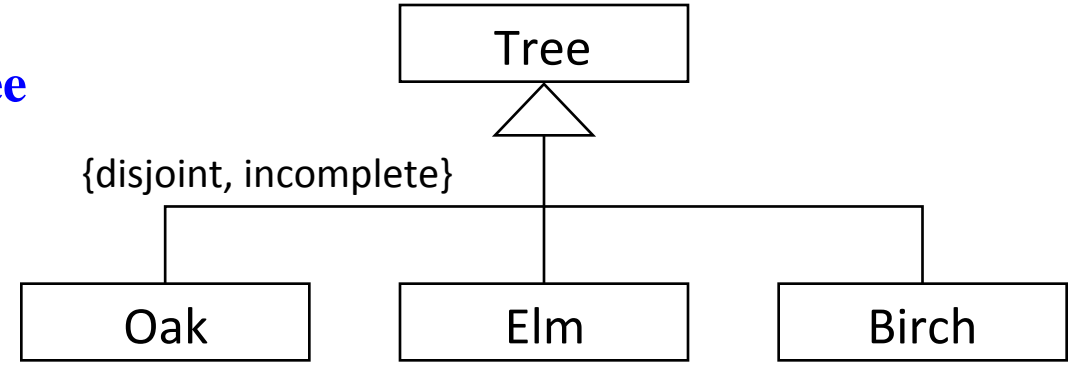
3. disjoint, complete



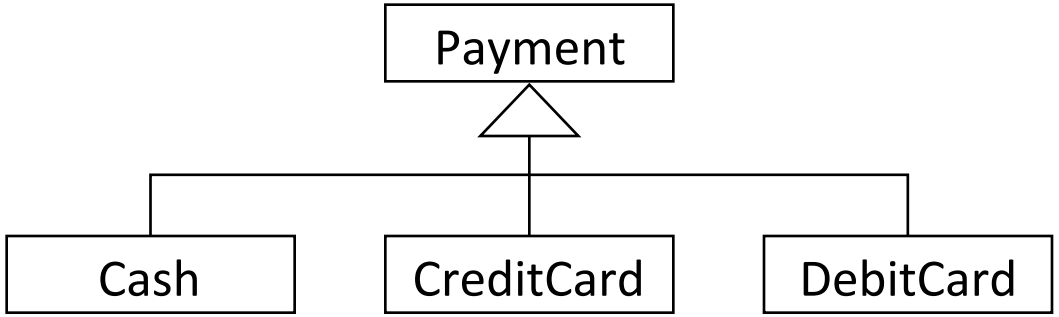


4. disjoint, incomplete

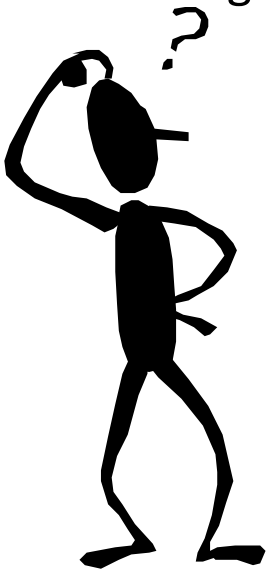




QUESTION?

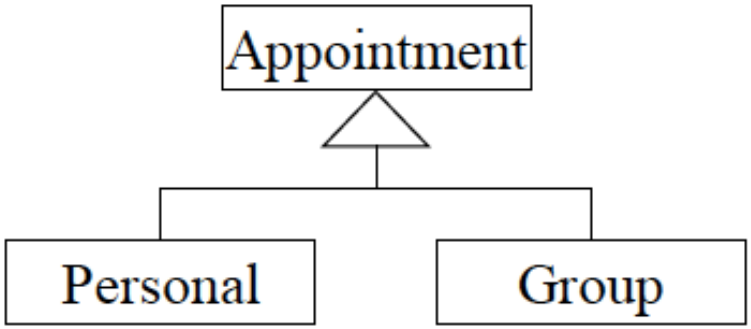


The coverage of the generalization shown above is:

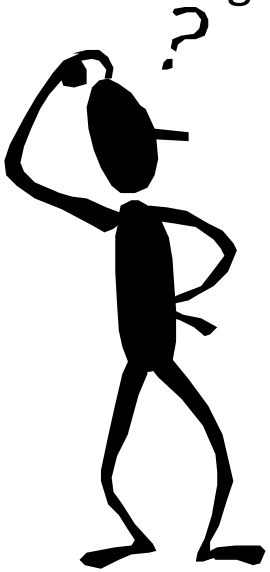


- overlapping, complete
- disjoint, complete
- overlapping, incomplete
- disjoint, incomplete

Project Question



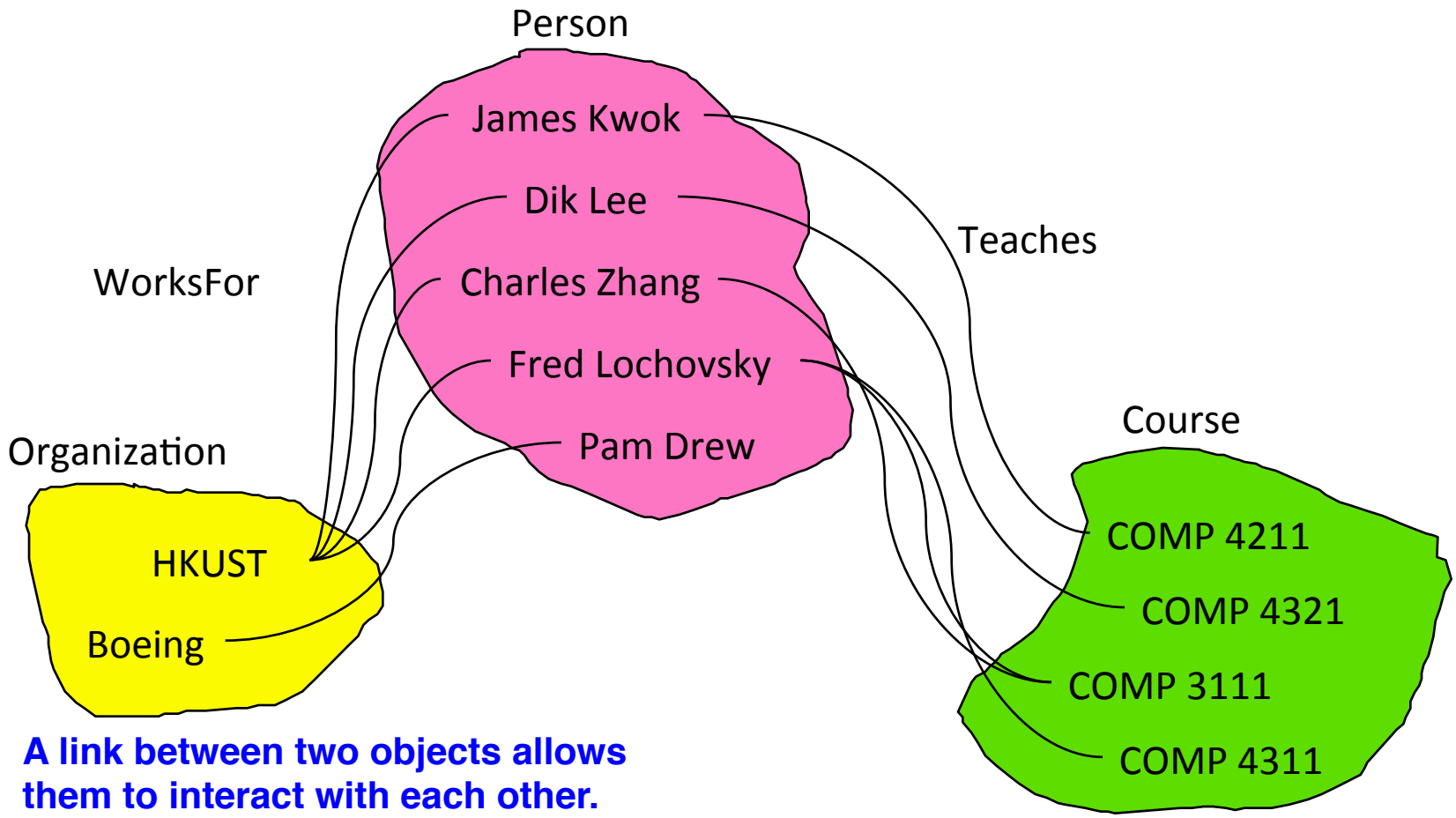
The coverage of the generalization shown above is:



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

LINK

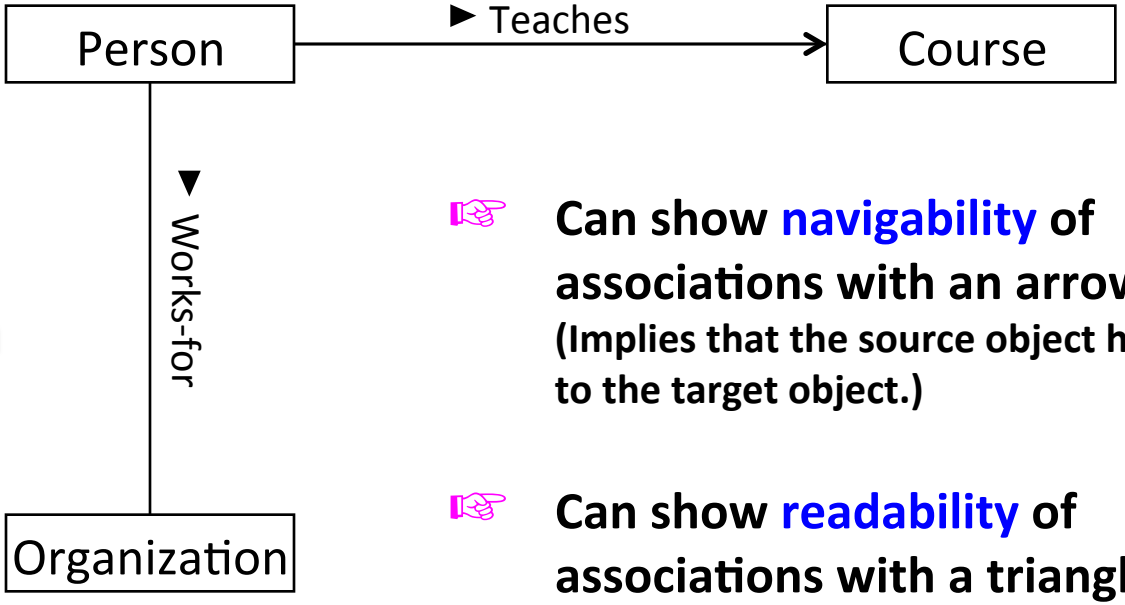
A *link* is a physical or conceptual relationship between objects.



☞ **A link between two objects allows them to interact with each other.**

Association

An *association* describes a group of links of the same kind (with common semantics).



class diagram

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

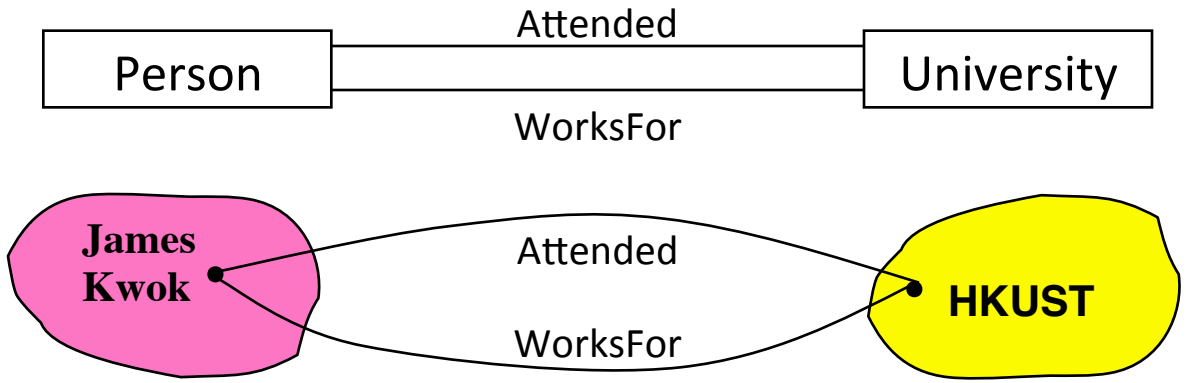
☞ **Can show readability of associations with a triangle.**

☞ An association is a classifier; a link is an instance.

☞ Conceptually, associations are inherently bi-directional.

Association (cont'd)

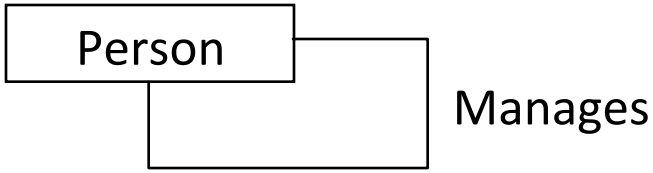
- There can be several associations between the same two classes.



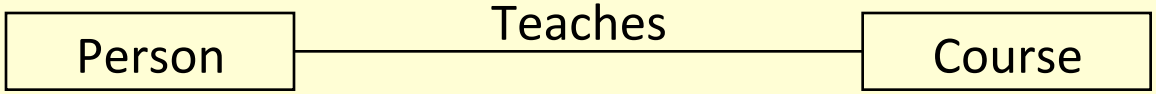
- Or even with the same class.

Association Degree

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

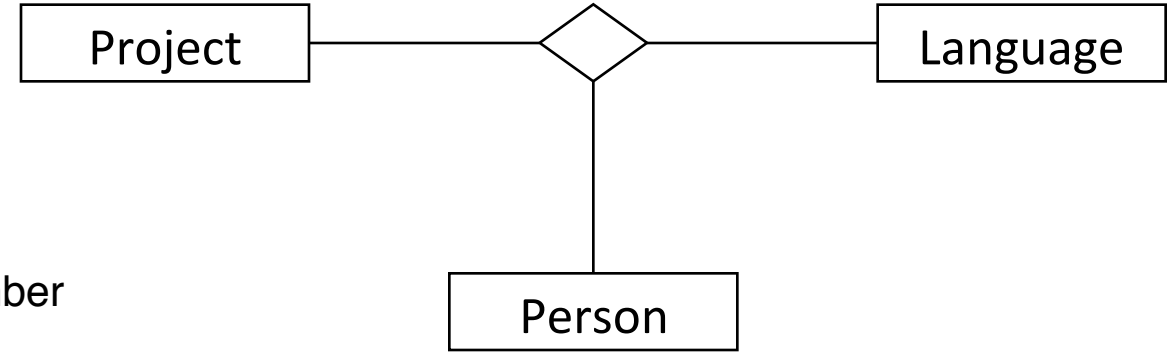


- **binary**
  relates two classes



- **ternary**
  relates three classes

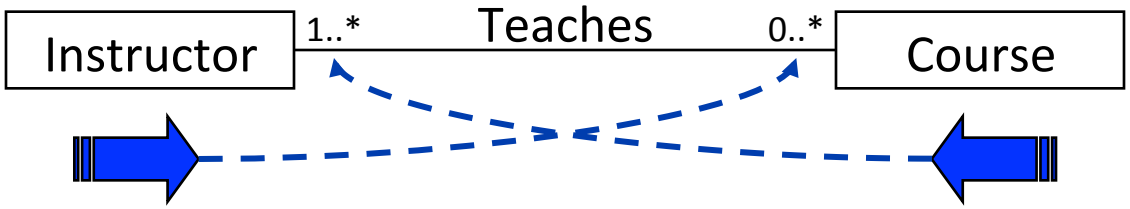
- **n-ary**
  relates any number of classes



In practice, the vast majority of associations are binary!

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 instructor, how many courses can he teach?

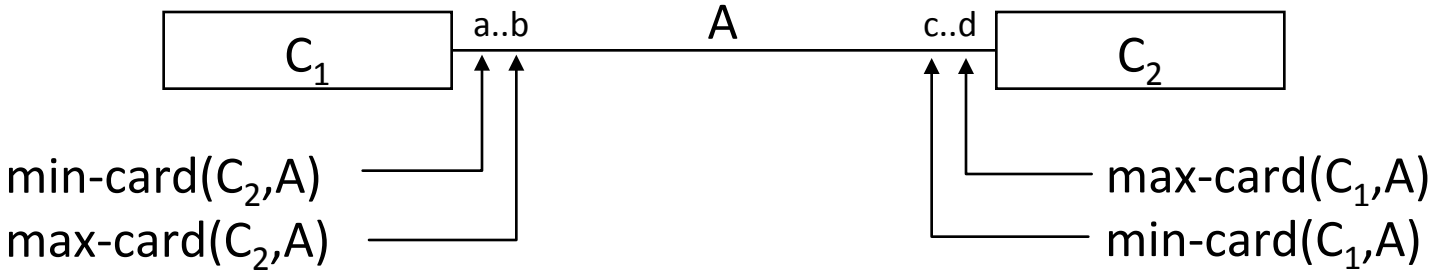
- An instructor does not have to teach any course but may teach more than one course in a given semester.

For a given course, how many instructors can teach it?

- Each course must be taught by one instructor but may have many.

Multiplicity is a *real-world constraint*!

Multiplicity (cont'd)



minimum cardinality (min-card)

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

min-card(C_1,A) = 0 → optional participation *(may not be related)*

min-card(C_1,A) > 0 → mandatory participation *(must be related)*

maximum cardinality (max-card)

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

Multiplicity (cont'd)

special cardinalities:

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

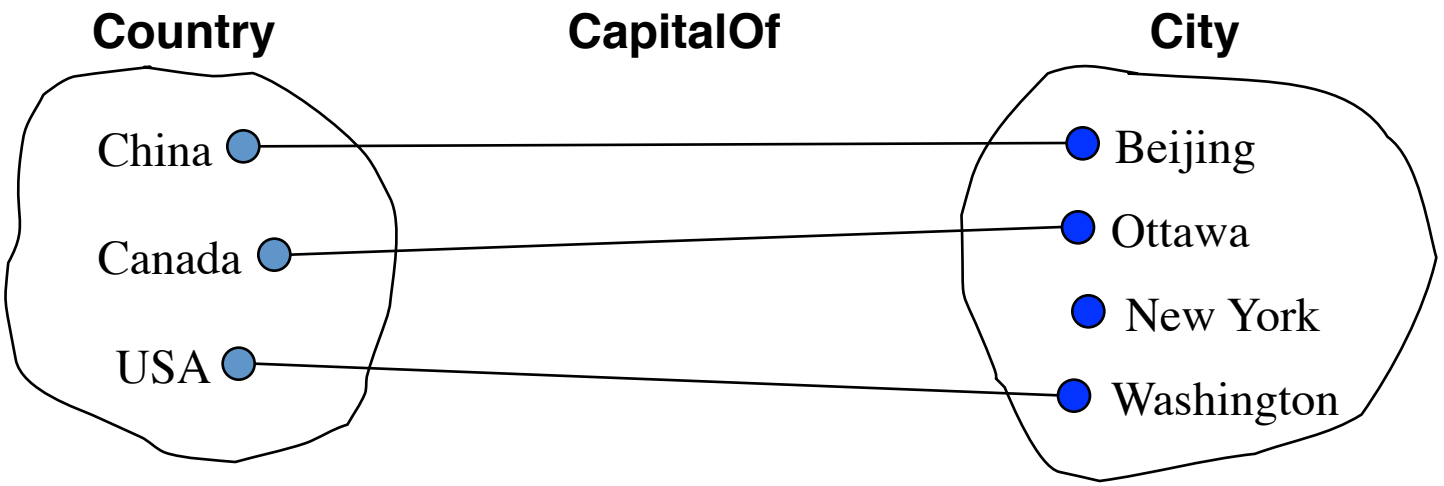
min-card = 1 and max-card = 1 → can use 1 by itself

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

Multiplicity(cont'd)

max-card(C1,A) = 1 and max-card(C2,A) = 1

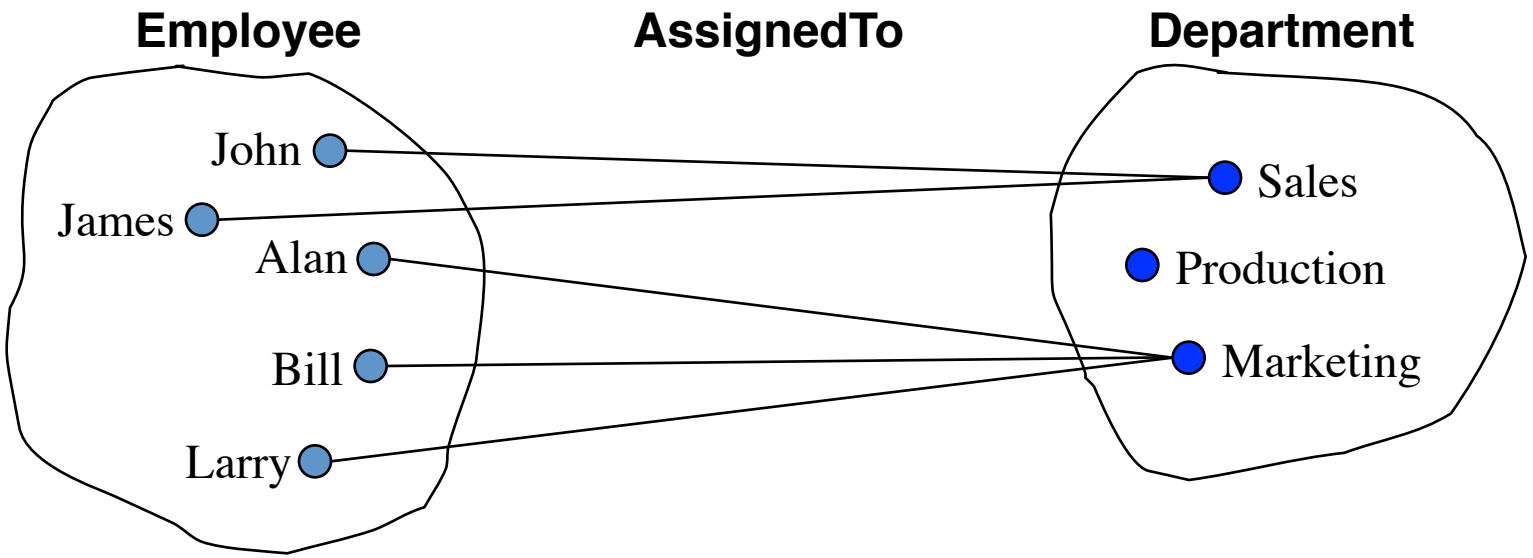
☞ one-to-one association (1:1)



Multiplicity (cont'd)

max-card(C1,A) = 1 and max-card(C2,A) = *

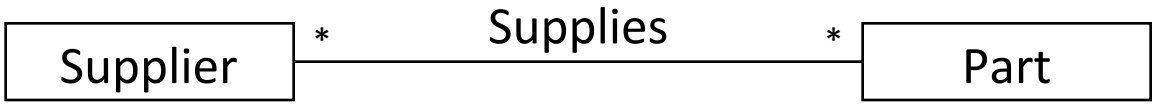
☞ one-to-many association (1:N)

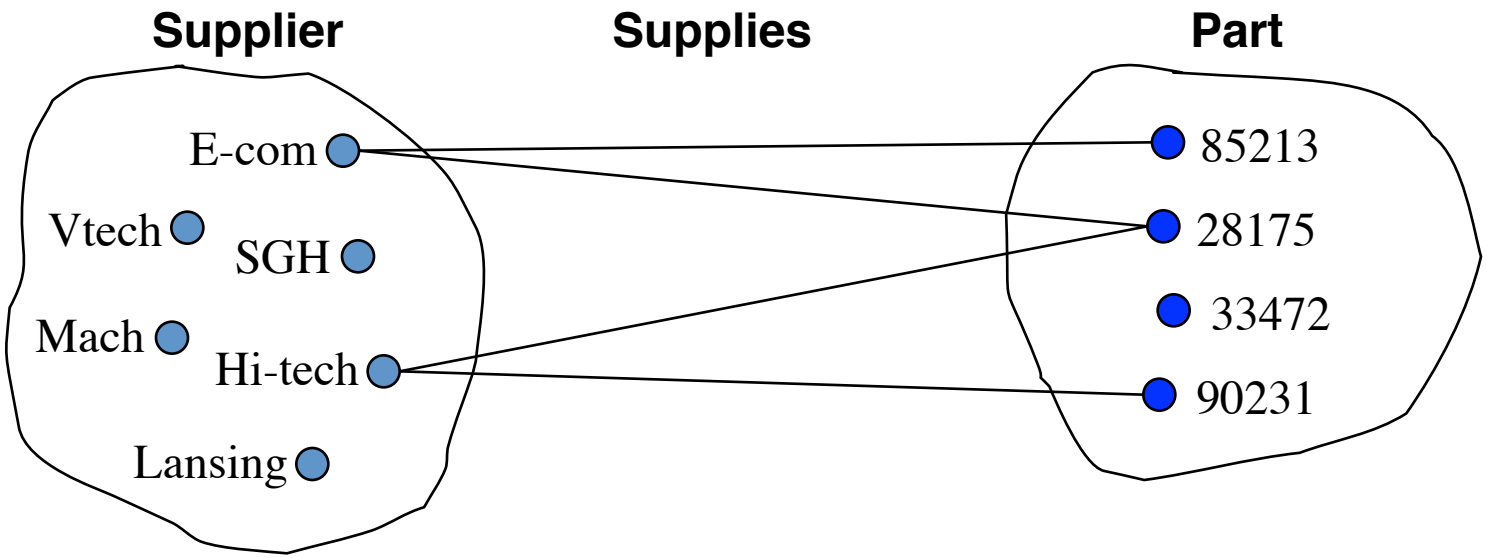


Multiplicity (cont'd)

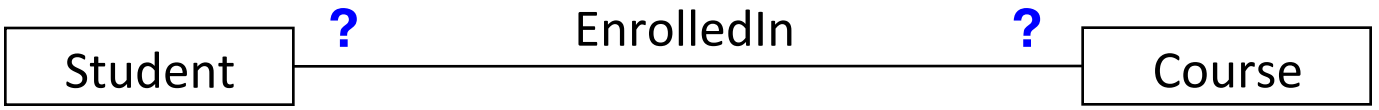
max-card(C1,A) = * and max-card(C2,A) = *

☞ many to many association (N:M)





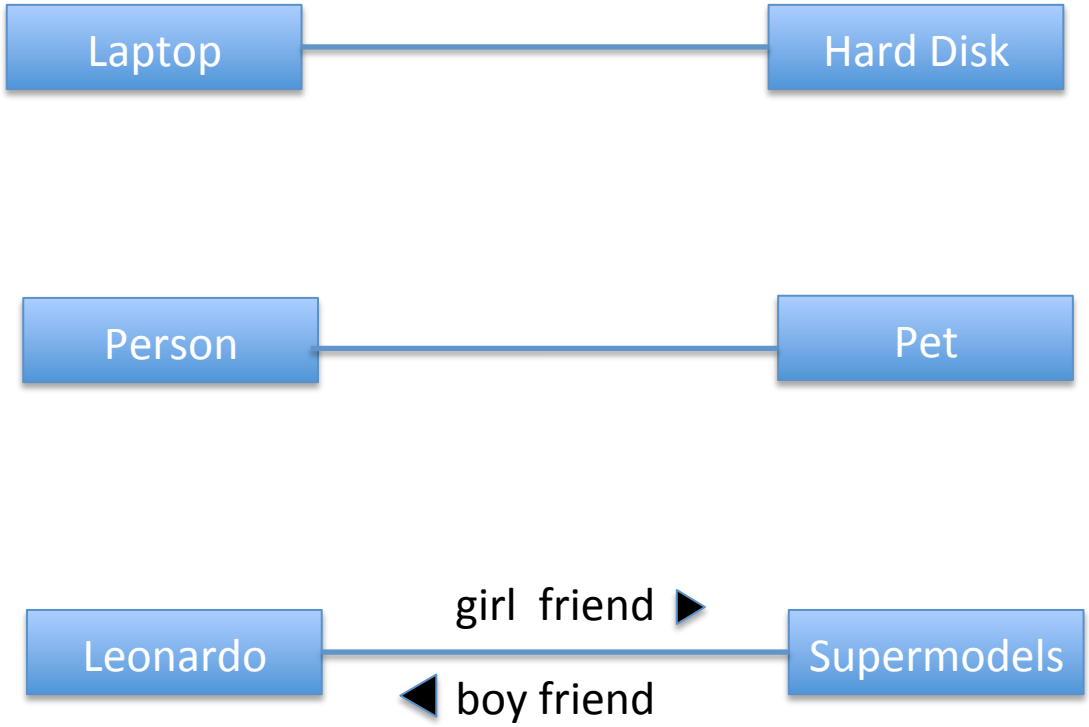
QUESTION?



- 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.

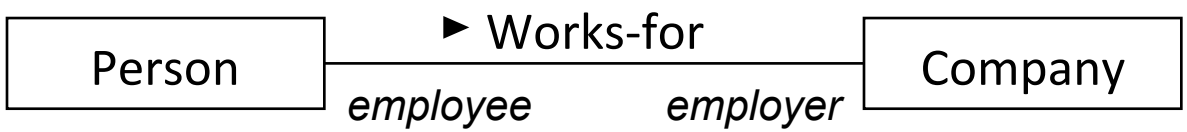
Direction? Multiplicity?

A. * B. 1..* C. 1



Role

A *role* describes how one end is involved in an association.

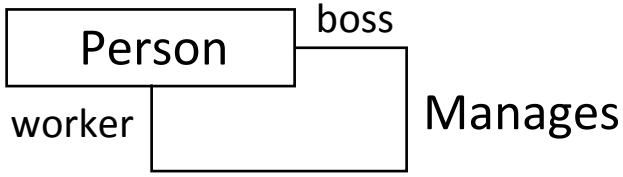


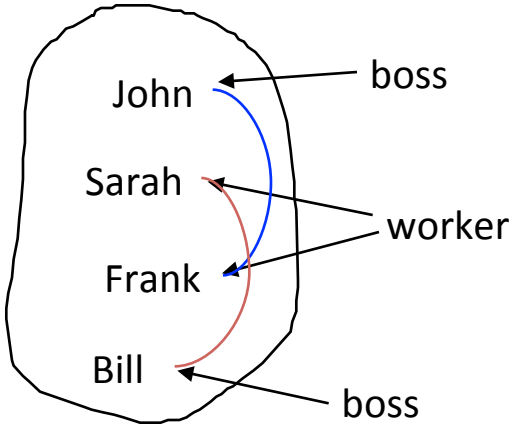
For unary and binary associations there are two roles.

Role names can be used to navigate an association.

Role (cont'd)

A *role* describes how one end is involved in an association.





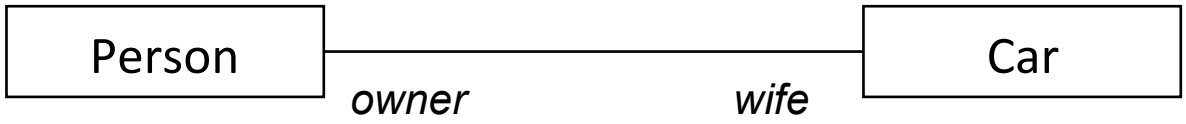
Who manages who?

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

Roles and variable names

```
class Person{
  String name = “David”;
  Car wife;
}
```

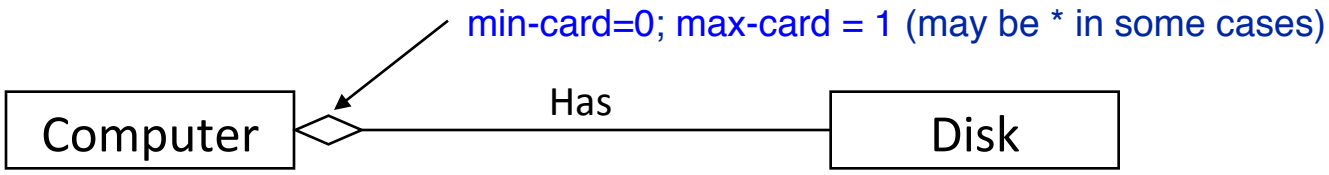
```
class Car{
   Person owner;
}
```



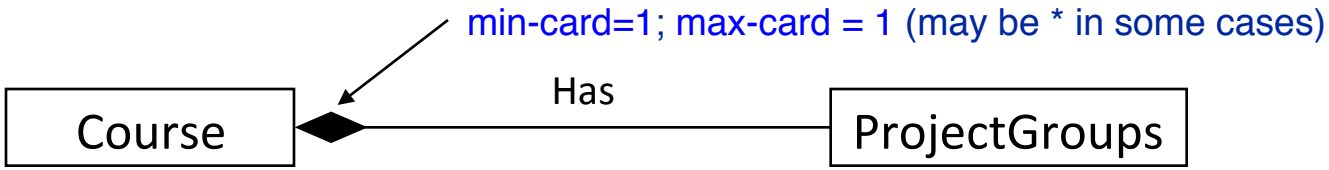
Aggregation/Composition

- 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. [◇ adornment]



☞ A component may not exist independent of the aggregate object of which it is a part → composition. [◆ adornment]



When to use?

- 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)?
- Are some attribute values propagated from the whole to all or some of the parts?

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! (When in doubt, use association!)

Composition and Aggregation (Prof. Zhang's secret)

Composition The "big" object ends the "small" objects also end

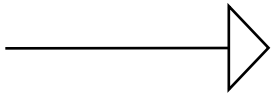
Aggregation The "big" object ends the "small" objects can join other "big" objects.

Exercises:

1. Jacket and pockets;
2. Book and pages;
3. Computer and keyboard;
4. Car and tires;

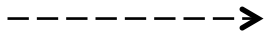
UML Relationships

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



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

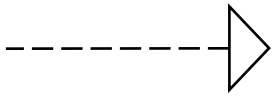
dependency relates classes whose behaviour or implementation affect other classes



flow — relates two versions of an object at successive times

usage — shows that one class requires another class for its correct functioning

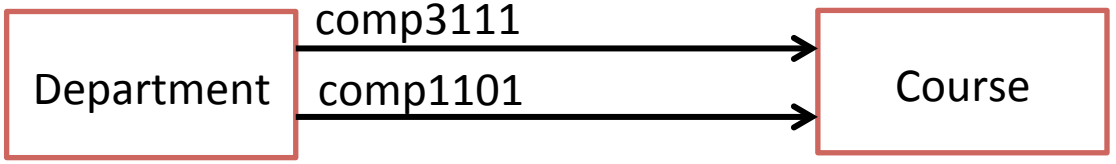
realization relates a specification to its implementation (e.g., an interface to the classes that implement it)



Association and Dependency (Prof. Zhang's secret)

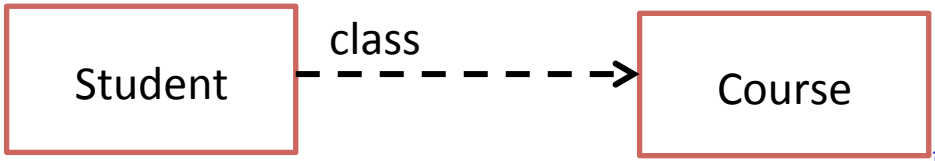
association One object has the other as a field

```
Department {
   Course comp3111;
   Course comp1101;
}
```

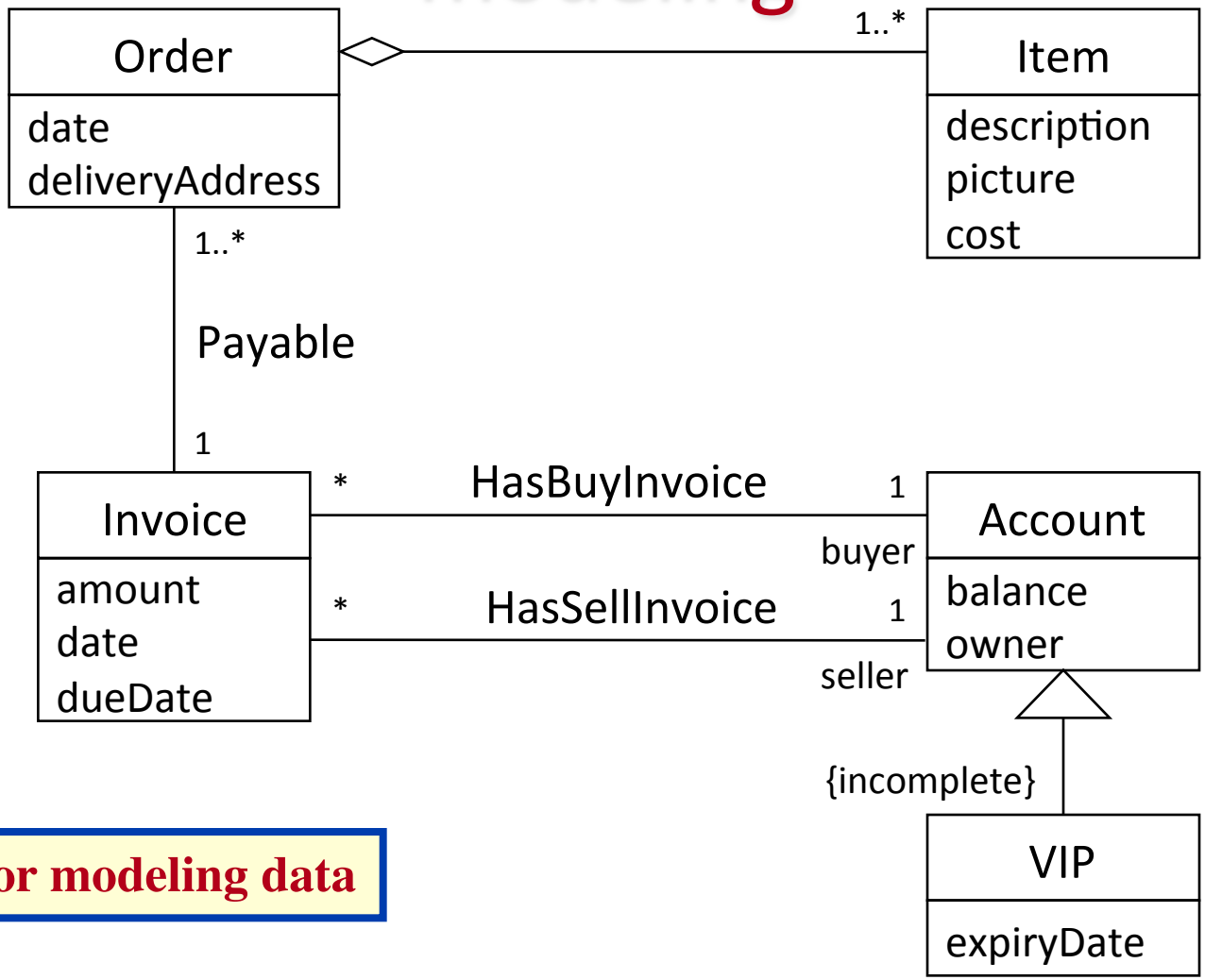


dependency One object has the other as method parameters

```
Student {
   void register( Course class){
    }
}
```

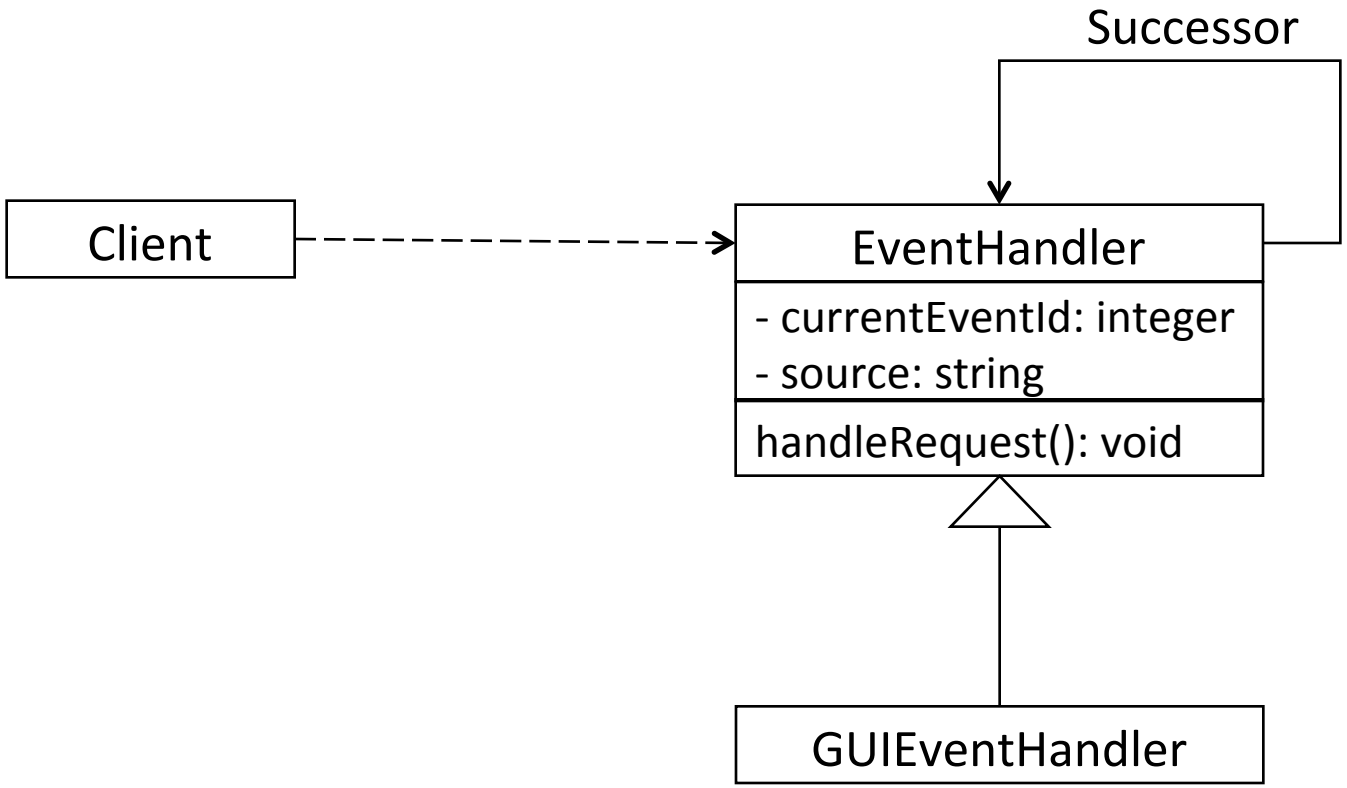


Class diagram example: Static modeling



For modeling data

Class diagram example: dynamic modeling



For modeling programs

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 *should not appear* in a class diagram. Such attributes should be either deleted or replaced with relationships. All such attributes conveniently have names ending in ID.

Persons, companies or banks may own cars. The car owner ID represents either the person, company or bank that owns the car. A car may have only one owner (person, company or bank). A car may have no car loan or multiple car loans. A car 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 car loan customer type indicate whether the car owner/loan holder is a person, company or bank.

Construct a class diagram in which all OIDs are either deleted or replaced with relationships. Use associations and generalizations as necessary to represent the relationships. 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
personID name age address

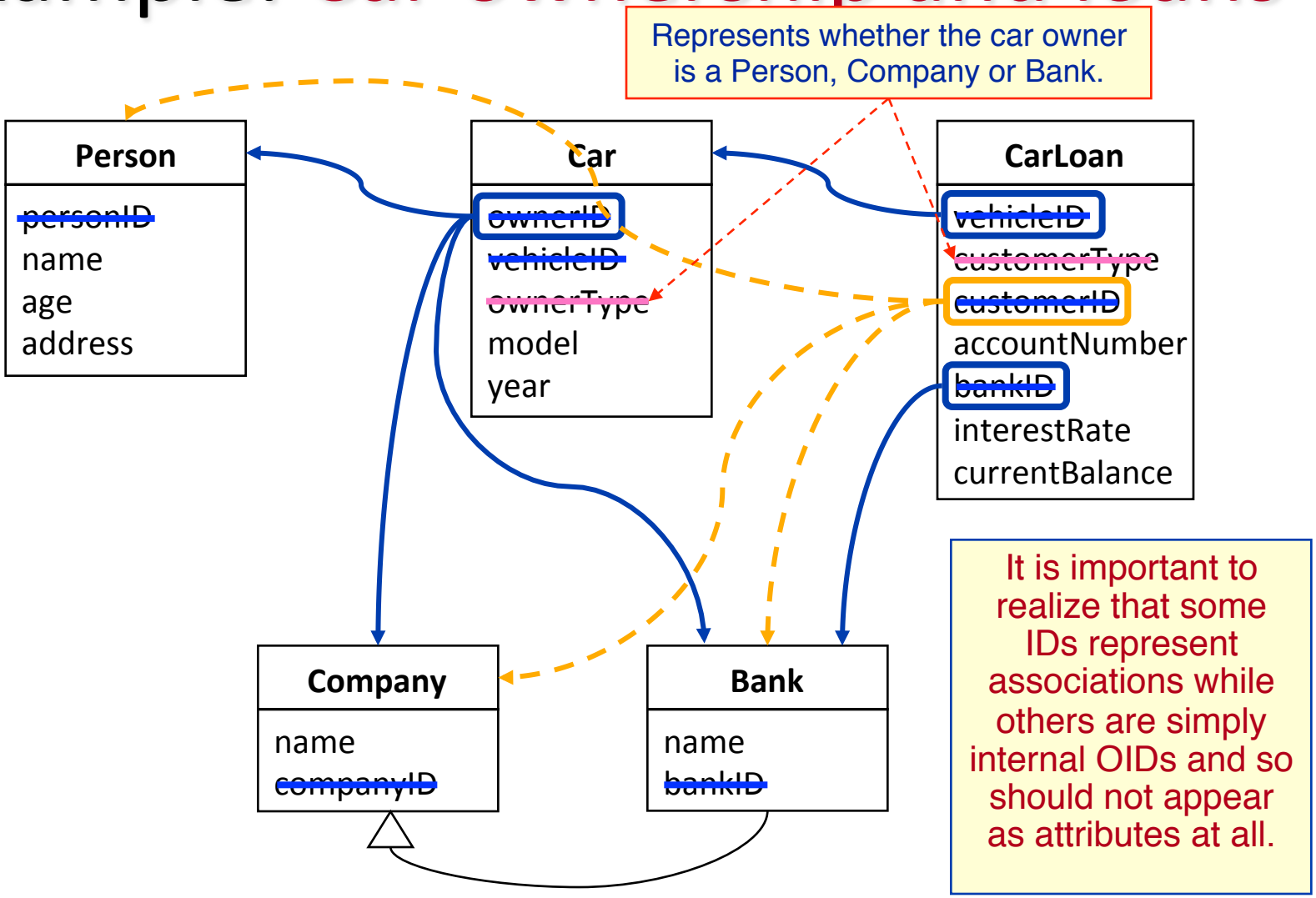
Car
ownerID vehicleID ownerType model year

CarLoan
vehicleID customerType customerID accountNumber bankID interestRate currentBalance

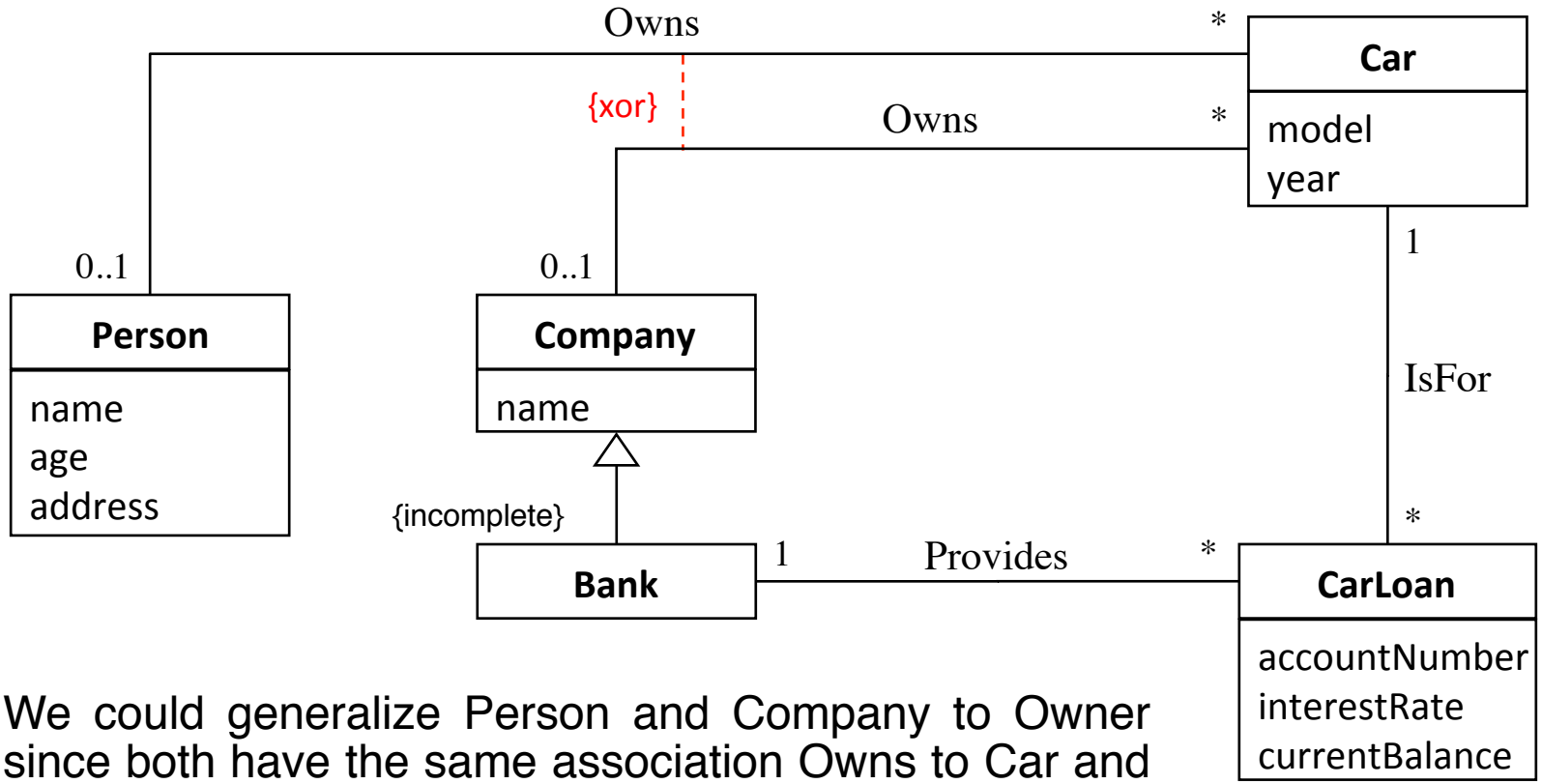
Company
name companyID

Bank
name bankID

Example: Car ownership and loans

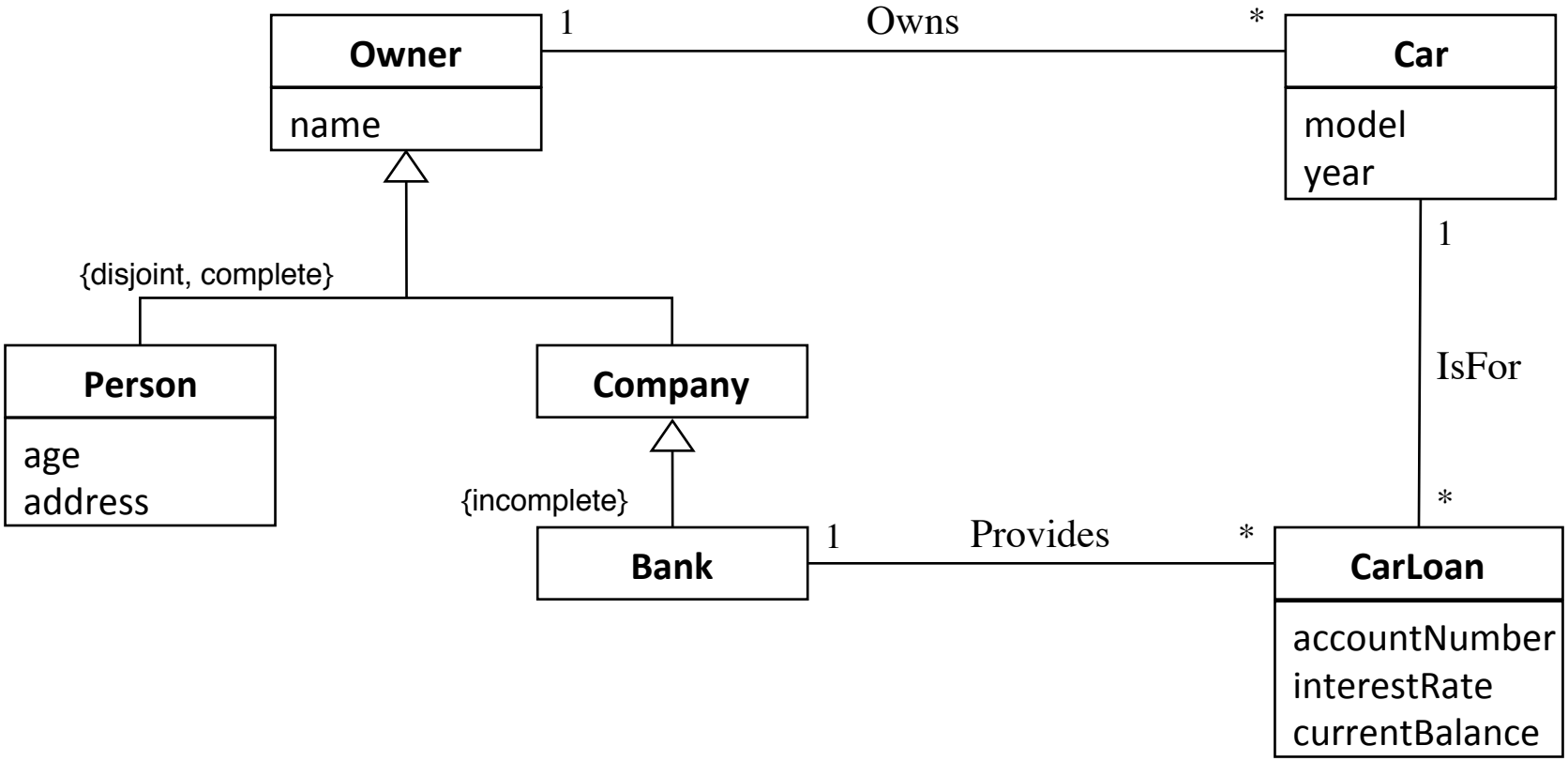


Example: Car ownership and loans—Initial solution



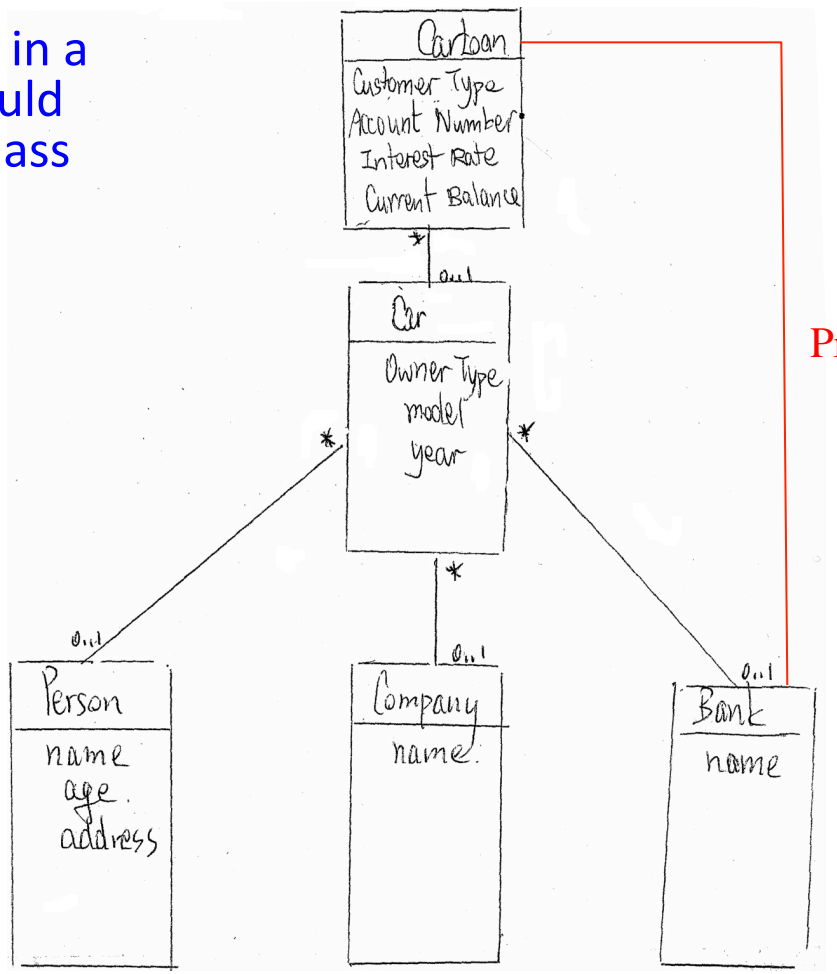
We could generalize Person and Company to Owner since both have the same association Owns to Car and both have an attribute, name, in common *(although the kinds of names are not exactly the same).*

Example: Car ownership and loans—Final Solution



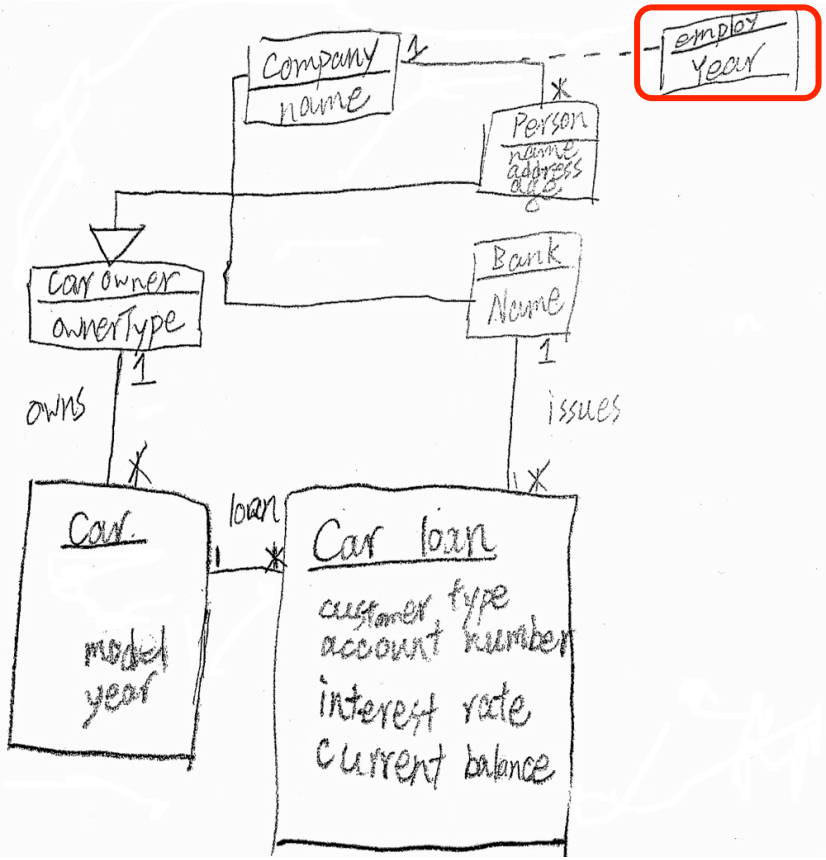
Common errors

- Missing classes or associations
  - All requirements stated in a problem statement should be represented in the class diagram, if possible.



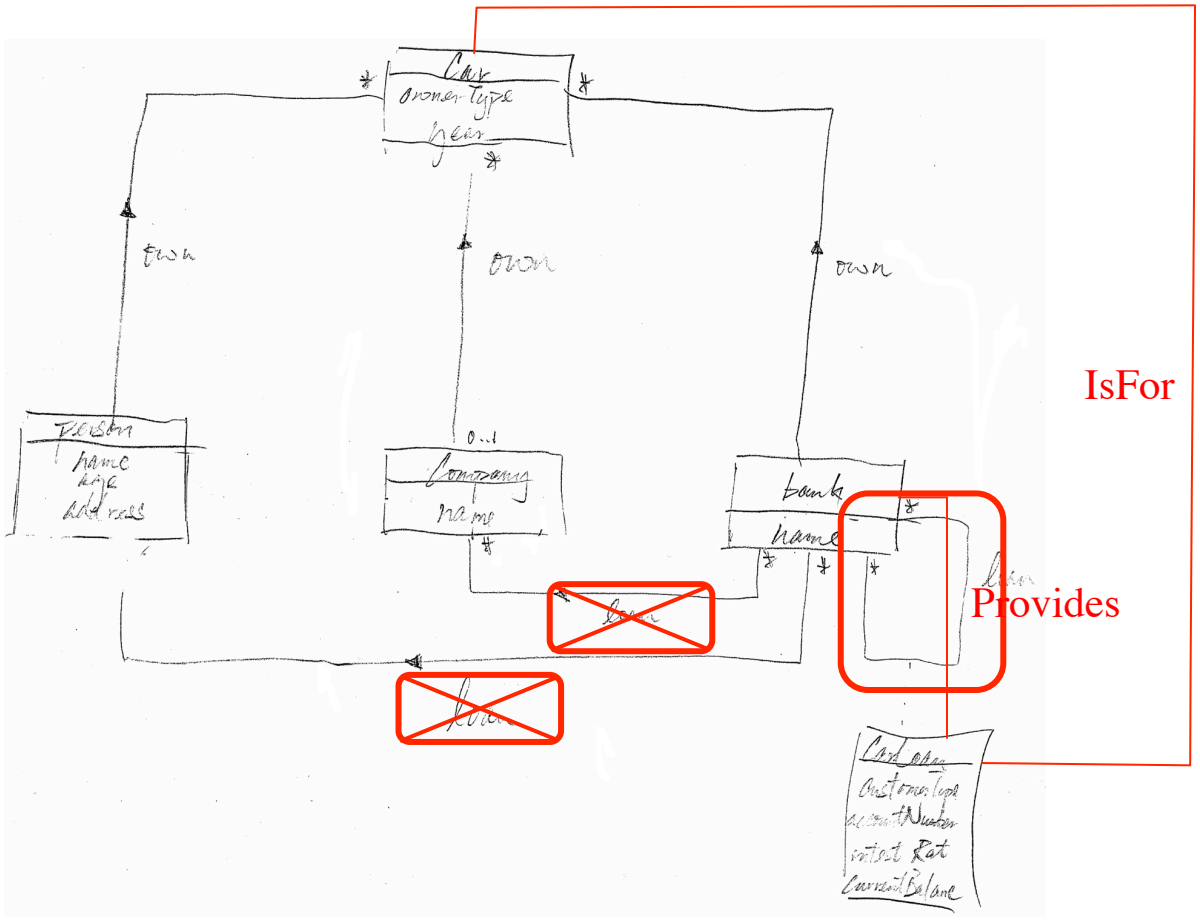
Common errors

- Extra classes or attributes not specified in the problem statement
  - Stick to what is stated in the problem statement; don't be creative.



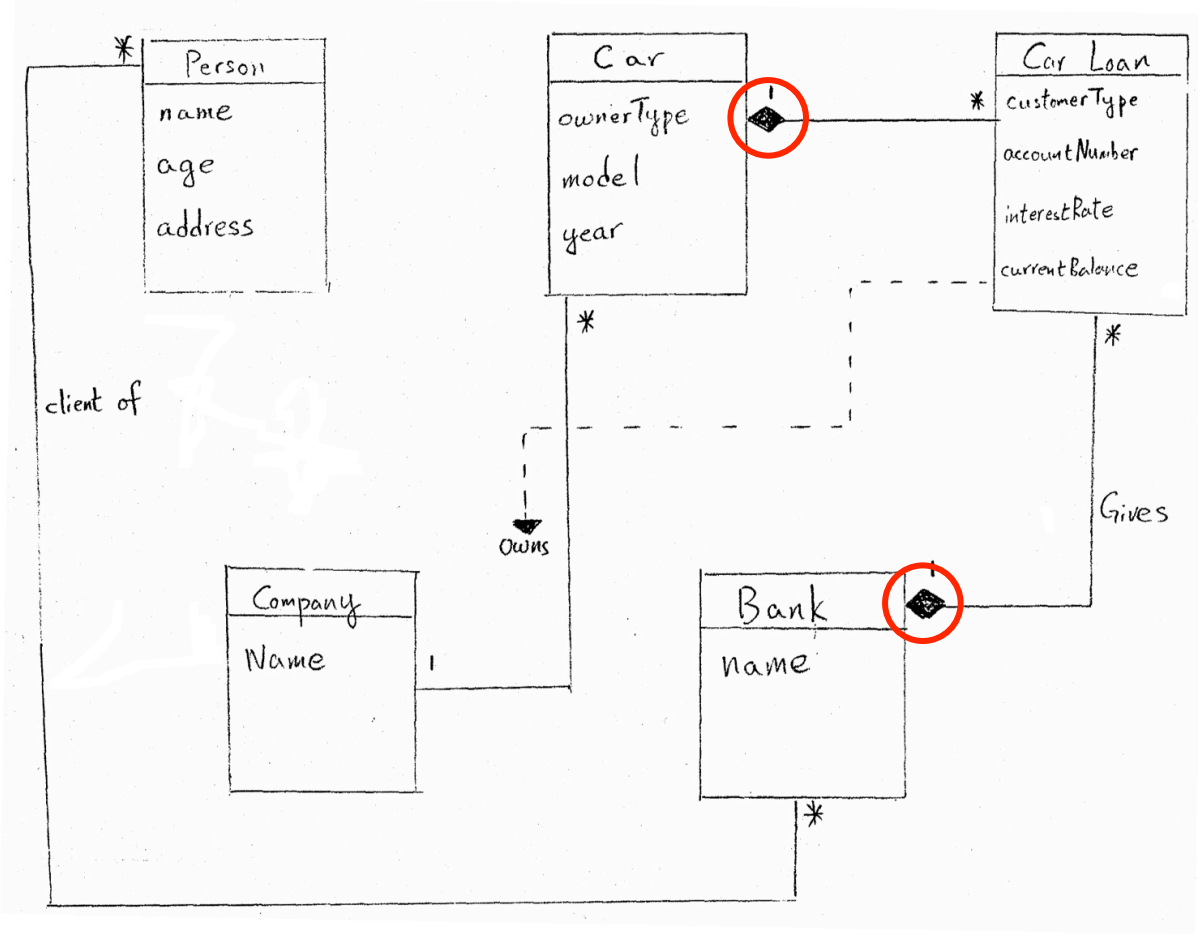
Common errors

- Incorrect associations
  - Be careful that your associations have the intended meaning.



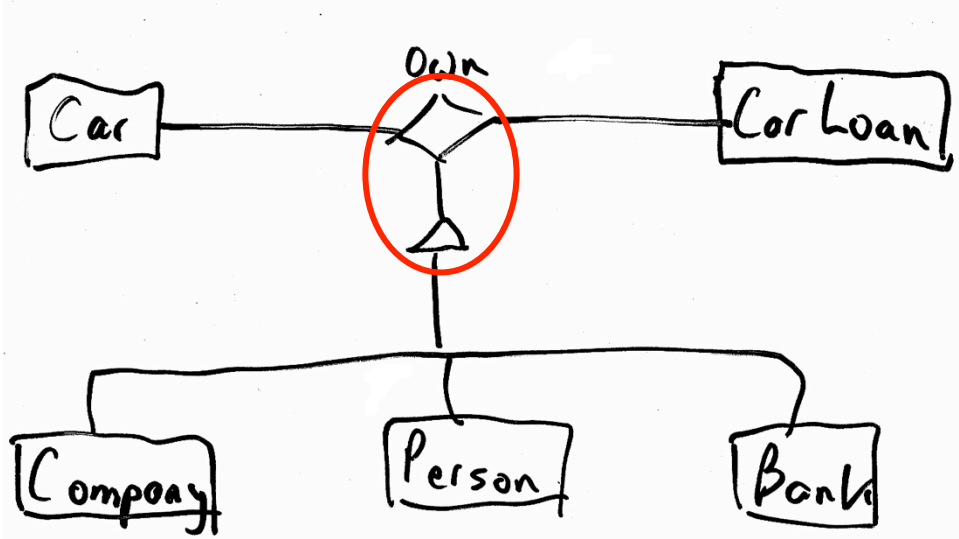
Common errors

- Incorrect use of aggregation
  - There is no aggregation in this example!



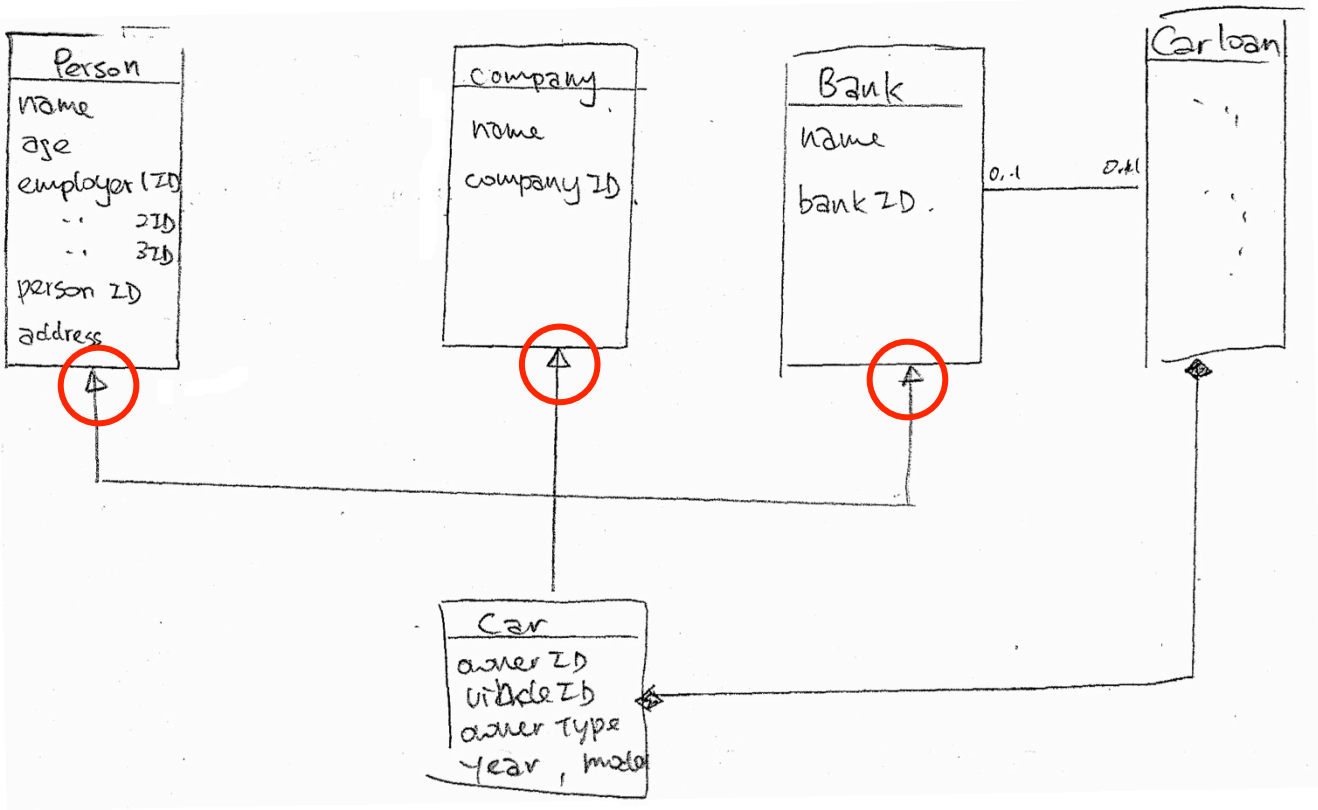
Common errors

- Incorrect use of relationships
  - Modeling elements can only be used in certain ways.



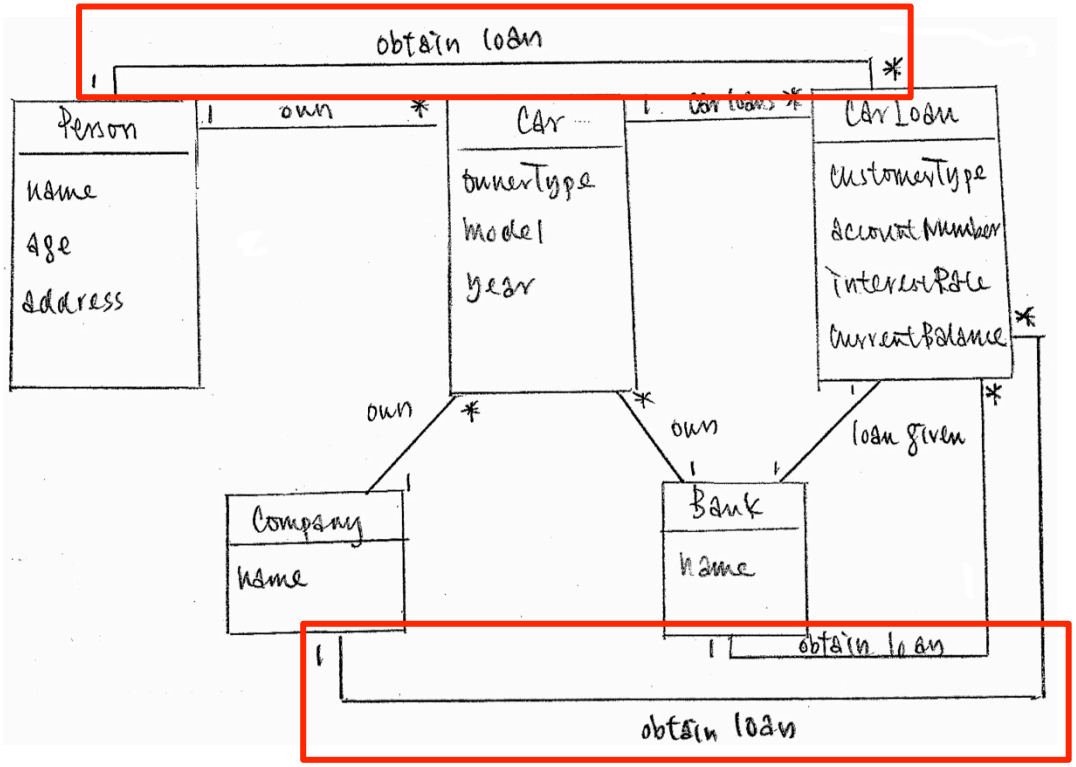
Common errors

- Incorrect use of relationships
  - Modeling elements can only be used in certain ways.



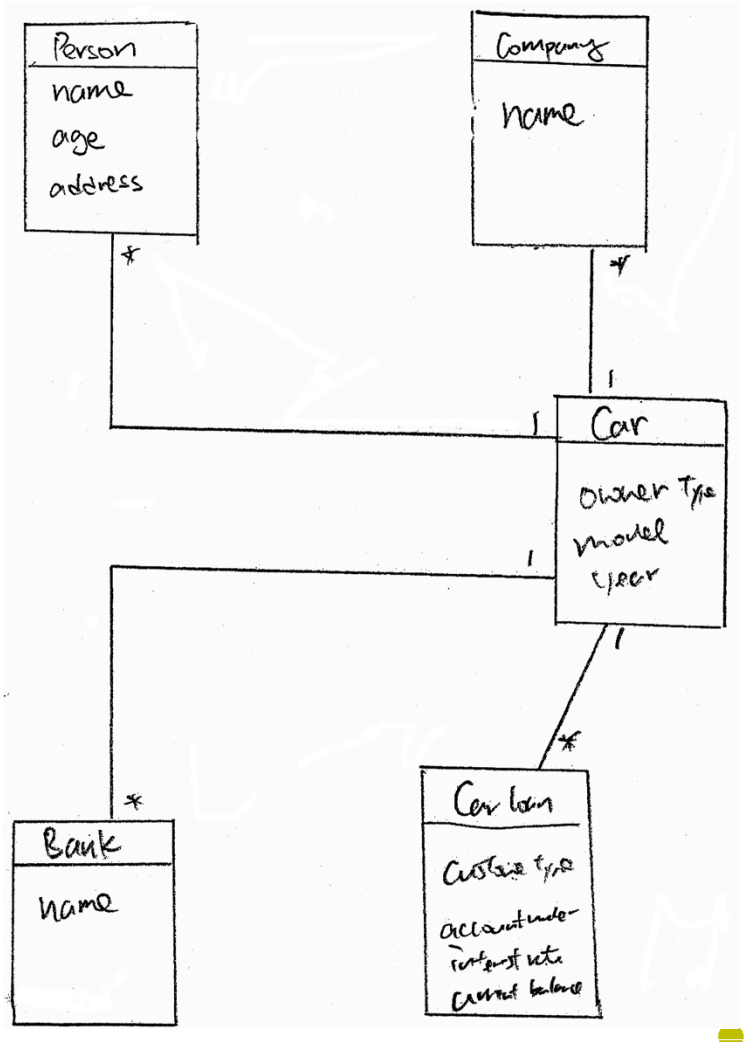
Common errors

- Redundant associations
  - Two paths in the class diagram that result in the same objects.



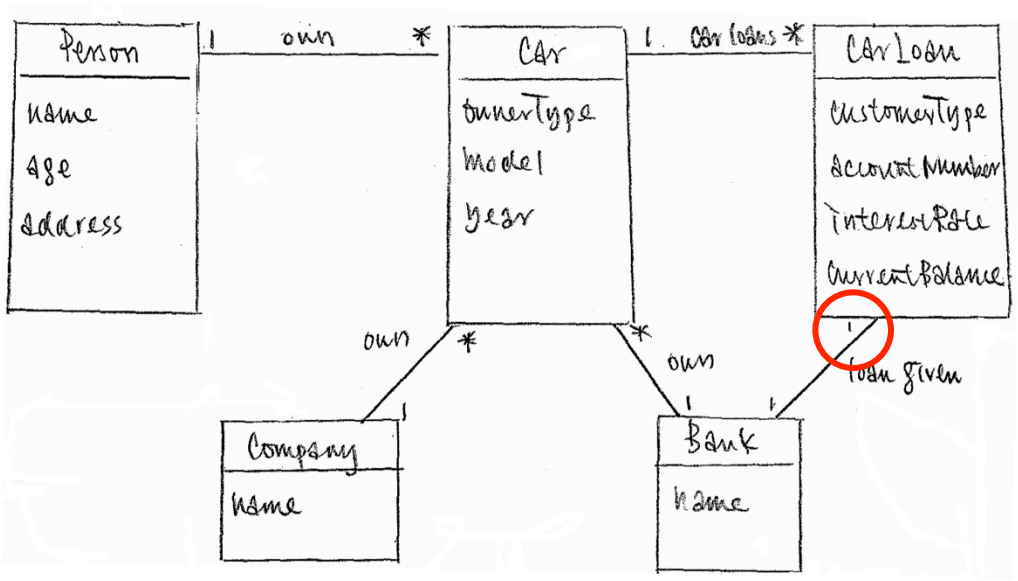
Common errors

- Missing association names
  - While associations names are not required, it is always a good idea to name them.



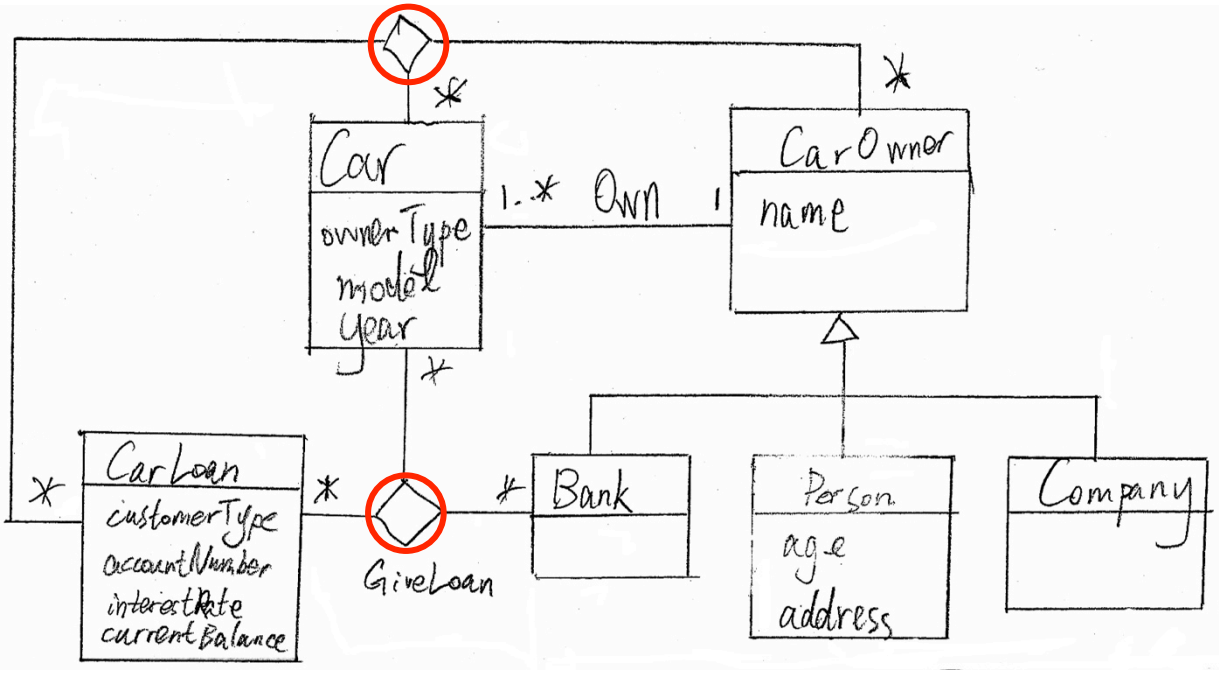
Common errors

- Missing or wrong multiplicities
  - (e.g., LoanGiven)



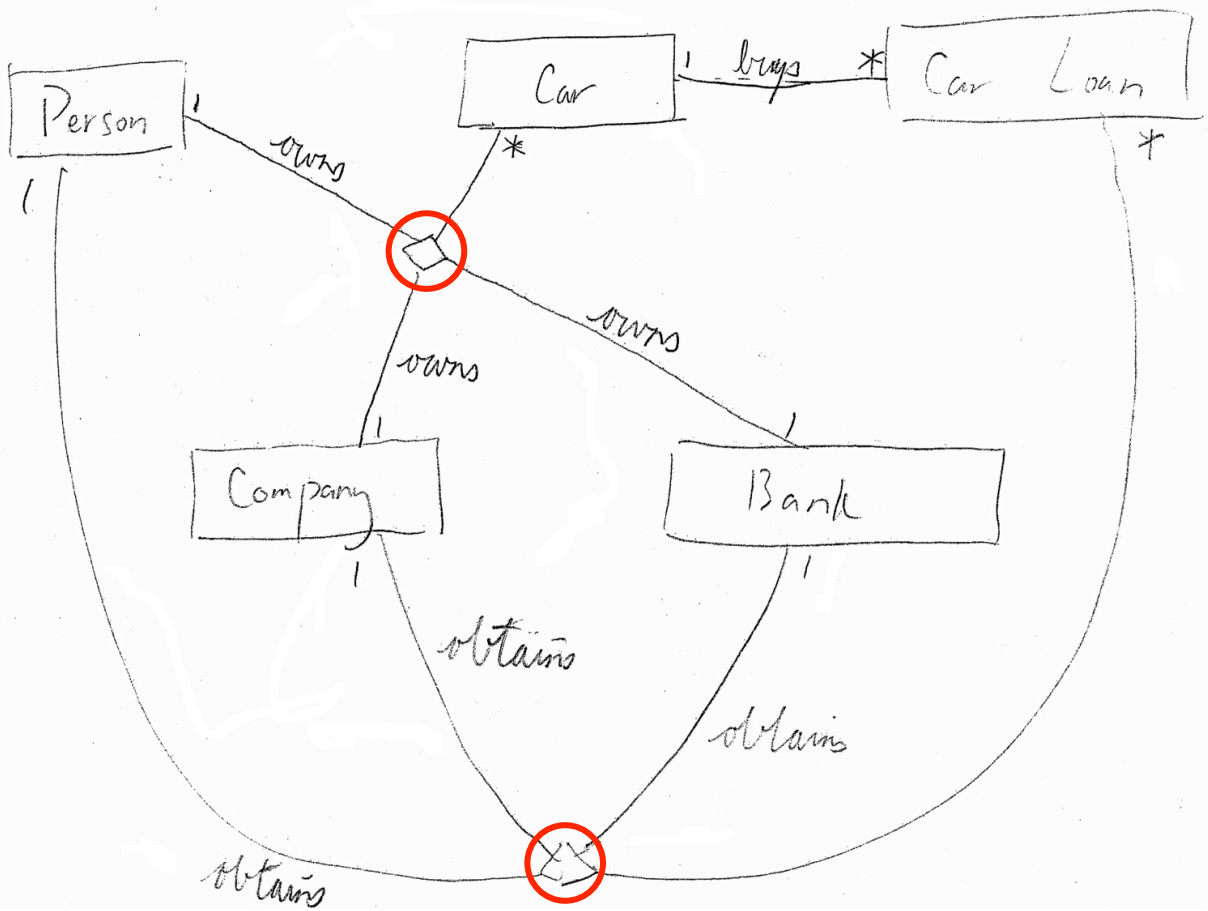
Common errors

- Using ternary associations
  - There are no ternary associations in this example.



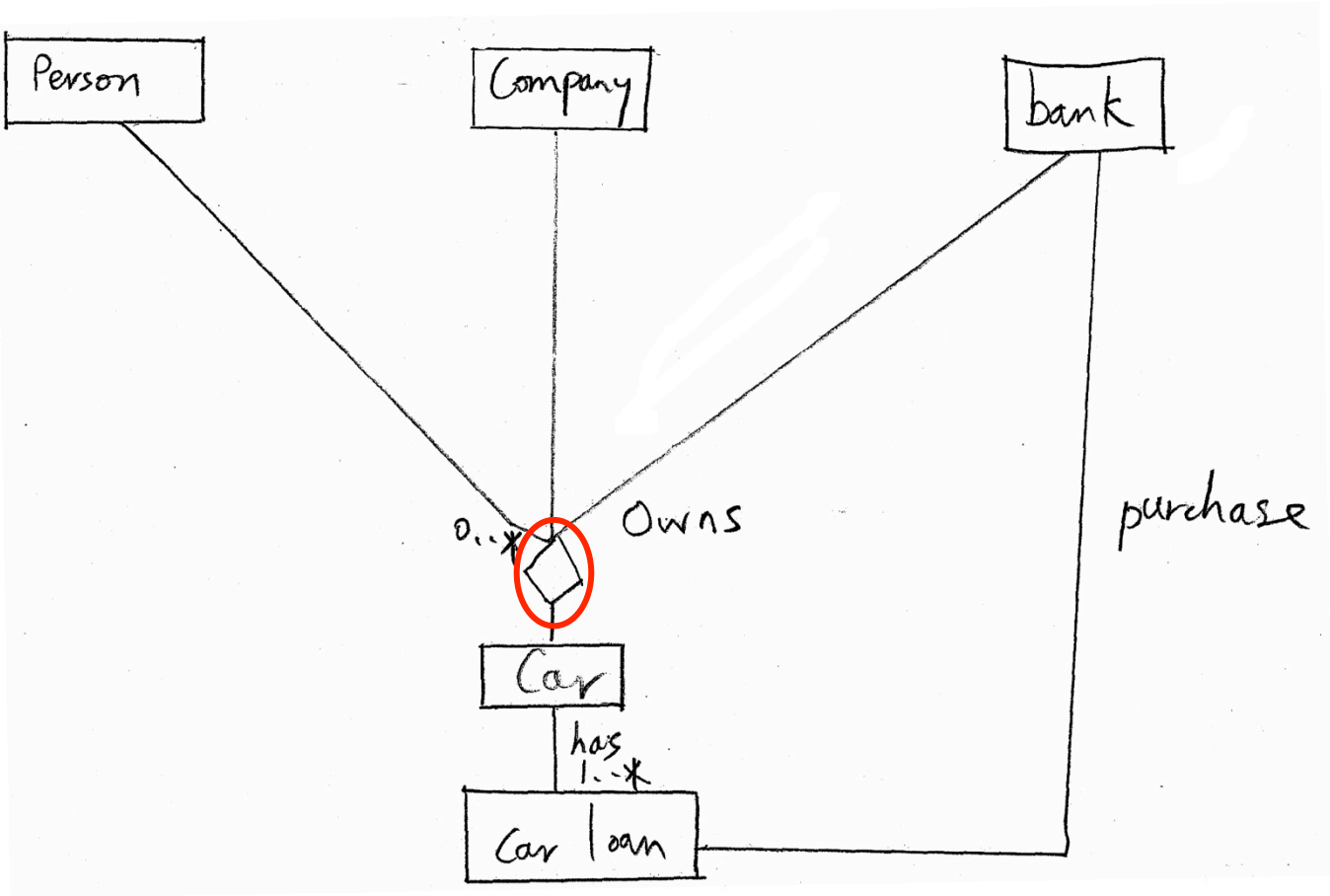
Common errors

- Using higher degree associations
  - There are no higher degree associations in this example.



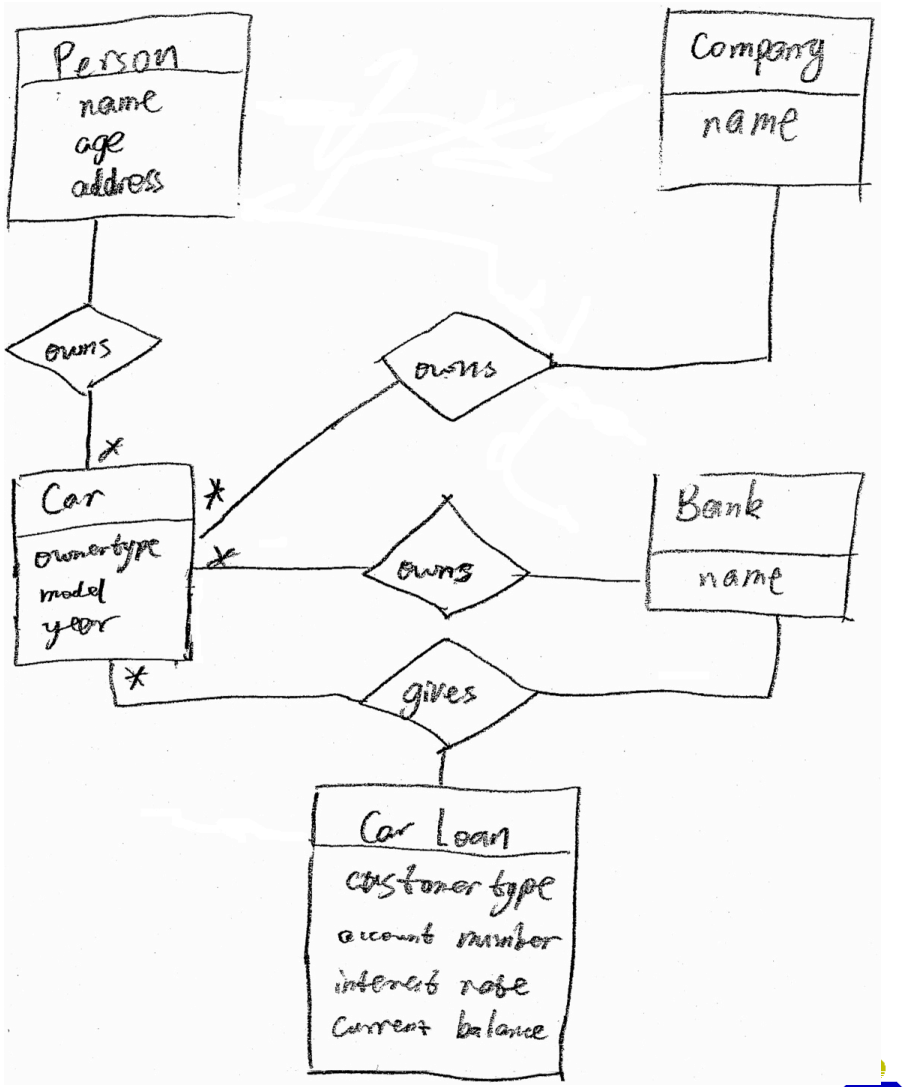
Common errors

- Using higher degree associations
  - There are no higher degree associations in this example.



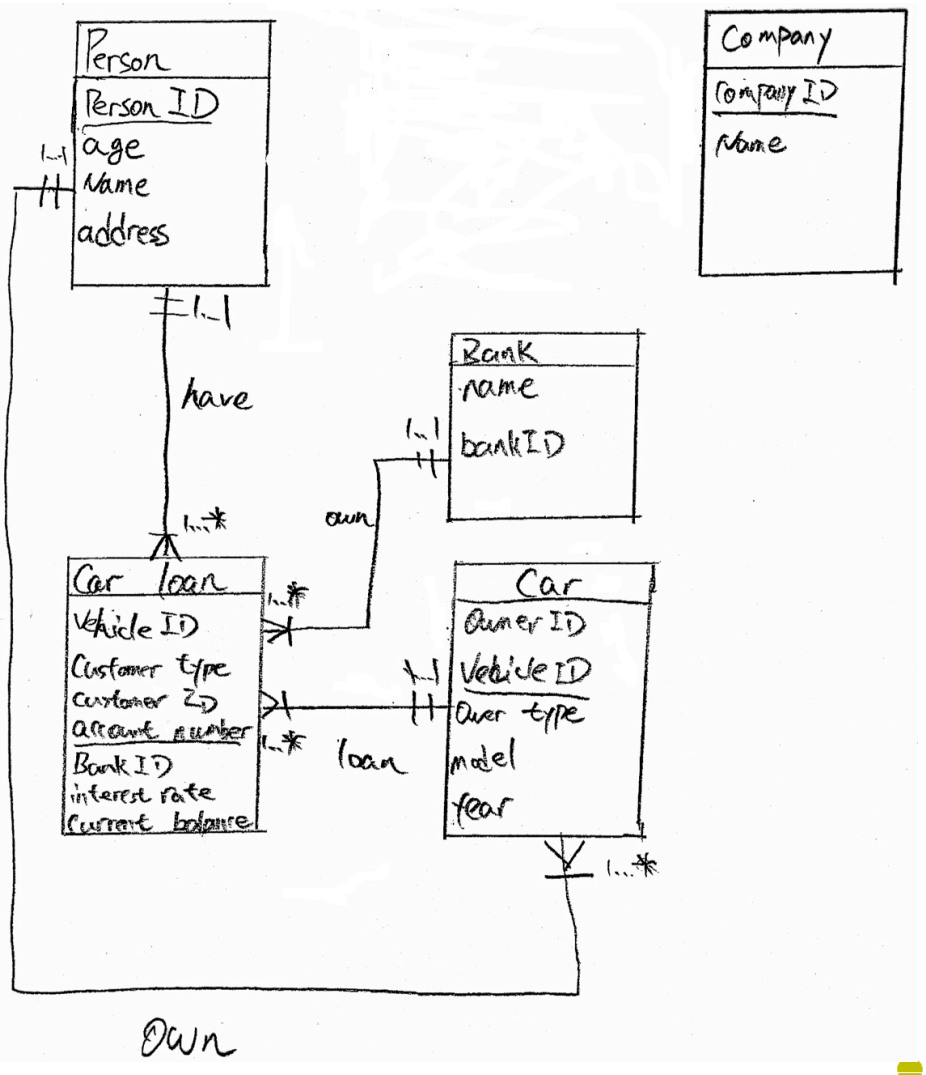
Common errors

- Using incorrect notation
  - (e.g., entity-relationship notation)



Common errors

- Using incorrect notation
  - (e.g., entity-relationship notation)



Reverse Engineering Exercise

```
public class Appt implements Schedulable{
        private int ID;
        private Vector<User> users;
        public void associate(Room r){}
}

public class Room extends User {
        private Vector<User> attendees;
}
```

```
public interface Schedulable {

}

public class User implements Schedulable {
        private String[] fullname;
        private Appt myAppointment;
}
```


Correct solution