



# Unified Modeling Language (UML): Class Diagram

Gunter Mussbacher

ECE, McGill University, Canada ◀▶ [gunter.mussbacher@mcgill.ca](mailto:gunter.mussbacher@mcgill.ca)

Based on material from: Bruegge & Dutoit, Lethbridge & Laganière,  
the Borland UML tutorial, K. Kostas, S. Somé, and D. Amyot

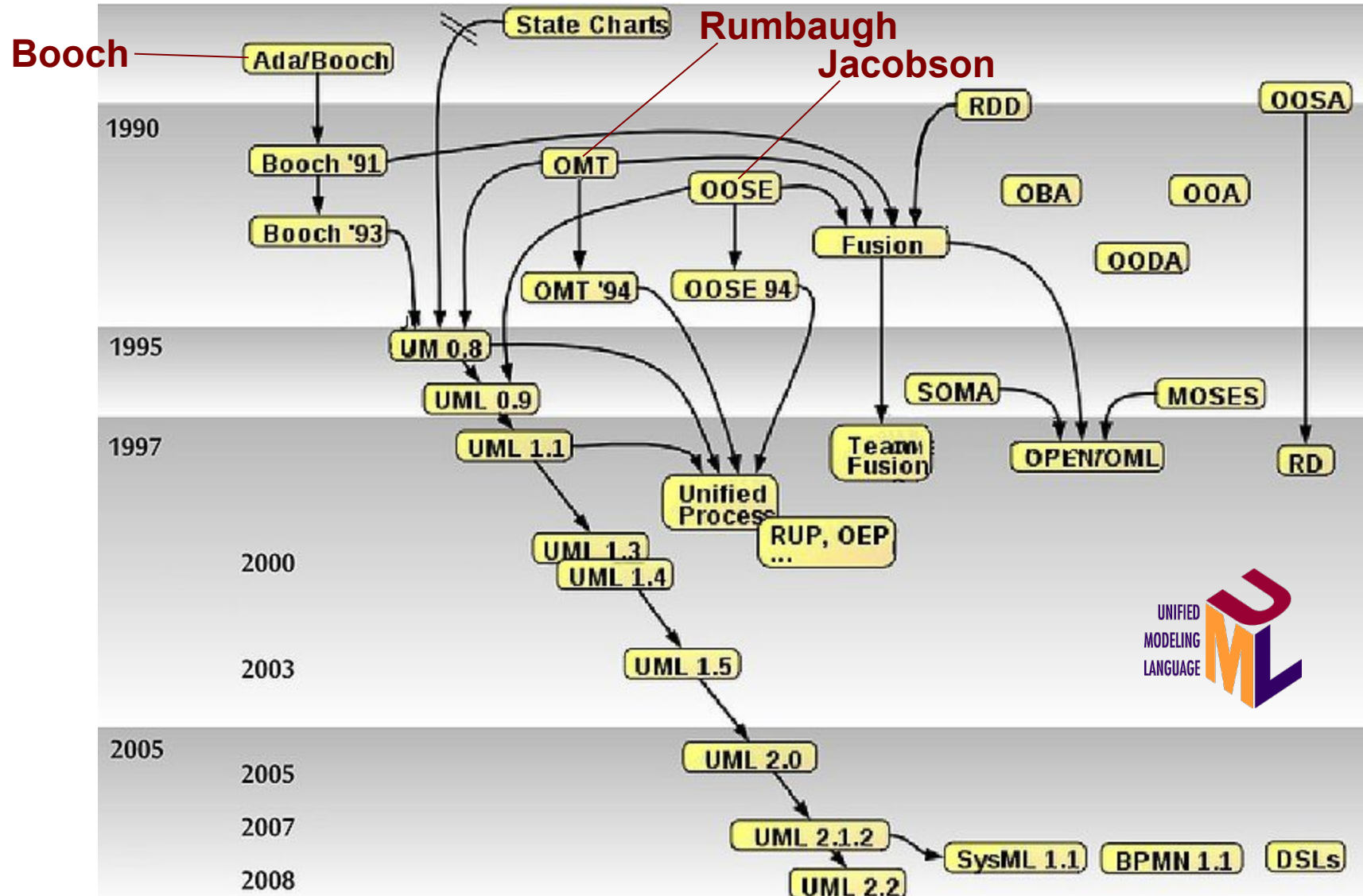
# Table of Contents

- Unified Modeling Language (UML)
  - Background on UML
  - Class Diagram
  - Object Diagram
  - Exercises
  - Advanced Concepts of Class Diagrams



# Background on UML

# History of UML



Source: [http://en.wikipedia.org/wiki/Unified\\_Modeling\\_Language](http://en.wikipedia.org/wiki/Unified_Modeling_Language)

# UML 2.x

- Object Management Group (OMG) standard
  - Version 2.0 released in 2005
  - Current version is 2.5 (March 2015)
  - <http://www.uml.org/>
- Thirteen Diagram Types in UML 2.x
  - Few changes compared to UML 1.x
    - Use case, object, package, deployment diagrams
  - Major improvements compared to UML 1.x
    - State machine, class, activity, and sequence diagrams
    - Component and communication (collaboration) diagrams
  - New additions
    - Timing, interaction overview, composite structure diagrams



# Classification of Diagram Types

- According to UML Reference Manual
  - Structural
    - **Class diagram**, **object diagram**, composite structure diagram, component diagram, and use case diagram
  - Dynamic
    - **State machine**, **activity diagram**, **sequence diagram**, communication diagram, timing diagram, and interaction overview diagram
  - Physical
    - Deployment diagram
  - Model Management
    - Package diagram



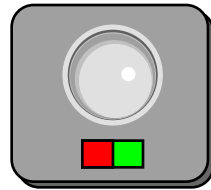
# Class Diagram

# Classes and Objects (1)

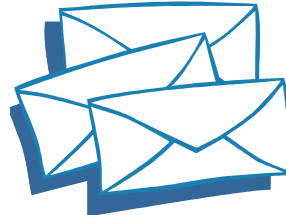
- A **class** describes a concept (a thing, an event, a role, an organizational unit, a place, a structure...)



Person



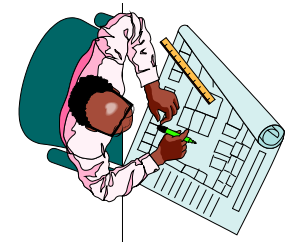
Movement Sensor



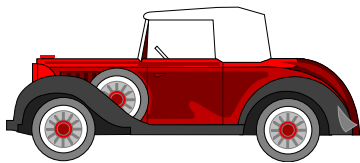
Invoice



Payment



Engineer



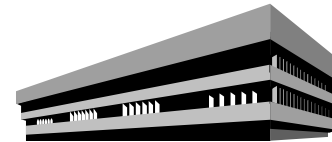
Car



Accounting  
Department



Student



Warehouse



Accident

- An **object** is a concrete member of a class
- Notation

**Class**

**object: Class**



# Classes and Objects (2)

- A **class** is the description for a **set of similar objects** that have the same structure and behavior, i.e., its **instances**
- All objects with the same features and behavior are instances of one class
- Organize **procedural abstractions** in the context of **data abstractions**
  - A class is a data abstraction which contains procedural abstractions that operate on its instances (objects)
- **Encapsulation** of data and behavior as well as **abstraction** leads to **information hiding**, which allows for **more maintainable, evolvable** systems
- Rules to differentiate a class from an instance:
  - In general, something should be a class if it could have instances
  - In general, something should be an instance if it is clearly a single member of the set defined by a class

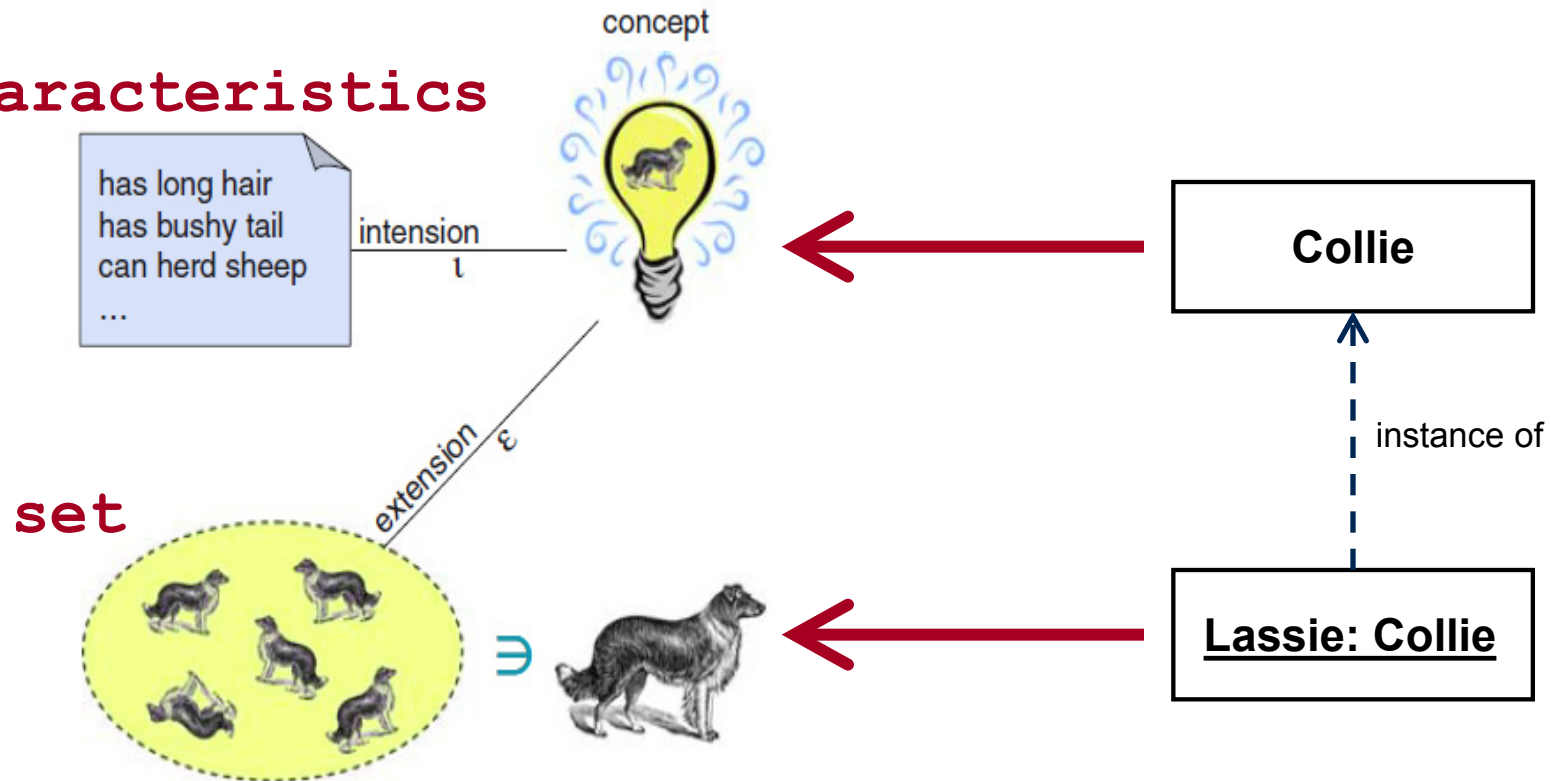
# Exercise: Class or Instance?

- General Motors **instance**
- Boing 777 **class (could be an instance)**
- Mary Smith **instance**
- Board game **class**
- University course ECSE123 **instance (a course offering is not the same as a course)**
- The game of chess between Tom and Jane which started today at 14:30 **instance**
- Automobile company **class**
- Computer science student **class**
- Game **class**
- Chess **instance (could be a class)**
- Airplane **class**
- The car with serial number JM 198765T4 **instance**

# Intension and Extension

- A software engineer is defining a new world and its **vocabulary**

## characteristics



- Power to **choose** which characteristics from the real world are important
- Characteristics expressed with **attributes, associations, and operations**



# This is Not a Pipe



René Magritte, 1929, "The Treachery of Images" (oil on canvas)

# Naming of Classes

- Noun
- Singular
- First letter capitalized; use camel case without spaces if needed
- Not too general, not too specific – at the right level of abstraction
- Avoid software engineering terms (data, record, table, information)
- Good: Hospital, Doctor, PartTimeEmployee
- Bad: register, Hospitals, doctor, PartTimeEmployeeData



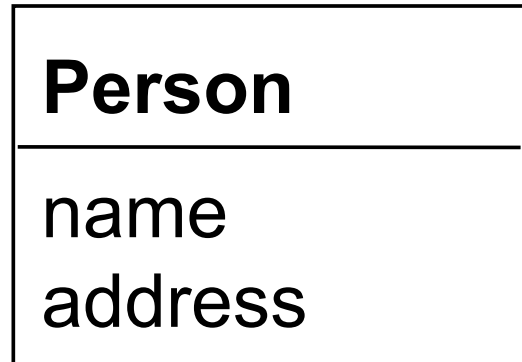
# Exercise: Naming of Classes

## In the context of a Train Management System:

- Train                    **ambiguous**  
                              **→ physical train (TrainConfiguration) ,**  
                              **scheduled train (ScheduledTrain) , or train on**  
                              **a particular day (SpecificTrain)?**
- Stop                    **the place where a train stops is called a**  
                              **station → Station**
- SleepingCarData      **software engineering term**  
                              **→ SleepingCar**
- arrive                  **no verbs, upper case → Destination**
- Routes                **no plural → Route**
- driver                  **upper case → Driver**
- SpecialTrainInfo      **not specific enough – what is a special**  
                              **train?, software engineering term**

# Attributes

- A class has attributes



**A class may also specify operations (shown in a third compartment). Domain models focus on concepts and their attributes, design models add operations.**

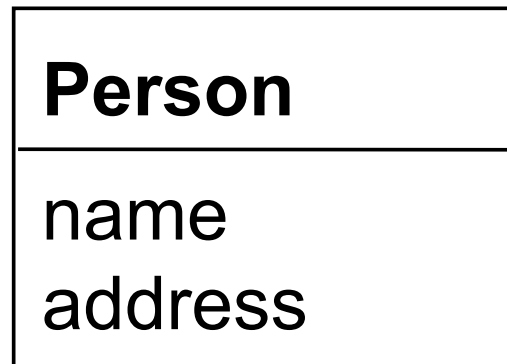
- An attribute is a **simple piece of data**
- More complex data is not modeled as an attribute
- Exists only when the object exists (has no existence on its own)
- First letter is lower case; use camel case without spaces if needed

# Exercise: Identify Attributes

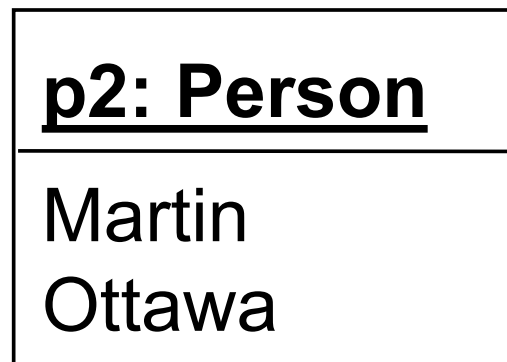
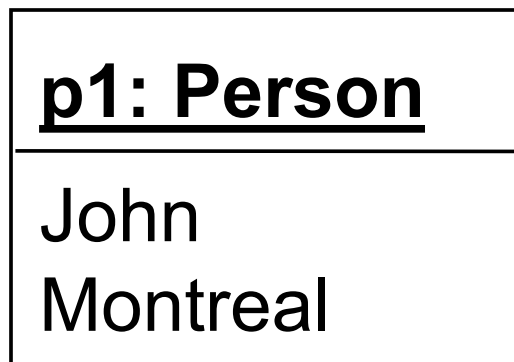
- Series (in a scheduling system for an independent television station)  
**title, description, numberOfSeasons, numberOfEpisodes**
- Passenger (in an airline system)  
**name, address, dateOfBirth, phoneNumber, email**
- Meeting (in a personal schedule system)  
**description, date, startTime, endTime, soundAlarmWhenStarting**
- Classroom (in a university course scheduling system)  
**roomCode, description, capacity**
- PhoneCall (in the system of a mobile telephone company)  
**startDateTime, length, isConnected, signalStrength, totalCost**
- AssemblyLine (in a factory automation system)  
**description, isActive**

# Objects

- A class



- An object of a class is **instantiated** (i.e., created by a software engineer)
- A **conscious decision** by a software engineer designates an object as an instance of a class
- Objects **interact** with each other at runtime





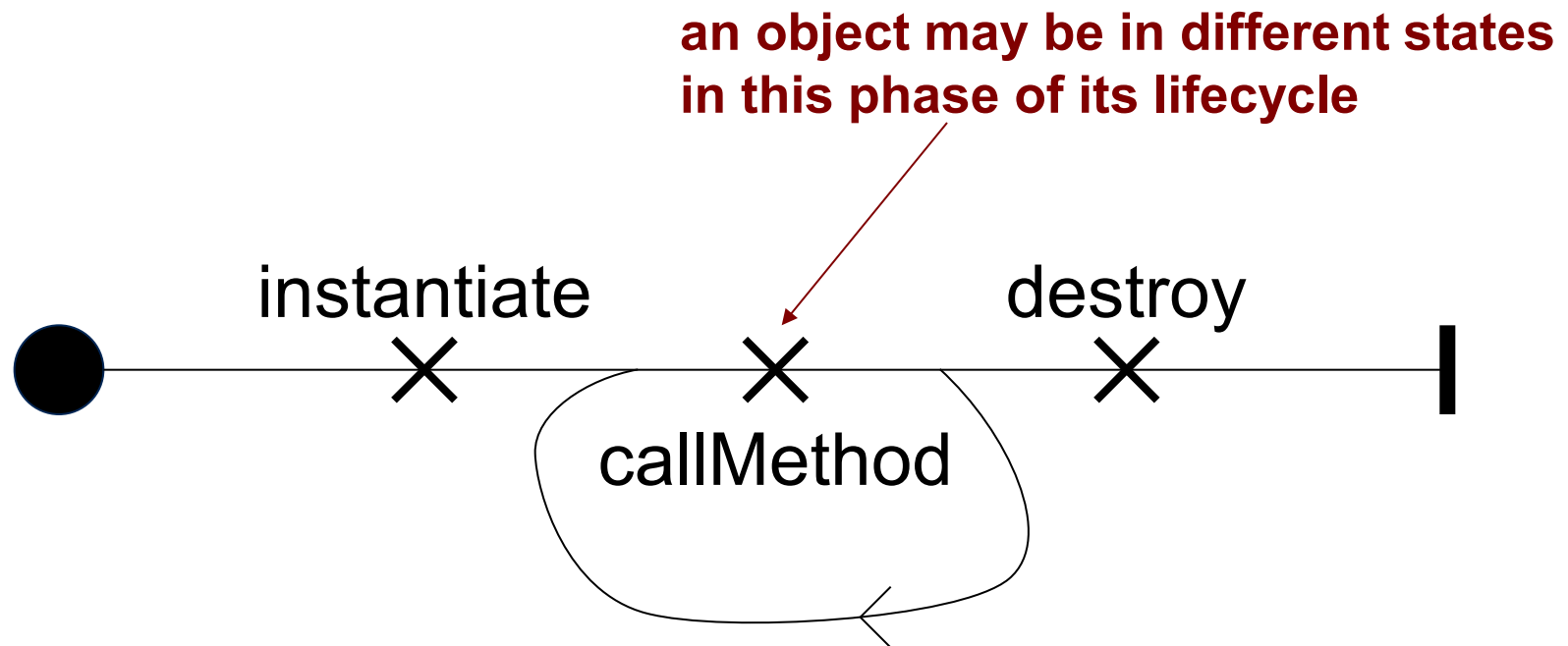
# Objects can be differentiated



identity



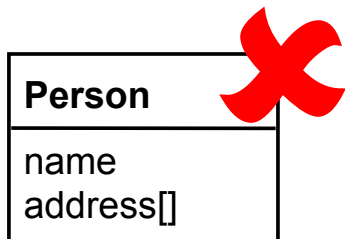
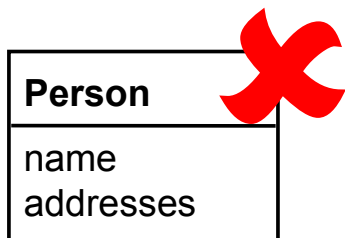
# Lifecycle of an Object



- The identity of an object **never** changes during its lifetime
- An attribute of an object does **not** have identity

# Identifying and Specifying Valid Attributes

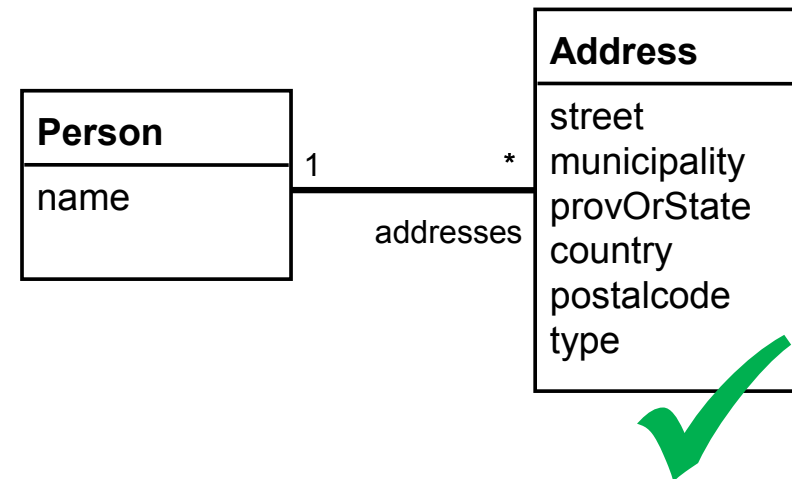
- It is not good to have many duplicate attributes
- If a subset of a class's attributes forms a coherent group, then create a distinct class containing these attributes



**Bad, due to a plural attribute**



**Bad, due to too many attributes, and the inability to add more addresses**



**Good solution; the type indicates whether it is a home address, business address etc.**

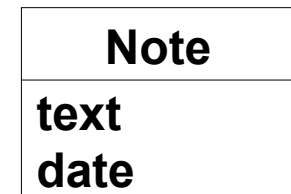
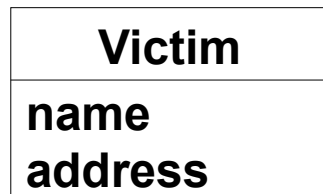
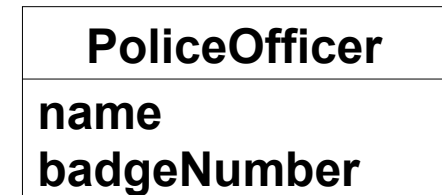
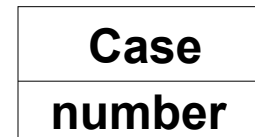
# Identifying Classes and Attributes

- Noun analysis helps identify classes and attributes

- Police Information (PI) System

This system helps the Java Valley police officers keep track of the cases they are assigned to do. Officers may be assigned to investigate particular crimes, which involves interviewing victims at their homes and entering notes in the PI system.

**instance of police station**



*work in progress – see later page for complete solution*

# Exercise: Identifying Classes and Attributes

- Use noun analysis to identify classes and attributes in the following description:

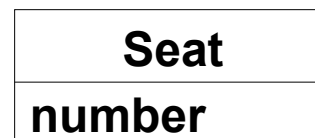
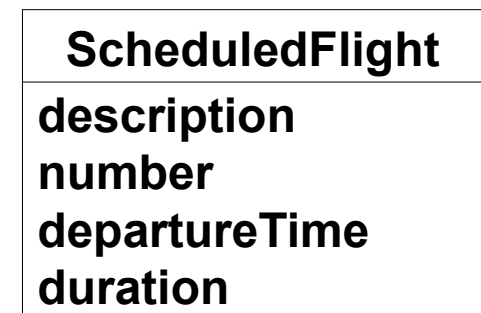
- Airline Reservation (AR) System

Ootumlia Airlines runs sightseeing flights from the Java Valley, the capital of Ootumlia. The AR system keeps track of passengers who will be flying in specific seats on various flights. Ootumlia Airlines runs several daily numbered flights on a regular schedule.

instance of airport

background info

a list of  
scheduled  
flights



work in progress – see later page for complete solution

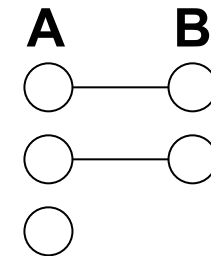
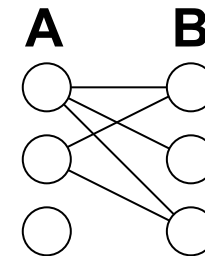
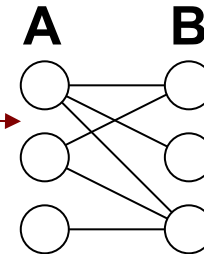
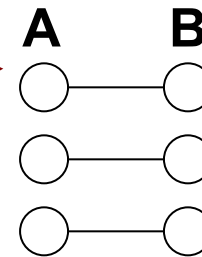


# Associations and Multiplicities: It's all about sets!

- An association models **relationships between two classes** (i.e., two sets of instances)
  - Each class plays a **role** in this relationship
- Multiplicity** (m) specifies how many instances of one class relate to how many instances of another class



- One:** Each instance of class A is related to one instance of class B
- Many:** Each instance of class A is related to one or many instances of class B
- Optional:** An instance of class A is not related to any instance of class B.

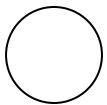
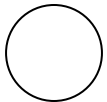
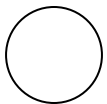


○ ... represents instance

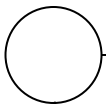
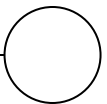
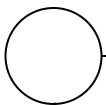
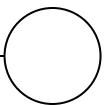


# Associations and Multiplicities (1)

- What is the most important property of a set?
  - Members of a set are **unique** → the same instance cannot be in the same set twice!
- **One instance of class A cannot be related to the same instance of class B more than once for the same association!**

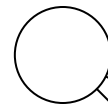
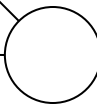
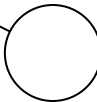
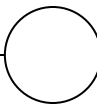
**A**

cannot be  
the same

**A****B****A****B**

X

Y

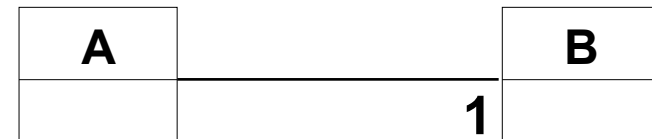
**A****B**

ok, because not the same  
association (different  
relationships X and Y)

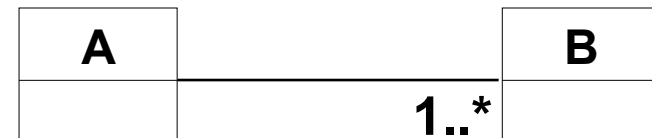
○ ... represents instance

# Associations and Multiplicities (2)

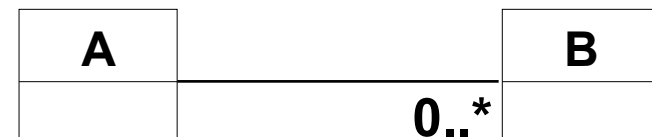
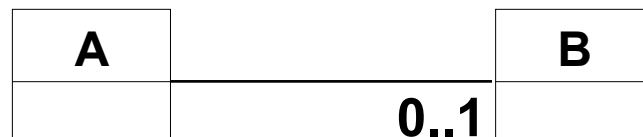
- Figure out multiplicities by asking yourself the following questions and finding concrete examples! Assuming a relationship exists between two classes A and B, start with each instance of class A being related to **one** (1) instance of class B



- If an instance of class A exist that is related to more than one instance of class B, then change the multiplicity of B to **one-or-many** (1..\*)

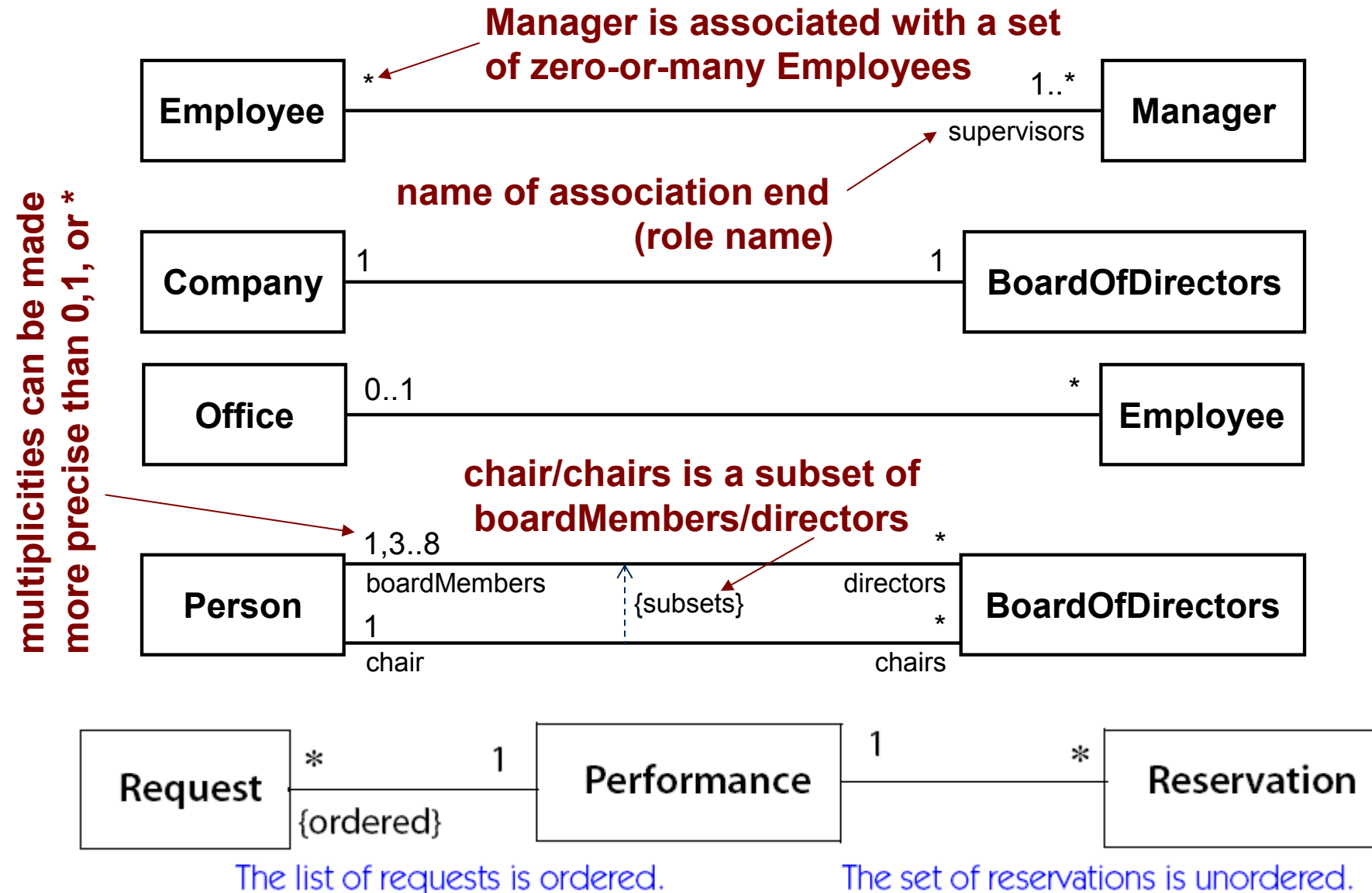


- If an instance of class A exist that is not related to any instance of class B, then make the association **optional** (0), i.e., **zero-or-one** or **zero-or-many**



- Do the same with roles of A and B reversed to determine A's multiplicity

# Associations and Multiplicities (3)



# Exercise: Identify Associations

- Series (in a scheduling system for an independent television station)  
**producers: ProductionCompany, leadActors: Actor, episodes: Episode**
- Passenger (in an airline system)  
**tickets: Ticket, loyaltyAccount: LoyaltyAccount**
- Meeting (in a personal schedule system)  
**location: Room**
- Classroom (in a university course scheduling system)  
**bookings: RoomBooking**
- PhoneCall (in the system of a mobile telephone company)  
**caller: Party, calledParty: Party**
- AssemblyLine (in a factory automation system)  
**manufacturedProduct: Product**



## Exercise: Identify Multiplicities

Country



one Country is on  
one Planet and one  
Planet consists of  
zero-or-many Countries      one


**Capital is a role played  
by a City; one Country  
has one Capital and one  
Capital is in one Country**



# Capital



# Planet

one Planet has zero-  
or-many Rivers and  
one River flows on  
 one Planet



# River

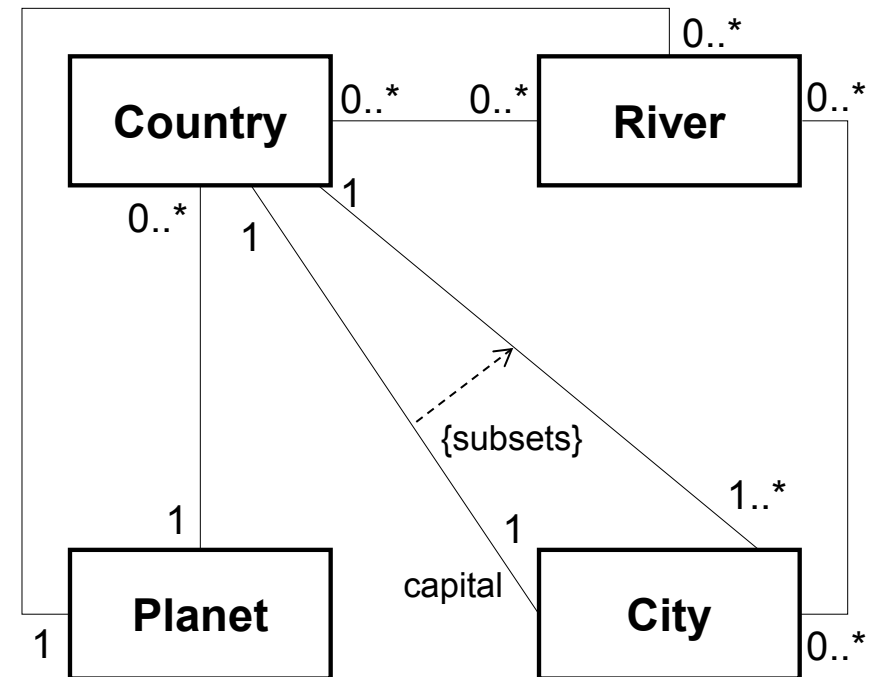
one Country has  
zero-or-many Rivers  
and one River flows  
through zero-or-  
many Countries

**one Country has  
one-or-many Cities  
and one City is in  
one Country**



City

one City has zero-or-many Rivers and one River flows through zero-or-many Cities



## Why is there no association between City and Planet?

## navigable via Country

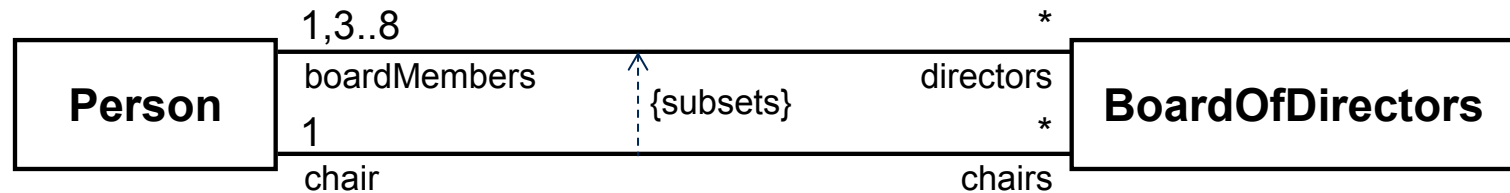
## Would it be ok to remove the association between River and Planet?

**no, cannot navigate from Planet via Country to River if a Planet does not have countries**

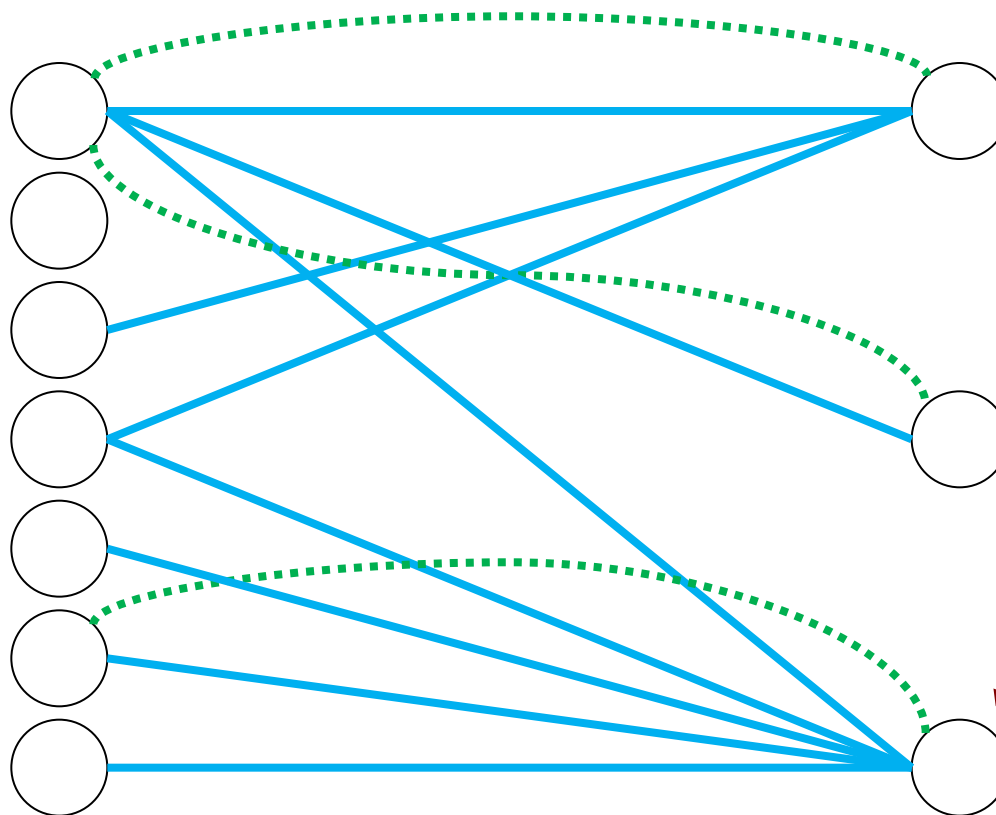


# Exercise: It's all about sets!

- Illustrate the two associations with lines between two sets of circles



a Person may exist who is not related to a BoardOfDirectors

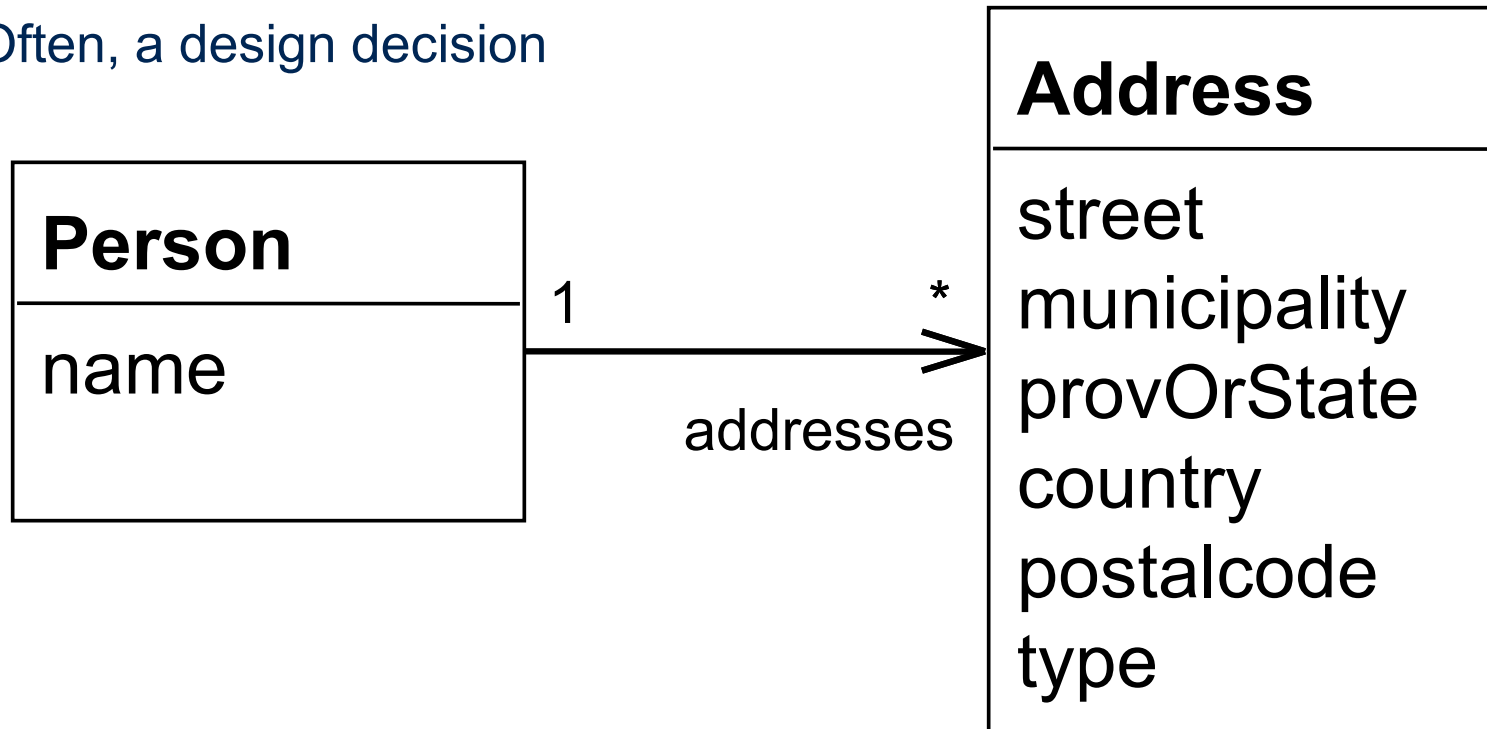


each BoardOfDirectors must have a chair and at least one boardMember

○ ... represents instance    — ... boardMembers/directors    ... chair/chairs

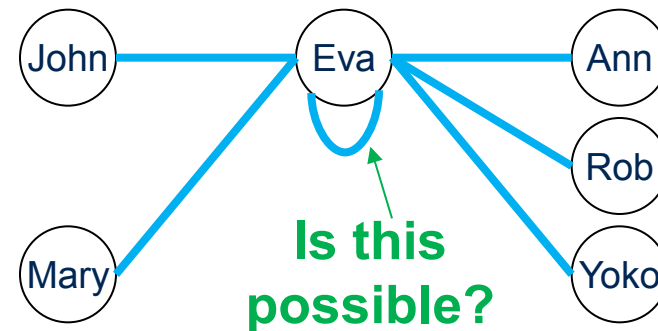
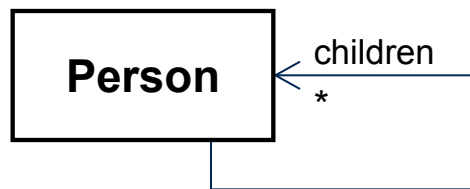
# Directionality in Associations (Navigability)

- Associations are by default **bi-directional**
- It is possible to limit the direction of an association by adding an arrow at one end
- In the example below, an Address object is **not aware** of the Person object at the other end of the association
- Often, a design decision



# A word about roles

- Person is a non-role term whereas daughter / son / father / mother / parents / children are role-terms, i.e., a Person plays the role of a daughter/son/father/mother/parent/child



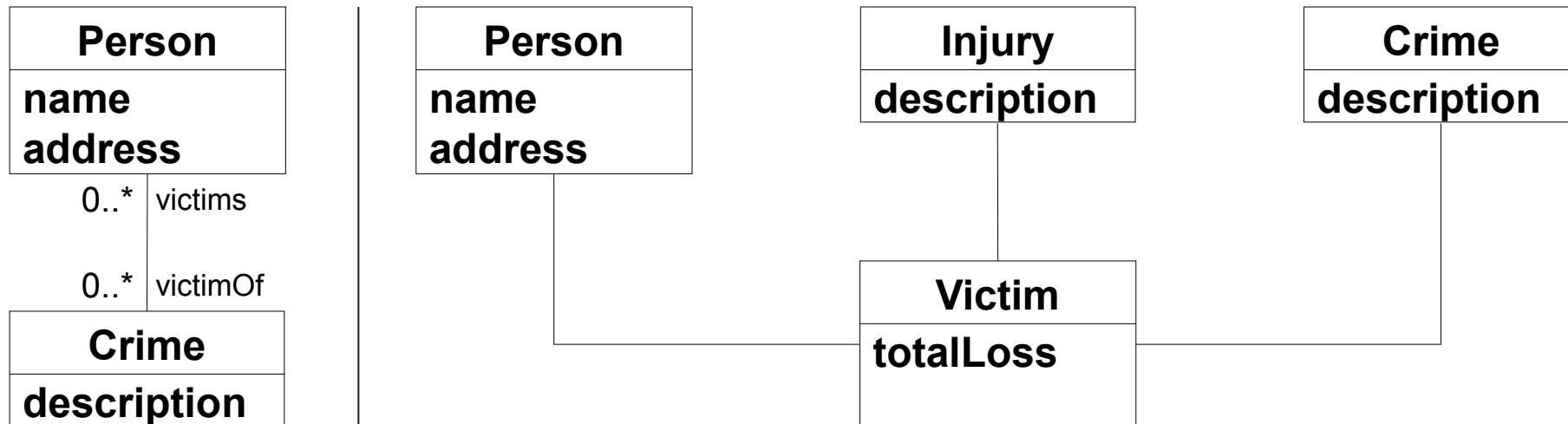
**Yes, but should be constrained!**

- Similarly, a Person plays the role of a victim in a Crime or the role of a passenger in an Airline Reservation System
- Similarly, object is a non-role term, while instance is a role-term
- An object is an instance of a class, i.e., the object plays the role of an instance of a particular class

○ ... represents instance of Person    — ... children

# Reification and Roles

- Process of making out of a concept an object (class) is called **reification**, meaning “making it an object” (making it a “first-class citizen”)
- Important: concept now has **identity** and may have **additional features**
- Reification can be applied to **associations (roles)**
  - A Victim is a reification of “a Person injured by a Crime”



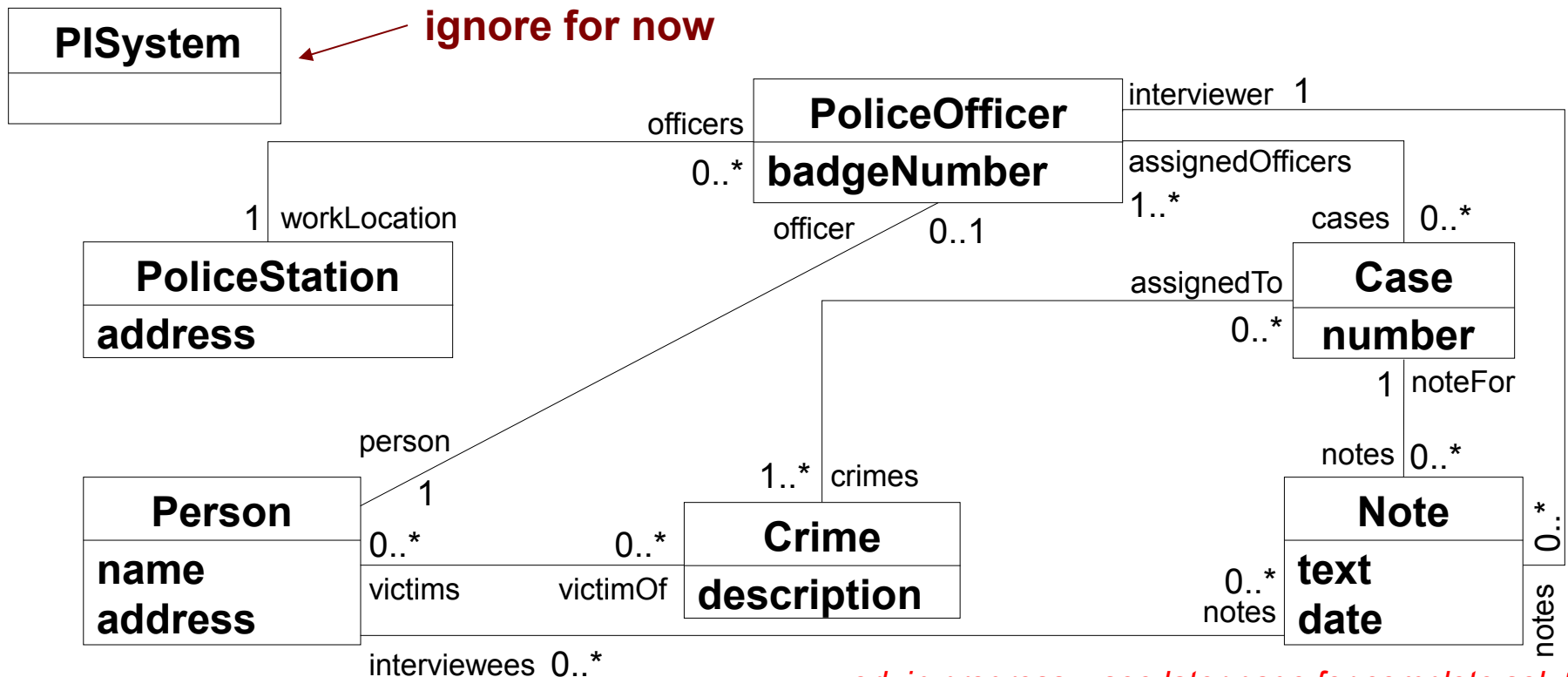
- A Flight is a reification of “an Aircraft scheduled on some Route on some Weekday”
- A Title is a reification of “a Person owning a Car”



# Identifying Associations and Roles

- Police Information (PI) System

This system helps the Java Valley police officers keep track of the cases they are assigned to do. Officers may be assigned to investigate particular crimes, which involves interviewing victims at their homes and entering notes in the PI system.



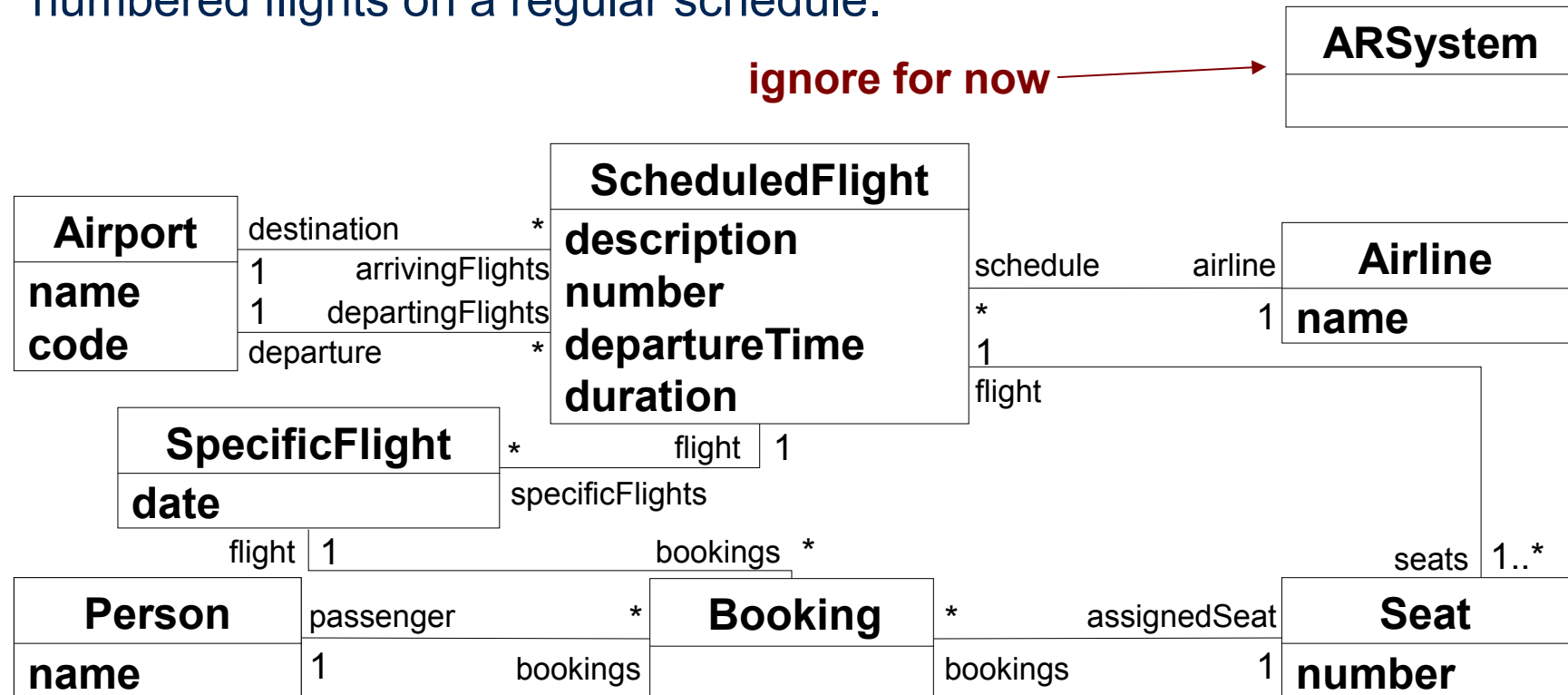
*work in progress – see later page for complete solution*

# Exercise: Identifying Associations and Roles

- Airline Reservation (AR) System

Ootumlia Airlines runs sightseeing flights from the Java Valley, the capital of Ootumlia. The AR system keeps track of passengers who will be flying in specific seats on various flights. Ootumlia Airlines runs several daily numbered flights on a regular schedule.

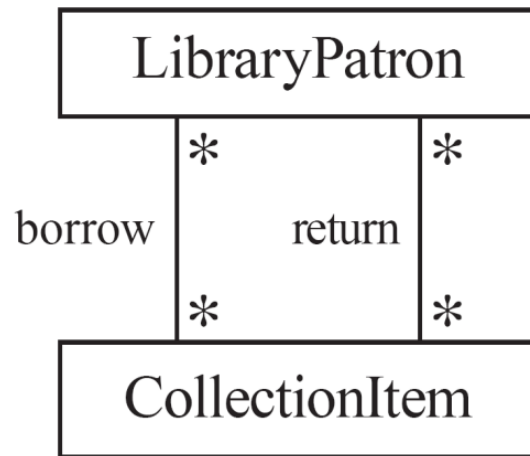
ignore for now



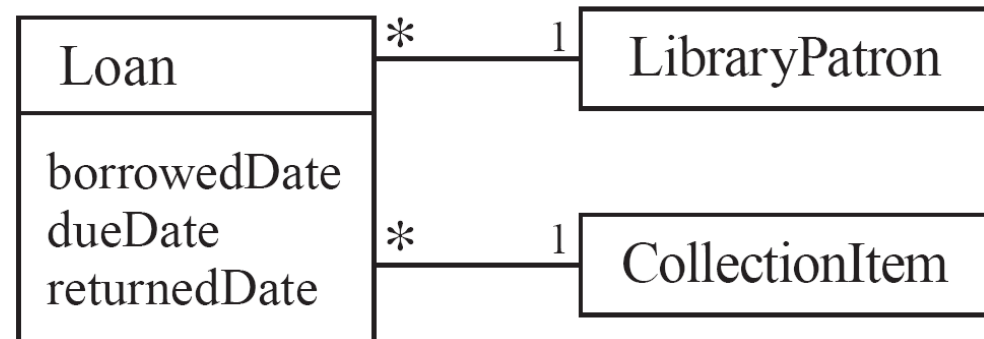
work in progress – see later page for complete solution

# Actions vs. Associations

- A common mistake is to represent actions as if they were associations



Bad, due to the use  
of associations that  
are actions



Better: The **borrow** operation creates  
a **Loan**, and the **return** operation sets  
the **returnedDate** attribute

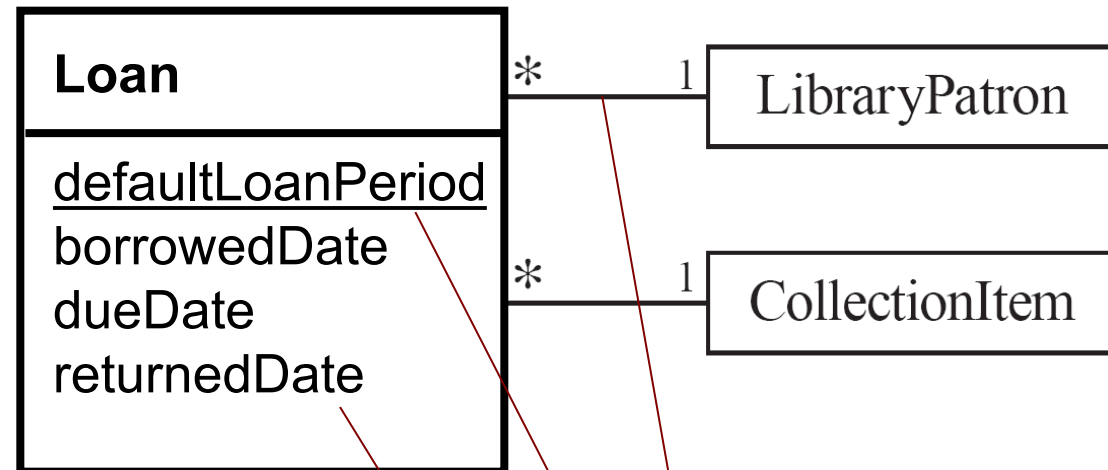
# Variables vs. Objects, Attributes, Associations (1)

- Two groups of **instance variables**: those used to implement attributes and those used to implement associations
- Class variable**: shared by all instances

- Useful for default values or constants
- Useful for lookup tables or similar structures used by operations
- Do not overuse!

**class variables  
(static)**

**instance variables**



```

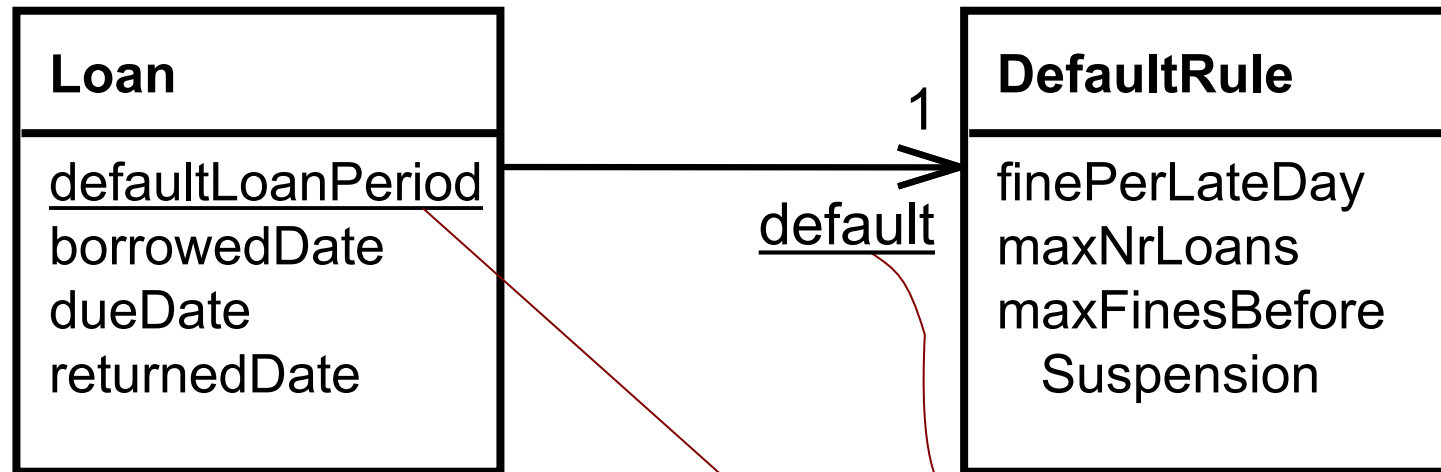
class Loan {
    static int defaultLoanPeriod;
    Date borrowedDate;
    Date dueDate;
    Date returnedDate;
    LibraryPatron libraryPatron;
    CollectionItem collectionItem;
    ...
}
  
```



# Variables vs. Objects, Attributes, Associations (2)

- Similar to **static attributes**, an association end/role may also be **static** (both shown with underline)

- E.g., the default is shared by all instances of Loan



```

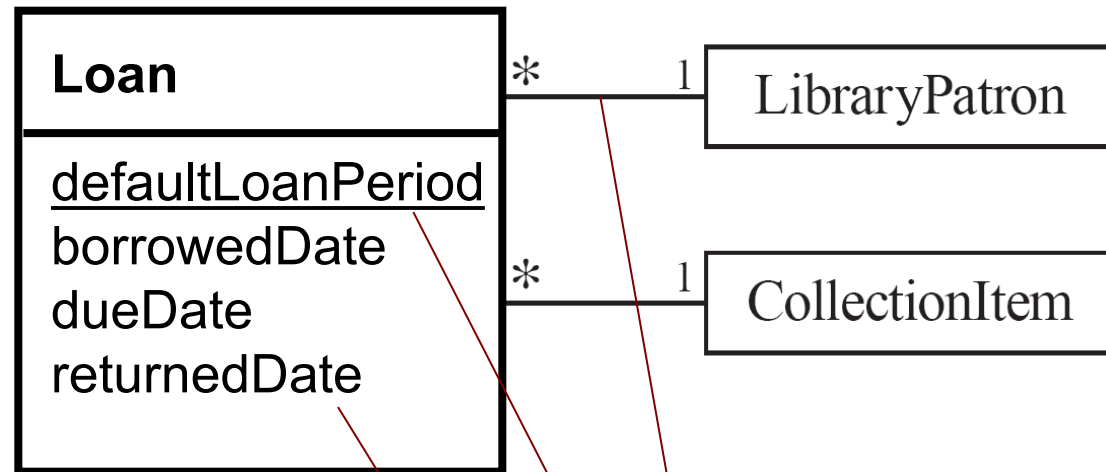
class Loan {
    static int <u>defaultLoanPeriod</u>;
    static DefaultRule <u>default</u>;
    Date borrowedDate;
    Date dueDate;
    Date returnedDate;
}
  
```

class variables  
(static)

instance variables

# Variables vs. Objects, Attributes, Associations (3)

- A variable is not an object, but rather **references** an object or no object at all
- An object may be temporarily referenced by a variable and live after the variable is destroyed as long as it is referenced by another variable
- The **type** of the variable determines what classes of objects it may contain



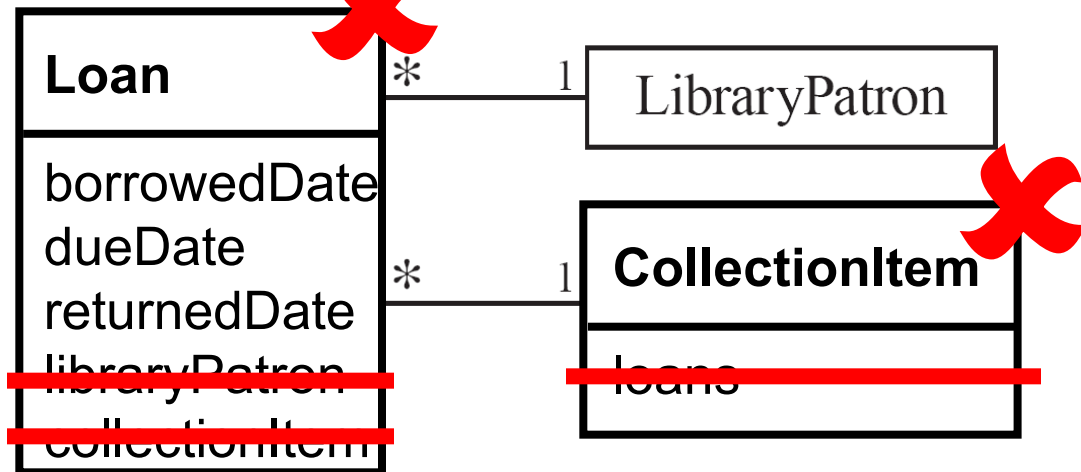
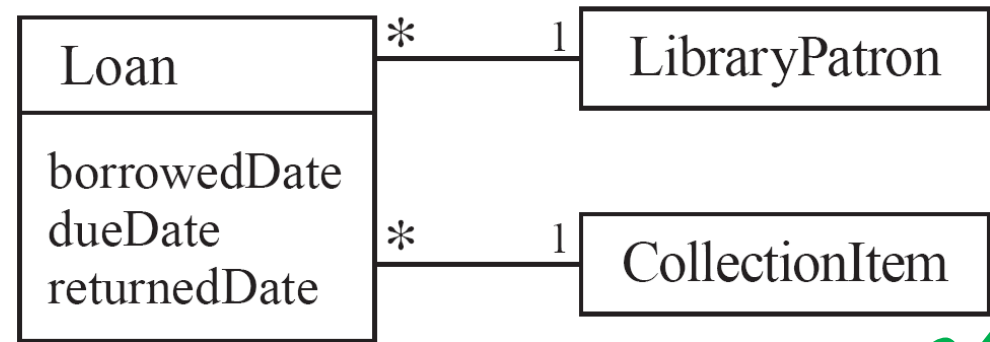
```

class Loan {
    static int defaultLoanPeriod;
    Date borrowedDate;
    Date dueDate;
    Date returnedDate;
    LibraryPatron libraryPatron;
    CollectionItem collectionItem;
    ...
}
  
```

**fields**  
(instance and  
class variables)

# Attributes vs. Associations (1)

- Another common mistake is to duplicate an association by adding an **attribute** in the class diagram
- An association is already implemented as a **field**
- A Java class for Loan that corresponds to the above class diagram will have the borrowedDate, dueDate, and returnedDate fields, but also a field for LibraryPatron and another one for CollectionItem
- No need to add it again!
- The same applies to loans in CollectionItem



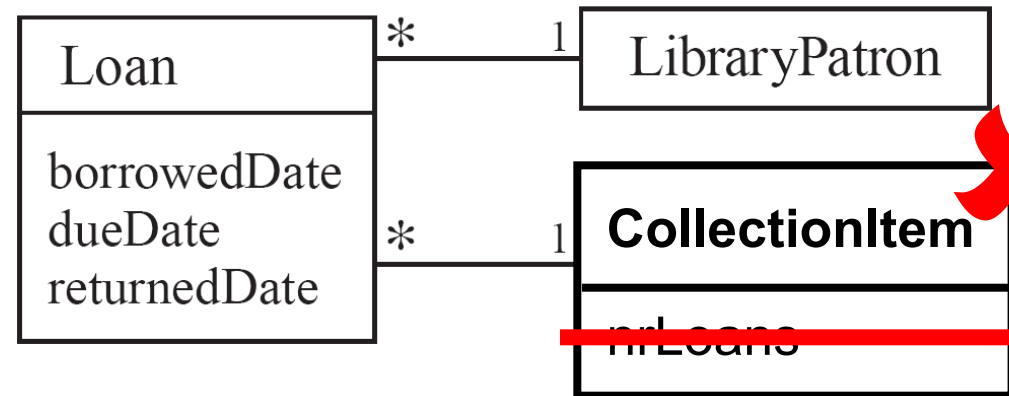
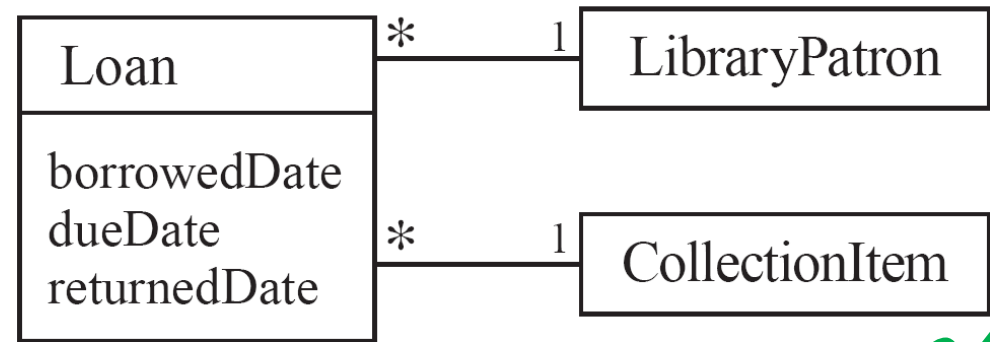
```

class CollectionItem {
    List<Loan> loans;
}
  
```



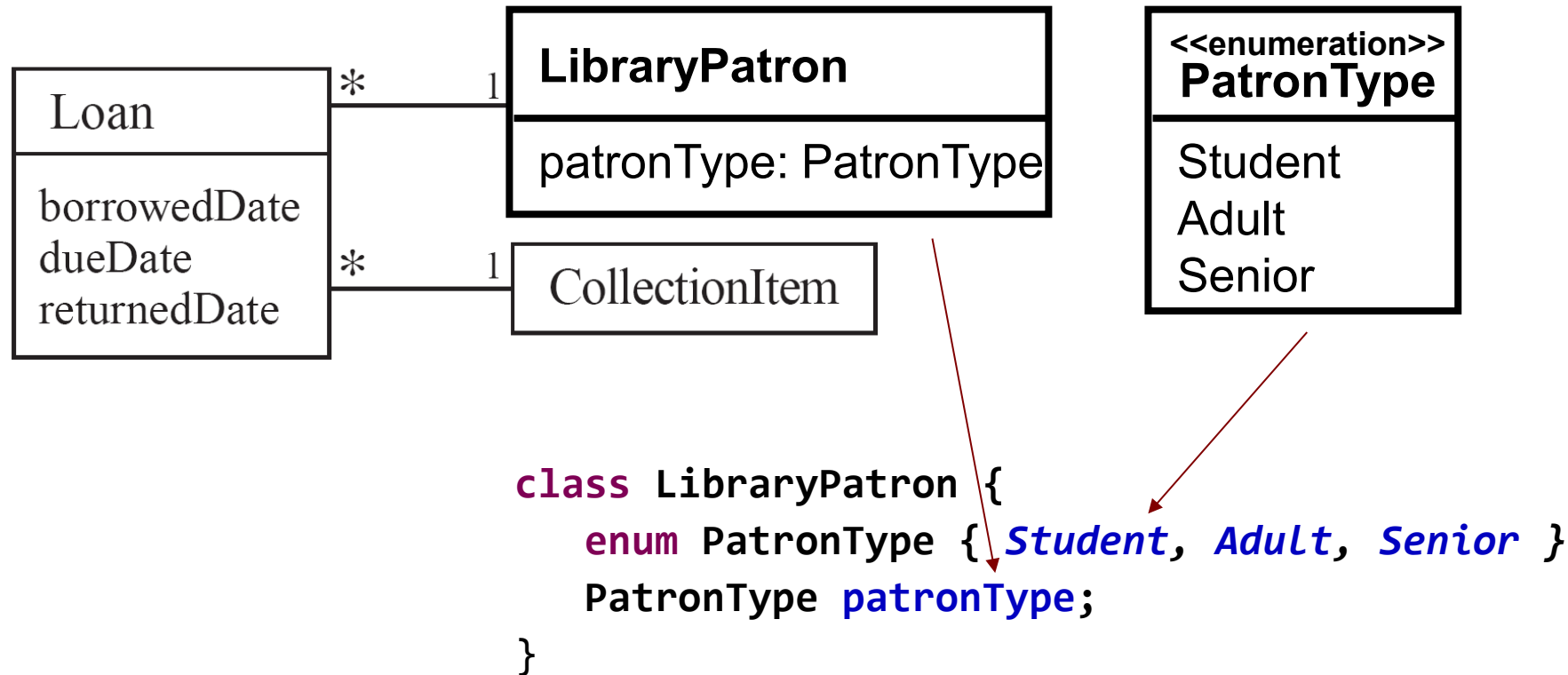
# Attributes vs. Associations (2)

- Similarly, it is also not necessary to duplicate information about an association such as the number of items by adding an **attribute** in the class diagram
- This can easily be determined by computing the **size** of the list representing the association
- In general not a clear-cut rule: trade-off between having to update this attribute (e.g., `nrLoans`) vs time it takes to compute the size (e.g., `loans.getSize()`)





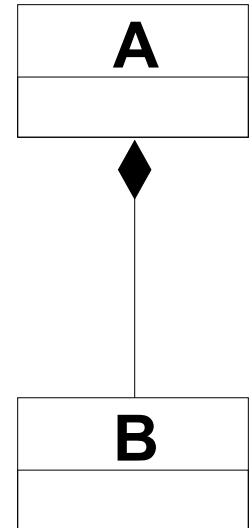
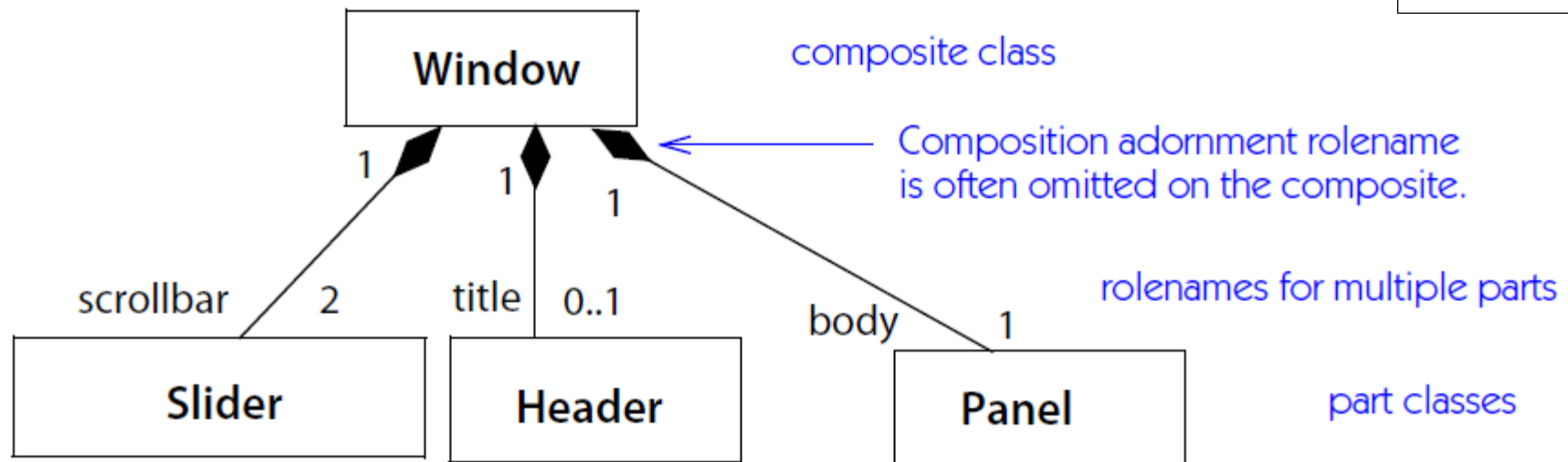
# Enumeration Classes



- An **enumeration** (e.g., `PatronType`) specifies a predefined list of choices
- Do not show association with an enumeration, indicate as type of attribute
- Often, the enumeration is defined within the class that needs the enumeration (unless it is used by several classes)

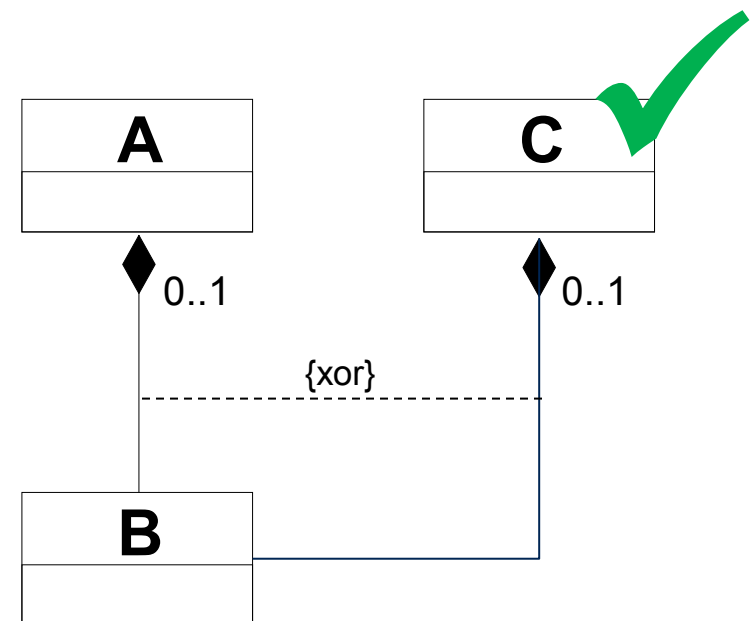
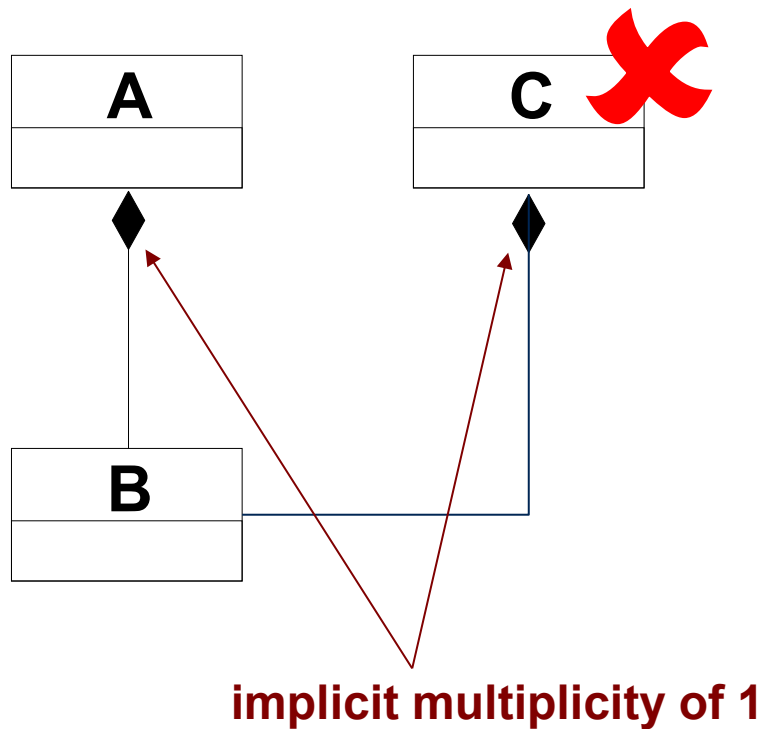
# Composition vs. Aggregation vs. Association (1)

- Composition
  - Part-whole relationship
  - Lifecycles of Slider, Header, and Panel objects (the parts) are tied to the lifecycle of their Window object (the parent or container): **destroy parts when parent is destroyed**
  - Transitive and antisymmetric (acyclic)



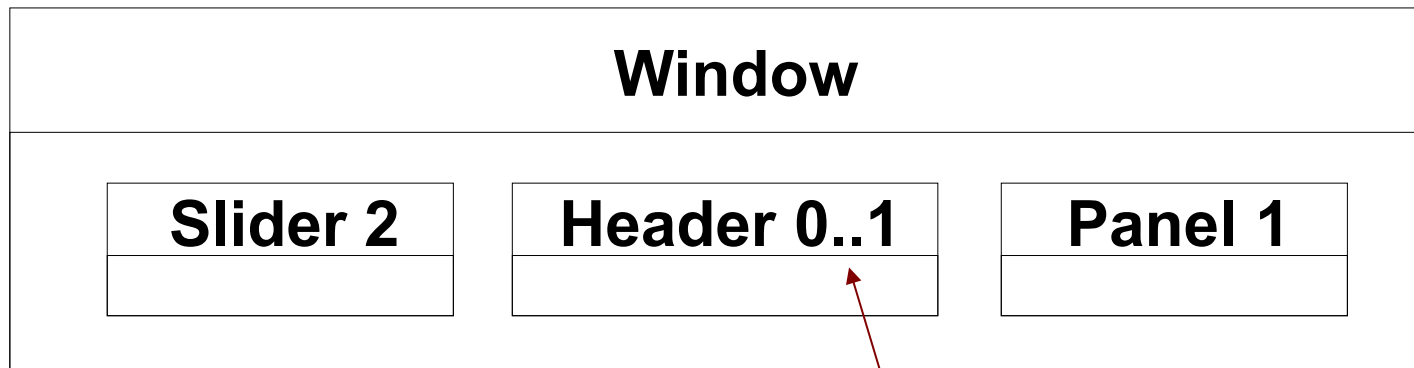
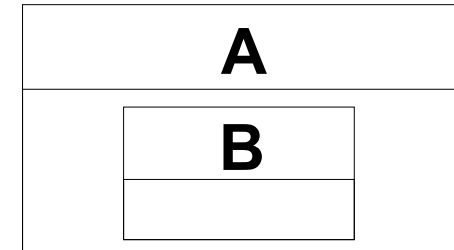
# Composition vs. Aggregation vs. Association (2)

- Consequently, a composed part cannot be contained in two parents at the same time, i.e., the example on the left is incorrect but the example on the right is possible



# Composition vs. Aggregation vs. Association (3)

- Alternative representation of Composition
  - More compact but does not allow role names to be specified



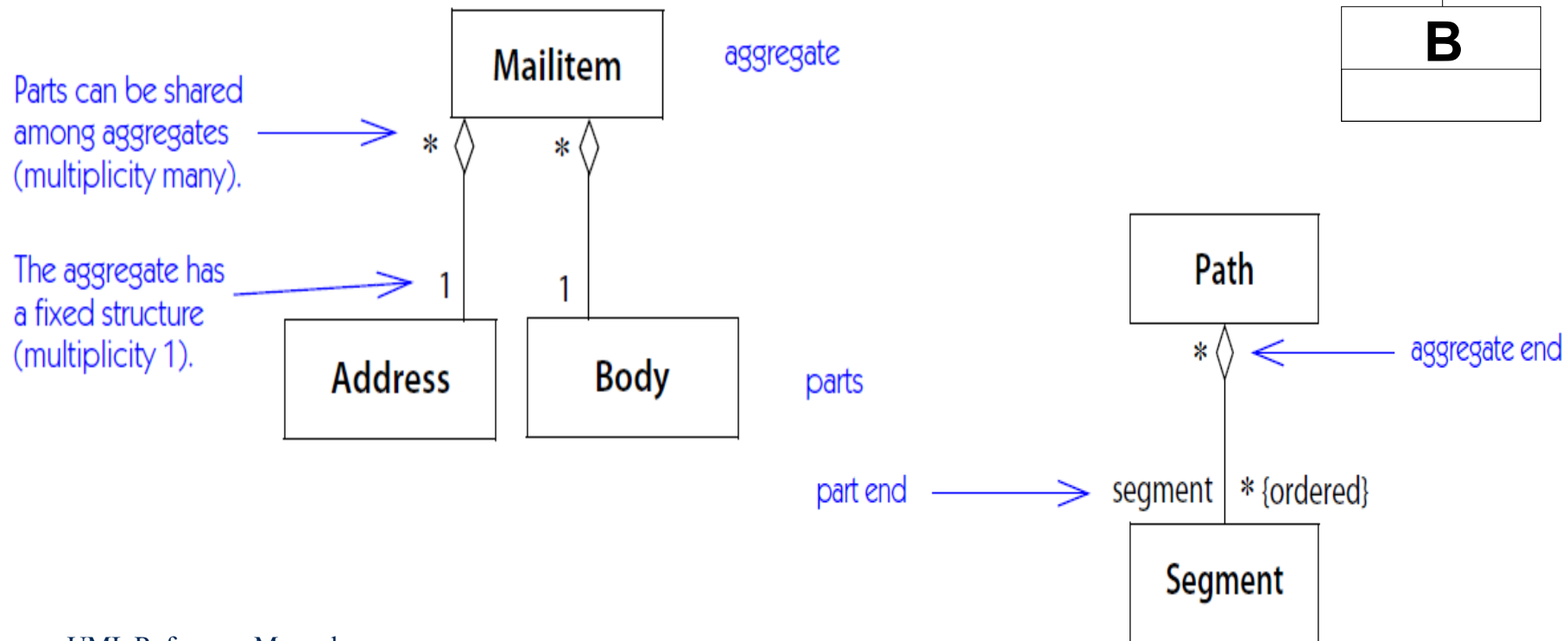
**multiplicity**



# Composition vs. Aggregation vs. Association (4)

- Aggregation

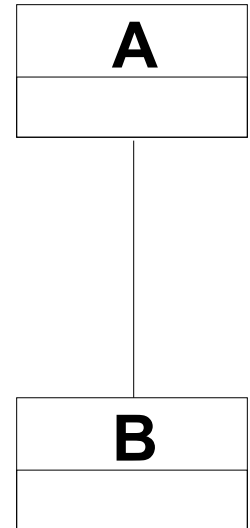
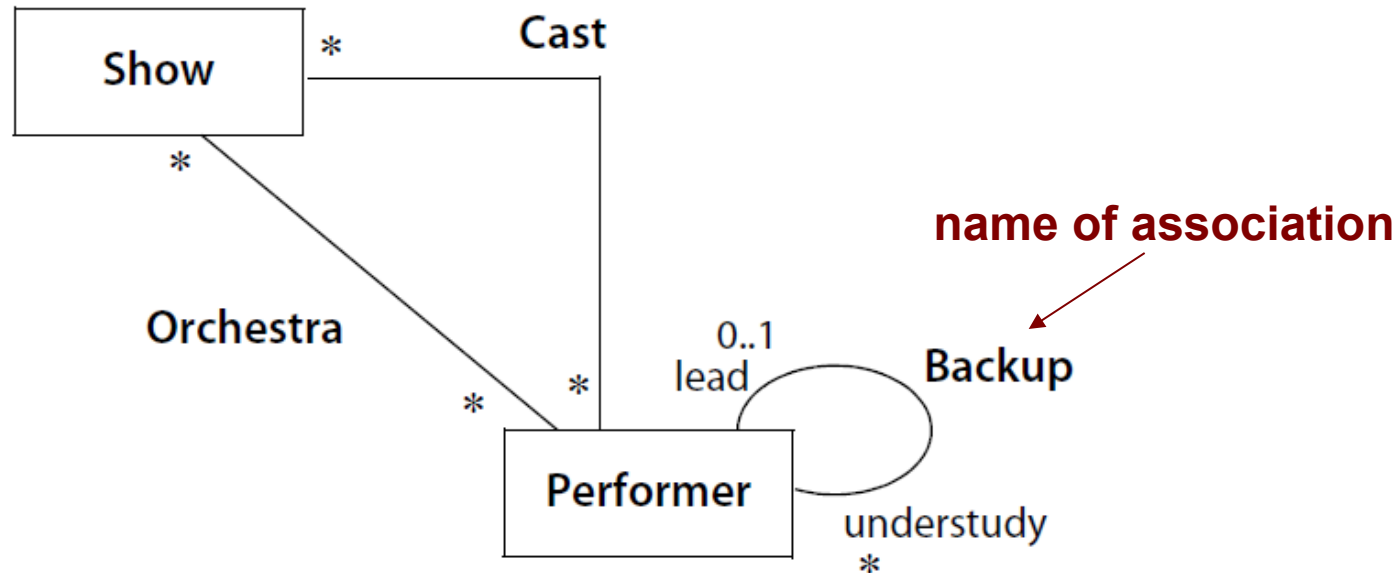
- Also part-whole relationship like composition, but not tied to lifecycle (least used relationship – typically not differentiated from association when generating code)
- Transitive and antisymmetric (acyclic)



Source: UML Reference Manual

# Composition vs. Aggregation vs. Association (5)

- Association
  - Basic relationship
  - An object can only interact with another object with which it has an explicit or implicit relationship



Source: UML Reference Manual

# Reification and Composition/Aggregation

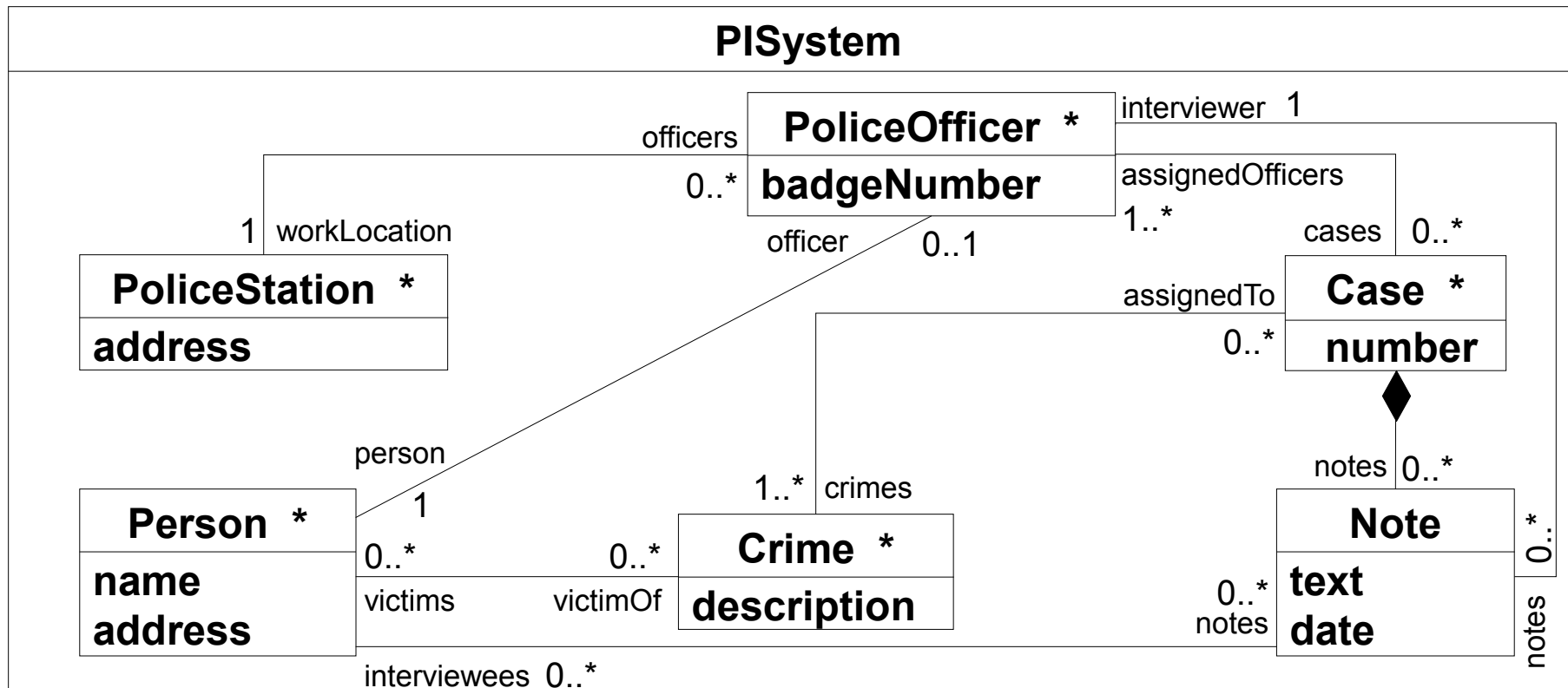
- The result of physical composition/aggregation is naturally an object (class)
  - E.g., a Car is composed of a Motor, Wheels, etc.
  - E.g., a Window is composed of an optional Header, two Sliders, and a Panel
  - E.g., a Path consists of Segments
- Reification is often applied to non physical composition/aggregation
  - E.g., a Department with its Personnel, Buildings, Budget, etc.

# Identifying Composition and Aggregation

- Police Information (PI) System

*work in progress – see later page for complete solution*

This system helps the Java Valley police officers keep track of the cases they are assigned to do. Officers may be assigned to investigate particular crimes, which involves interviewing victims at their homes and entering notes in the PI system.



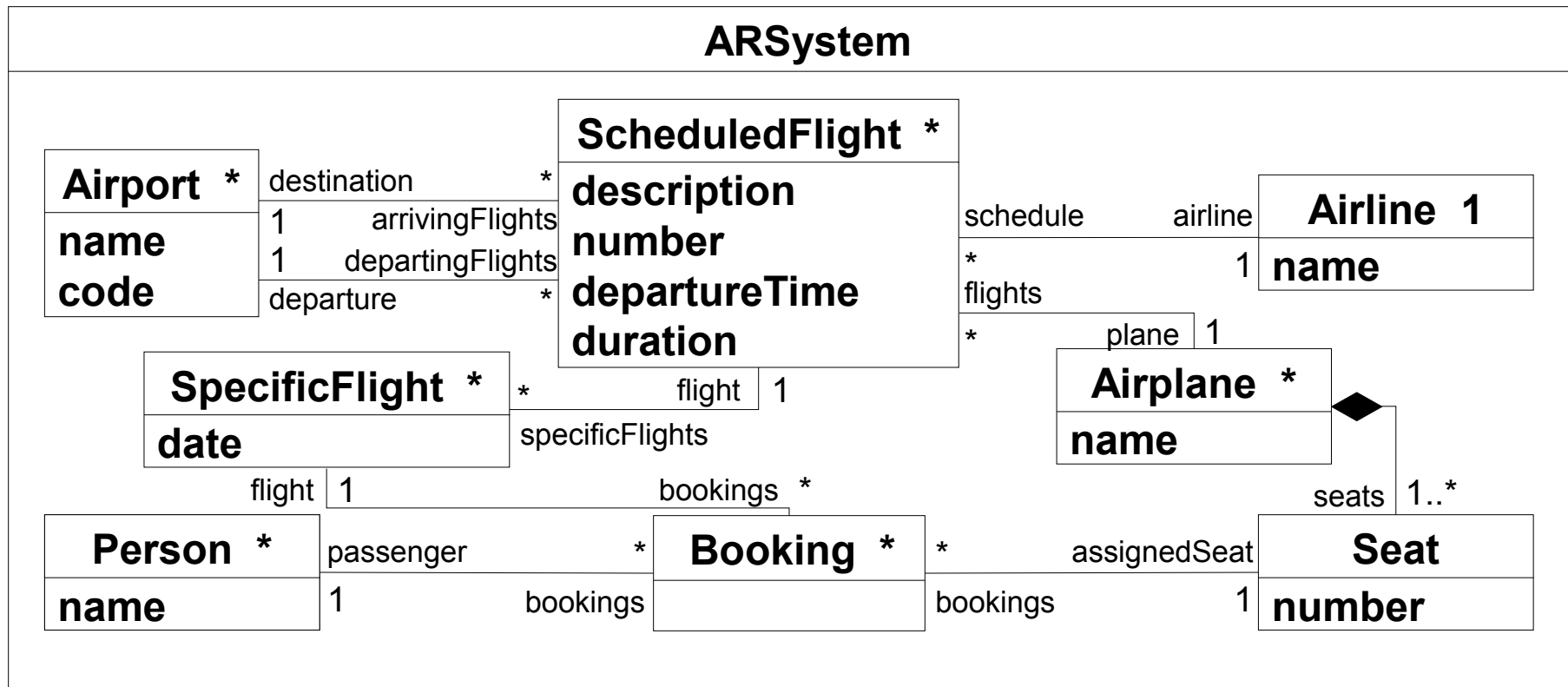


# Exercise: Identifying Composition and Aggregation

- Airline Reservation (AR) System

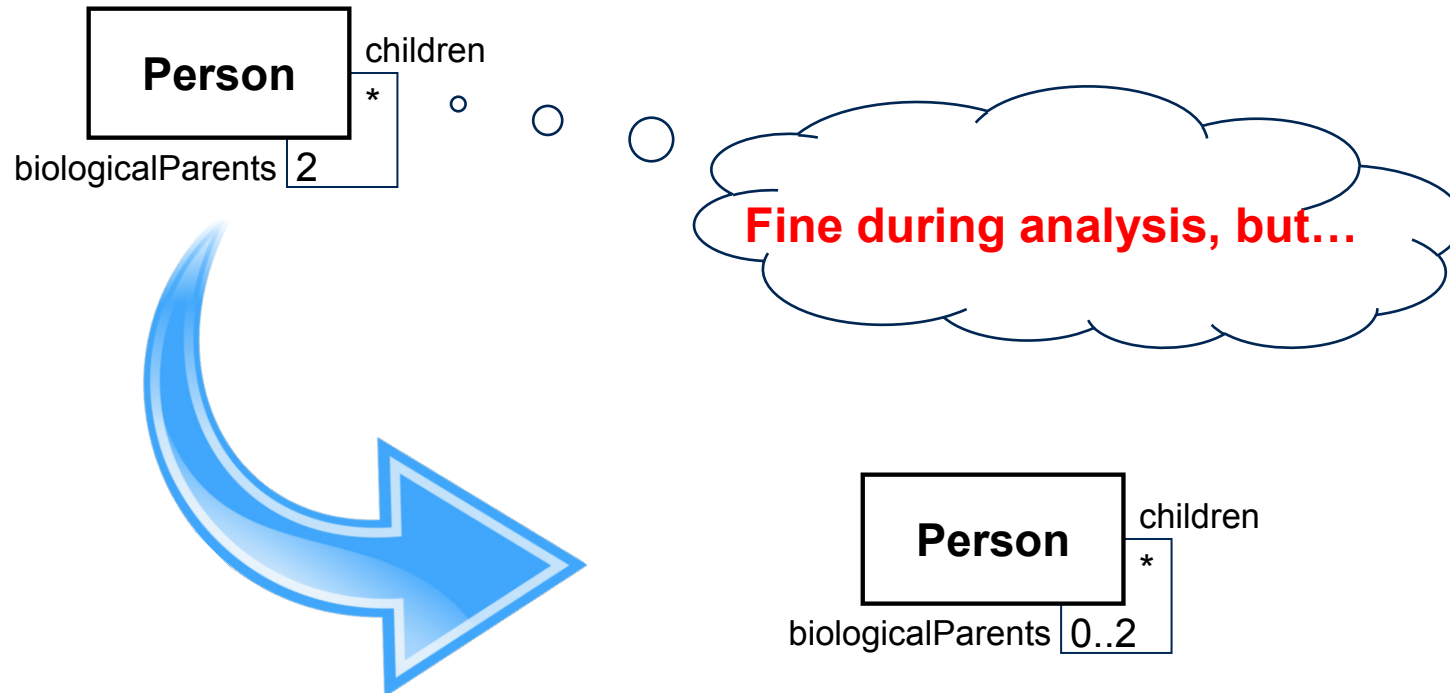
*work in progress – see later page for complete solution*

Ootumlia Airlines runs sightseeing flights from the Java Valley, the capital of Ootumlia. The AR system keeps track of passengers who will be flying in specific seats on various flights. Ootumlia Airlines runs several daily numbered flights on a regular schedule.



# Modeling Reality vs. Computer Program

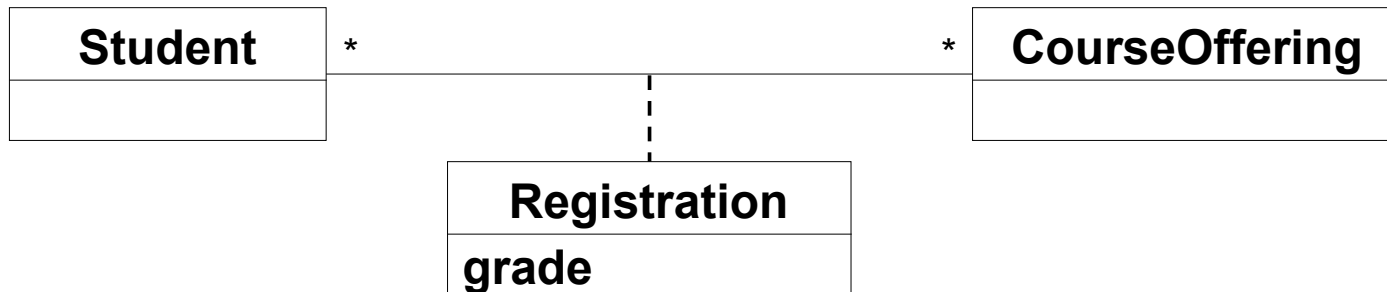
- Do you see a problem with the class diagram on the left?



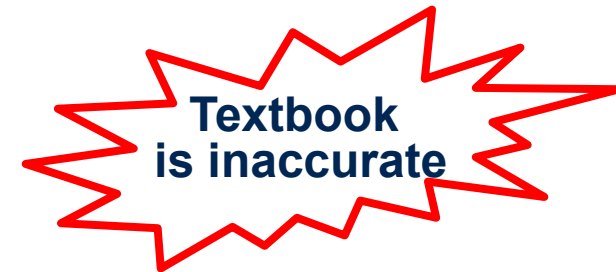
- Cannot model a **Person** as having two biological parents because that would force the parents to have two parents, ad infinitum

# Association Classes

- Sometimes, an attribute that relates to two associated classes cannot be placed in either of the classes



- One Registration for each associated Student/CourseOffering pair in the system



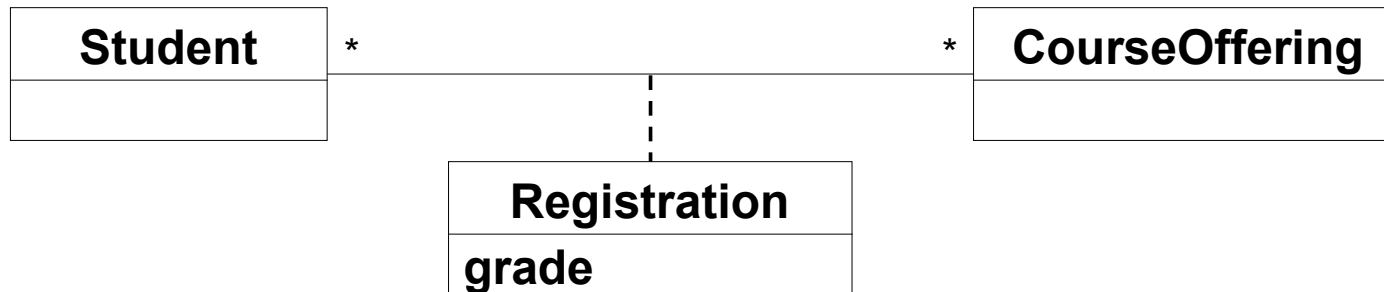
- The following alternative is almost equivalent. What's the difference?



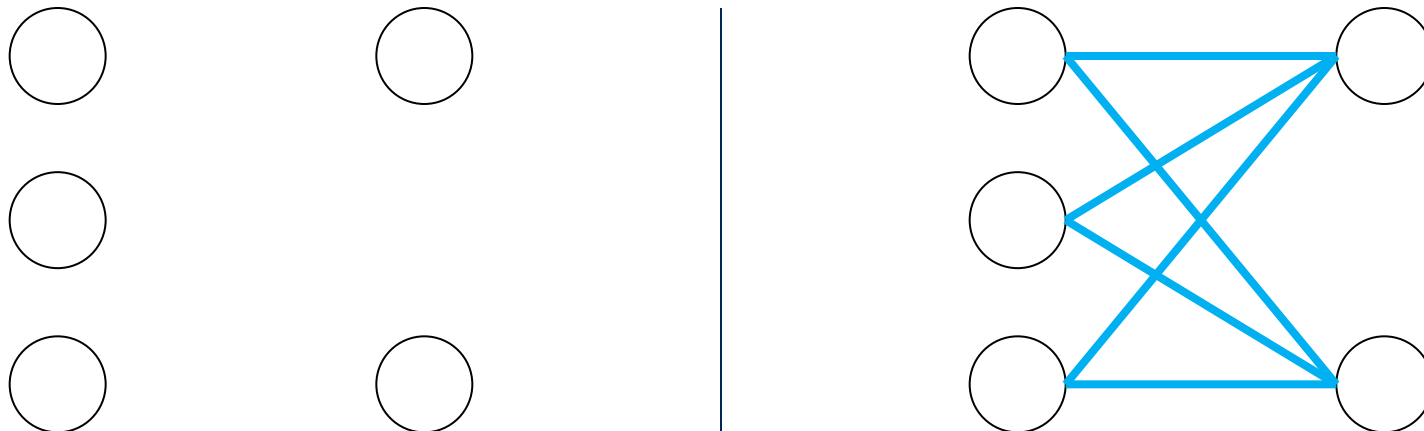
- Twins allowed in contrast to association class (i.e., more than one Registration for the same Student and same CourseOffering may exist)

# Exercise: Association Classes (1)

- How many Registration objects can minimally and maximally exist given the following class diagram?



- Minimally zero; maximally  $\# \text{Students} * \# \text{CourseOfferings}$



... represents instance

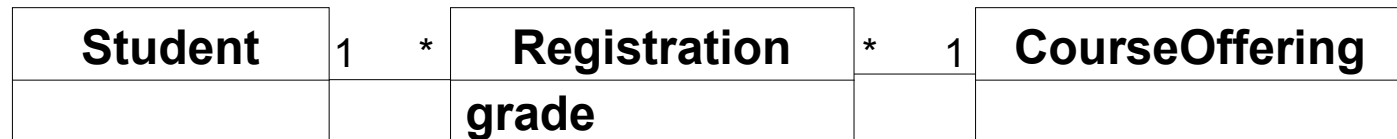


... link (one registration instance exists for each link)

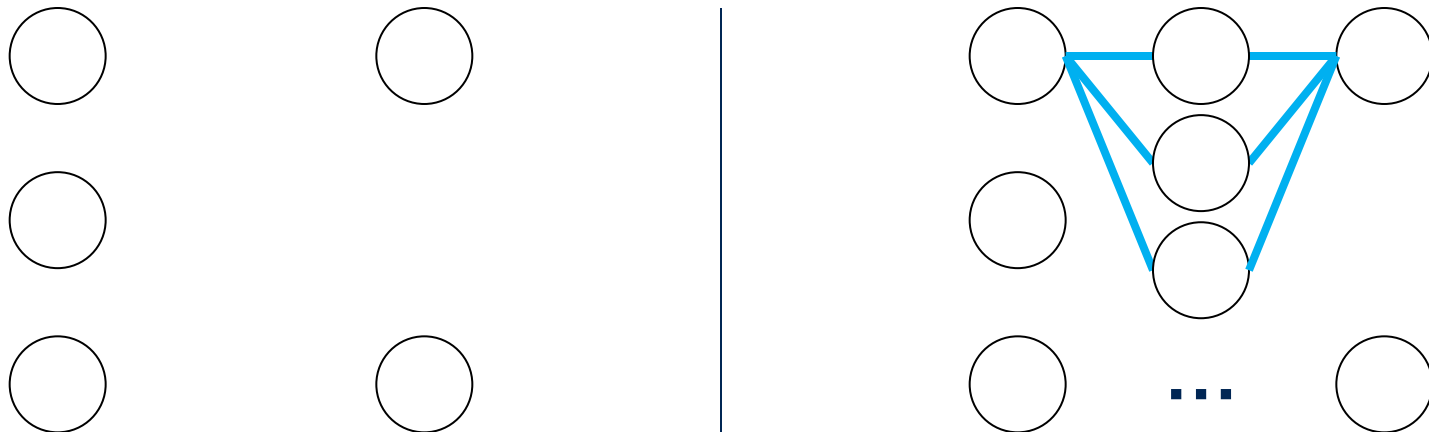


# Exercise: Association Classes (2)

- How many Registration objects can minimally and maximally exist given the following class diagram?



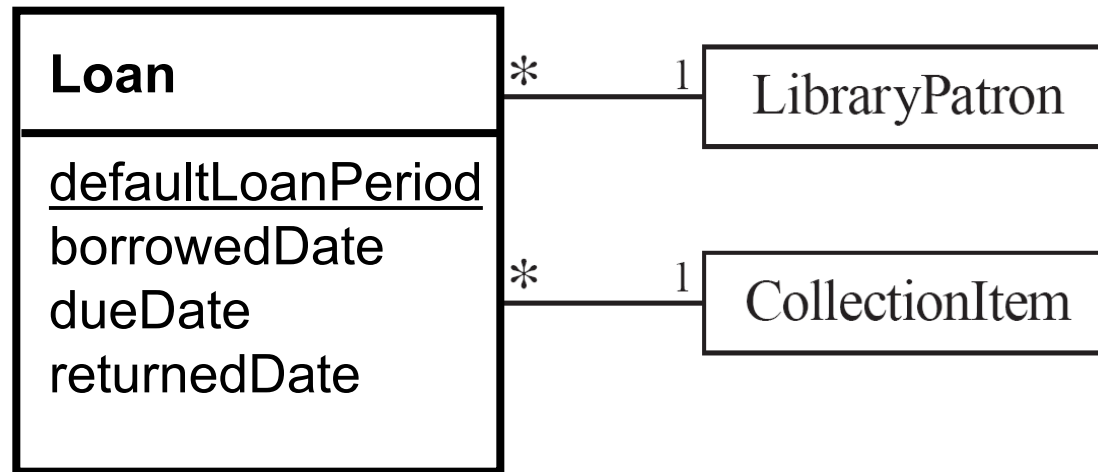
- Minimally zero; maximally unlimited



○ ... represents instance    — ... link

## Exercise: Association Classes (3)

- Should the **Loan** class be an association class?



- No, because it is possible that the same library patron is borrowing the same collection item multiple times
- In other words, many Loan objects with the same library patron/collection item pair may exist

# Exercise: Terminology

- **Correct the following statements by changing at the most one word per sentence.**

**class**

- An ~~instance~~ represents a set of objects.

**link**

- There is an ~~association~~ between library patron Mike and his last loan.

**attribute value**

- Mike's loan has an ~~attribute~~ that is the next Saturday.

**link**

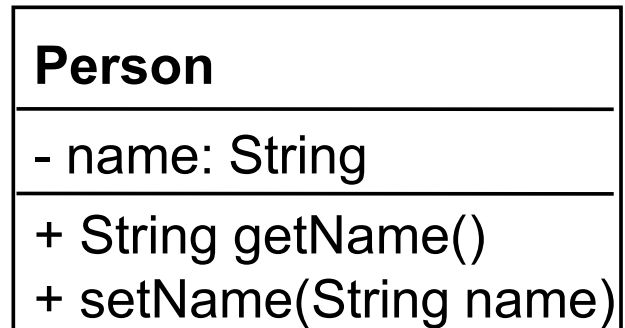
- An ~~association~~ is a relationship between two objects.

**classes**

- An association is a relationship between two ~~objects~~.

# Operations and Methods: Procedural Abstractions

- **Operation**: specification of a transformation or query that an object may be called to execute
- **Method**: implementation of an operation
- Different kinds of operations
  - **Constructor**: create, build, and initialize an object
  - **Observer**: retrieve information about the state of an object
  - **Modifier**: alter the state of an object
  - **Destructor**: destroy an object
  - **Iterator** (for objects that encapsulate a collection of other objects): access all parts of a composite object, and apply some action to each of the parts

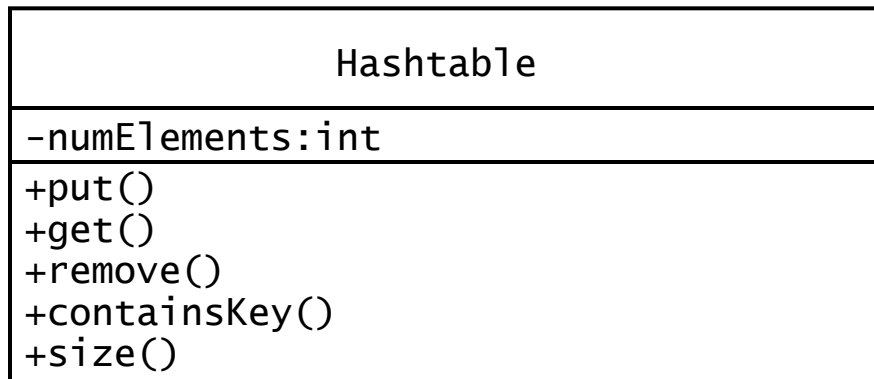


```
public class Person {  
    private String name;  
    public Person(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

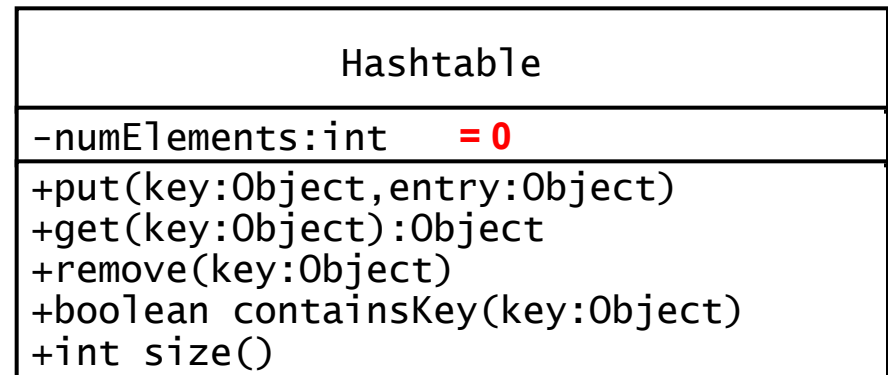


# Type Signature Information

Initial values may  
also be provided



**Attributes and operations  
without type information  
may be acceptable  
during analysis, but are  
added during design**



# Data Types

- UML supports many pre-defined and user-defined data types
  - Predefined ones include: Integer, UnlimitedInteger, String, Time, Boolean, and many others targeted for defining UML itself and for defining UML models
  - Many tools support other common types: Float, Char, Date, etc – these may be influenced by the target programming language (Java, C, ...)
  - User-defined types also include user classes
- Data types can be used for operation signatures (parameters and return value)
  - Useful for static checking of models and for code generation

# Attributes, Operations, and Visibility

- Syntax for Attributes
  - [visibility] name [: type] [multi] [= default] [{prop}]
    - **+ id : String {readOnly}**
    - **num = 1234**
  - readOnly is the most common attribute property
- Syntax for Operations
  - [visibility] [type] name [(params)] [{prop}]  
where each parameter is defined using:  
[direction] name : type [multi] [= default]
    - **+ boolean containsKey(key : Object)**
- UML defines four levels of visibility
  - Private (-), Protected (#) , Public (+), Package (~) (default)

# Visibility in UML and in Java

- UML visibility

Modifier	Class	Package	Subclass	World
Public	Y	Y	Y	Y
Protected	Y	N	Y	N
Package (default)	Y	Y	N	N
Private	Y	N	N	N

- Java visibility

Modifier	Class	Package	Subclass	World
Public	Y	Y	Y	Y
Protected	Y	Y	Y	N
No modifier	Y	Y	N	N
Private	Y	N	N	N



# Operations: Directions and Properties

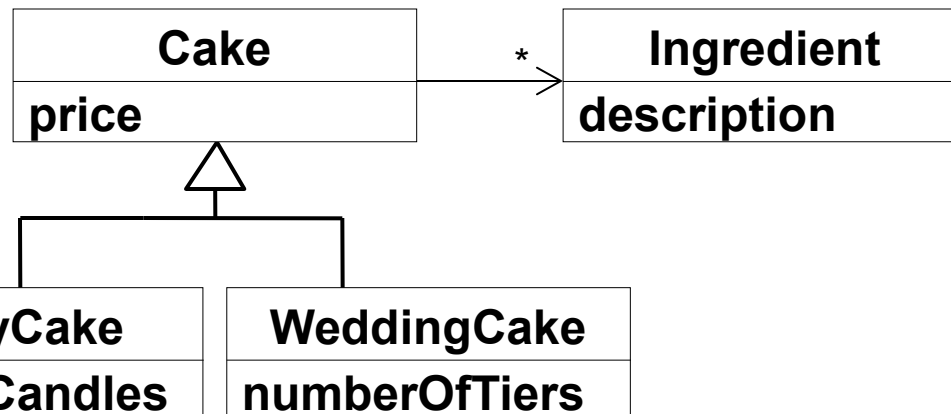
- Directions
  - **in**: the operation cannot modify the parameter and the caller does not need to see it again
  - **out**: the operation can modify the parameter and returns it to the caller
  - **inout**: the operation can modify the parameter
- Properties
  - **concurrent**: multiple calls may arrive simultaneously, and they all proceed concurrently
  - **guarded**: multiple calls may arrive simultaneously, but only one can proceed at a time
  - **sequential**: only one call may come at any time

# Generalization – Inheritance (1)

- Reuse – avoid duplication
  - Structure and behavior specified for a superclass also applies to the subclass
  - Subclass inherits from superclass
  - Transitive and antisymmetric (acyclic)
  - BirthdayCake and WeddingCake both also have Ingredients and a price, but only BirthdayCake has candles and only WeddingCake has tiers



**Generalization:**  
A taxonomic relationship between a more general and a more specific element. The more specific element is fully consistent with the more general element and contains additional information.

**Inheritance:**  
The mechanism by which more specific elements incorporate structure and behavior defined by more general elements.



Source: UML Reference Manual

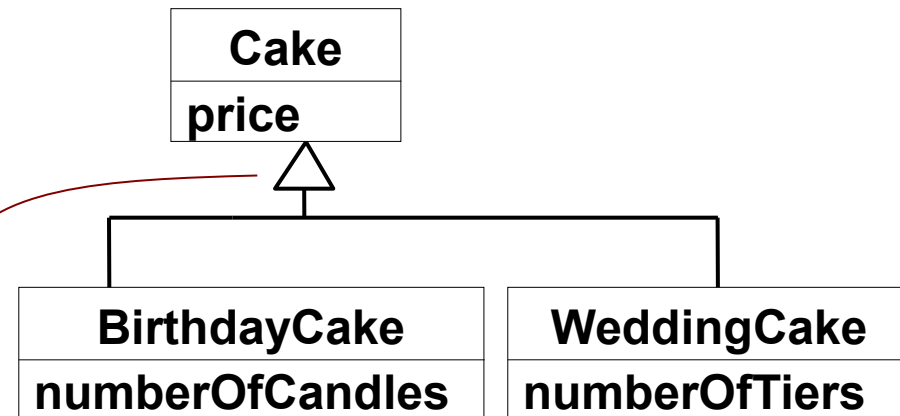
# Generalization – Inheritance (2)

- Definition: **implicit possession** by a subclass of features defined in a superclass (attributes, associations, operations)
  - Subclass may add additional features
- However, not enough → need principle of substitutability
- **Liskov Substitution Principle**
  - S is a subclass of T, if and only if any instance of T can be substituted by an instance of S, without any visible effect
  - If you have a variable whose type is a superclass (e.g., Account), then the program should work properly if you replace an instance of that superclass or any of its subclasses (e.g., CheckingAccount or SavingsAccount) in that variable. The program using the variable should not be able to tell which class is being used, and should not care!
- Always check generalizations to ensure they obey the **isa rule**
  - A CheckingAccount is an Account 
  - A Province is a Country 

# Inheritance in Java

```
public class Cake {
    private int price;

    public Cake(int price) {
        this.price = price;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
}
```



```
public class BirthdayCake extends Cake {
    private int numberOfCandles;

    public BirthdayCake(int price, int numberOfCandles) {
        super(price);
        this.numberOfCandles = numberOfCandles;
    }
    public int getNumberOfCandles() {
        return numberOfCandles;
    }
    public void setNumberOfCandles(int numberOfCandles) {
        this.numberOfCandles = numberOfCandles;
    }
}
```



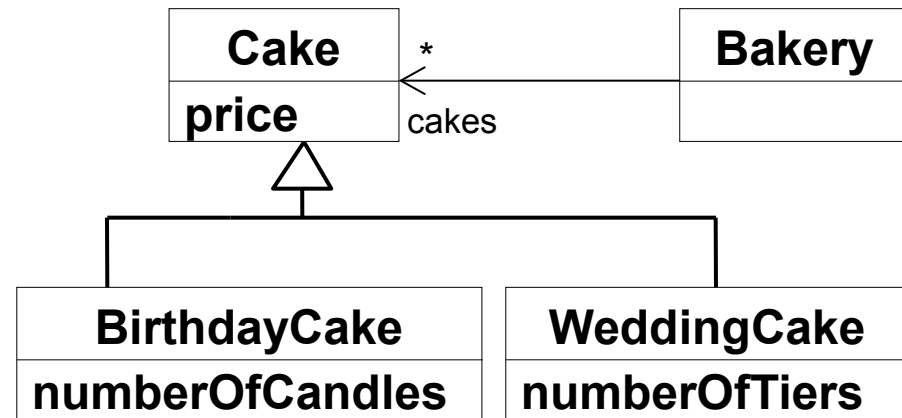
# Example: Liskov Substitution Principle

```
// in this example, ingredients are not
// specified for any cake
Cake simpleCake = new Cake(10);
Cake fancyCake = new WeddingCake(30, 3);
BirthdayCake happyBirthday =
    new BirthdayCake(25, 18);
WeddingCake justMarried =
    new WeddingCake(45, 5);
```

```
Bakery bakery = new Bakery();
bakery.addCake(simpleCake);
bakery.addCake(happyBirthday);
bakery.addCake(fancyCake);
bakery.addCake(justMarried);
```

```
int totalPrice = 0;
for (Cake cake : bakery.getCakes()) {
    totalPrice += cake.getPrice();
}
```

```
int nrCandles =
    happyBirthday.getNumberOfCandles();
```



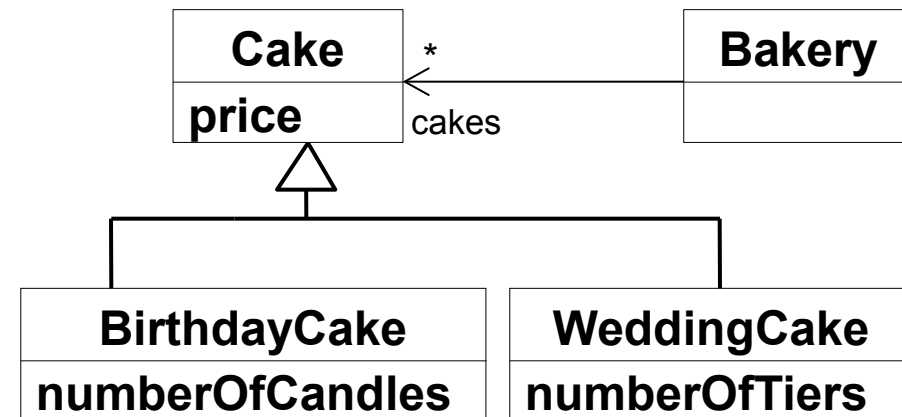
**because of the substitution principle, a software engineer does not have to care about whether a cake in the bakery's list of cakes is a regular cake, a birthday cake, or a wedding cake → they all behave like a cake**

**the variable happyBirthday holds an object of type BirthdayCake → therefore, it is possible to get the number of candles**

# Exercise: Liskov Substitution Principle

```

Cake simpleCake = new Cake(10);
Cake fancyCake = new WeddingCake(30, 3);
BirthdayCake happyBirthday =
    new BirthdayCake(25, 18);
WeddingCake justMarried =
    new WeddingCake(45, 5);
  
```



- Which of the following statements is allowed?  
Which one is not allowed?

**CORRECT** simpleCake.getPrice();  
**WRONG** simpleCake.getNumberOfCandles();  
**CORRECT** happyBirthday.getNumberOfCandles();  
**CORRECT** happyBirthday.getPrice();  
**WRONG** fancyCake.getNumberOfTiers();  
**CORRECT** fancyCake.getPrice();  
**CORRECT** justMarried.getNumberOfTiers();  
**CORRECT** justMarried.getPrice();

this is possible but  
typically should not  
be done

need to cast fancyCake to  
WeddingCake for this to work

```
((WeddingCake) fancyCake)
    .getNumberOfTiers();
```

# Exercise: Identify Good Generalizations

- Which of the following would not form good superclass-subclass pairs (generalizations), and why? Look for violations of the isa rule, poor naming, and other problems.

## Superclass – Subclass:

## Answer:

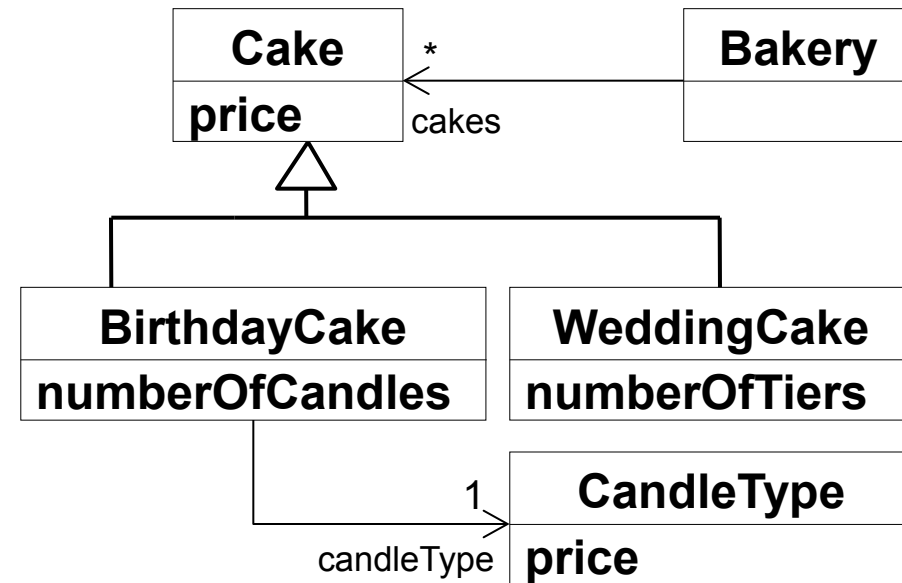
- Money – CanadianDollar **Bad: You can't say CanadianDollar is a Money (also, CanadianDollar is an instance of Currency)**
- Bank – Account **Bad: Account  $\neq$  a Bank (isa check fails)**
- OrganizationUnit – Division **Good**
- SavingsAccount – CheckingAccount **Bad, isa check fails**
- Account – Account12876 **Bad, Account12876 is not a class, but an instance**
- Person ✓
- People – Customer **Bad, isa check fails (Person ok)**
- Continent – Country **Bad, isa check fails**
- Municipality – Neighborhood **Bad, isa check fails**



# Overriding Methods

```
public class Cake {
    private int price;

    public Cake(int price) {
        this.price = price;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
}
```



```
public class BirthdayCake extends Cake {
    private int numberOfCandles;
    private CandleType candleType;
    ...
    @Override
    public int getPrice() {
        return super.getPrice() + numberOfCandles * candleType.getPrice();
    }
    ...
}
```

**overridden method**

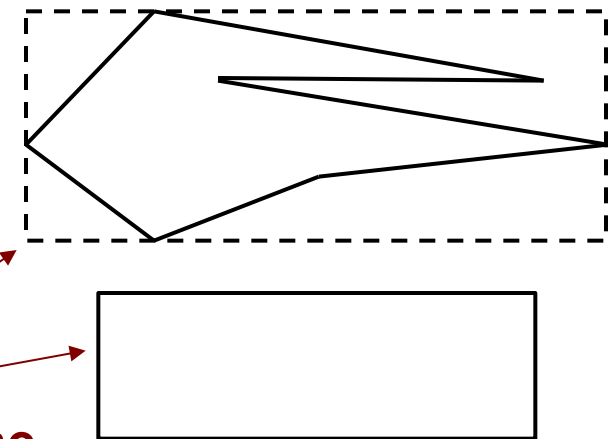


# Three Reasons for Overriding Methods

- **Overriding** = a method would be inherited from a superclass, but a subclass contains a new version of the method instead
- Three reasons:
  - For **restriction** (e.g., it is possible to debit money from an Account, but its subclass MortgageAccount restricts debit to a single withdrawal when the account is opened)
  - For **extension** (e.g., a BirthdayCake charges an extra fee for the candles)
  - For **optimization** (e.g., the bounding rectangle of a Shape may be quite complicated to calculate, but it is much more straightforward to calculate it for a subclass of Shape such as Rectangle)

**bounding  
rectangle**

**the rectangle is the  
bounding rectangle**



# Polymorphism

- **Polymorphism** (from the Greek meaning "having multiple forms")
  - A property of object oriented software by which an abstract operation may be performed in different ways in different classes
  - Requires that there be **multiple methods of the same name** (e.g., `getPrice()` in the earlier example)
  - Reduces the need for programmers to code many if-else or switch statements

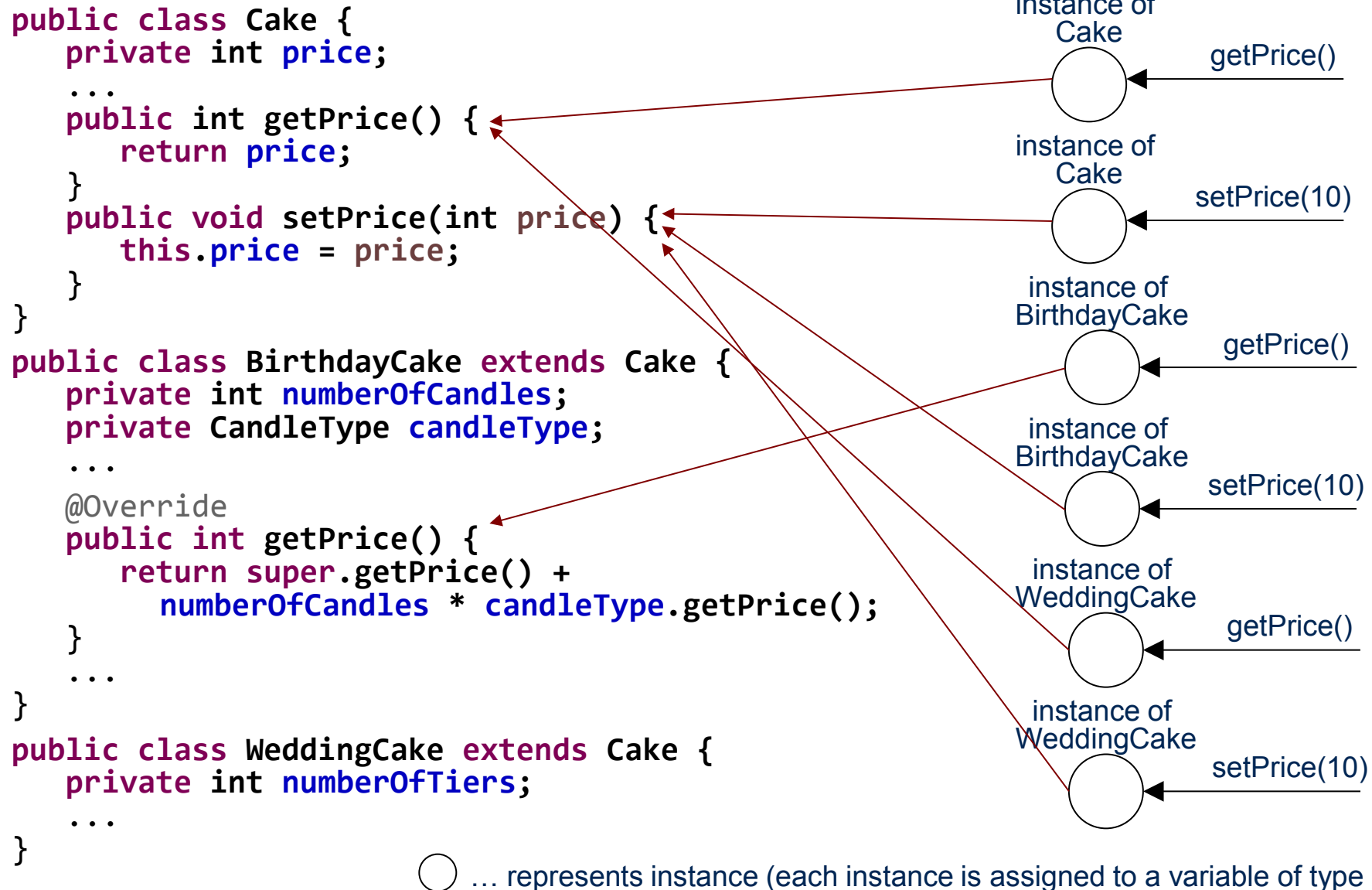
```
public class CakeNo00 {  
    private int price;  
    public enum CakeType { Regular,  
                           BirthdayCake, WeddingCake };  
    private CakeType cakeType;  
  
    ...  
    public int getPrice() {  
        if (cakeType ==  
            CakeType.BirthdayCake) {  
            // additional flat fee of  
            // $10 for the candles  
            return price + 10;  
        }  
        return price;  
    }  
    ...  
}
```

# Dynamic Binding

- **Dynamic binding** occurs when the decision about which method to run can only be made at runtime
- Needed when:
  - A variable is declared to have a superclass as its type, and
  - There is more than one possible polymorphic method that could be run among the type of the variable and its subclasses
- The choice of which polymorphic method to execute depends on the class of the object that is in a variable
  - 1.If there is a concrete method for the operation in the current class, run that method
  - 2.Otherwise, check in the immediate superclass to see if there is a method there; if so, run it
  - 3.Repeat step 2, looking in successively higher superclasses until a concrete method is found and run it
  - 4.If no method is found, then there is an error (in Java the program would not have compiled)



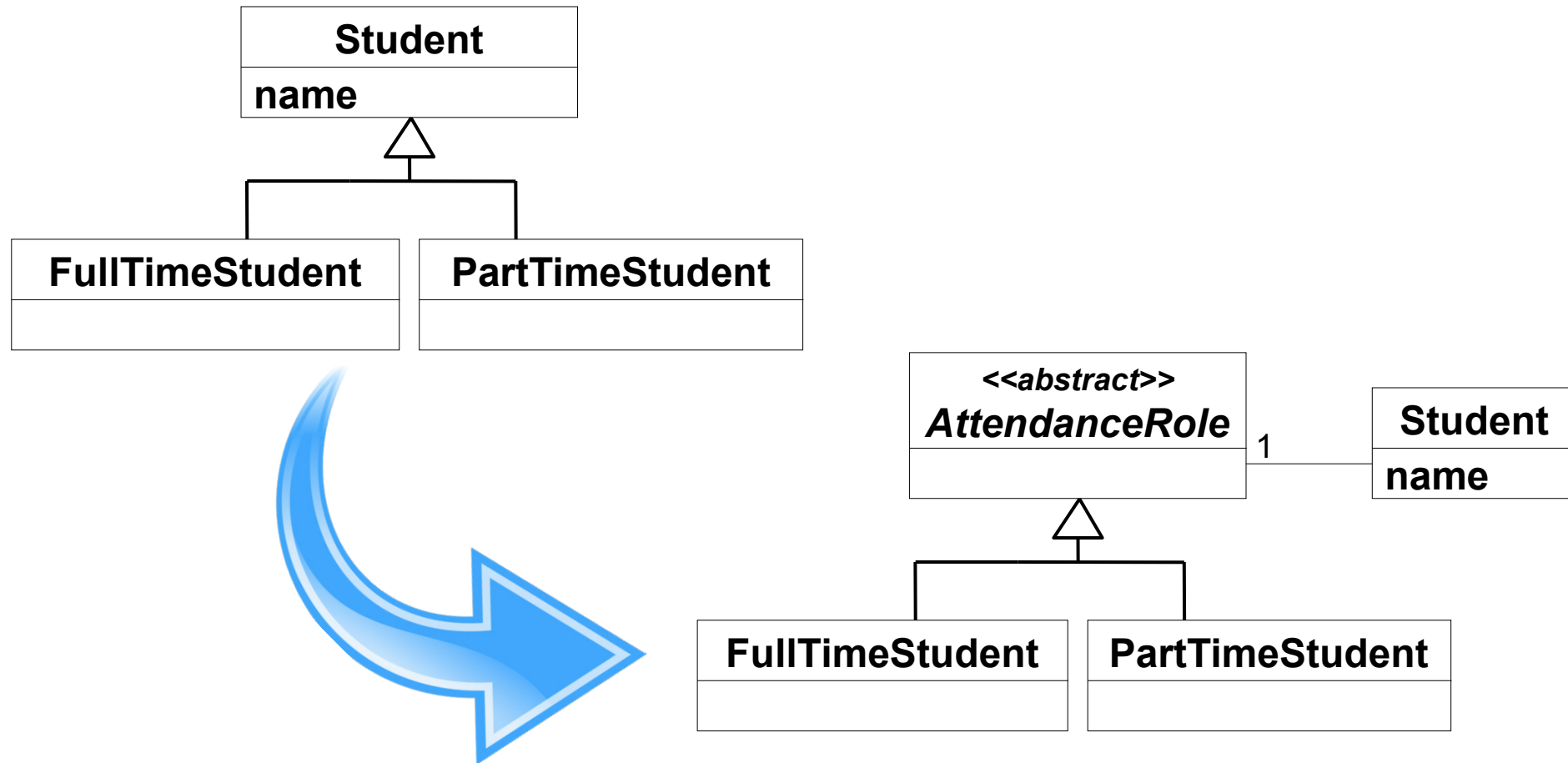
# Exercise: Dynamic Binding





# Avoid Having Instances Change Class

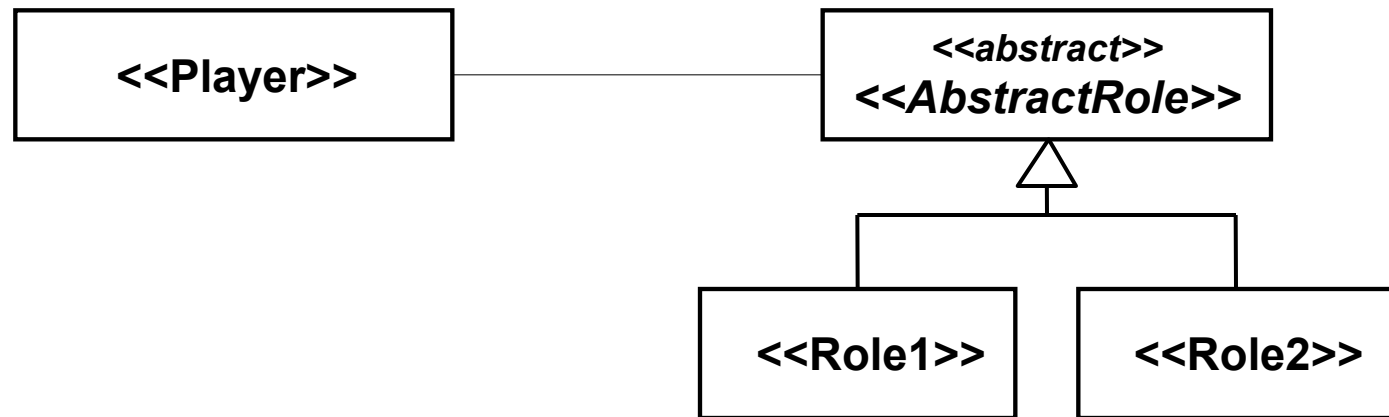
- Do you see a problem with the class diagram on the left?



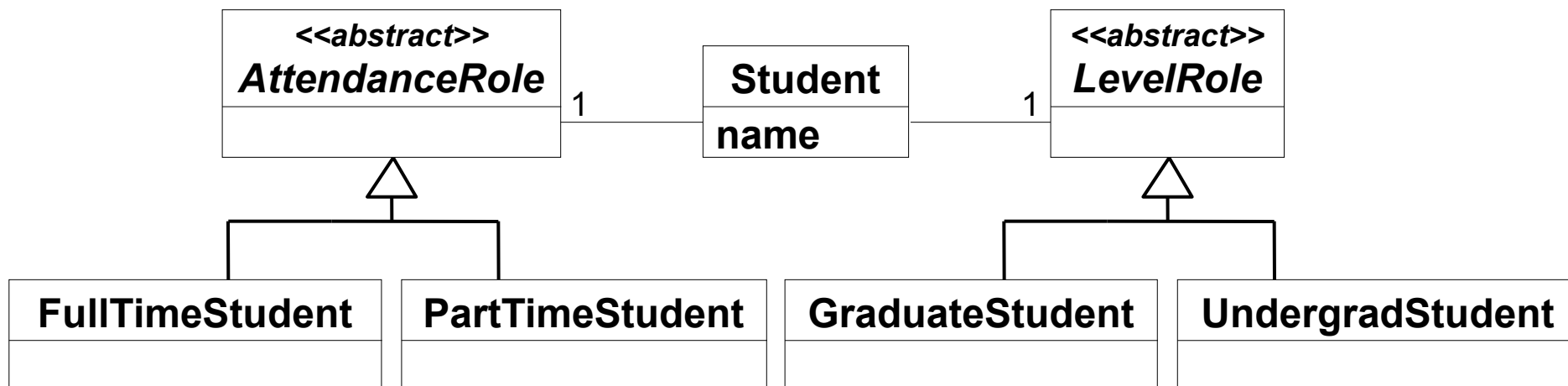
- Inappropriate hierarchy of classes, because a **Student** may be a full-time student at some point and a part-time student at another point

# Player-Role Pattern (1)

- An object may **play** different **roles** in different contexts
- What is the best way to model players and roles, so that a player can change roles or possess multiple roles without changing the class of an instance?

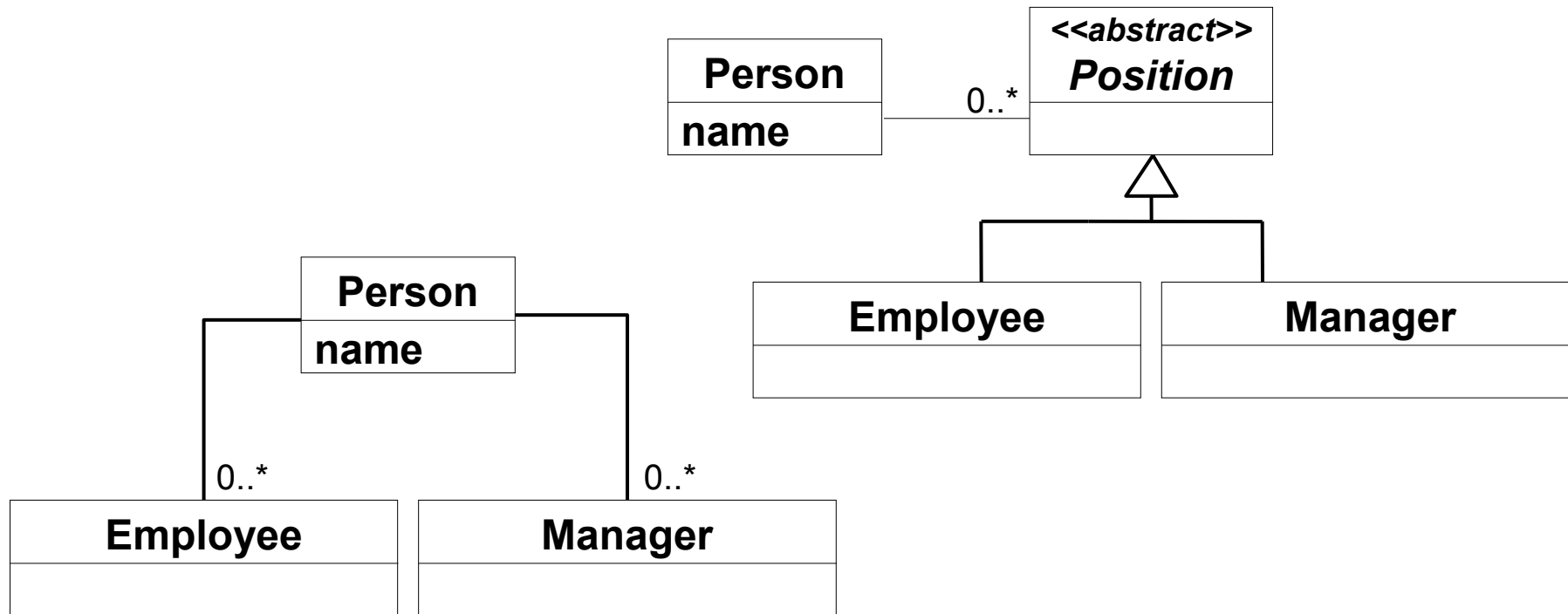


- Example



# Player-Role Pattern (2)

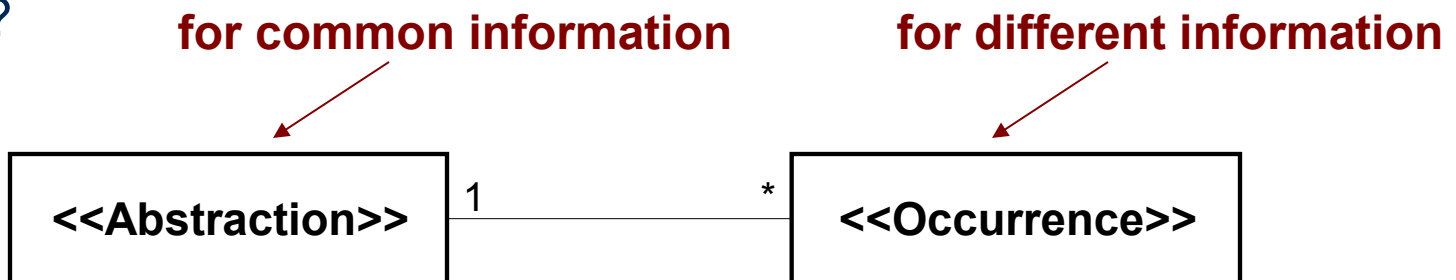
- Is there a difference between these two solutions?



- Equivalent from a player-role point of view, but bottom solution does not allow **Employee** and **Manager** to be treated the same way
- E.g., how many positions does a **Person** have?  
`getPositions().size()` vs `getEmployees().size() + getManagers().size()`

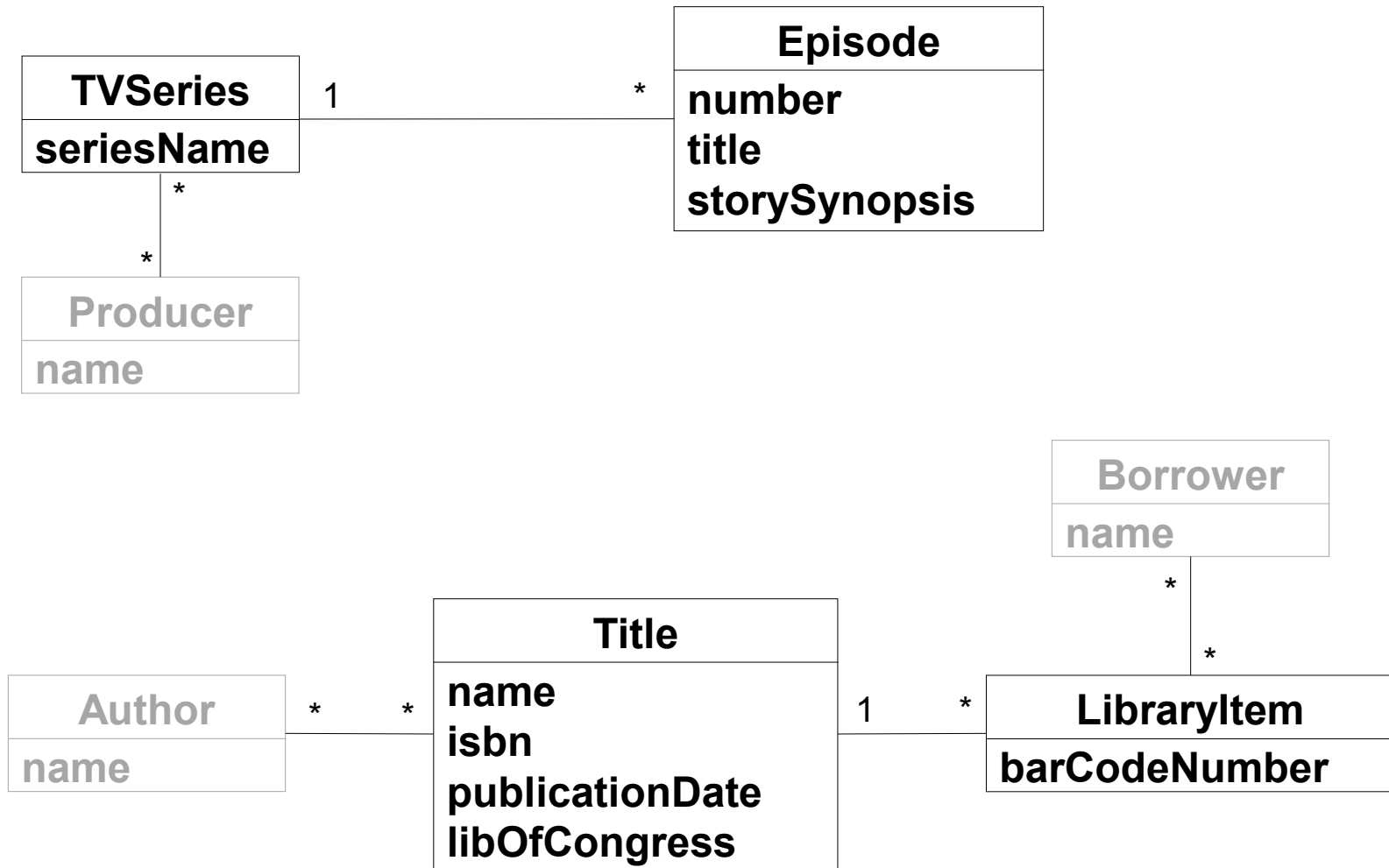
# Abstraction-Occurrence Pattern (1)

- **Occurrences**: a set of related objects that share common information but also differ from each other in important ways
  - E.g., episodes of a television series have the same producer and same series title, but different storylines
  - E.g., flights with the same flight number that leave at the same time every day to the same destination, but on different dates and with different passengers and crew
  - E.g., all copies of a book have the same title and author, but different barcode identifiers and different borrowers
- What is the best way to represent such sets of occurrences in a class diagram?

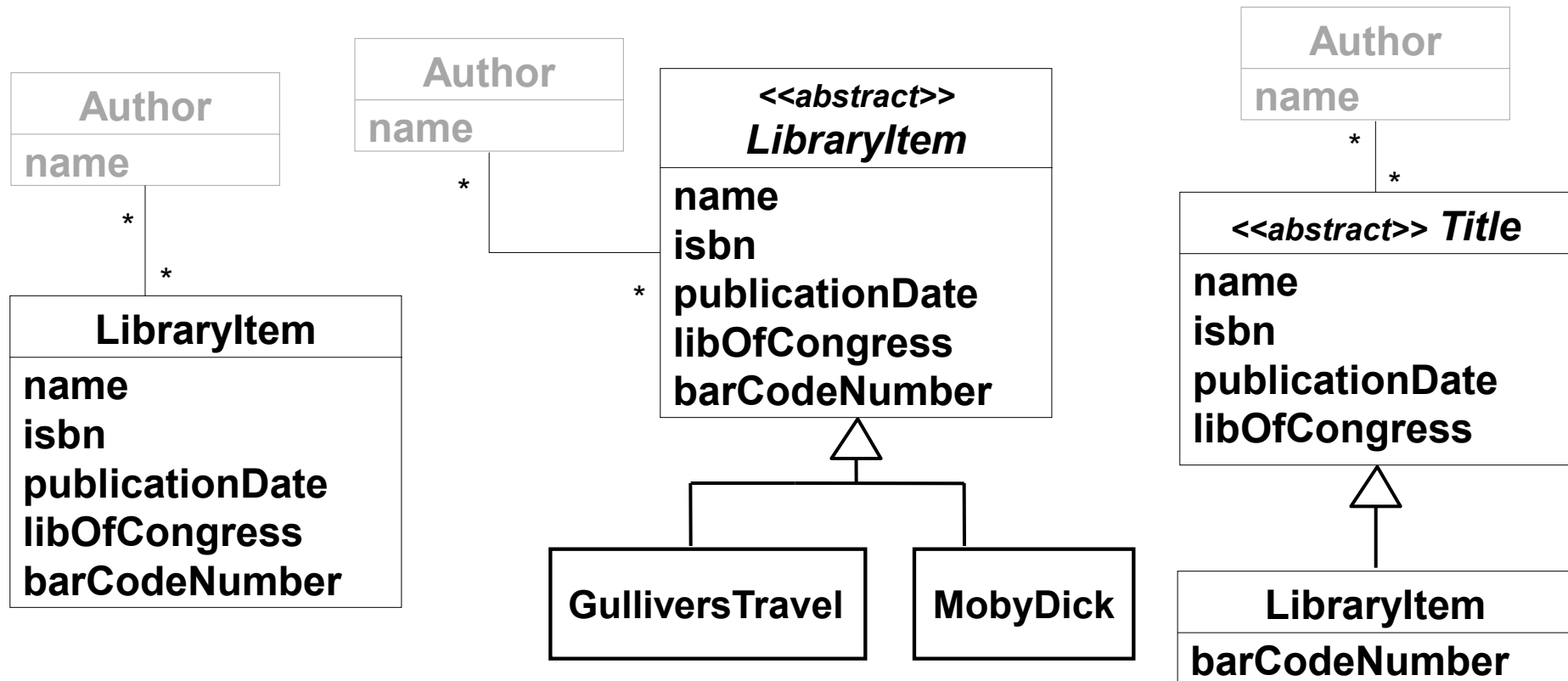




# Abstraction-Occurrence Pattern (2)

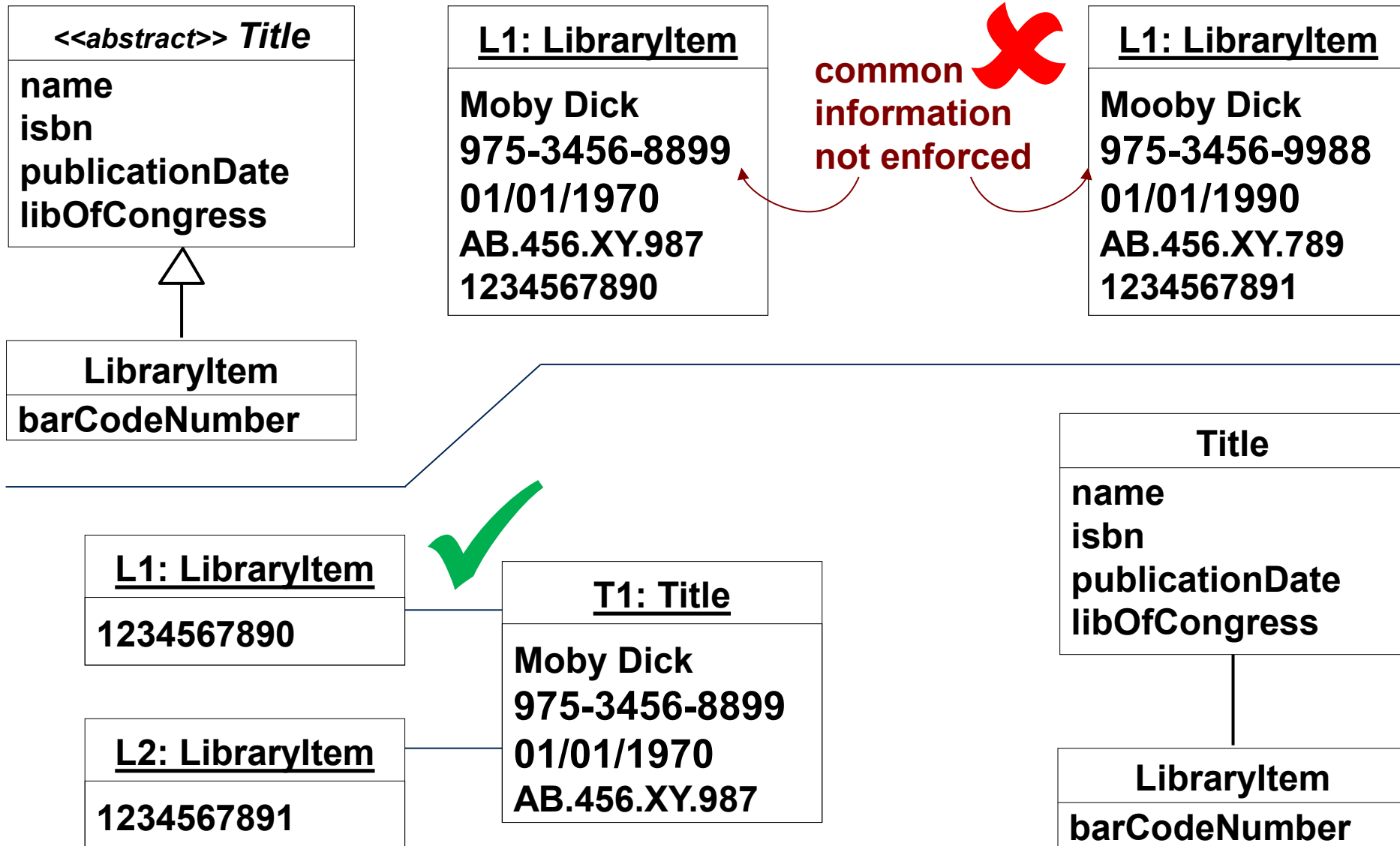


# Abstraction-Occurrence Antipatterns (1)



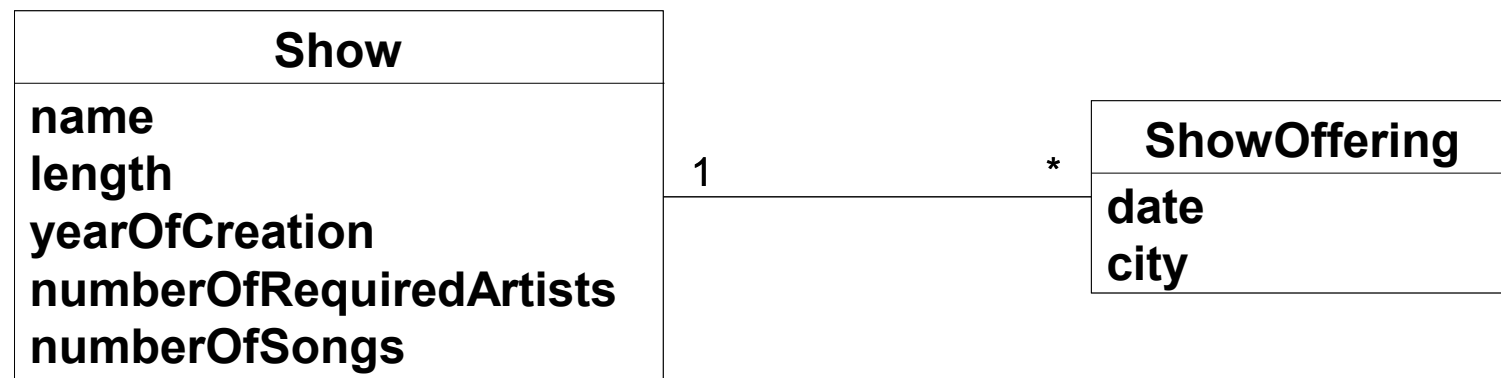
- Why are these solutions inappropriate?
- Repetition of common information (all attributes except **barCodeNumber**)
- Instances as subclasses
- Common information is not enforced (e.g., name of each **LibraryItem** may be different; see next slide for illustration)

# Abstraction-Occurrence Antipatterns (2)



# Exercise: Abstraction-Occurrence

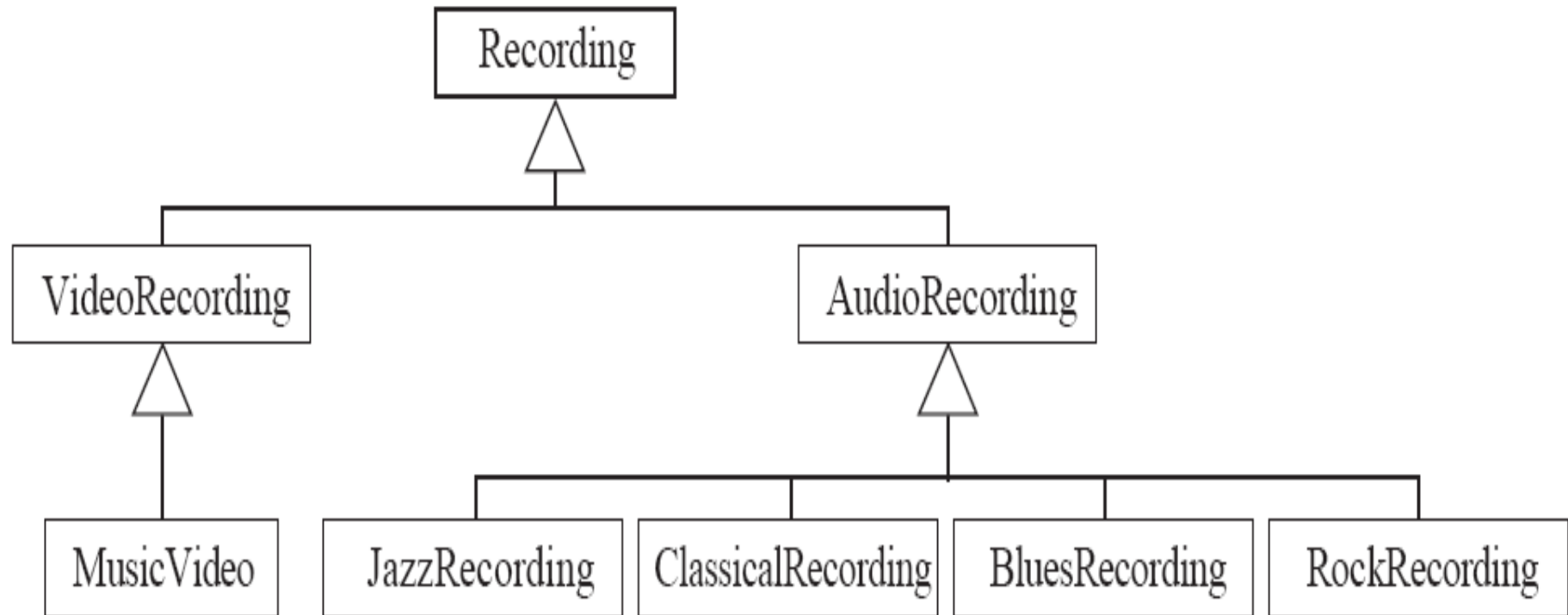
- **Model the concepts and relationships based on the description below:**
- Cirque du Soleil's show management system keeps track of their touring shows including the name of the show, the length of the show, the year the show was created, the date of each show, each city the show is visiting, the number of artists needed for the show, and the number of songs in the show.





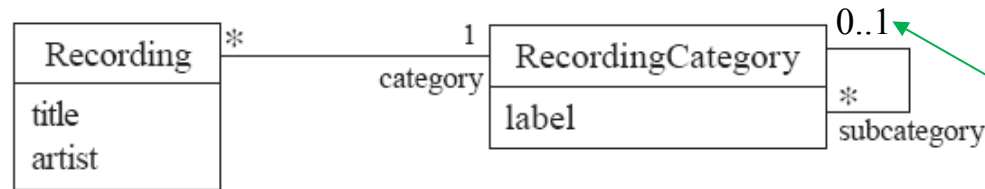
# Avoid Unnecessary Generalizations (1)

- Do you see a problem with this class diagram (in the context of a music library application)?

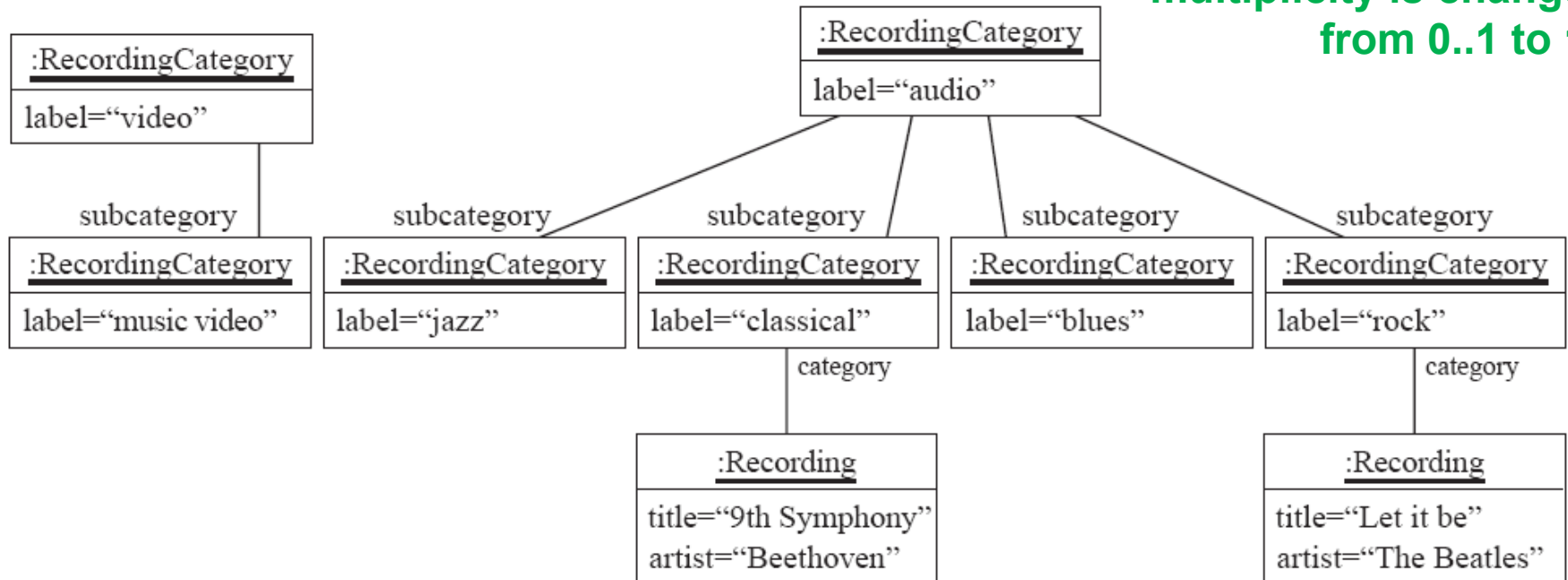


- Inappropriate hierarchy of classes, which should be instances because they are not treated/do not behave in any different way (if they are/do, subclasses may be appropriate)

# Avoid Unnecessary Generalizations (2)



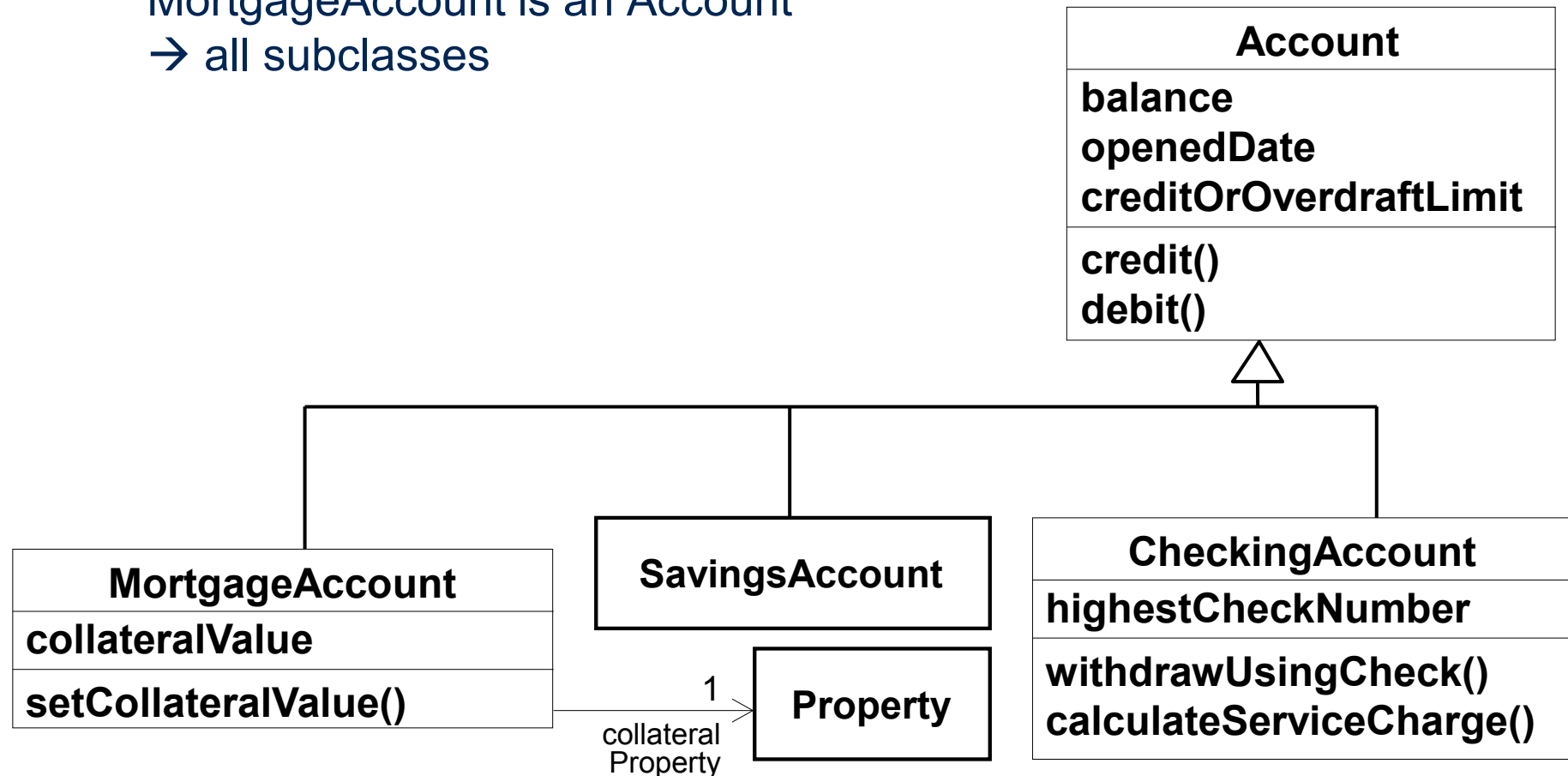
What happens if the multiplicity is changed from 0..1 to 1?



- Improved class diagram, with its corresponding object (instance) diagram

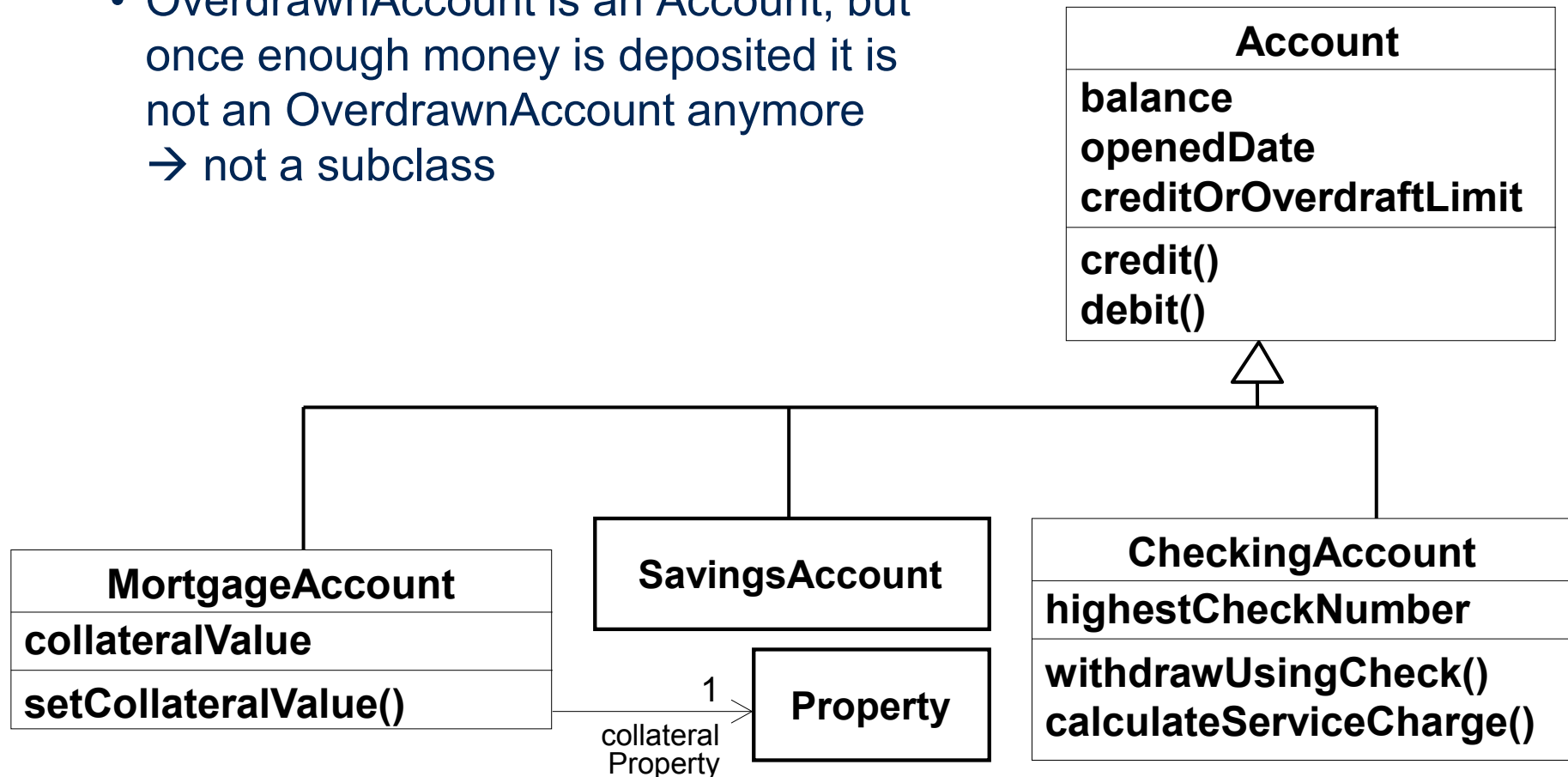
# First Check for Proper Generalization

- Always check generalizations to ensure they obey the **isa rule**
  - SavingsAccount is an Account, CheckingAccount is an Account, MortgageAccount is an Account  
→ all subclasses



# Second Check for Proper Generalization

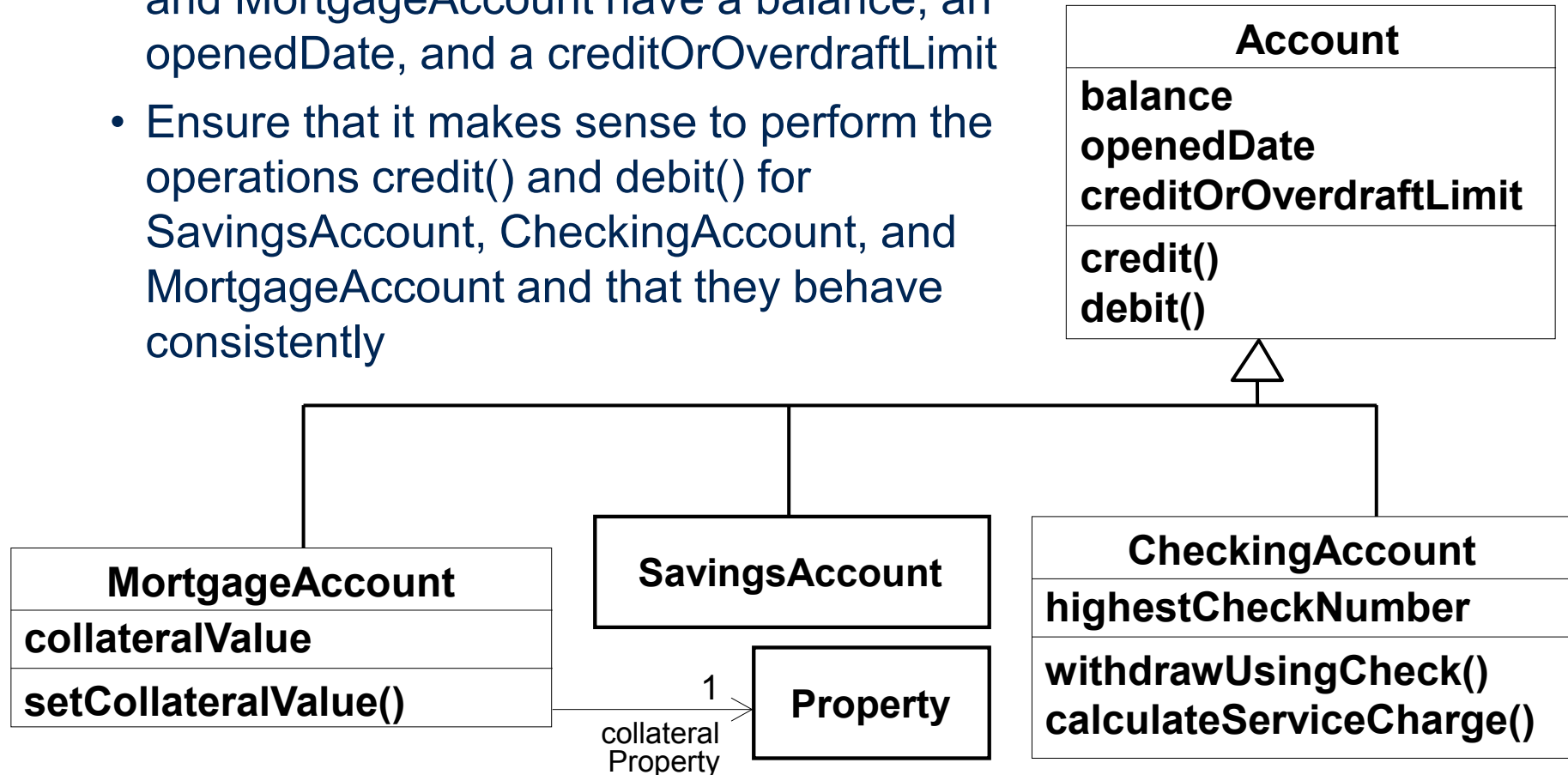
- A subclass must **retain its distinctiveness** throughout its life (in other words: an object cannot change its type/class)
  - OverdrawnAccount is an Account, but once enough money is deposited it is not an OverdrawnAccount anymore  
→ not a subclass





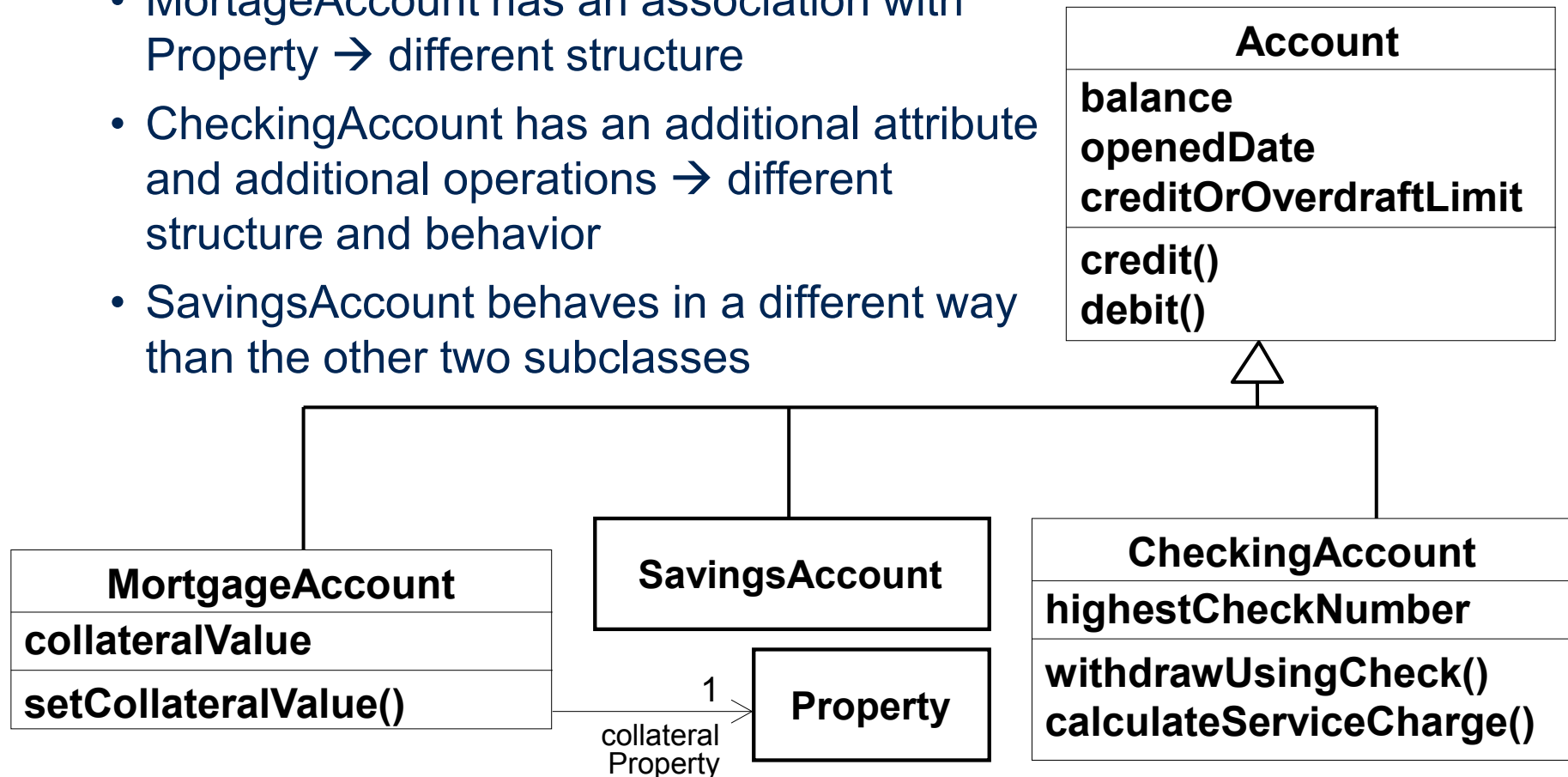
# Third Check for Proper Generalization

- All inherited features must **make sense** in each subclass
  - Ensure that SavingsAccount, CheckingAccount, and MortgageAccount have a balance, an openedDate, and a creditOrOverdraftLimit
  - Ensure that it makes sense to perform the operations credit() and debit() for SavingsAccount, CheckingAccount, and MortgageAccount and that they behave consistently



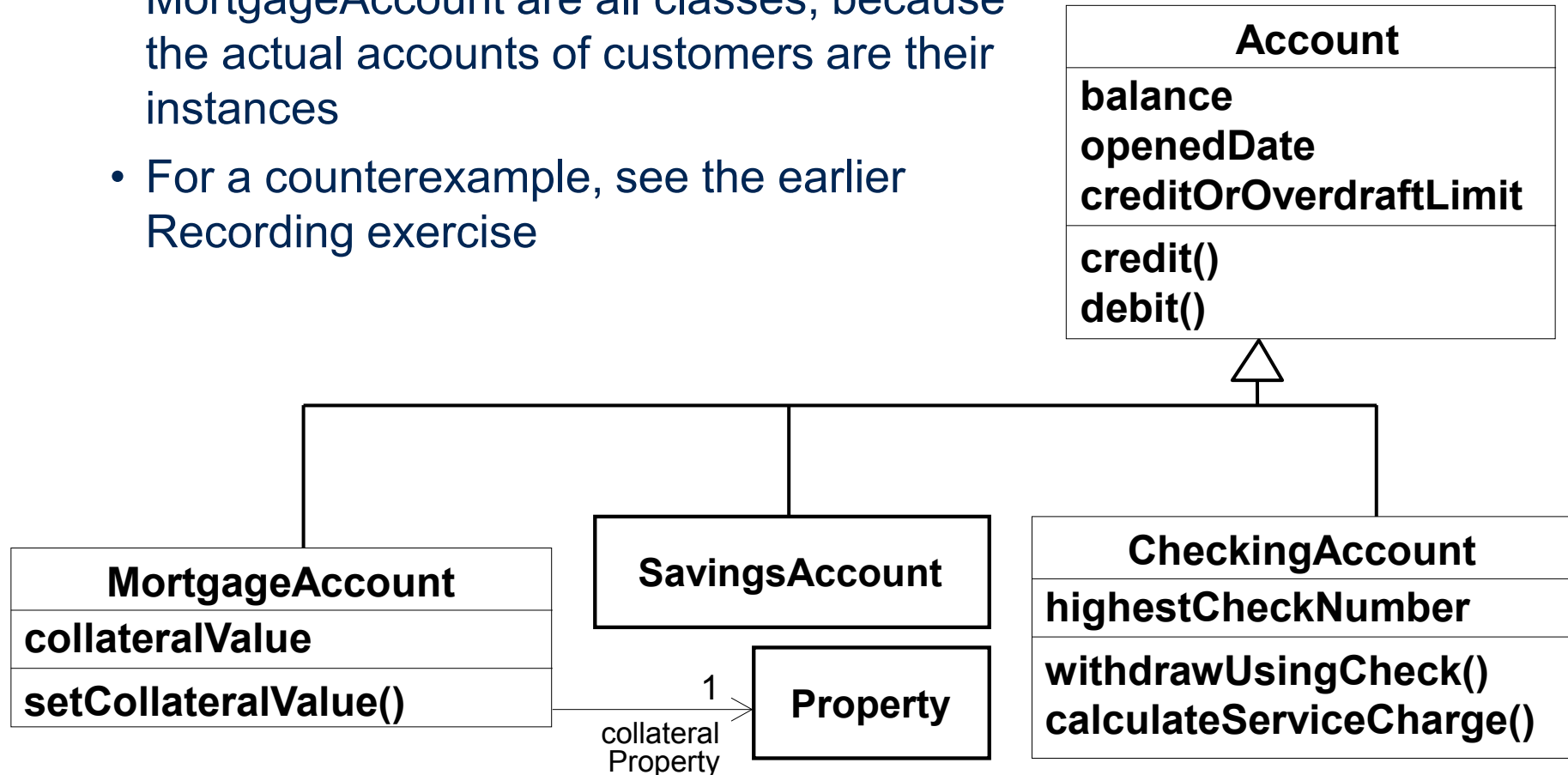
# Fourth Check for Proper Generalization

- A subclass must be **different than superclass and other subclasses** in terms of behavior or structure
  - MortgageAccount has an association with Property → different structure
  - CheckingAccount has an additional attribute and additional operations → different structure and behavior
  - SavingsAccount behaves in a different way than the other two subclasses



# Fifth Check for Proper Generalization

- A subclass must **not be an instance**
  - SavingsAccount, CheckingAccount, and MortgageAccount are all classes, because the actual accounts of customers are their instances
  - For a counterexample, see the earlier Recording exercise

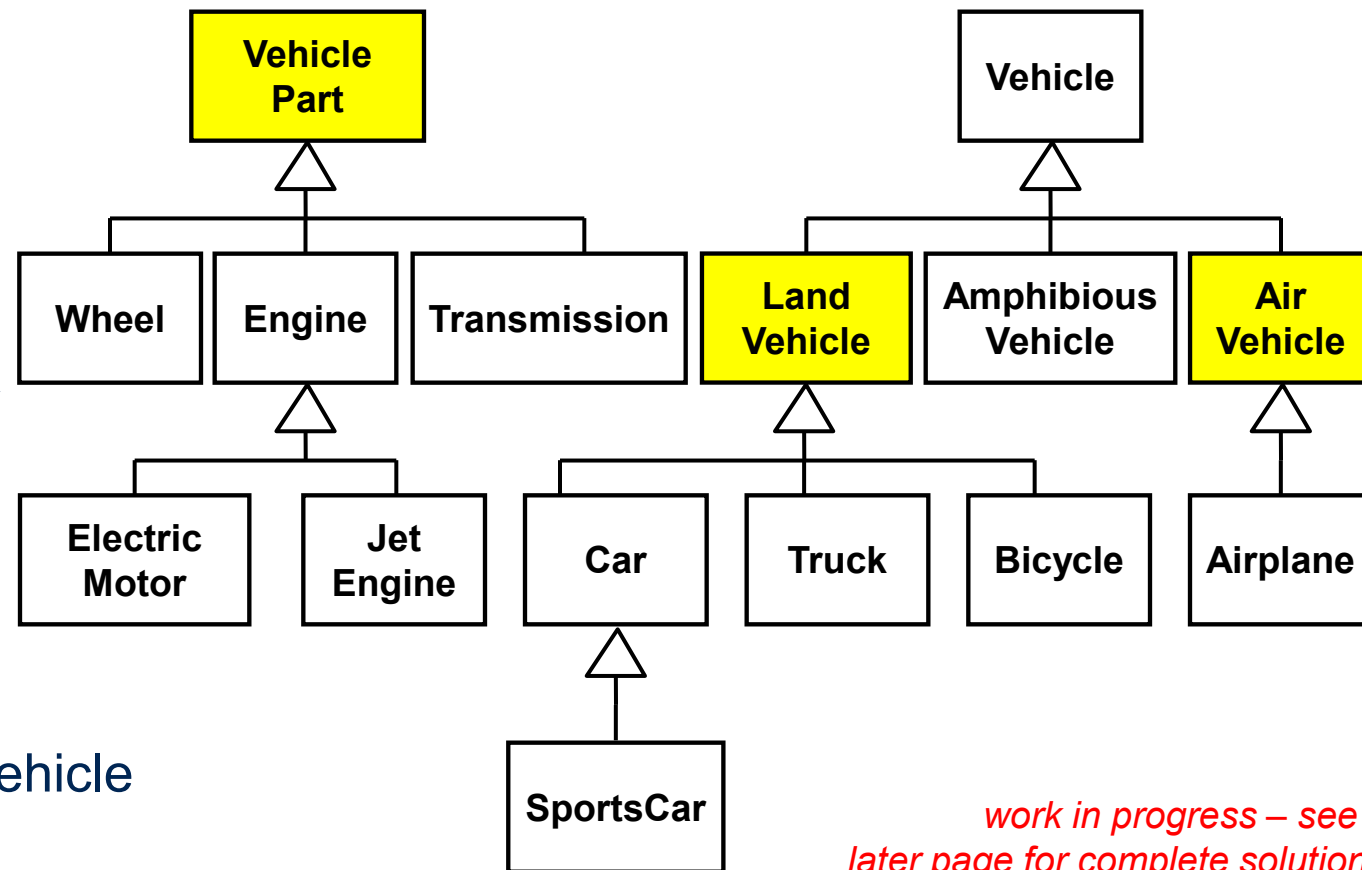


# Generalization Hierarchies

- Organize the following set of items into several distinct inheritance hierarchies of classes. You may need to add additional classes to act as superclasses. You may need to change some names. You may discover that two items correspond to a single class:

added  
class

- Vehicle
- Airplane
- Jet engine
- Transmission
- Car
- Electric motor
- Truck
- Sports car
- Engine
- Wheel
- Amphibious vehicle
- Bicycle

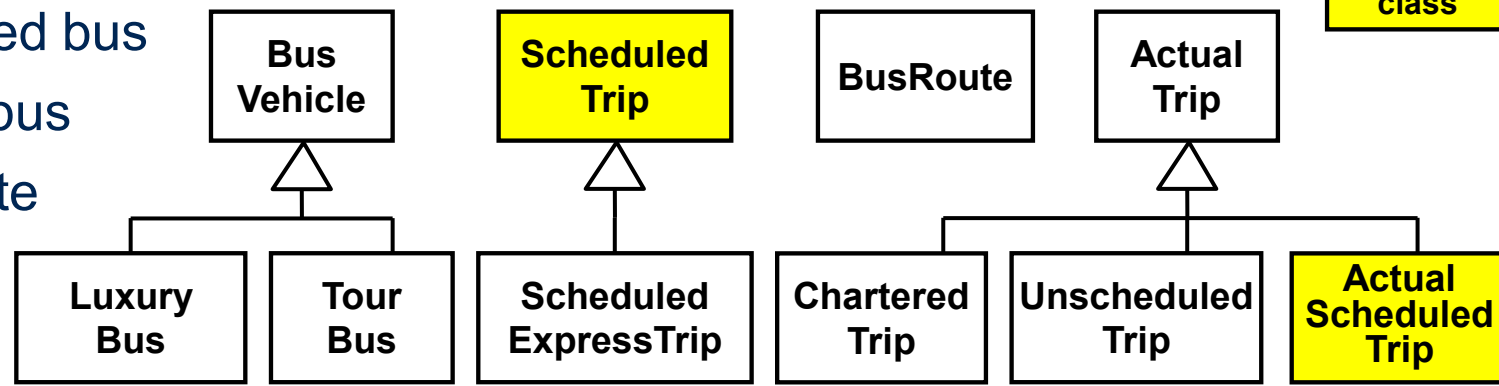




# Exercise: Generalization Hierarchies (1)

- Organize the following set of items into several distinct inheritance hierarchies of classes. You may need to add additional classes to act as superclasses. You may need to change some names. You may discover that two items correspond to a single class:

- Chartered bus
- Luxury bus
- Bus route
- Bus
- Trip
- Route
- Tour bus
- Schedule
- Express bus
- Unscheduled trip



added class

in everyday English, bus may mean three different things: the vehicle, a scheduled trip, or an actual trip

there is a difference between something that is scheduled and an actual trip

added to properly complement UnscheduledTrip

a list of scheduled trips

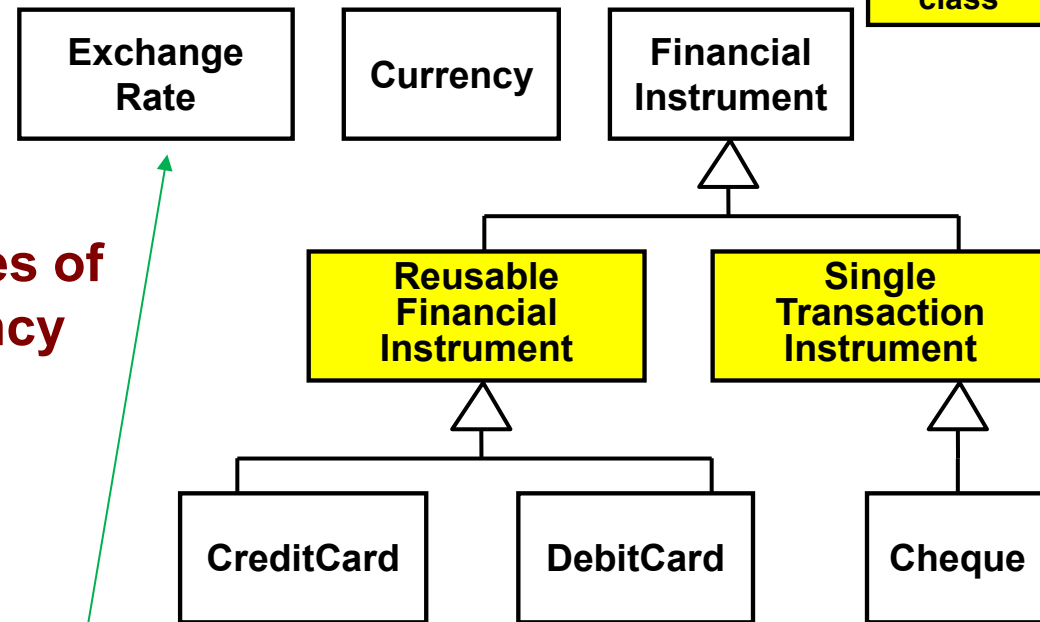
work in progress – see later page for complete solution

# Exercise: Generalization Hierarchies (2)

- Organize the following set of items into several distinct inheritance hierarchies of classes. You may need to add additional classes to act as superclasses. You may need to change some names. You may discover that two items correspond to a single class:

- Currency
- Financial instrument
- Cheque
- Exchange rate
- Bank account
- US dollars
- Visa
- Credit card
- Credit union
- MasterCard
- Bank branch
- Bank
- Debit card
- Bank machine
- Loan
- Canadian dollars

**instances of  
Currency**



**Which attributes and associations might exist for ExchangeRate?** **rate; effectiveDate;**  
**fromCurrency; toCurrency**

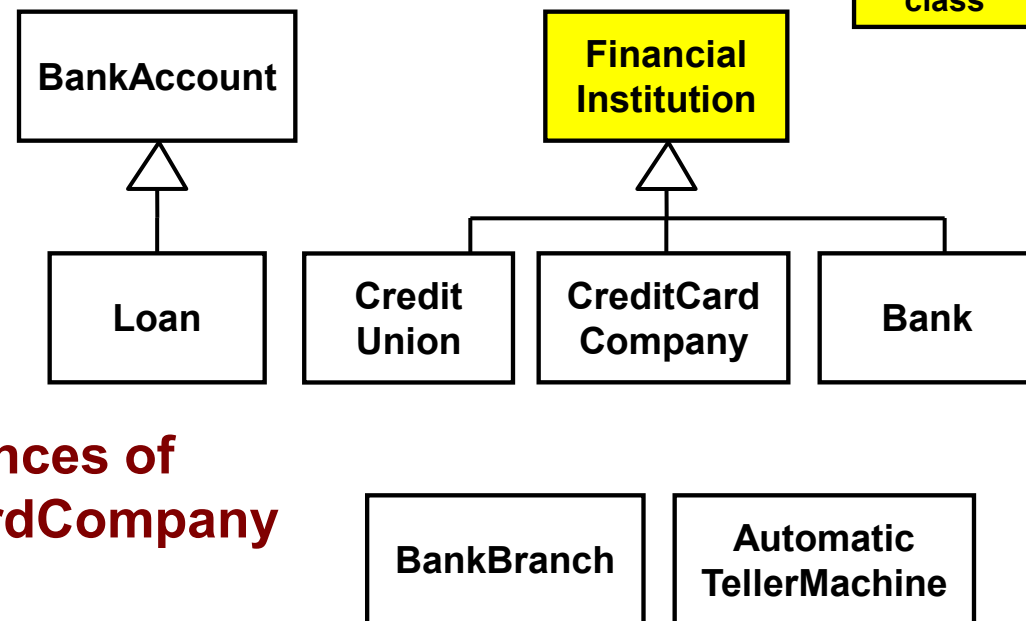
*work in progress – see later page for complete solution*

# Exercise: Generalization Hierarchies (3)

- Organize the following set of items into several distinct inheritance hierarchies of classes. You may need to add additional classes to act as superclasses. You may need to change some names. You may discover that two items correspond to a single class:

- Currency
- Financial instrument
- Cheque
- Exchange rate
- Bank account
- US dollars
- Visa
- Credit card
- Credit union
- MasterCard
- Bank branch
- Bank
- Debit card
- Bank machine
- Loan
- Canadian dollars

**instances of  
CreditCardCompany**



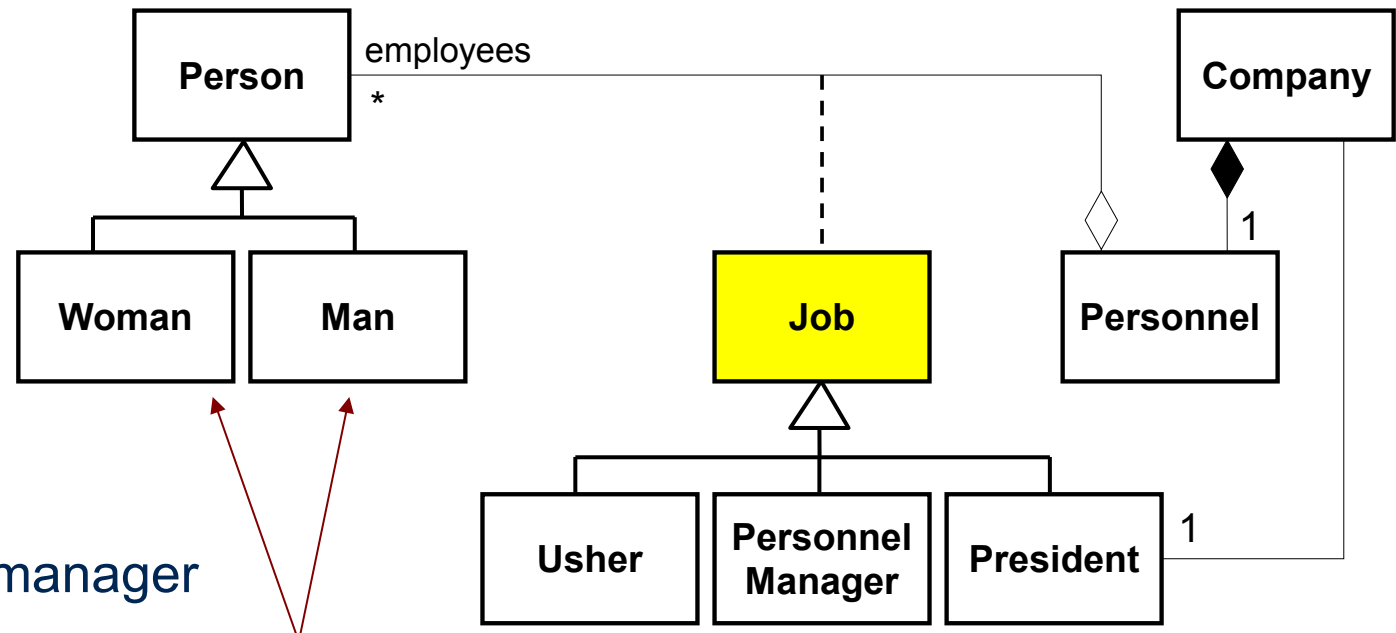
*work in progress – see later page for complete solution*



# Exercise: Relationships

- Use associations, aggregation, composition, and inheritance as needed to describe the relationships of the following set of concepts. You may need to add additional classes to act as superclasses. You may need to change some names. You may discover that two items correspond to a single class:

- Personnel
- Woman
- Man
- Usher
- Person
- President
- Employee
- Personnel manager
- Company



**should only be subclasses if the behavior / structure of the subclasses is different in some way**

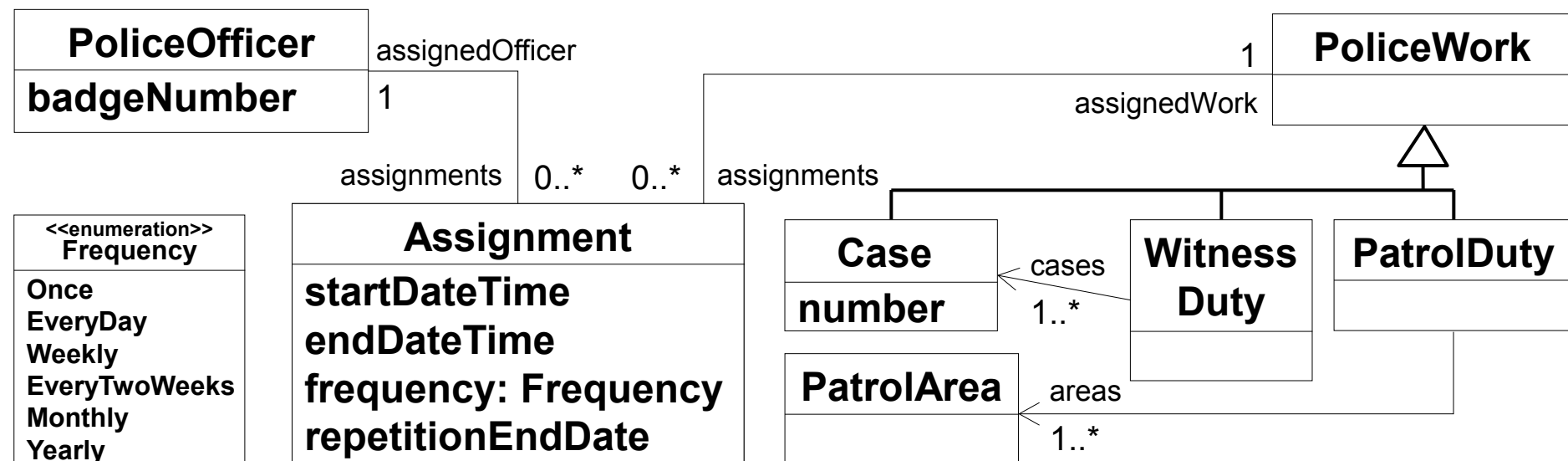
*work in progress – see later page for complete solution*



# Identifying Generalizations

- Police Information (PI) System

This system helps the Java Valley police officers keep track of the cases they are assigned to do. Officers may be assigned to investigate particular crimes, which involves interviewing victims at their homes and entering notes in the PI system. In addition to particular cases, an officer may also be assigned to patrol particular areas or to attend particular events such as being a witness for a case in court. Some work assignments are regular ongoing assignments, while others are for a particular period of time.

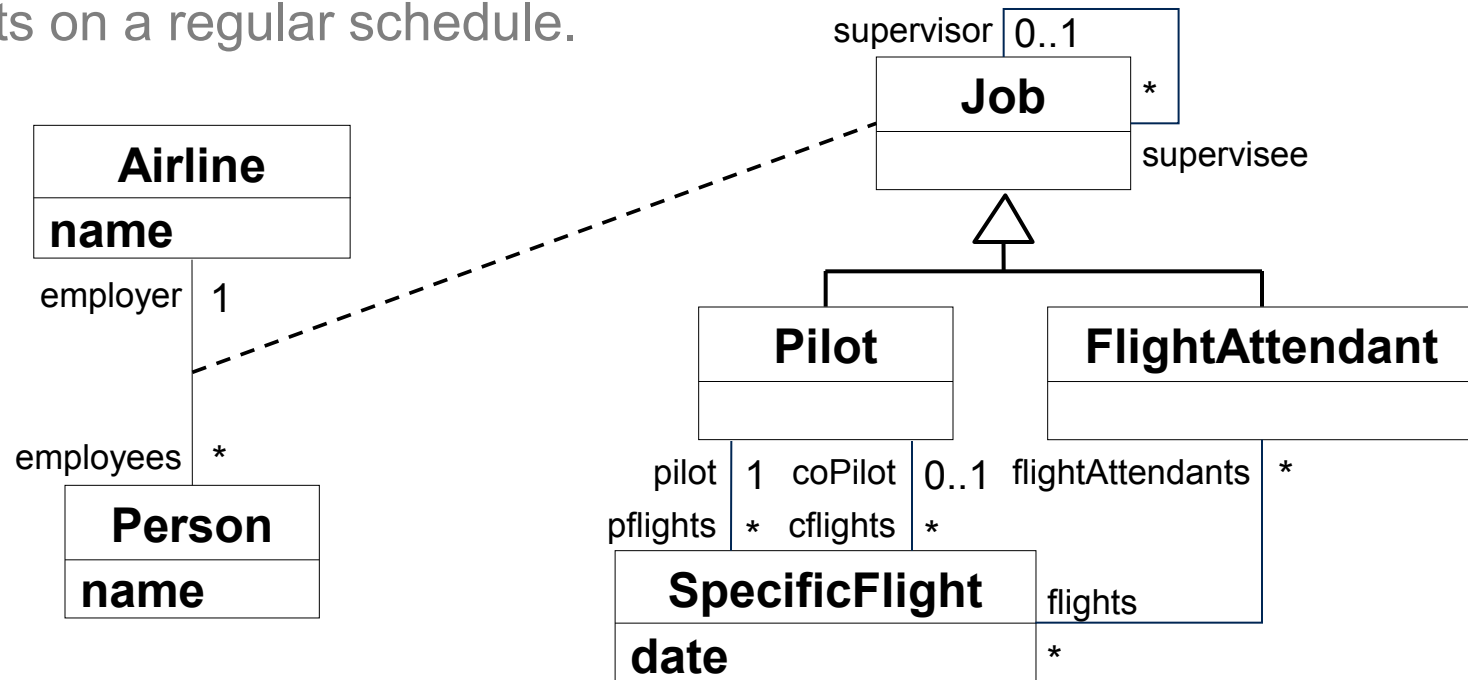


*some classes from earlier solution not shown again; work in progress – see later page for complete solution*

# Exercise: Identifying Generalizations

- Airline Reservation (AR) System

Ootumlia Airlines runs sightseeing flights from the Java Valley, the capital of Ootumlia. The AR system keeps track of passengers who will be flying in specific seats on various flights, as well as people who will form the crew. For the crew, the system needs to track what everyone does, and who supervises whom. Ootumlia Airlines runs several daily numbered flights on a regular schedule.



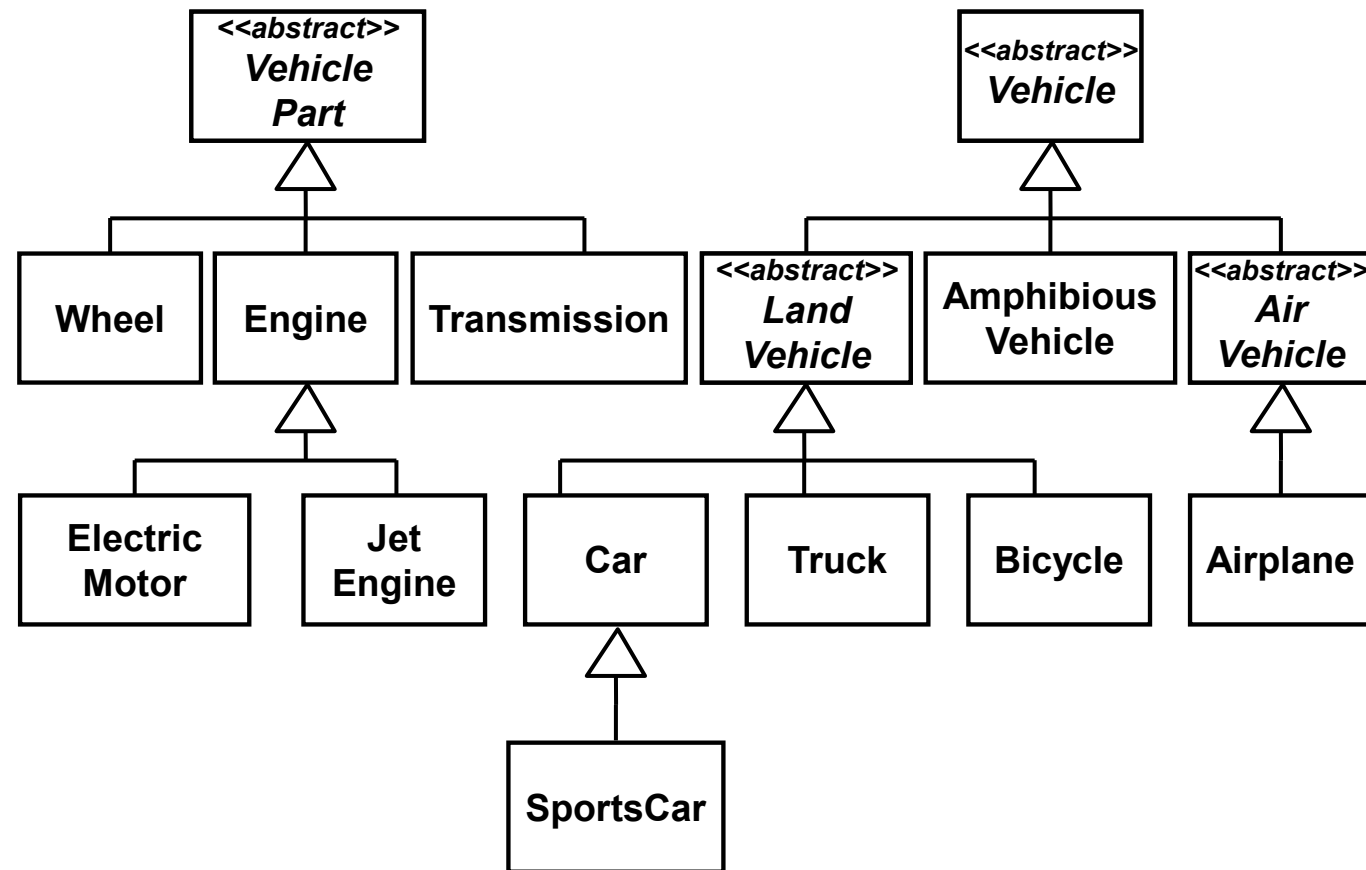
*some classes from earlier solution not shown again; work in progress – see later page for complete solution*

# Abstract Classes

- An abstract class is just like a regular class (i.e., it can have attributes, associations, operations...), but it **cannot** be instantiated
- At the modeling level, it only makes sense to have an abstract class, if it eventually has a (direct or indirect) subclass that can be instantiated

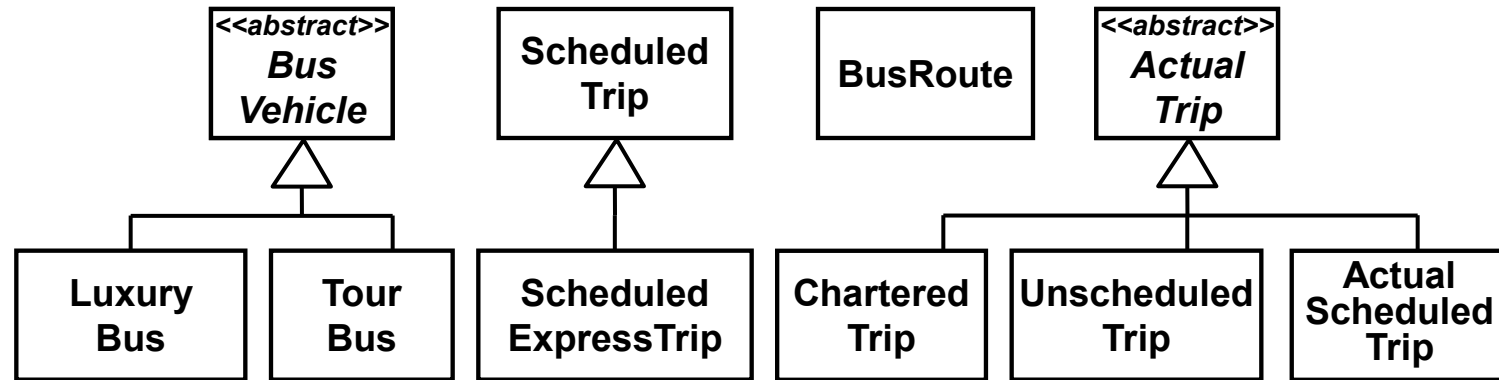
- An abstract class is indicated with a name in italic font and/or the `<<abstract>>` keyword

`<<abstract>>`  
***AbstractClass***

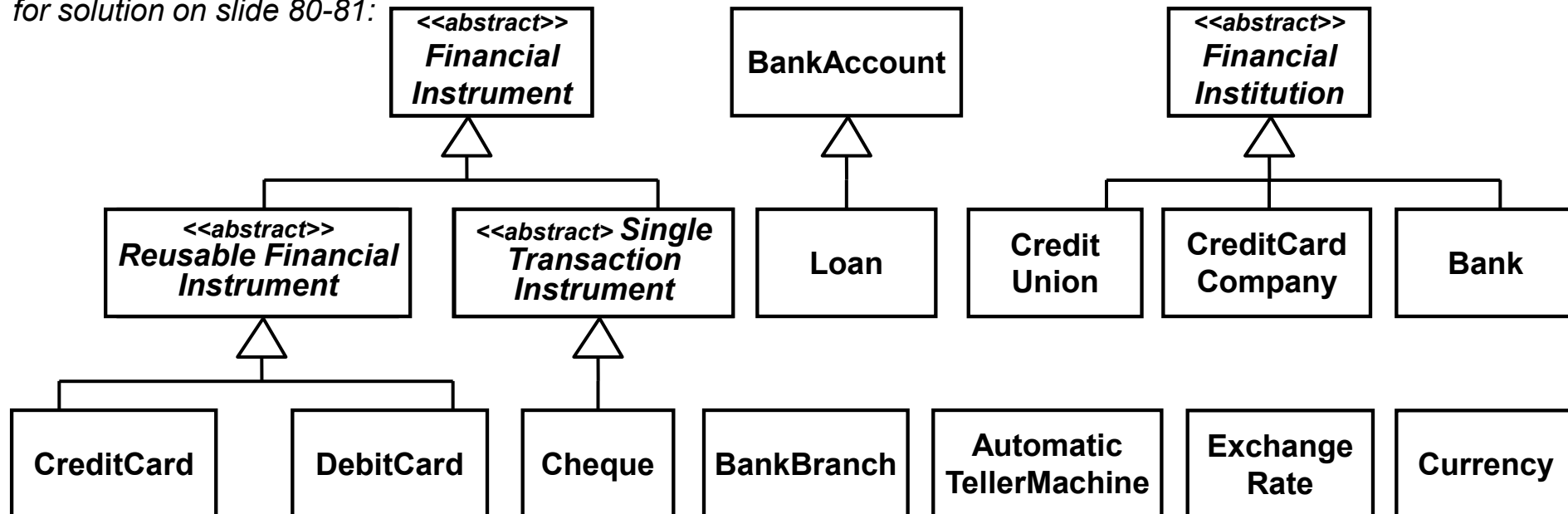


# Exercise: Identify Abstract Classes (1)

for solution on slide 79:



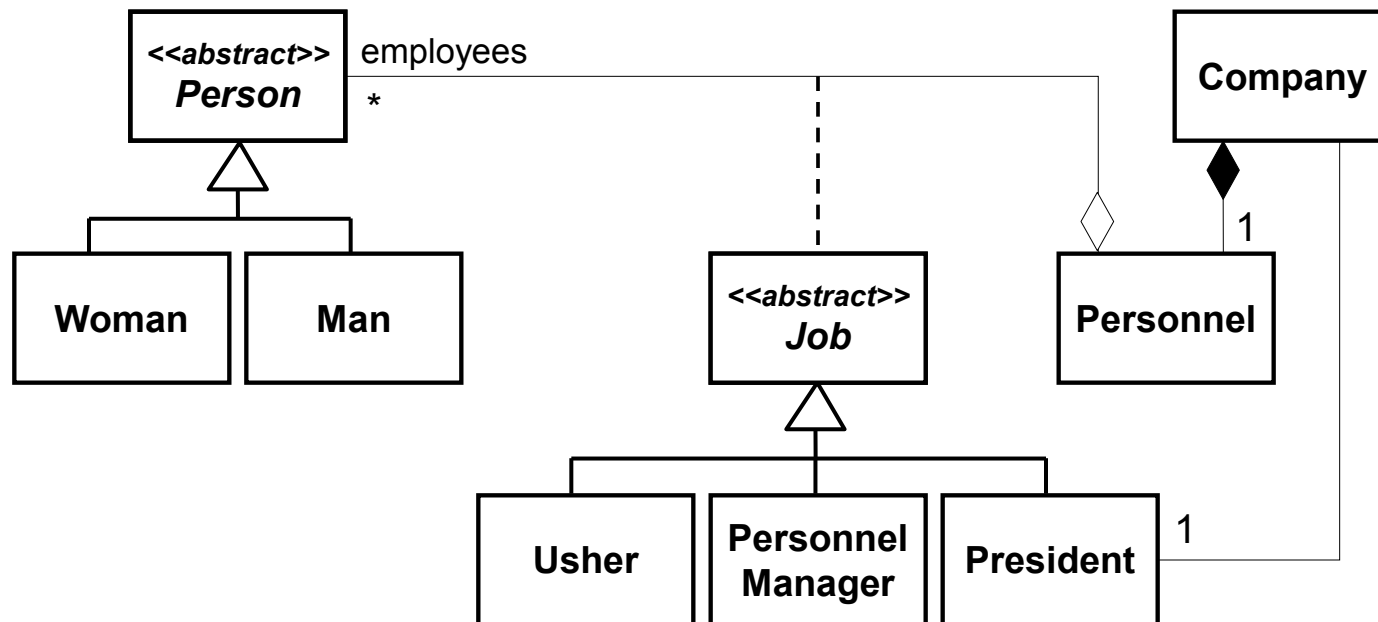
for solution on slide 80-81:





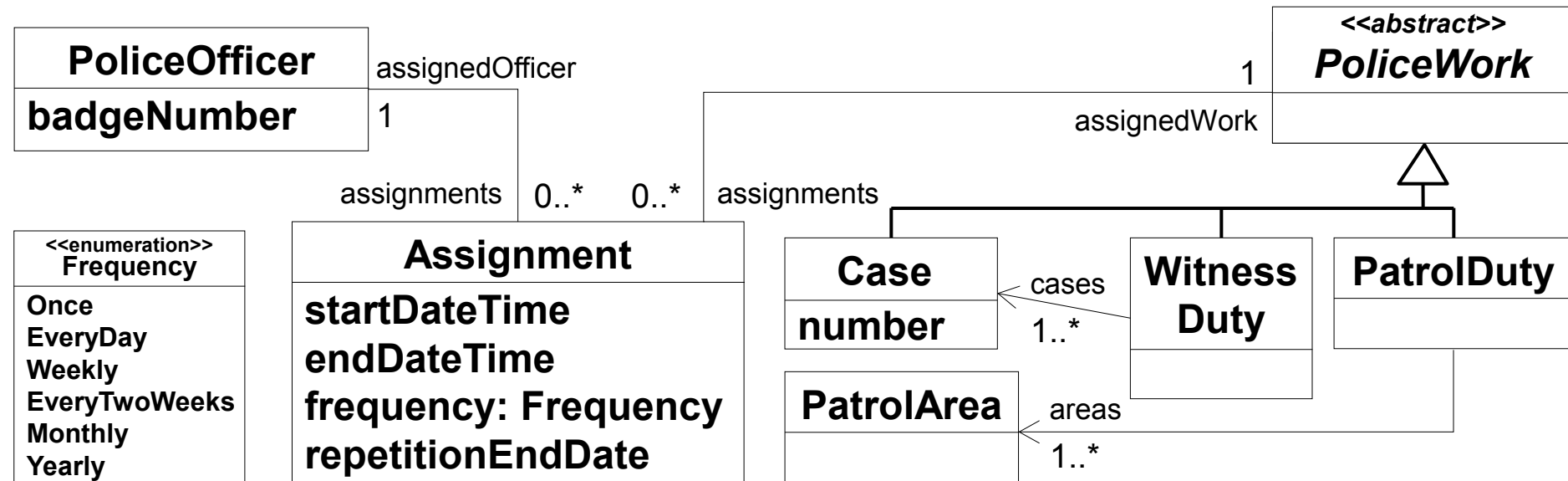
# Exercise: Identify Abstract Classes (2)

for solution on slide 82:



# Exercise: Identify Abstract Classes (3)

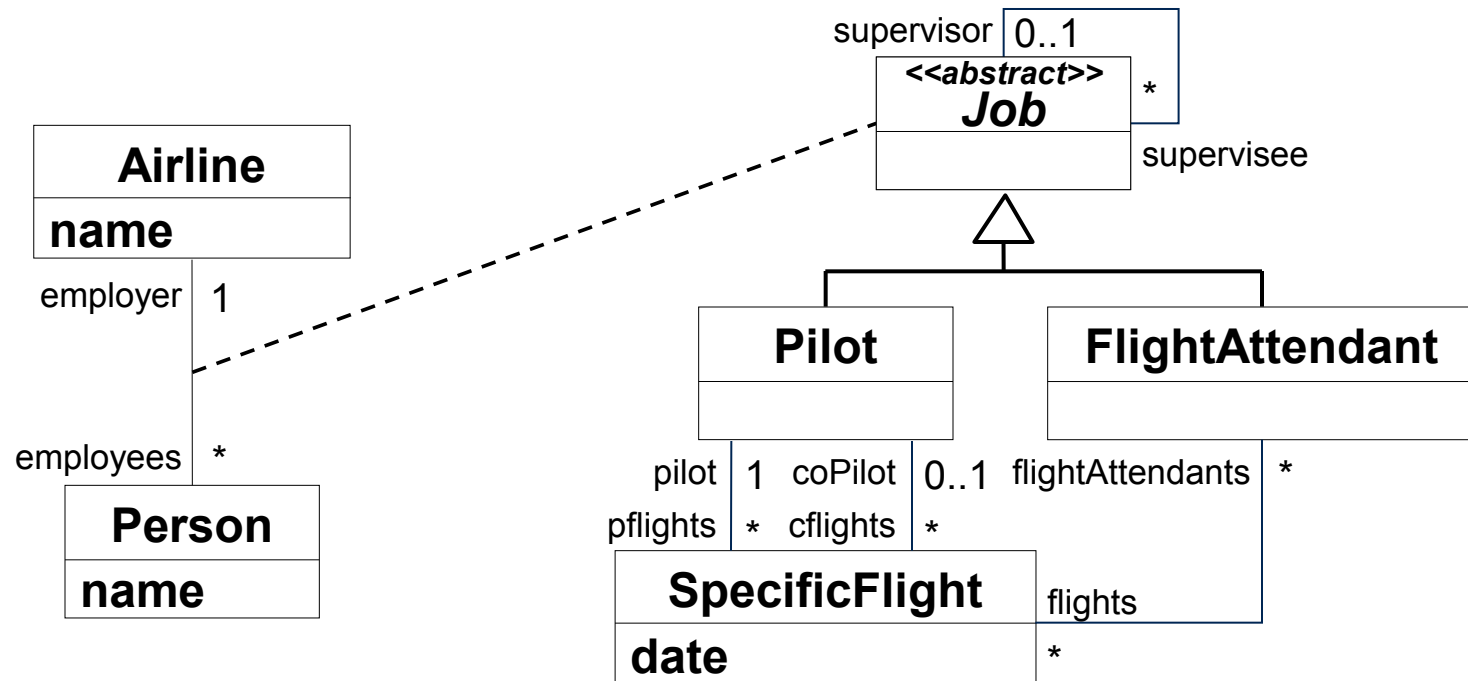
for solution on slide 83:



some classes from earlier solution not shown again; work in progress – see later page for complete solution

# Exercise: Identify Abstract Classes (4)

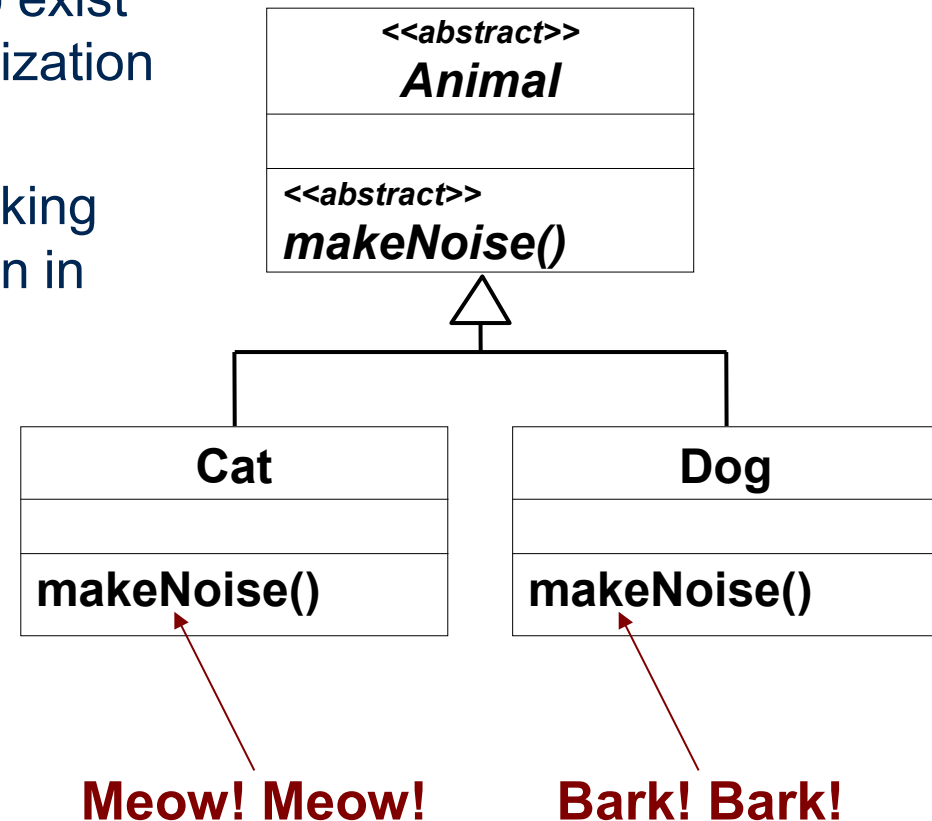
for solution on slide 84:



some classes from earlier solution not shown again; work in progress – see later page for complete solution

# Abstract Methods

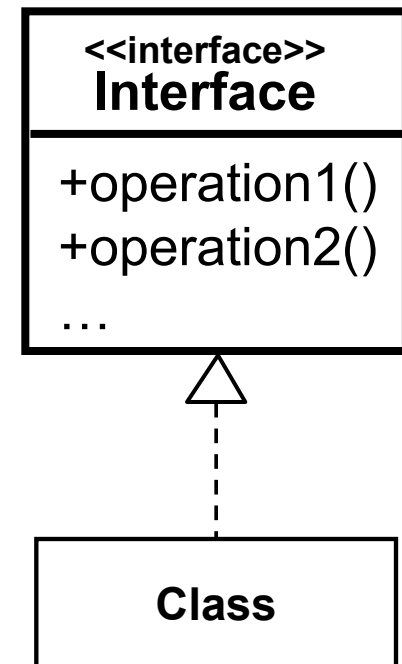
- An operation should be declared to exist at the **highest class** in the generalization hierarchy where it **makes sense**
- The operation may be abstract (lacking implementation) at that level (shown in italic font just like abstract classes)
- If so, the class also must be abstract
  - No instances can be created
  - The opposite of an abstract class is a concrete class
- If a superclass has an abstract operation then its subclasses at some level must have a concrete method for the operation
  - Leaf classes must have or inherit concrete methods for all operations
  - Leaf classes must be concrete





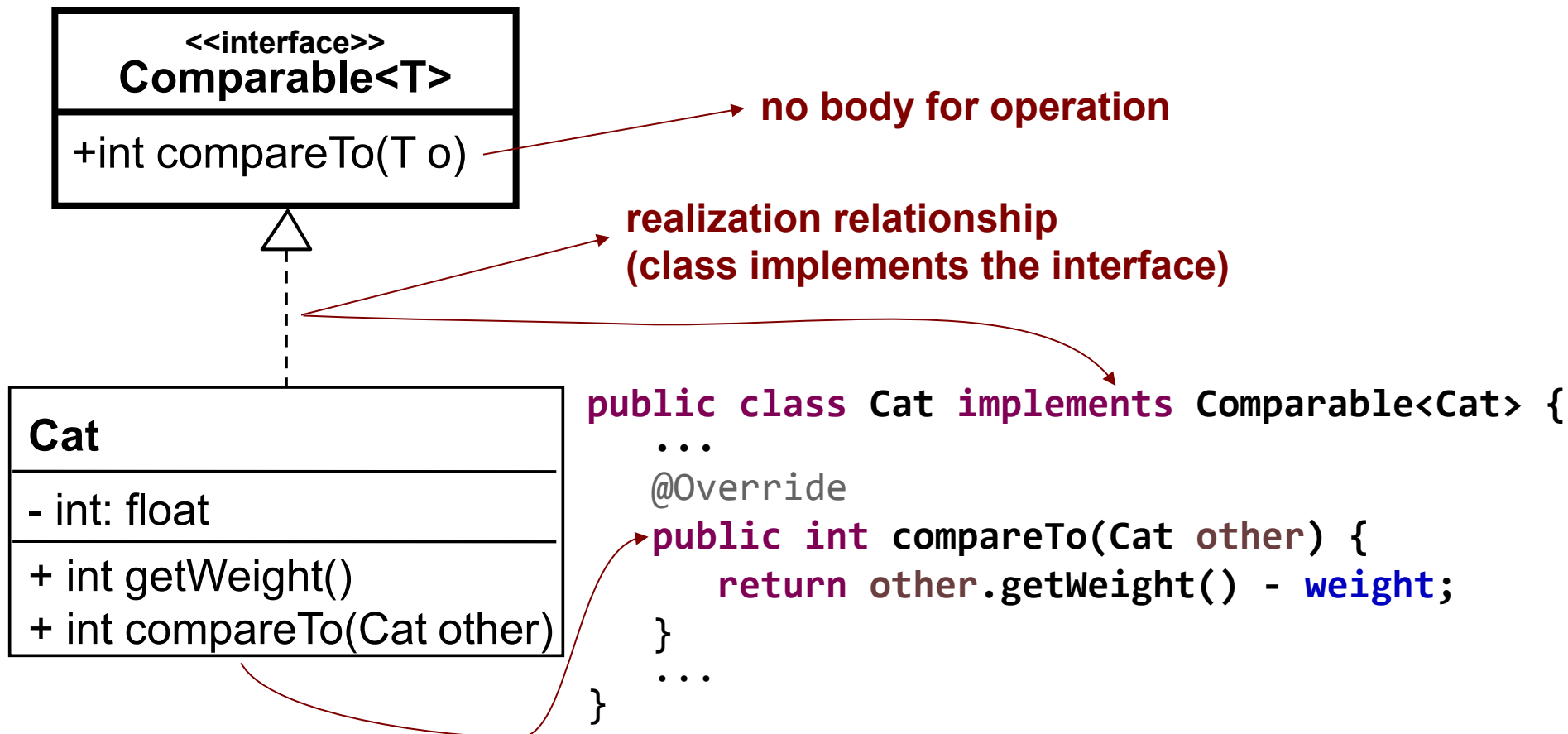
# Interfaces

- An **interface** is essentially like an abstract class, but cannot have executable statements
- Defines a set of operations that make sense in several classes
- A class can implement **any number** of interfaces
  - The class must have concrete methods for the operations
- You can declare the type of a variable to be an interface
  - This is just like declaring the type to be an abstract class
- An interface may have a generalization relationship with another interface
- Everything in an interface is **public**, since an interface has no “inside” that could use something that is not public



# Java Interfaces

- Important interfaces in Java's library include
  - Runnable, Collection, Iterator, Comparable, Cloneable



# Key Concepts of Object Orientation (1)

- **Identity**

- Each object is distinct from each other object, and can be referred to
- Two objects (twins) are distinct even if they have the same data

- **Classes**

- The code is organized using classes, each of which describes a set of objects

- **Inheritance**

- The mechanism where a subclass inherits from a superclass in the inheritance hierarchy

- **Polymorphism**

- The mechanism by which several methods can have the same name and implement the same abstract operation (requires **dynamic binding**)

# Key Concepts of Object Orientation (2)

- **Abstraction** (creating a simplified representation of something for a purpose)
  - Object: abstraction of something in the world
  - Class: abstraction of a set of objects, and abstract container of methods that operate on those objects
  - Superclass: abstraction of a set of subclasses
  - Operation: abstraction of a set of methods
  - Method: procedural abstraction of its implementation
  - Attributes and associations: abstractions of underlying instance variables used to implement them



# Key Concepts of Object Orientation (3)

- **Modularity**

- Code can be constructed entirely from a set of classes

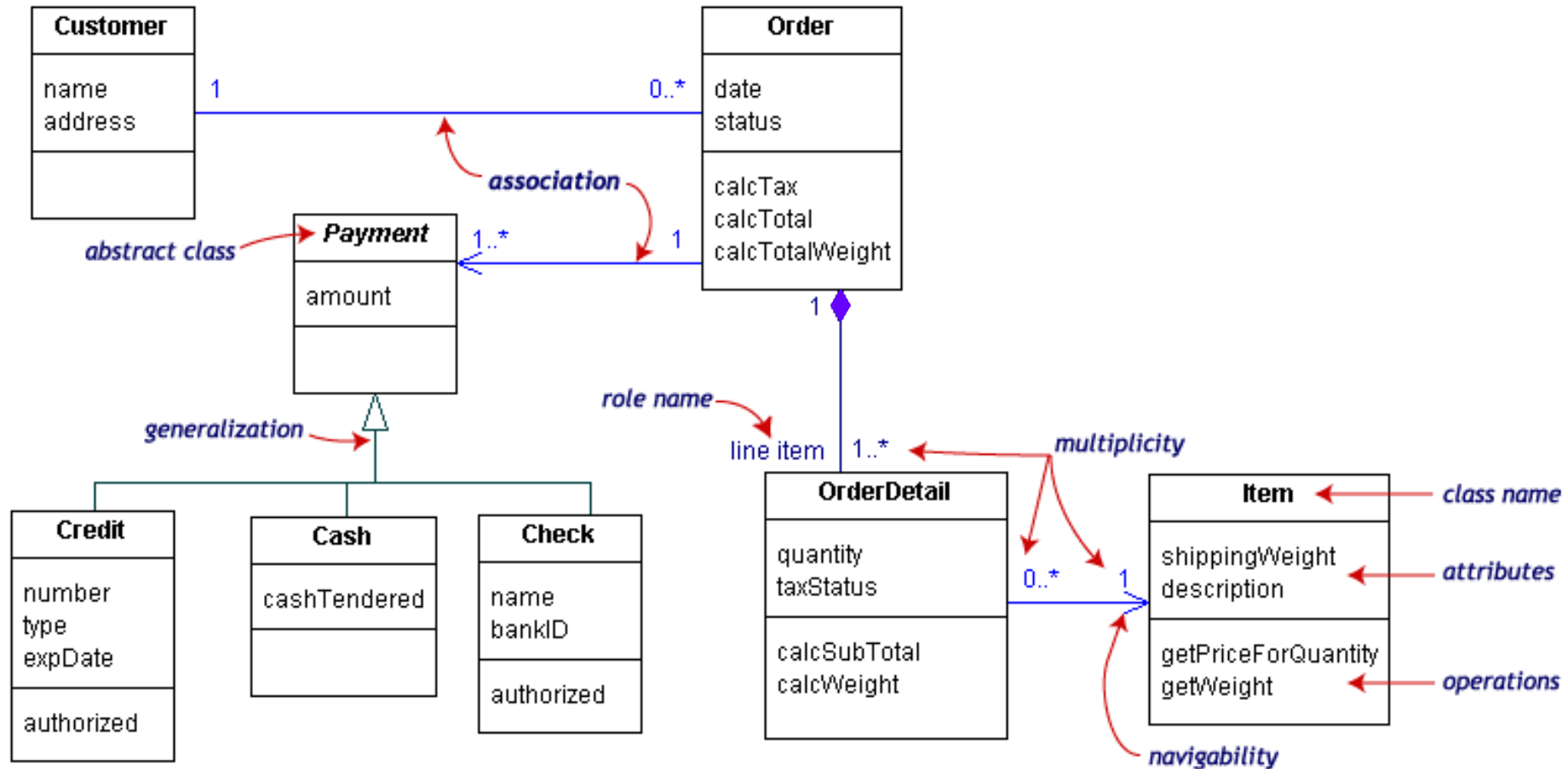
- **Encapsulation**

- Details can be hidden in classes
- Class acts as a container to hold its features (variables and methods) and defines an interface that allows only some of them to be seen from the outside

- Abstraction, modularity, and encapsulation give rise to **information hiding**:

- Programmers do not need to know all the details of a class
- Easier to maintain and evolve a software system

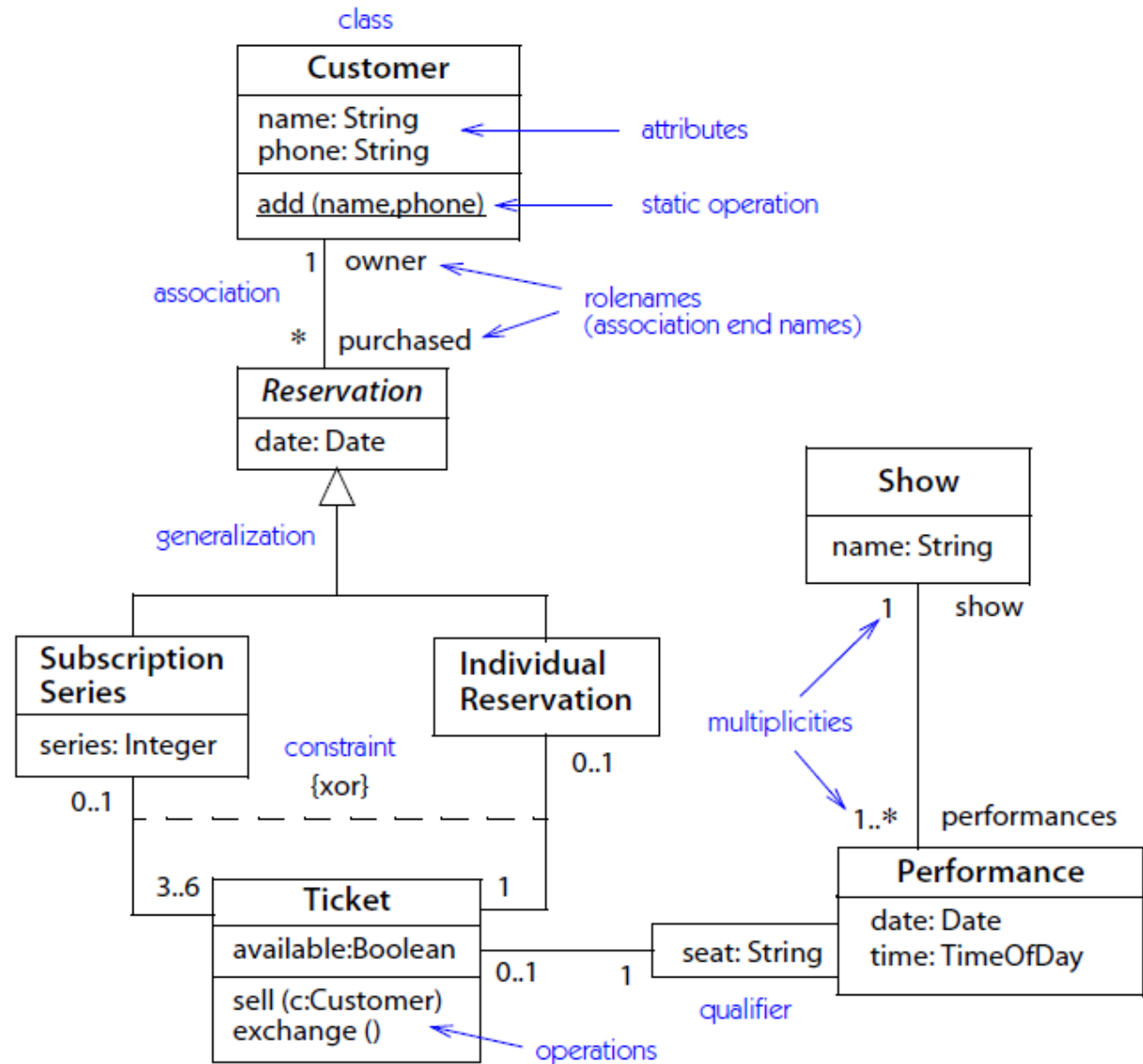
# Class Diagrams: Overview (1)



- A class diagram describes the structural aspects of the system
  - Classes, attributes, relationships (association, aggregation, composition, generalization), operations

# Class Diagrams: Overview (2)

- Another example of a class diagram with key concepts highlighted



Source: UML Reference Manual

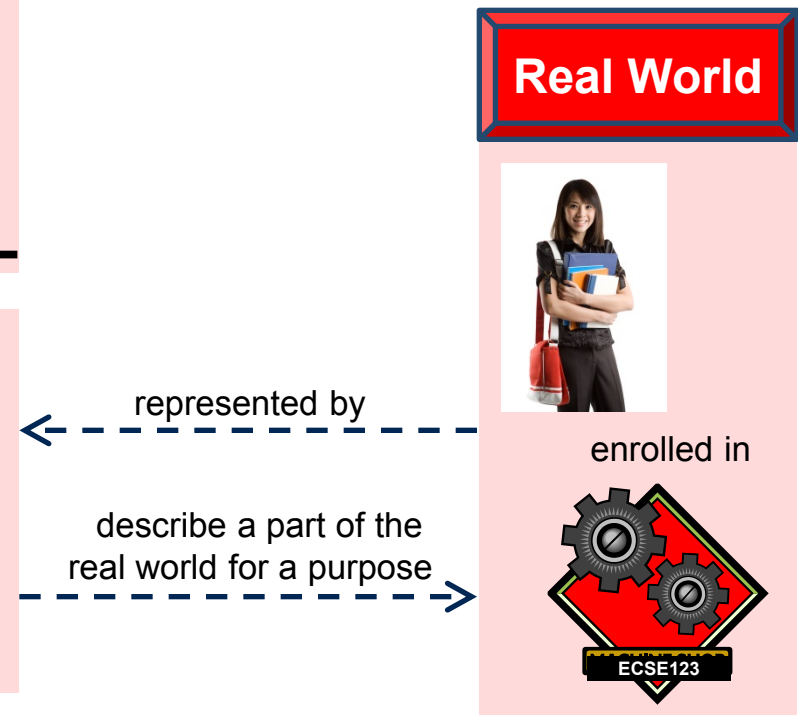
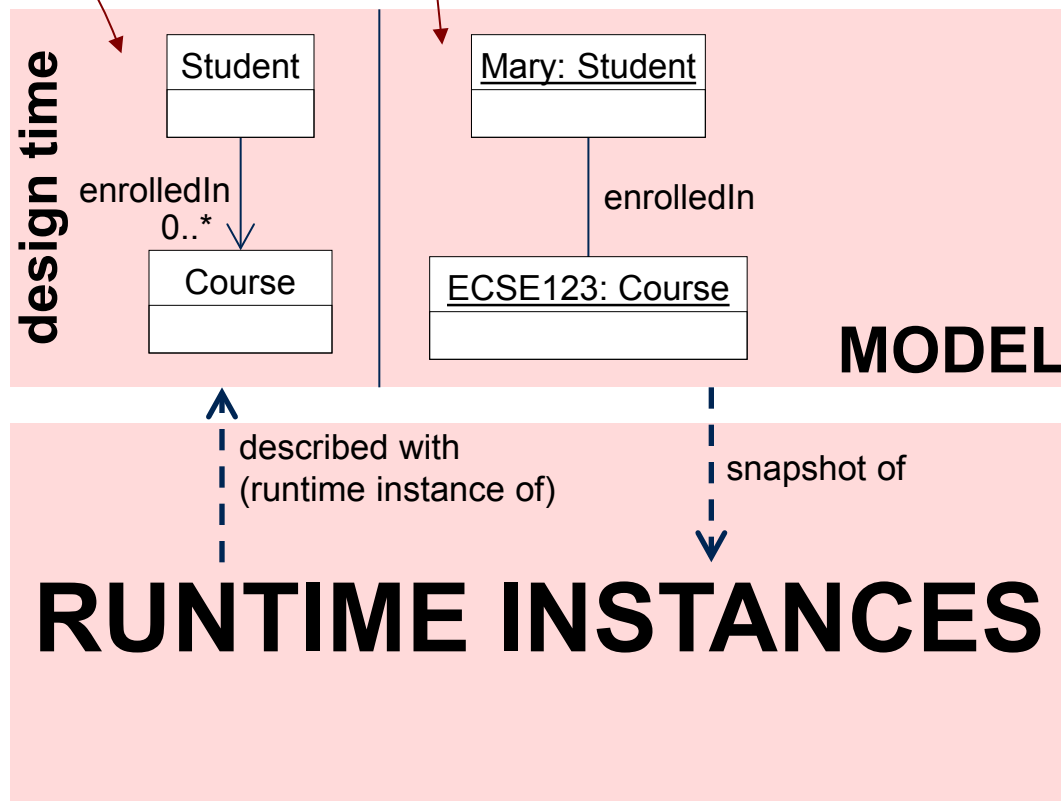


# Object Diagram

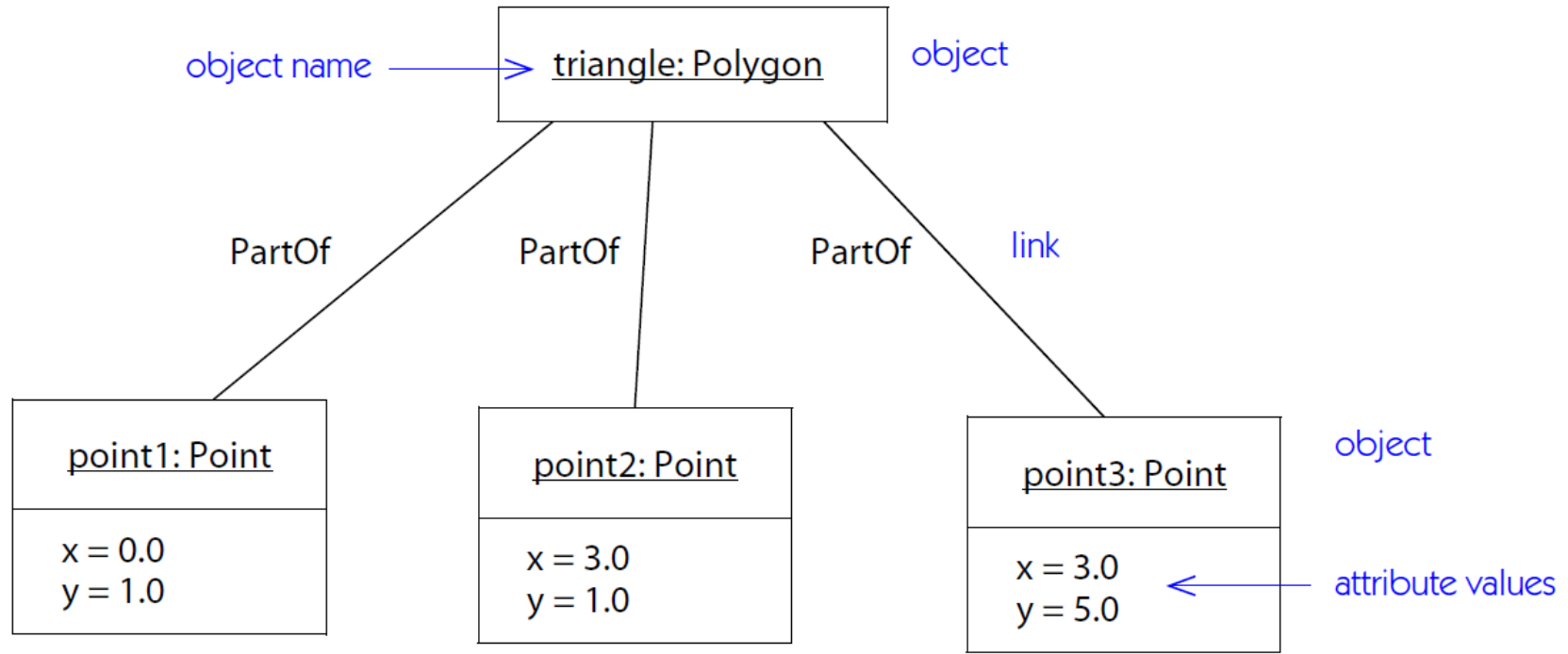


# Object Diagram (1)

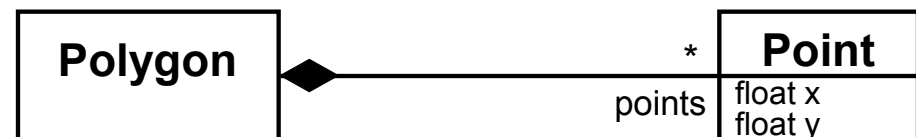
- A **class diagram** defines all possible sets of runtime instances and their relationships
- An **object diagram** visualizes one specific set of instances, i.e., a snapshot of the system as it executes



# Object Diagram (2)



- Object diagrams do not have generalizations, multiplicities...
- Object diagrams only have **objects**, **links**, and **attribute values**
- **What does the corresponding class diagram look like?**



Source: UML Reference Manual



# Exercises

# Scenario Analysis (1)

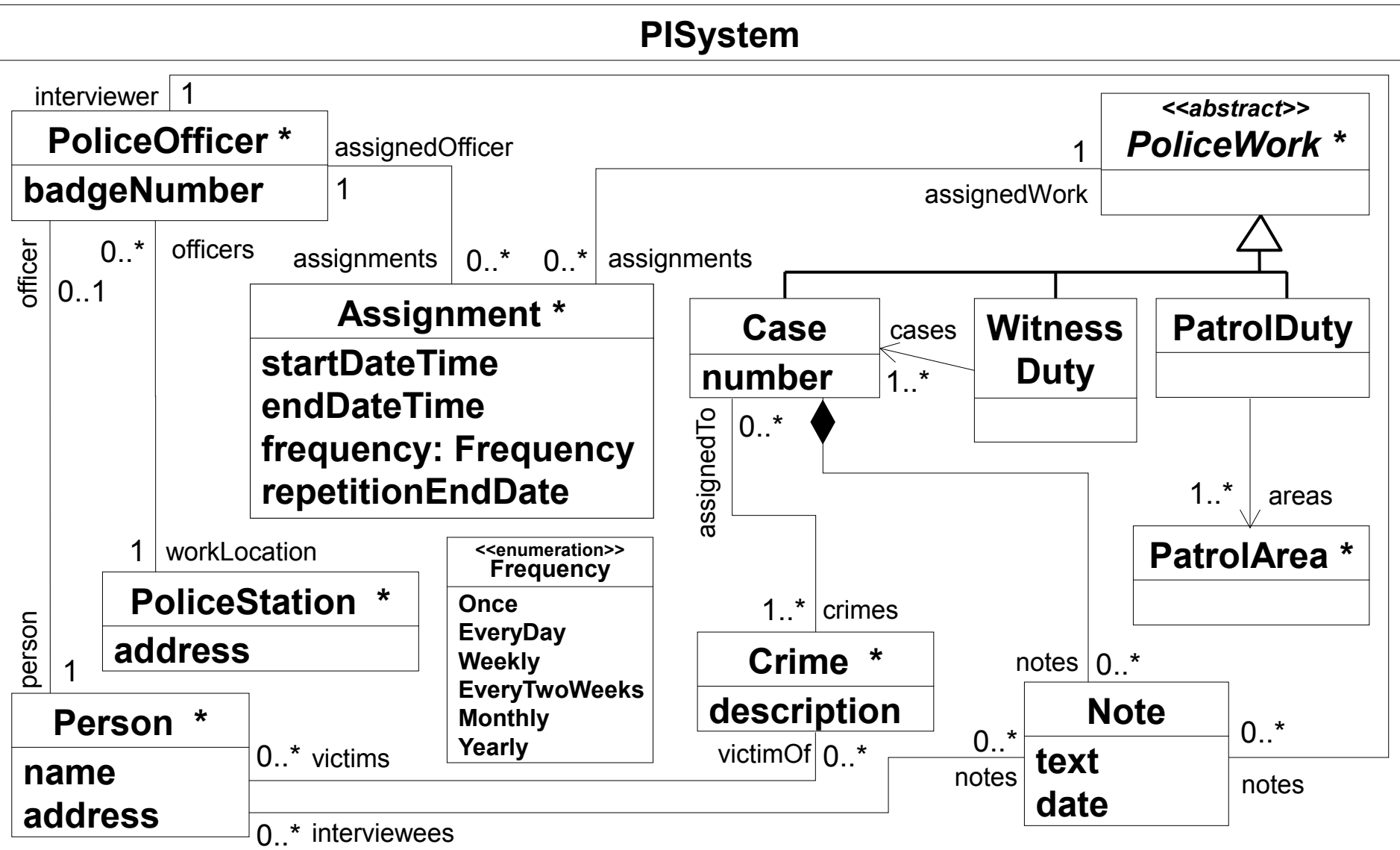
- Performed at the end of an iteration of your class diagram (after identifying classes, attributes, and the various relationships – and operations depending on the development phase)
- Helps confirm that your domain model sufficiently supports the required functionality (features) of the future system
- Pick a feature, run through its **scenario**, and figure out what each class needs to do
- Do not focus on CRUD features (create, read, update, delete) of individual classes, but rather on features that require the **interaction of several classes**
- E.g., do not focus on CRUD features for police officer, crime, case, note, patrol duty, witness duty
- Rather focus on (i) assigning a police officer to a case, (ii) unassigning the police officer, (iii) assigning the police officer again to the same case at the same time, (iv) assigning a police officer for two separate periods of time, (v) doing an interview with a number of victims and taking notes



# Scenario Analysis (2)

- Scenario analysis results in a **list of operations** each class needs to have to support the desired system behavior
- For a domain model, use scenario analysis as a check
- For a system model, use scenario analysis to determine the operations of the classes
  - Do not bother defining setters/getters in the class diagram – implied by attributes and their properties
  - Do not bother defining operations for associated elements (e.g., add, remove, ...) – implied by navigable association

# Police Information (PI) System – Scenarios (1)



# Police Information (PI) System – Scenarios (2)

- Assign a police officer to a case
  - Assume PoliceOfficer and Case instances already exist
  - Create an Assignment with one assignedOfficer (the PoliceOfficer), one assignedWork (the Case), a startDateTime, an endDateTime, frequency = Once, and repetitionEndDate = startDateTime (or null)
  - Contain Assignment in PISystem

**Note that Umple takes care of many things (see referential integrity). E.g., links and containment are handled because the constructor for Assignment includes parameters for PoliceOfficer, Case, and PISystem.**

- Unassign a police officer from a case
  - Delete Assignment if startDateTime is after right now; set endDateTime to today if startDateTime is not after right now
- Assign a police officer again to the same case at the same time
  - Check if Assignment with the PoliceOfficer and Case with overlapping times already exists, and if yes reject Assignment
- Assign a police officer for two separate periods of time to a case
  - Create two Assignments (see first bullet)



# Police Information (PI) System – Scenarios (3)

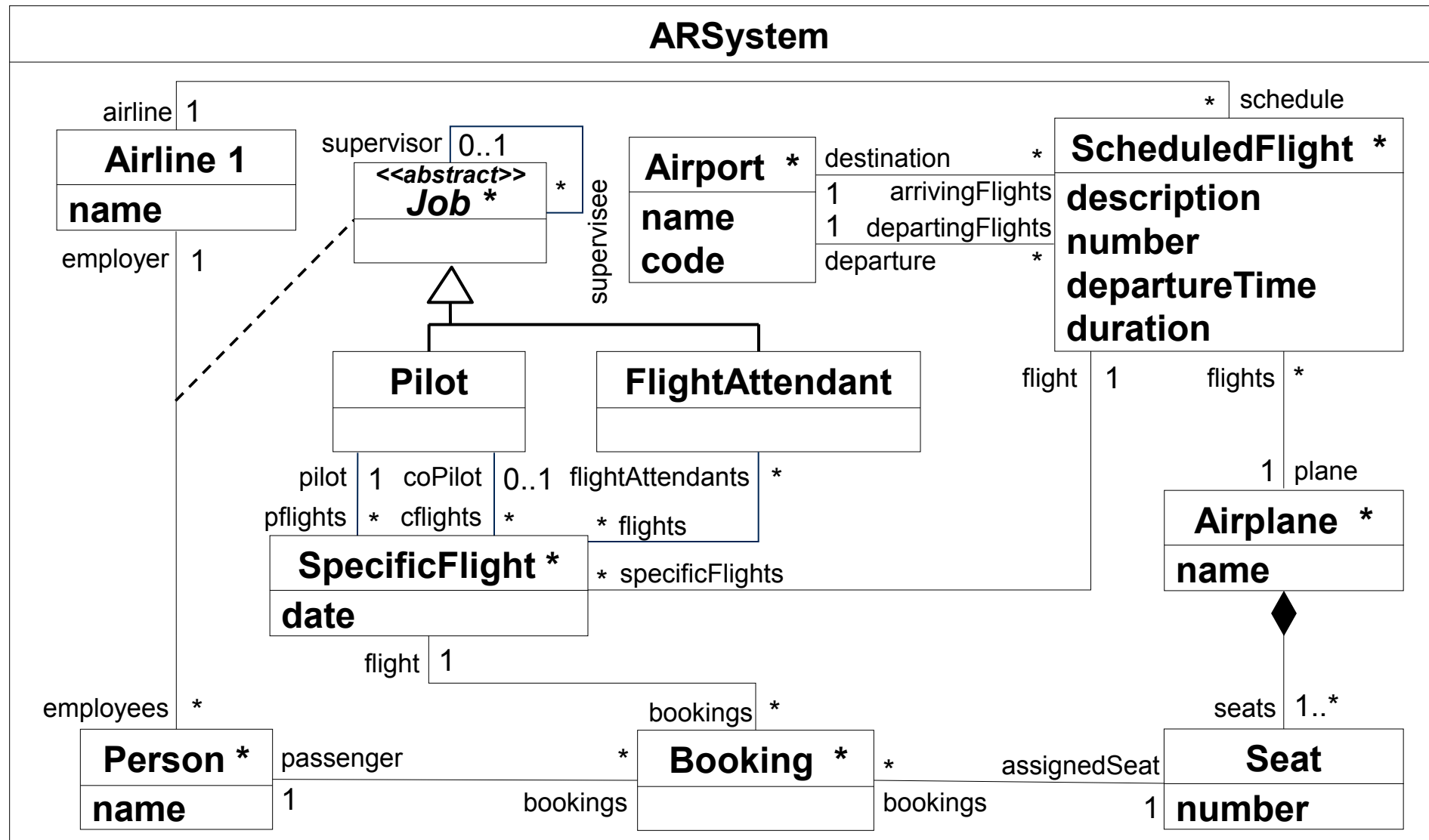
- Extending the assignment of a police officer to a case by a certain time
  - Set `endDateTime` to desired time and date
- A police officer does an interview with a number of victims and takes notes
  - Assume `PoliceOfficer` and `Case` instances already exist
  - Create a `Person` for each interviewed victim (unless the `Person` is already in the system); contain it in `PISystem`
  - Create a `Note`, set its containment to the `Case`, its interviewee to the `Persons`, and its interviewer to the `PoliceOfficer`
- A police officer is assigned to patrol duty in a new patrol area every Friday morning until the end of the year
  - Assume `PoliceOfficer` instance already exists
  - Create the new `PatrolArea`; contain it in `PISystem`; create a `PatrolDuty` with the new `PatrolArea` as its areas; contain it in `PISystem`
  - Create an `Assignment` with one assignedOfficer (the `PoliceOfficer`), one assignedWork (the `PatrolDuty`), a `startDateTime` (this Friday 8:00), an `endDateTime` (this Friday 12:00), `frequency = Weekly`, and `repetitionEndDate = December 31 of this year`; contain it in `PISystem`



# Exercise: PI System – Navigation

- Which persons did the police officer *myOfficer* interview?
  - ```
List<Note> notes = myOfficer.getNotes();  
List<Person> interviewees = new ArrayList<Person>();  
for (Note n : notes) {  
    interviewees.addAll(n.getInterviewees());  
}
```
- At which police stations was the case *myCase* investigated?
  - ```
List<Assignment> assignments = myCase.getAssignments();  
List<PoliceOfficer> officers = new ArrayList<PoliceOfficer>();  
for (Assignment a : assignments) {  
    officers.add(a.getAssignedOfficer());  
}  
List<PoliceStation> stations = new ArrayList<PoliceStation>();  
for (PoliceOfficer o : officers) {  
    stations.add(o.getWorkLocation());  
}
```

# Exercise: AR System – Scenarios (1)



work in progress – see later page for complete solution

## Exercise: AR System – Scenarios (2)

- The Airline creates a Schedule with several daily ScheduledFlights
  - Assume Airline, Airport, and Airplane instances already exist
  - Create as many ScheduledFlights as needed with description, number, departureTime, duration, the departure and destination (both the Airport), and the plane; assign each ScheduledFlight to the schedule of the Airline; contain each ScheduledFlight in ARSystem
  - If an Airplane does not yet exist, first create an Airplane with a name; given the number of seats on the Airplane, create Seats with a unique number for the Airplane and set their containment to the Airplane

## Exercise: AR System – Scenarios (3)

- A new passenger views the ScheduledFlights and then books a seat on one SpecificFlight
  - List all ScheduledFlights in the schedule
  - Upon entering the name and selecting the date for the desired ScheduledFlight, create a Person with the name (contain it in ARSystem)
  - Create a SpecificFlight for the desired date with the selected ScheduledFlight as its flight (assuming that such a SpecificFlight has not yet been created); a Pilot would also have to be assigned to this SpecificFlight); contain the SpecificFlight in ARSystem
  - Create a Booking with the passenger (the Person), on the SpecificFlight, in an empty Seat; contain the Booking in ARSystem

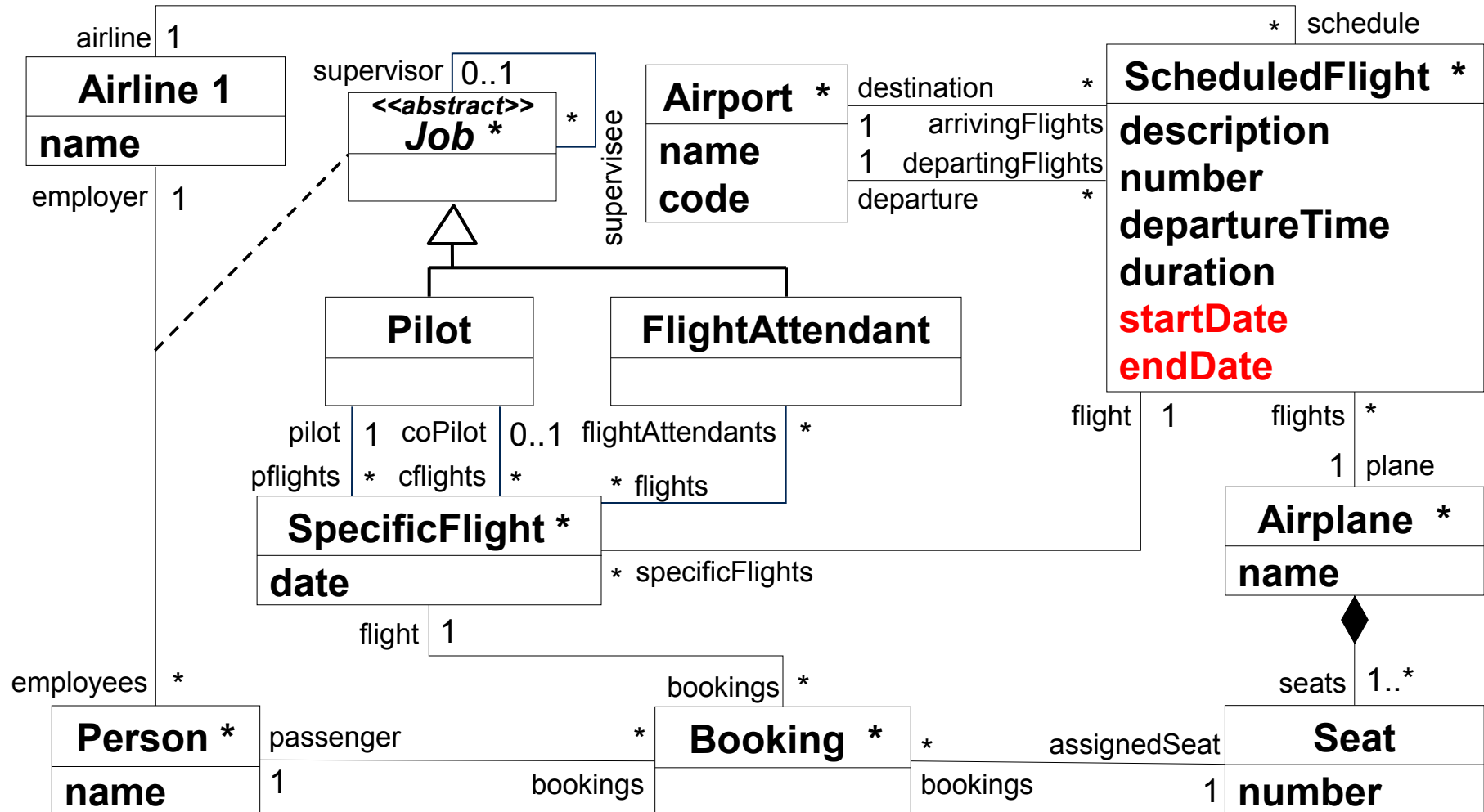


## Exercise: AR System – Scenarios (4)

- The Airline updates the schedule for the current year
  - Cannot remove the existing ScheduledFlights, because they are needed for the SpecificFlight that were booked (if a **history** of past flights is not needed, then old ScheduledFlights could be removed including their SpecificFlights, etc)
  - If an Airline adds a new ScheduledFlight, is the flight available right away or does the new schedule go into effect at a later date?  
(→ **startDate for ScheduledFlight missing**)
  - If an Airline needs to keep a history of all ScheduledFlights, how does it know which flights are currently scheduled and which ones are not  
(→ **endDate missing for ScheduledFlight**)
  - Therefore to update the schedule, the endDate for each ScheduledFlights is set accordingly and new ScheduledFlights are created with the desired startDates and endDates
  - The Airline could also have another association with ScheduledFlights called **currentlyScheduled**, which subsets the schedule association

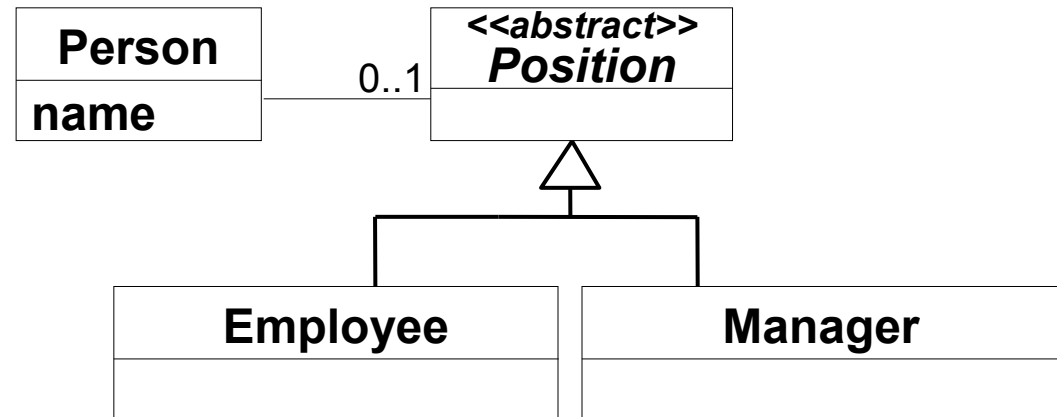
# Exercise: AR System – Scenarios (5)

## ARSystem

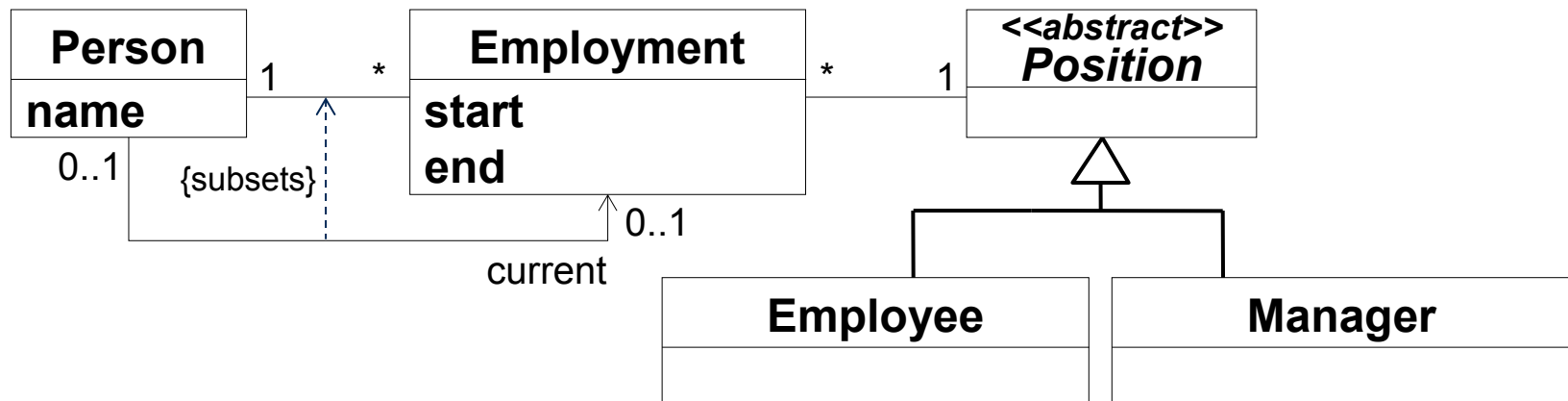


# Exercise: Current State vs. History

- How does the following have to change to capture the job history of a person?



- Keeping track of history in addition to the current state usually requires additional classes and attributes and increases the complexity of the domain model



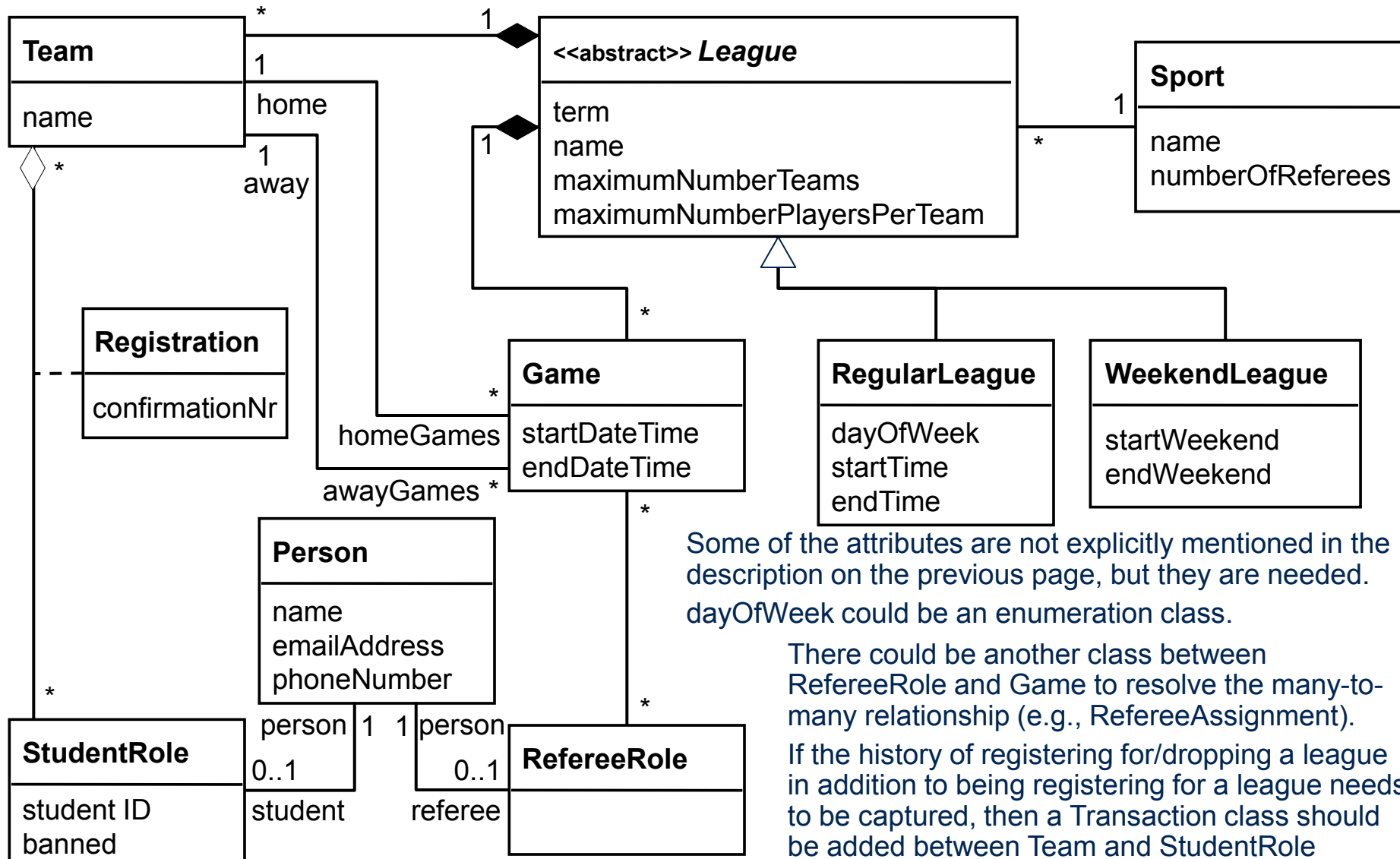


# Exercise: Intramural Registration System (IRS)

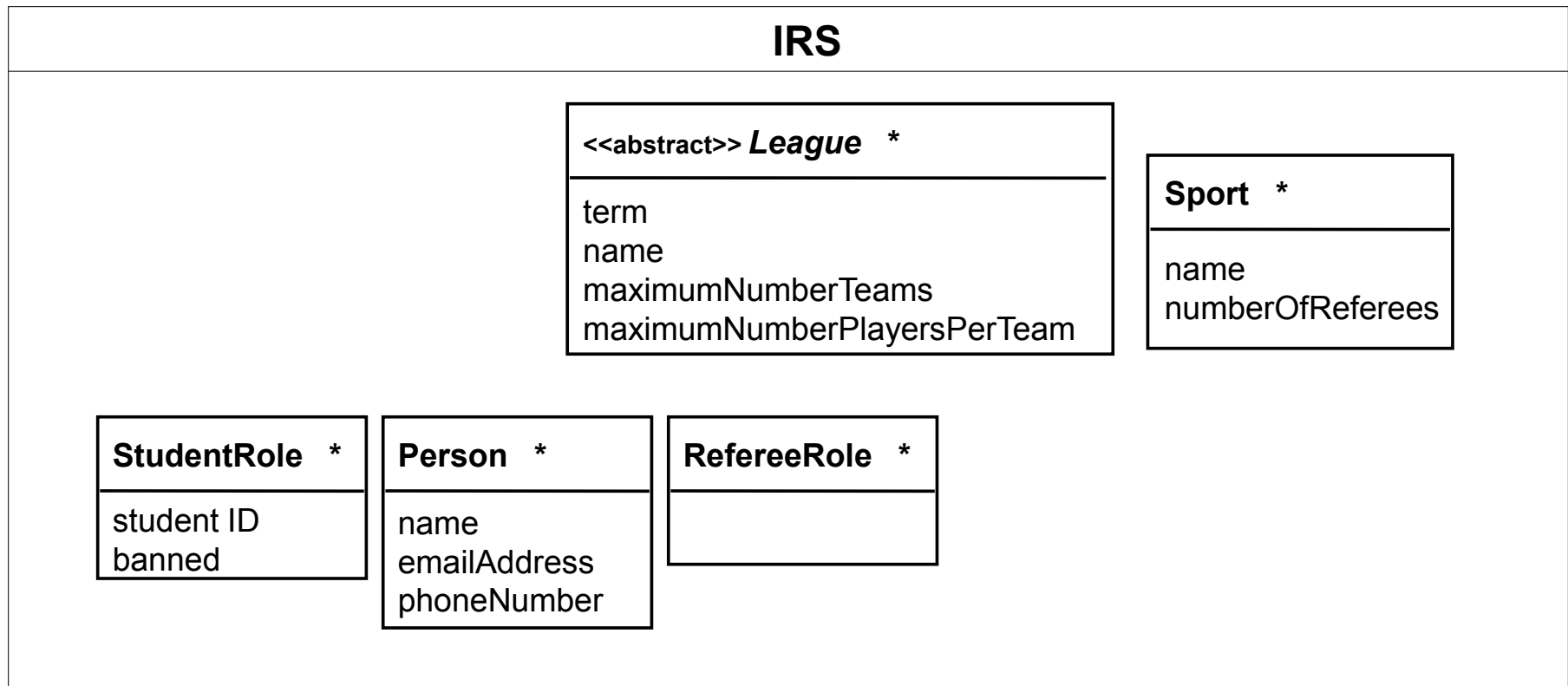
Top University is considering the implementation of an Intramural Registration System (IRS). The IRS allows students to register in and drop various Intramural leagues such as squash, table tennis, volleyball, soccer, ice hockey, etc. on a per term basis. A student is only allowed to register for one team per league. A student, however, may register for several teams in different leagues. The IRS allows to add/delete/modify/search past and current Intramural leagues. The IRS uses the league's name as a code. A regular league is held at a certain time on one day of the week. A weekend league is held on two consecutive weekends (i.e., Saturday and Sunday from 10:00 until 16:00). The maximum number of teams allowed to play in a league and the maximum number of players allowed per team varies from league to league. When registering, a student must provide her/his student's ID, name, email address, and phone number. Furthermore, the IRS provides the ability to ban a student from Intramural leagues until further notice because of unsportsmanlike conduct. Referees also make use of the IRS to sign up for games. Students are not allowed to be referees for the league in which they are also playing in a team. Some sports require only one referee (e.g., table tennis), while others require several referees per game (e.g., ice hockey) or none. **Create the domain model for the IRS and state key constraints.**



# IRS Exercise: Domain Model



# IRS Exercise: Containment Hierarchy



- Game and Team are already contained in League.
- RegularLeague and WeekendLeague are covered by League.
- Registration is an association class.

# IRS Exercise: Constraints

- For each League  $L$ , the number of elements in  $L.teams$  is less than or equal to  $L.maximumNumberTeams$ .
- For each Team  $T$ , the number of elements in  $T.students$  is less than or equal to  $T.league.maximumNumberPlayersPerTeam$ .
- For all pairs of Teams  $T1$  and  $T2$  of StudentRole  $S$ ,  $T1.league$  is not the same as  $T2.league$ .
- For each Game  $G$ ,  $G.startDateTime$  is before  $G.endDateTime$ .
- For each Game  $G$  of a League  $L$  of type RegularLeague,  $G.startDateTime$  and  $G.endDateTime$  have the same day of the week as  $L.dayOfWeek$ ,  $G.startDateTime$  has a time that is after or the same as  $L.startTime$  and before  $L.endTime$ , and  $G.endDateTime$  has a time that is after  $L.startTime$  and before or the same as  $L.endTime$ .
- For each Game  $G$  of a League  $L$  of type WeekendLeague,  $G.startDateTime$  and  $G.endDateTime$  fall on a Saturday or Sunday and in the range of  $L.startWeekend$  and  $L.endWeekend$ .
- If a Person  $P$  has both a StudentRole  $S$  and a RefereeRole  $R$ , then all the Leagues  $L_n$  of the Games  $R.games$  of the RefereeRole  $R$  do not contain any of the Leagues  $L_m$  of the Teams  $S.teams$  of the StudentRole  $S$