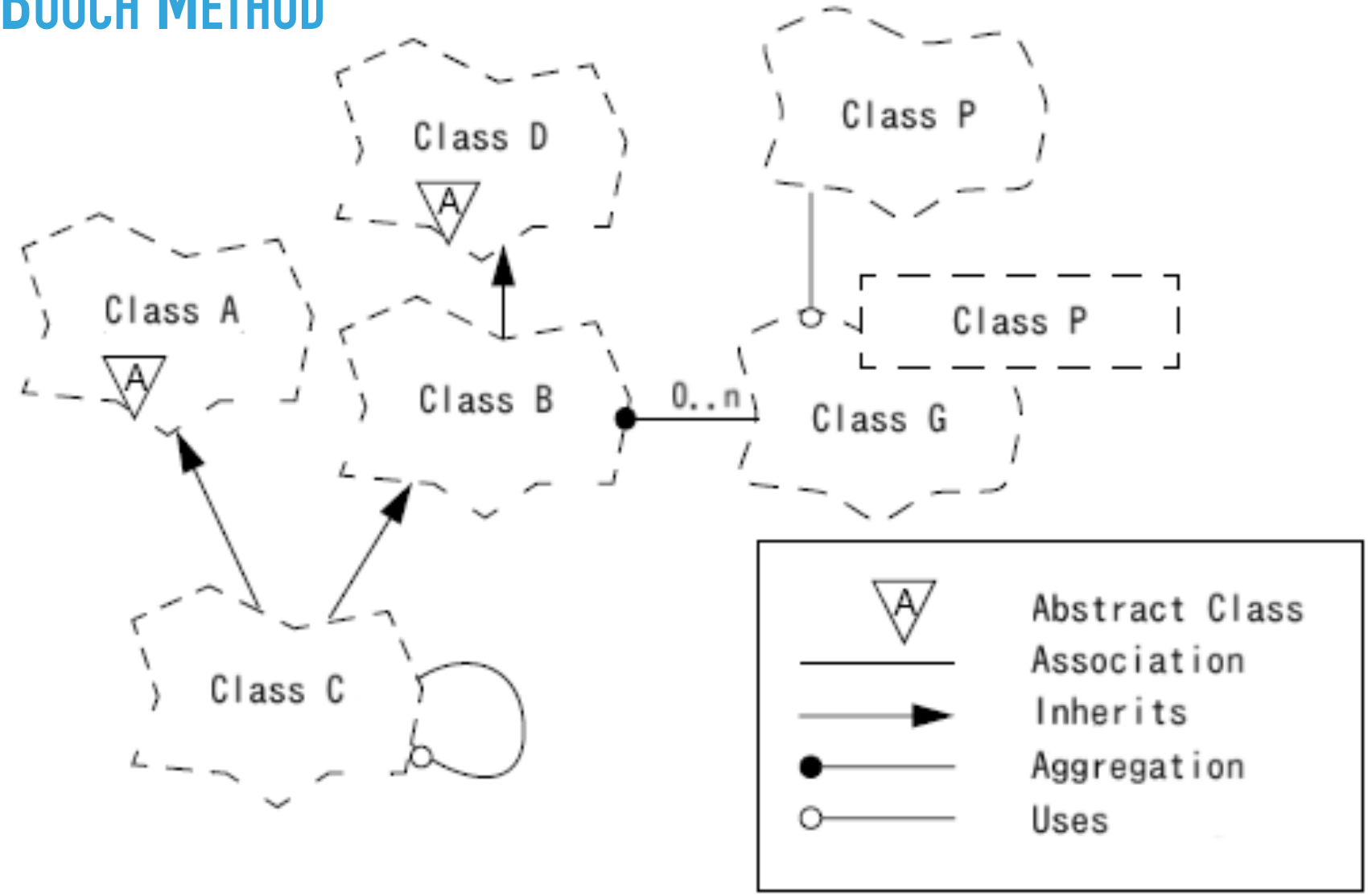
SEG 2105 - LECTURE 05

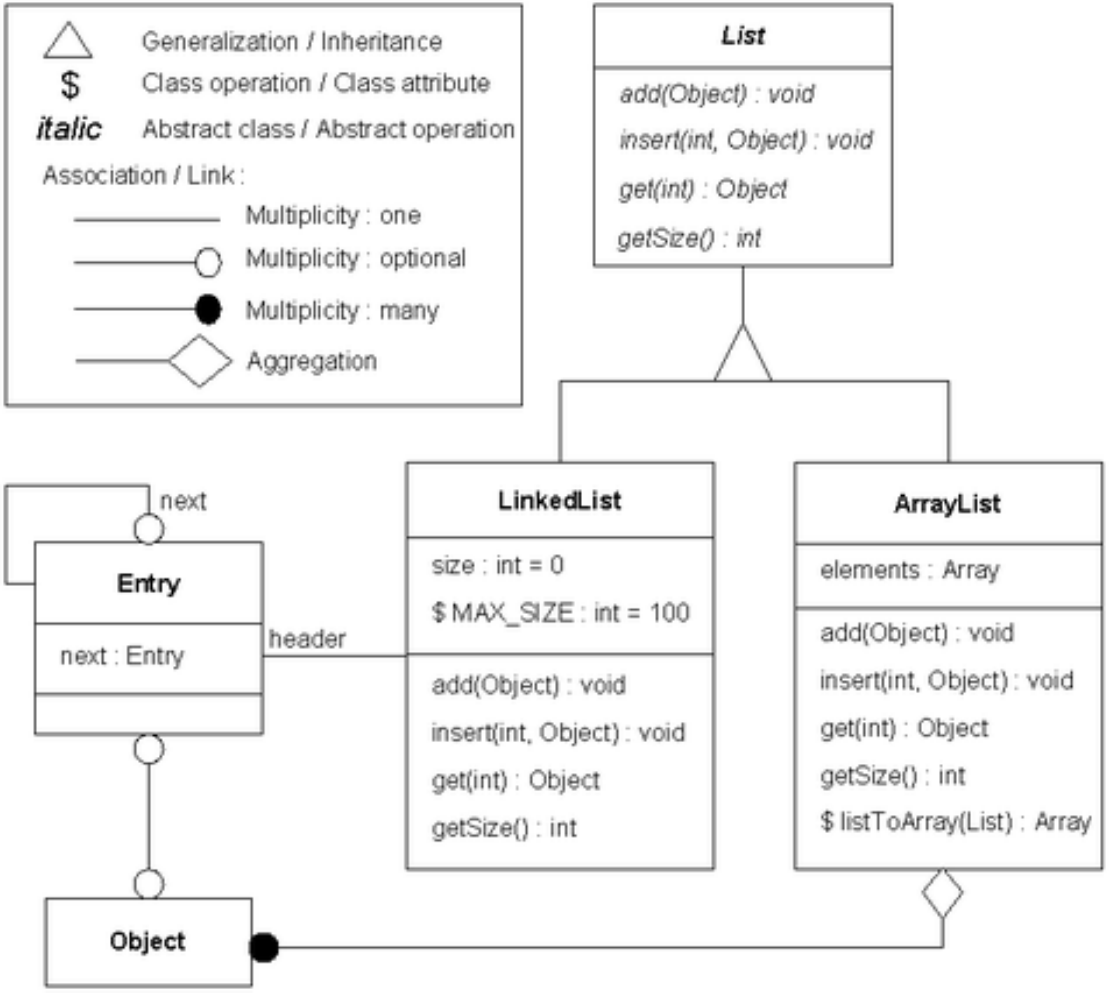
Modeling with Classes

THE BOOCH METHOD



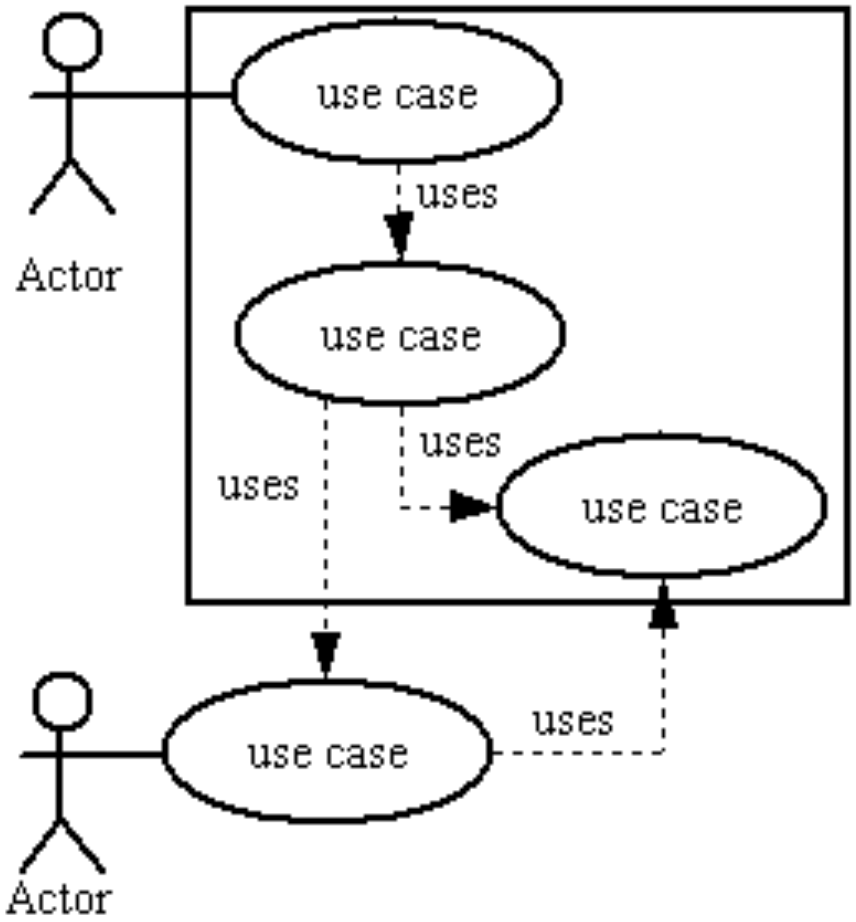
https://en.wikipedia.org/wiki/Booch_method

OBJECT MODELING TECHNIQUE (OMT)



https://en.wikipedia.org/wiki/Object-modeling_technique

OOSE (Object Oriented Software Engineering)



https://www.conceptdraw.com/How-To-Guide/examples-oose-method

Object Modeling Technique	JAMES RUMBAUGH 1991
Booch Method with OO Analysis	GRADY BOOCH 1994
OOSE and Use Cases	IVAN JACOBSON 1992

Rational Software Corporation

1995

JAMES RUMBAUGH
OMT

GRADY BOOCH
BOOCH

IVAN JACOBSON
OOSE

U
Unified

M
Modeling

L
Language

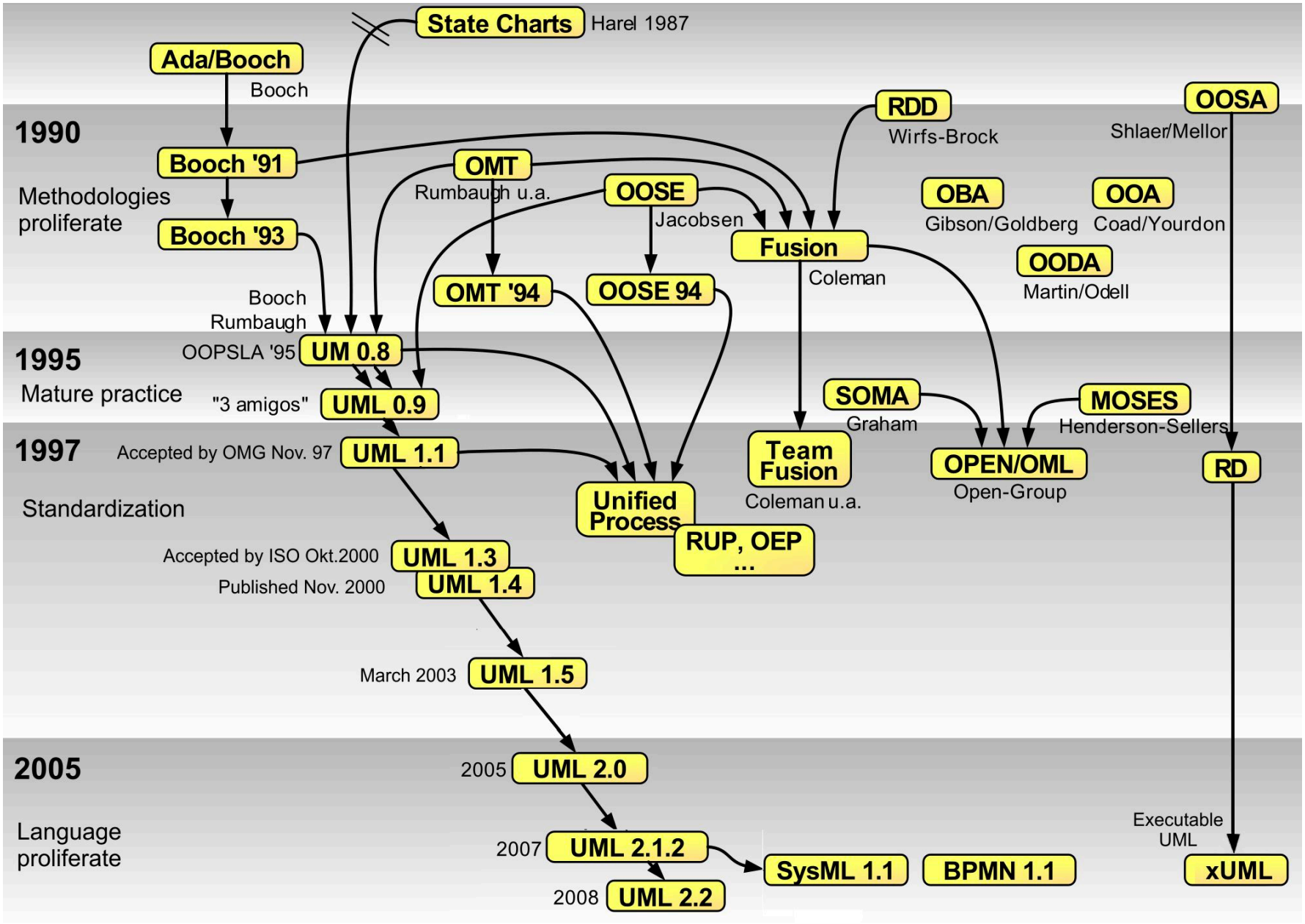
By Rumbaugh, Booch and Jacobson

Object Management Group (OMG)

UML STANDARDIZATION

UML 1.X 1997

UML 2.X 2005



https://en.wikipedia.org/wiki/Unified_Modeling_Language

FORMAL VERSIONS

VERSION	ADOPTION DATE	URL
2.5.1	December 2017	https://www.omg.org/spec/UML/2.5.1
2.4.1	July 2011	https://www.omg.org/spec/UML/2.4.1
2.3	May 2010	https://www.omg.org/spec/UML/2.3
2.2	January 2009	https://www.omg.org/spec/UML/2.2
2.1.2	October 2007	https://www.omg.org/spec/UML/2.1.2
2.0	July 2005	https://www.omg.org/spec/UML/2.0
1.5	March 2003	https://www.omg.org/spec/UML/1.5
1.4	September 2001	https://www.omg.org/spec/UML/1.4
1.3	February 2000	https://www.omg.org/spec/UML/1.3
1.2	July 1999	https://www.omg.org/spec/UML/1.2
1.1	December 1997	https://www.omg.org/spec/UML/1.1

https://www.omg.org/spec/UML/2.5.1

ISO ADOPTED VERSIONS

- UML Superstructure version 2.4.1
- UML Infrastructure version 2.4.1
- OCL (Object Constraint Language) version 2.3.1
- XML Metadata Interchange version 2.4.2.

https://www.omg.org/spec/#M&M

https://www.omg.org/spec/UML

https://www.omg.org/spec/OCL

UML

Diagrams

The Unified Modeling Language

The **Unified Modeling Language™ (UML®)** is a standard visual modeling language intended to be used for

- modeling business and similar processes,
- analysis, design, and implementation of software-based systems

UML is a common language for business analysts, software architects and developers used to describe, specify, design, and document existing or new business processes, structure and behavior of artifacts of software systems.

UML can be applied to diverse **application domains** (e.g., banking, finance, internet, aerospace, healthcare, etc.) It can be used with all major object and component **software development methods** and for various **implementation platforms** (e.g., J2EE, .NET).

UML is a standard modeling **language**, not a **software development process**. UML 1.4.2 Specification explained that process:

- provides guidance as to the order of a team's activities,
- specifies what artifacts should be developed,
- directs the tasks of individual developers and the team as a whole, and
- offers criteria for monitoring and measuring a project's products and activities.

UML is intentionally **process independent** and could be applied in the context of different processes. Still, it is most suitable for use case driven, iterative and incremental development processes. An example of such process is **Rational Unified Process** (RUP).

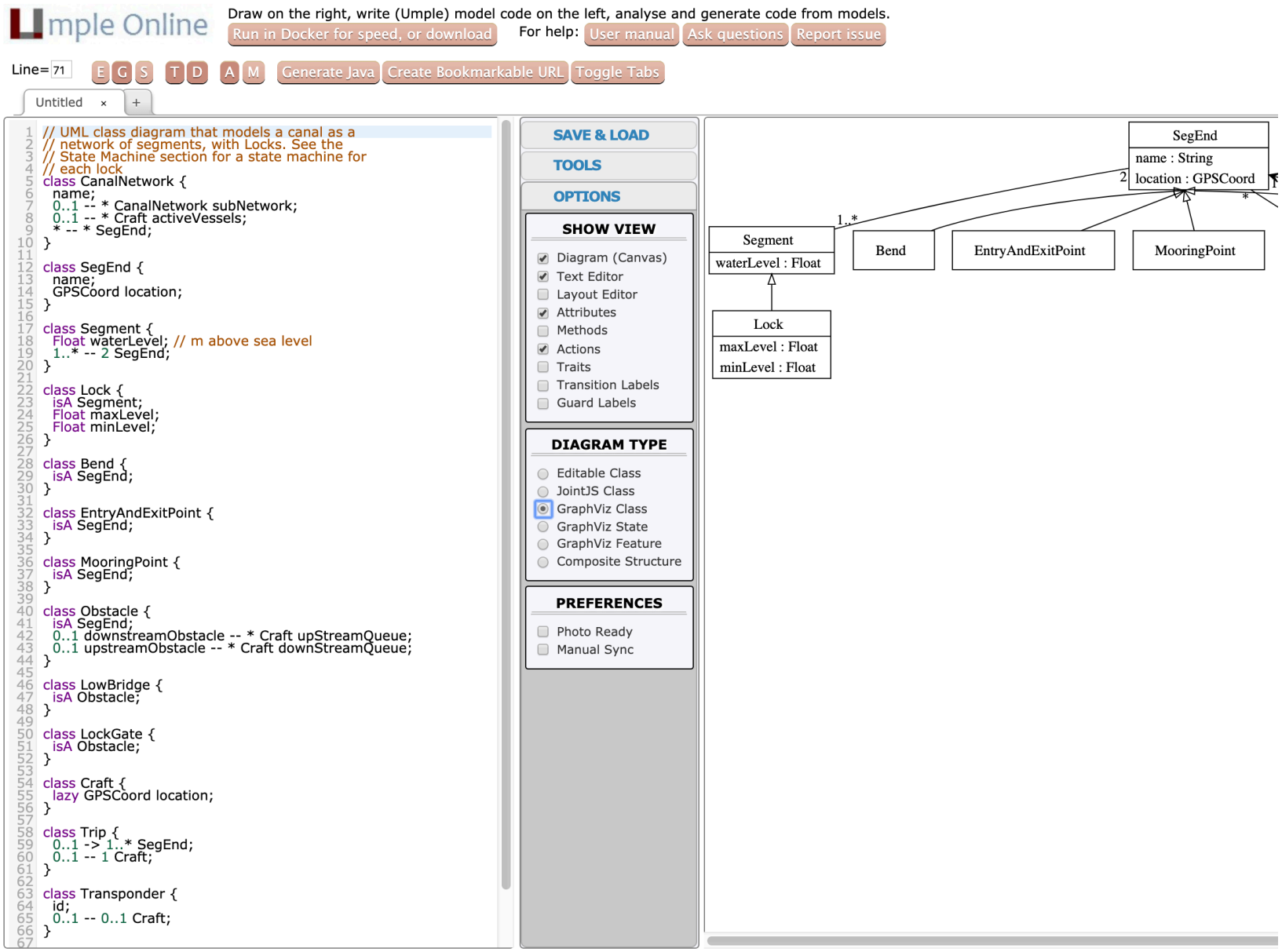
UML is not complete and it is not completely visual. Given some UML diagram, we can't be sure to understand depicted part or behavior of the system from the diagram alone. Some information could be intentionally omitted from the diagram, some information represented on the diagram could have different interpretations, and some concepts of UML have no graphical notation at all, so there is no way to depict those on diagrams.

For example, semantics of **multiplicity of actors** and **multiplicity of use cases** on **use case diagrams** is not defined precisely in the UML specification and could mean either concurrent or successive usage of use cases.

Name of an **abstract classifier** is shown in italics while **final classifier** has no specific graphical notation, so there is no way to determine whether classifier is final or not from the diagram.

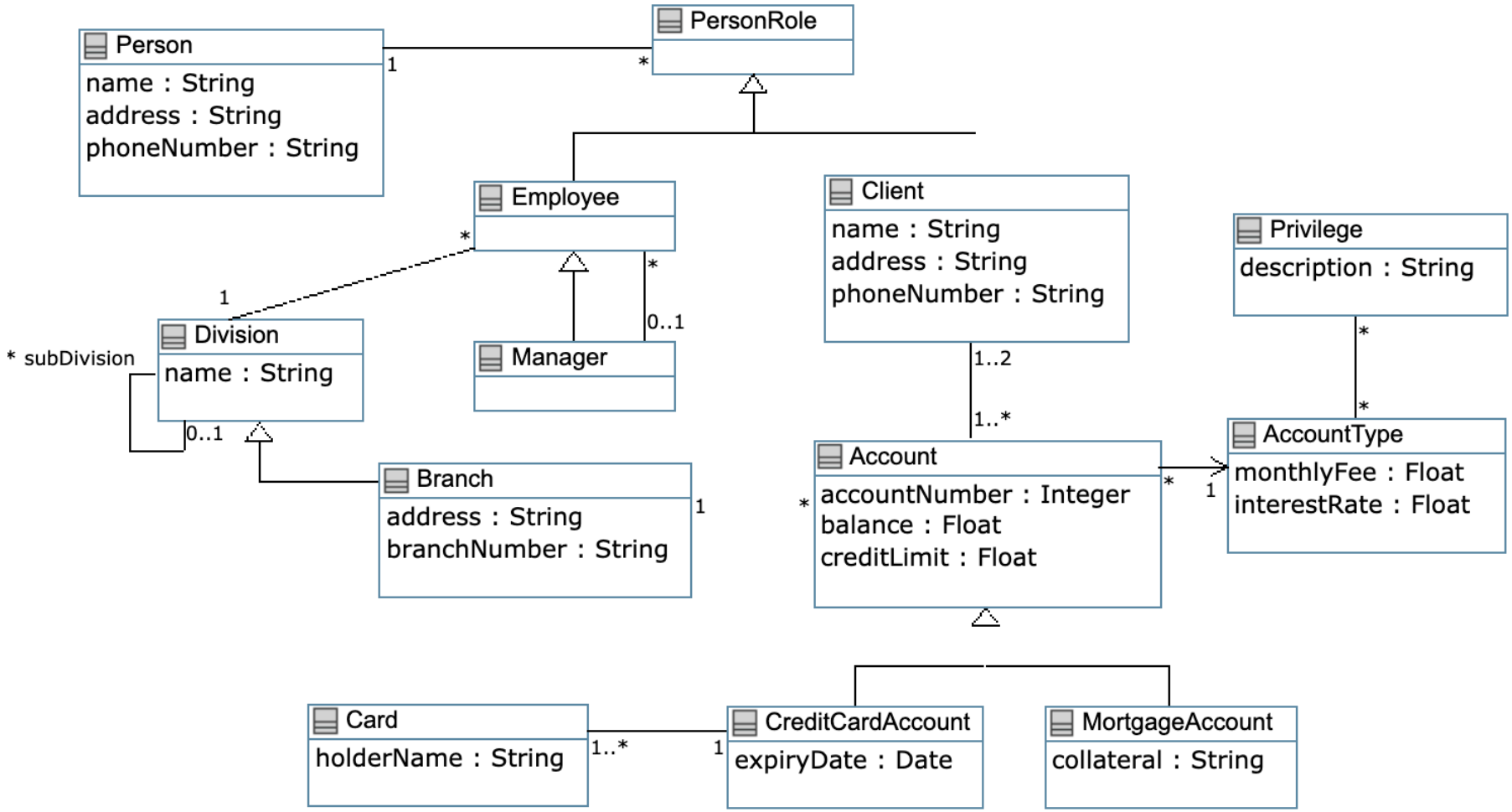
---

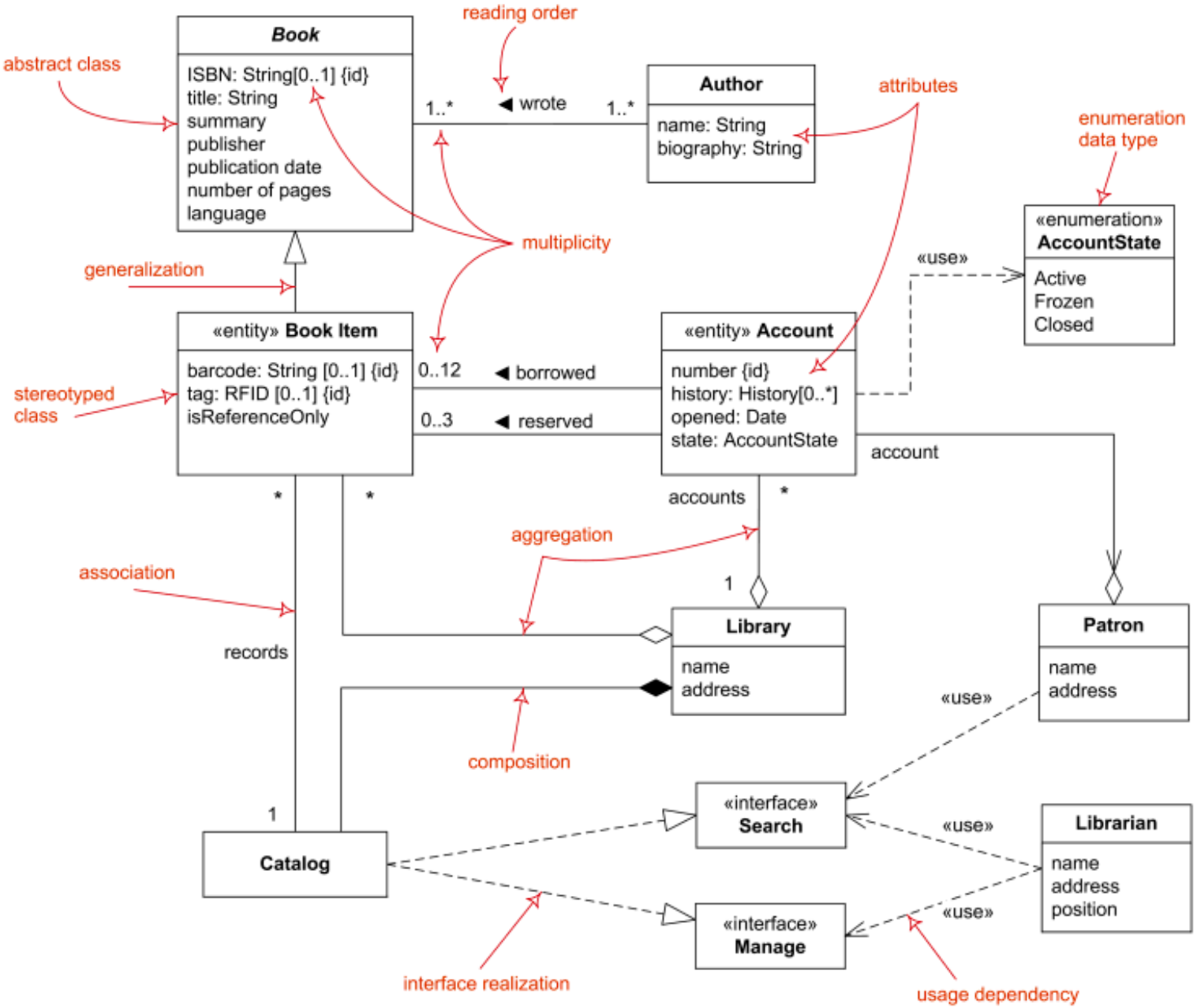
https://www.uml-diagrams.org/



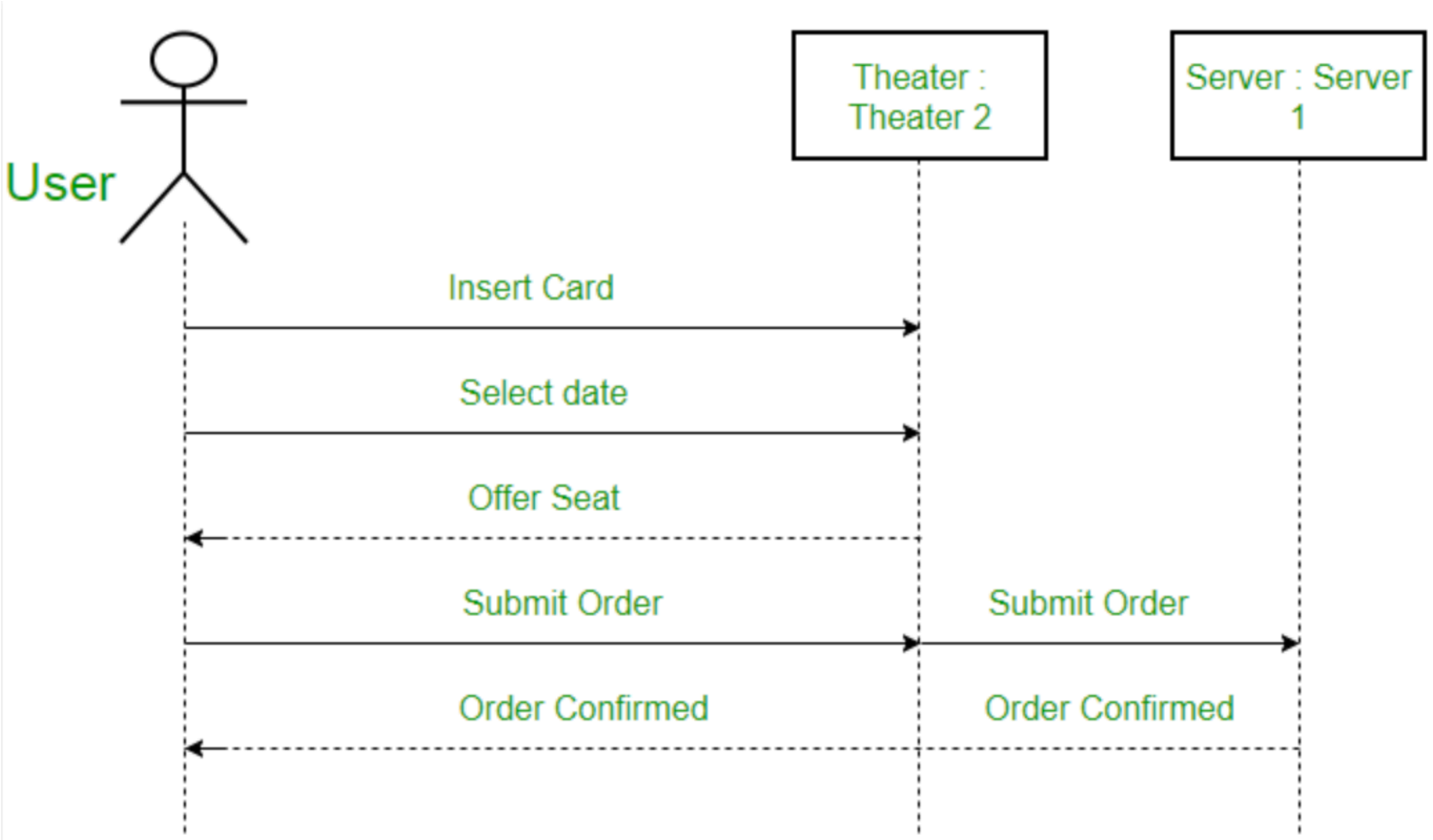
http://try.umple.org

CLASS DIAGRAMS

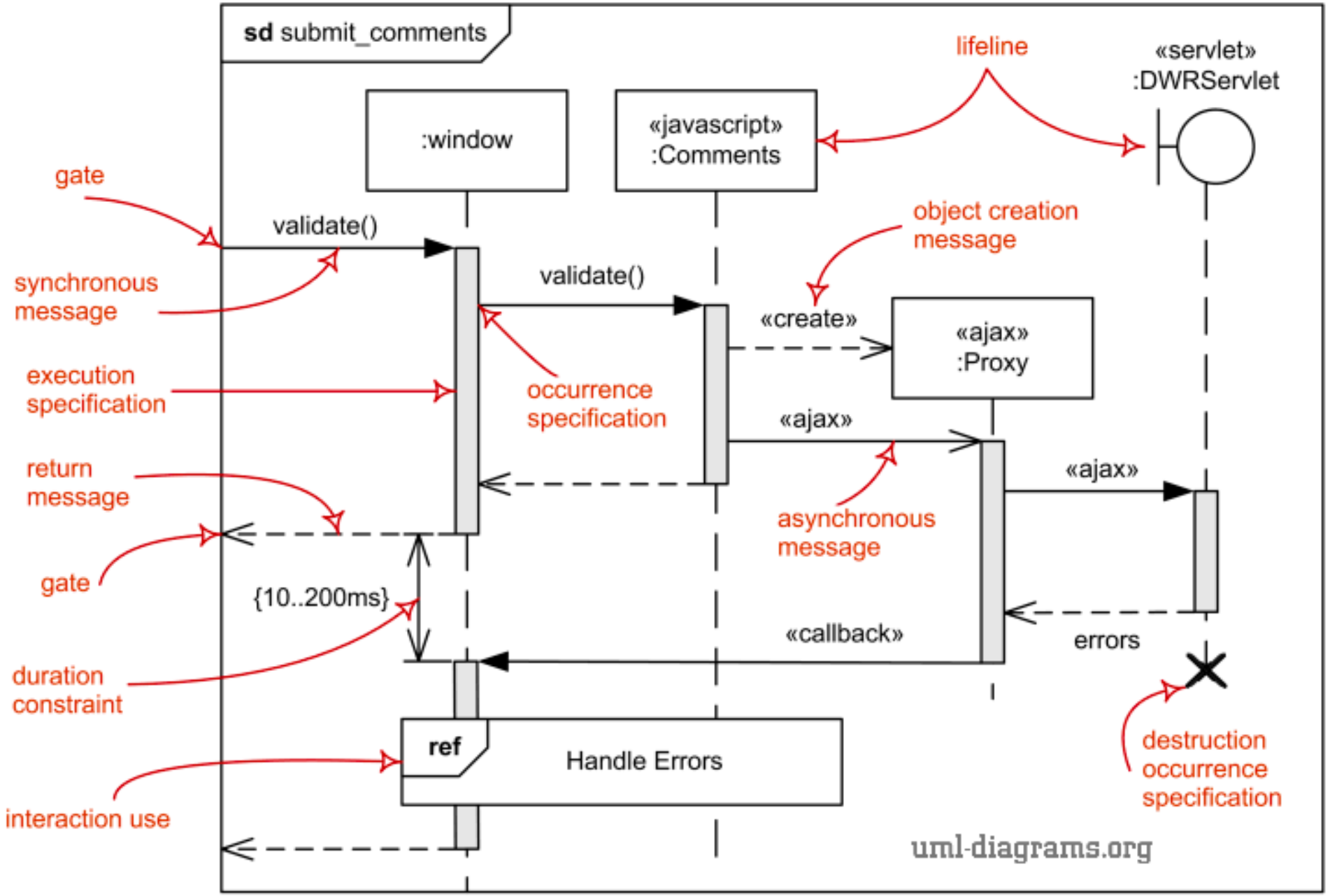



Book
ISBN: String[0..1] {id}
title: String
summary
publisher
publication date
number of pages
language
abstract class
reading order
1..*
wrote
1..*
Author
name: String
biography: String
attributes
enumeration data type
«enumeration» AccountState
Active
Frozen
Closed
«use»
multiplicity
generalization
«entity» Book Item
barcode: String [0..1] {id}
tag: RFID [0..1] {id}
isReferenceOnly
stereotyped class
0..12
borrowed
0..3
reserved
«entity» Account
number {id}
history: History[0..*]
opened: Date
state: AccountState
account
accounts
*
*
*
aggregation
association
1
Library
name
address
Patron
name
address
records
composition
«use»
1
«interface» Search
«use»
Librarian
name
address
position
Catalog
«interface» Manage
«use»
interface realization
usage dependency

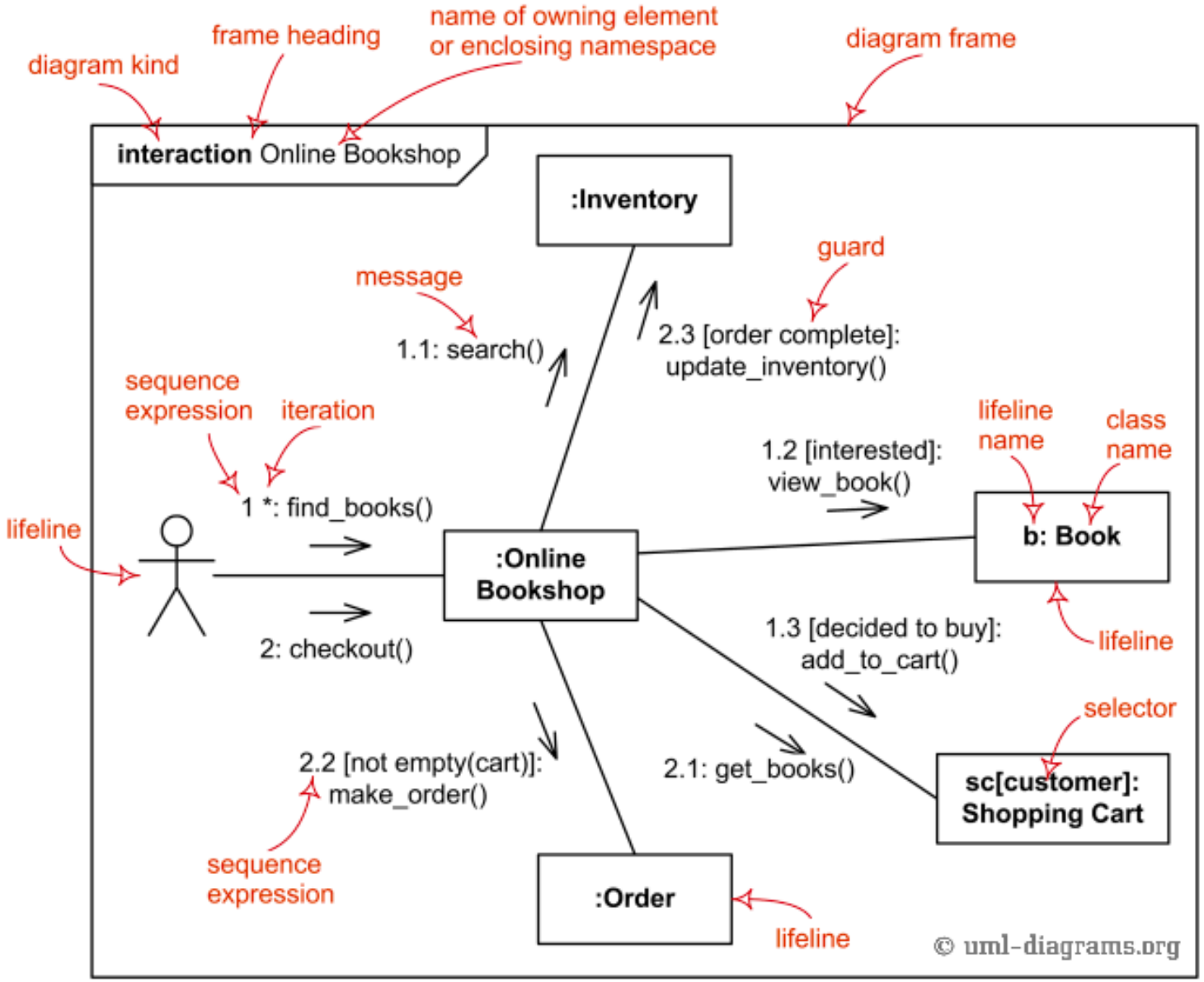
SEQUENCE DIAGRAMS (INTERACTIONS)



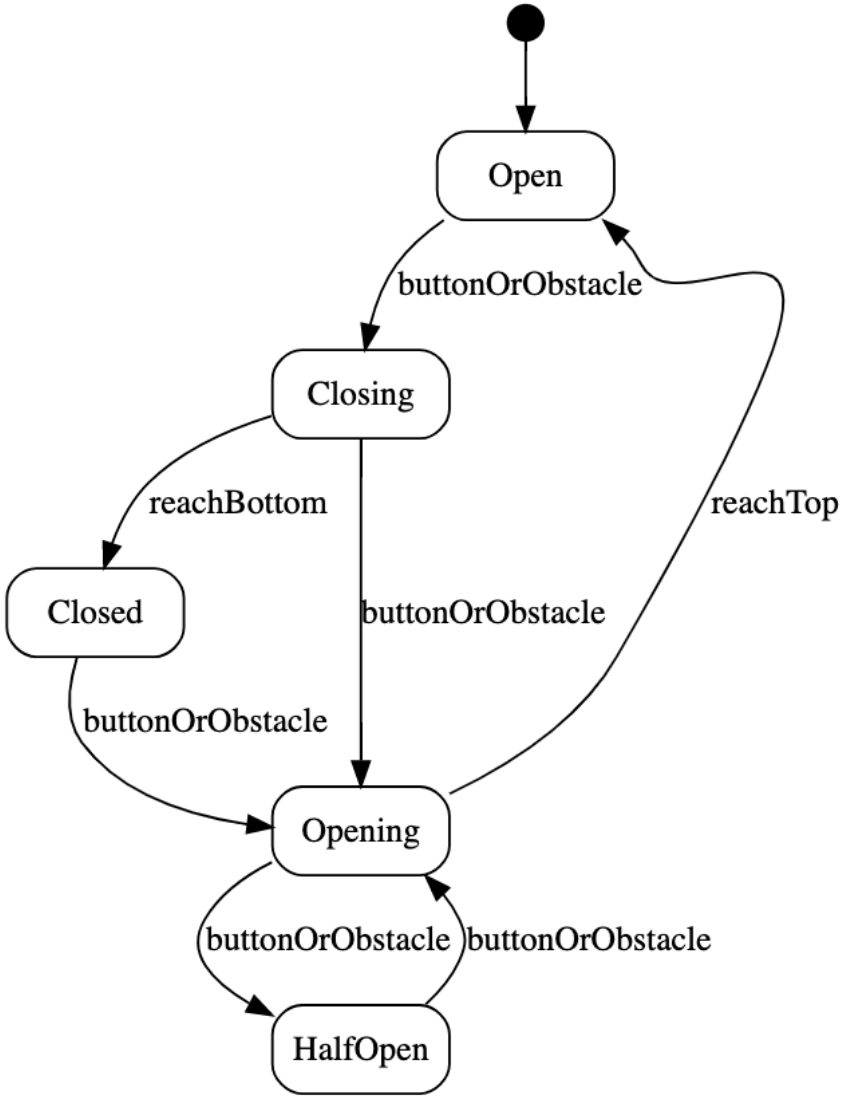
https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/


sd submit_comments
lifeline
«servlet»
:DWRServlet
:window
«javascript»
:Comments
gate
validate()
object creation
message
synchronous
message
validate()
«create»
«ajax»
:Proxy
execution
specification
occurrence
specification
«ajax»
return
message
«ajax»
asynchronous
message
gate
{10..200ms}
«callback»
errors
duration
constraint
ref
Handle Errors
destruction
occurrence
specification
interaction use
uml-diagrams.org

COMMUNICATION DIAGRAMS (INTERACTIONS)



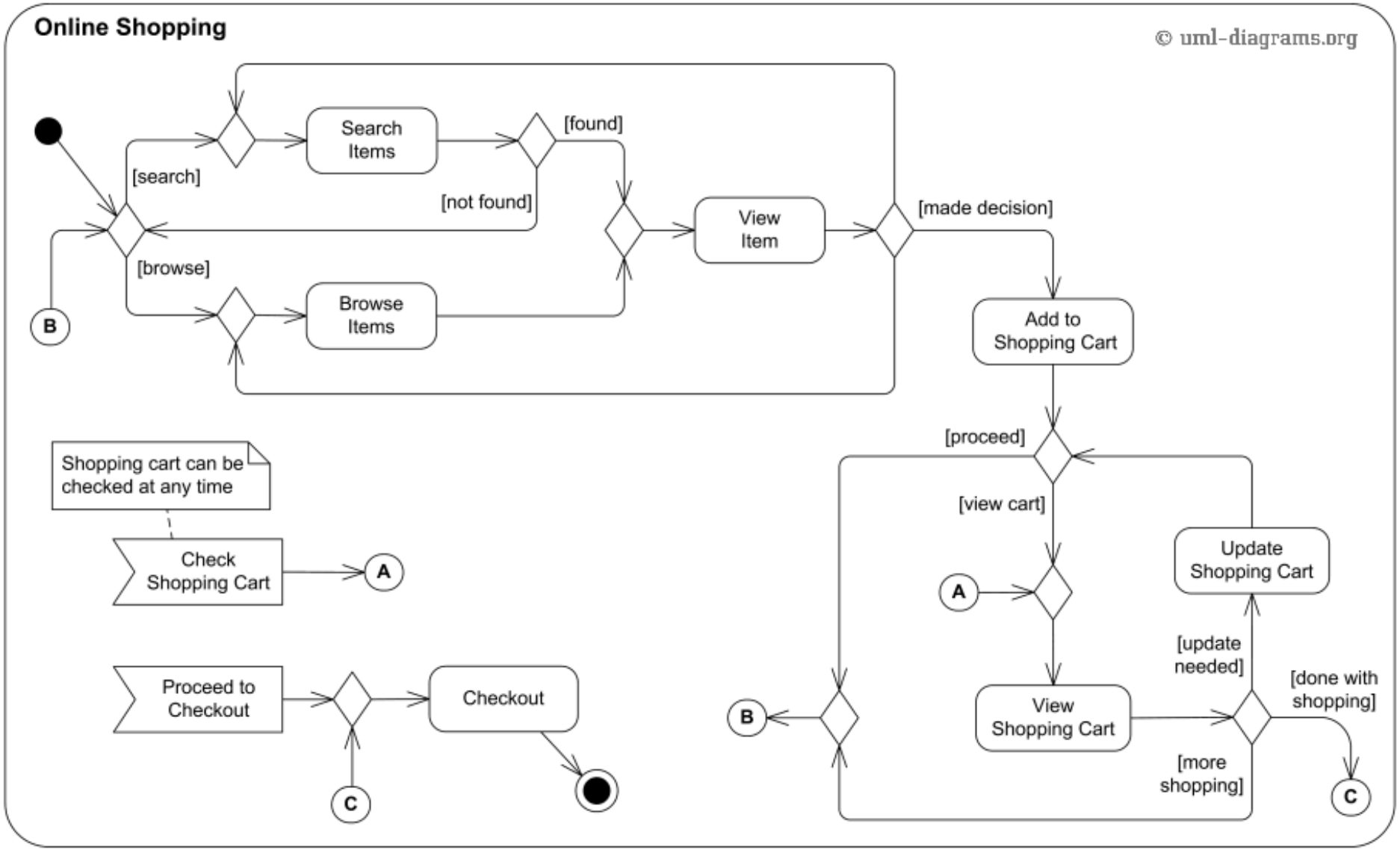
State diagrams

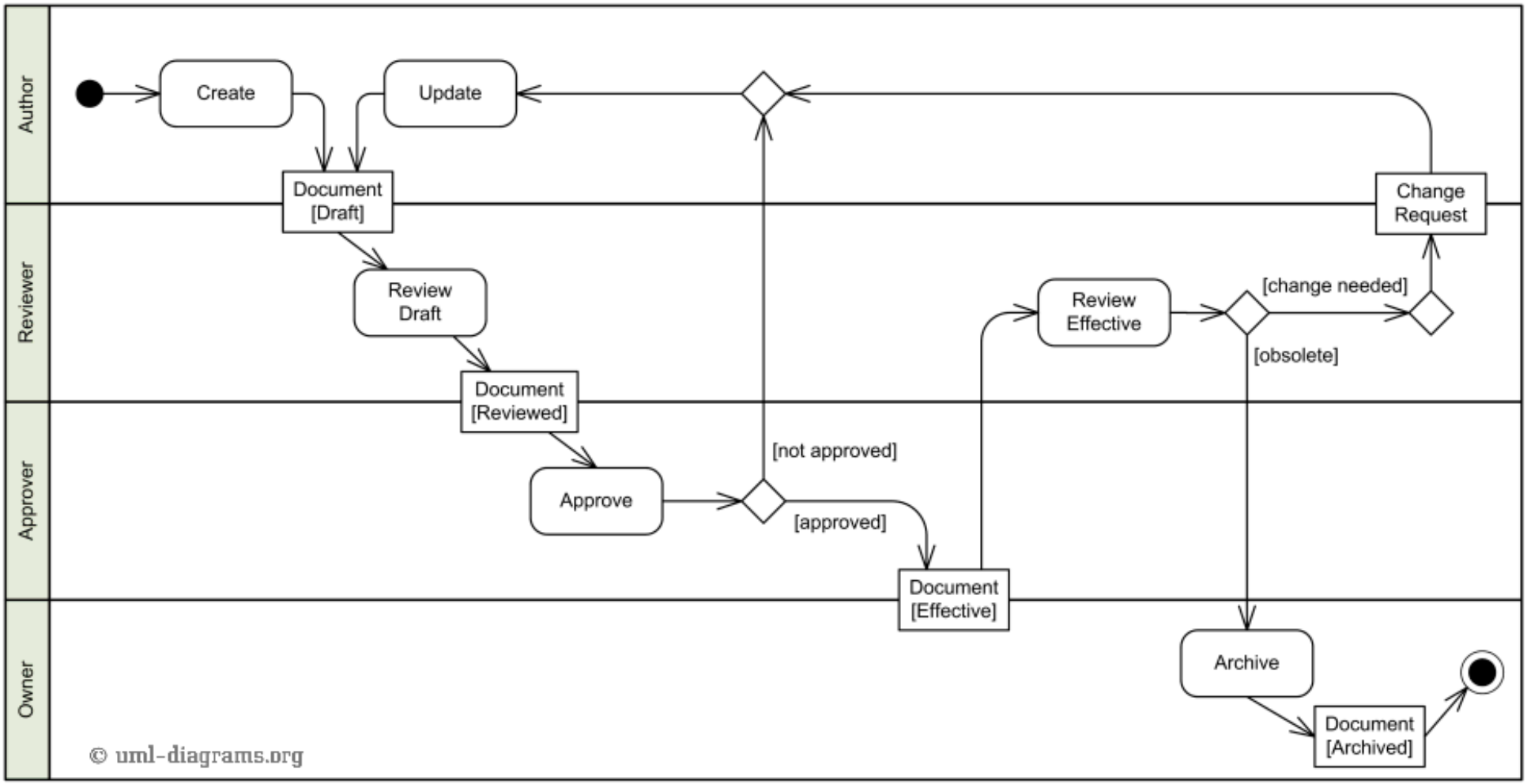


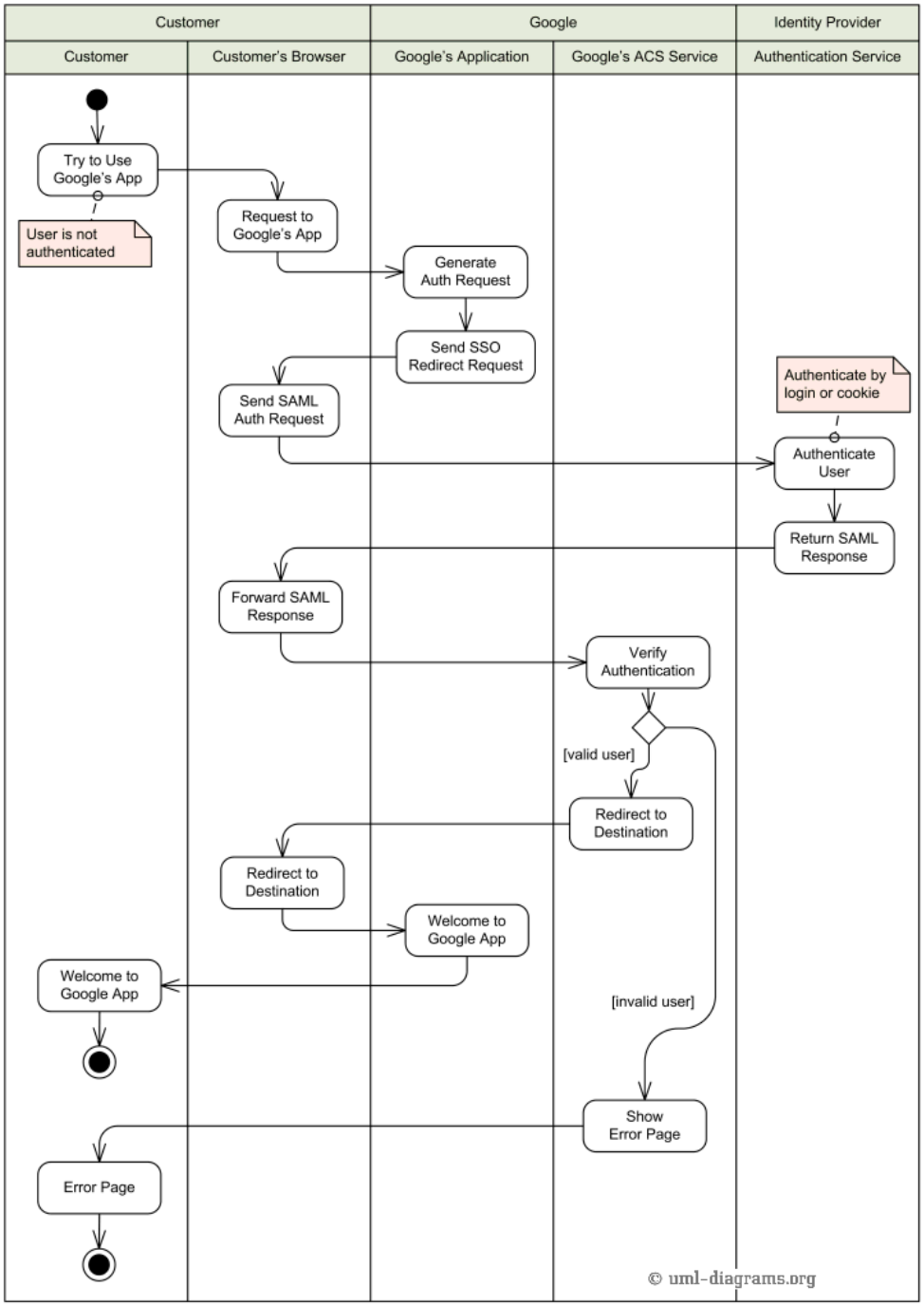
```
public boolean buttonOrObstacle()
{
  boolean wasEventProcessed = false;

  Status aStatus = status;
  switch (aStatus)
  {
    case Open:
      setStatus(Status.Closing);
      wasEventProcessed = true;
      break;
    case Closing:
      setStatus(Status.Opening);
      wasEventProcessed = true;
      break;
    case Closed:
      setStatus(Status.Opening);
      wasEventProcessed = true;
      break;
    case Opening:
      setStatus(Status.HalfOpen);
      wasEventProcessed = true;
      break;
    case HalfOpen:
      setStatus(Status.Opening);
      wasEventProcessed = true;
      break;
    default:
      // Other states do respond to this event
  }

  return wasEventProcessed;
}
```

```
public boolean reachBottom()
{
  boolean wasEventProcessed = false;

  Status aStatus = status;
  switch (aStatus)
  {
    case Closing:
      setStatus(Status.Closed);
      wasEventProcessed = true;
      break;
    default:
      // Other states do respond to this event
  }

  return wasEventProcessed;
}

public boolean reachTop()
{
  boolean wasEventProcessed = false;

  Status aStatus = status;
  switch (aStatus)
  {
    case Opening:
      setStatus(Status.Open);
      wasEventProcessed = true;
      break;
    default:
      // Other states do respond to this event
  }

  return wasEventProcessed;
}
```

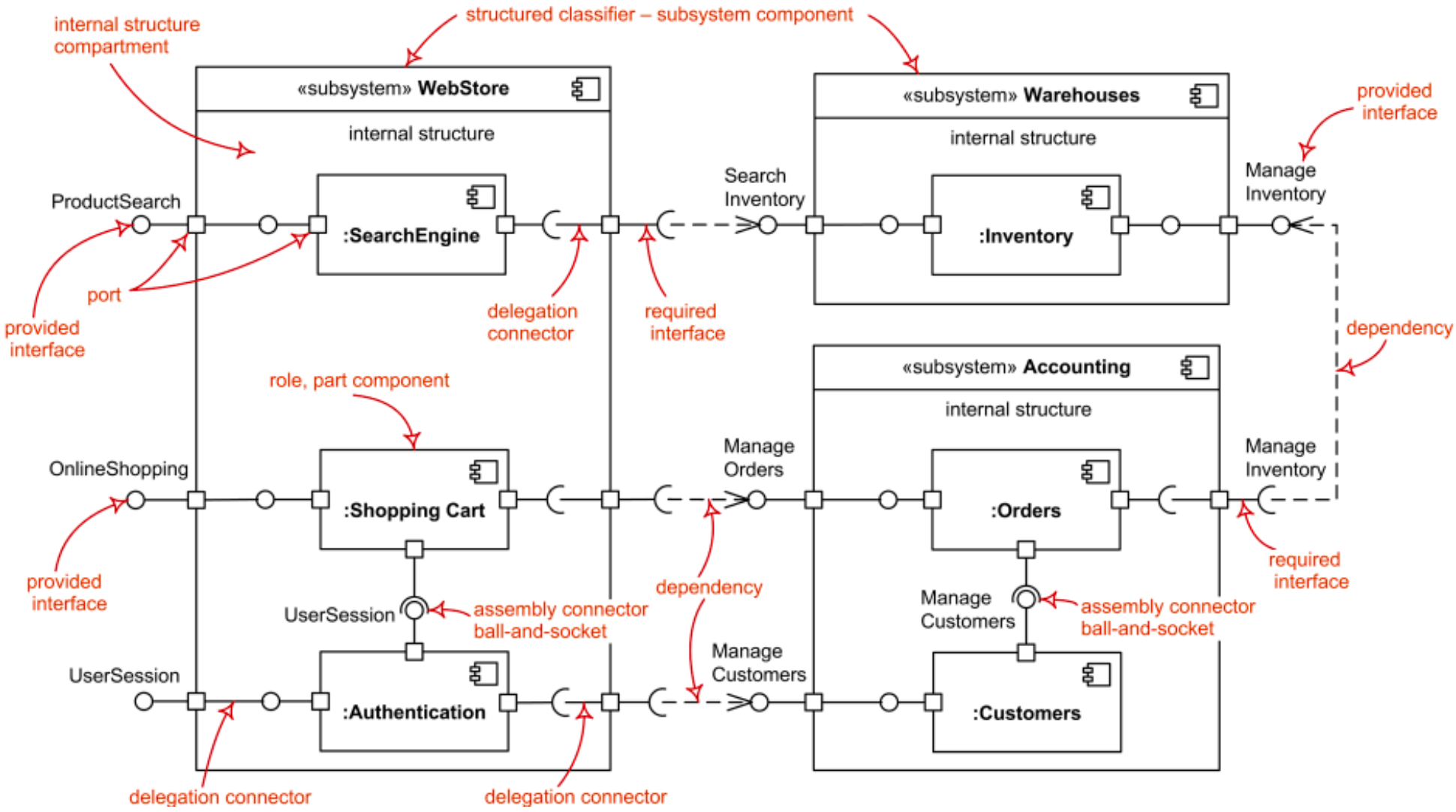
ACTIVITY DIAGRAMS



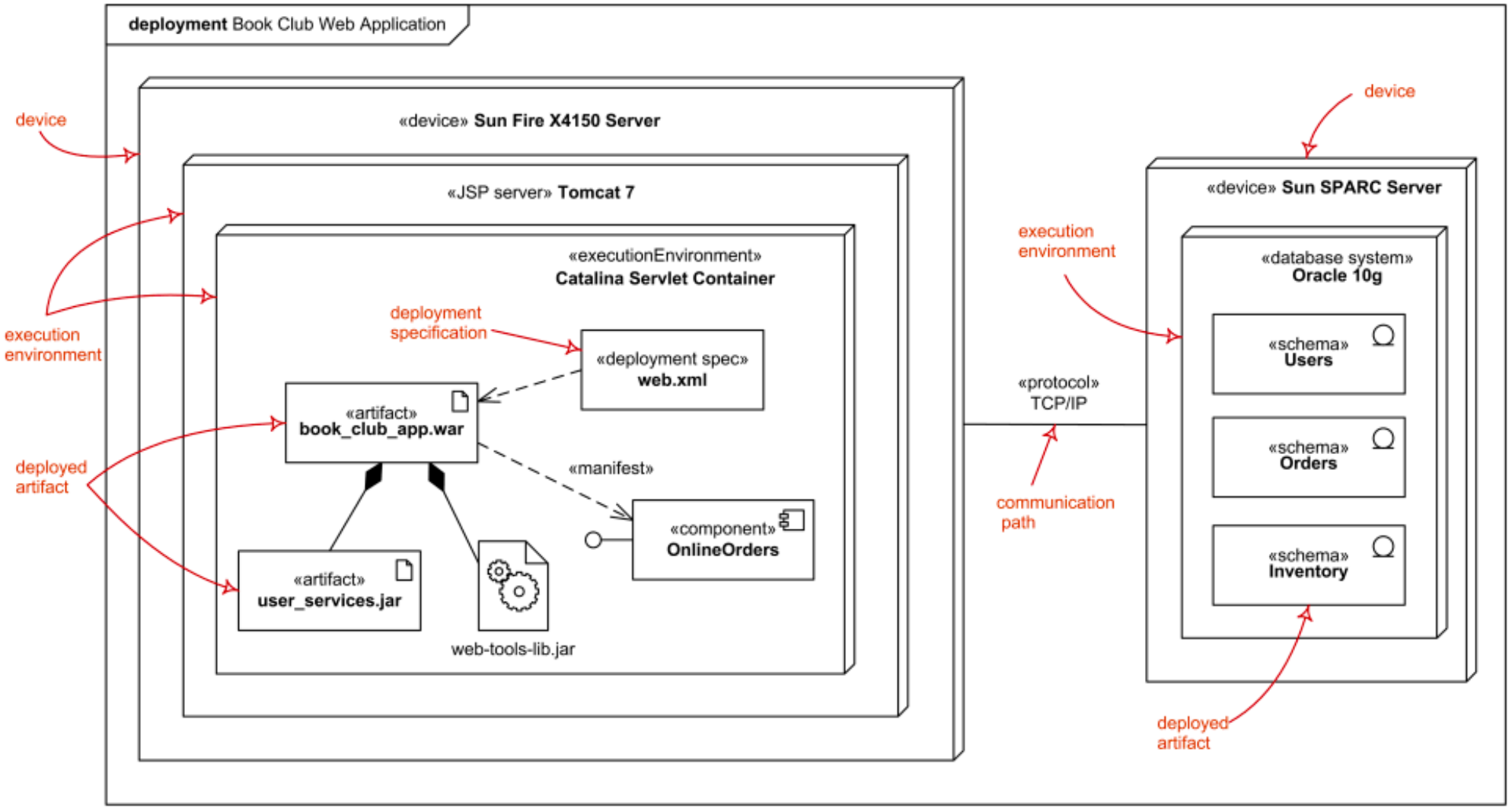

Author
Reviewer
Approver
Owner
Create
Update
Document [Draft]
Review Draft
Document [Reviewed]
Approve
[not approved]
[approved]
Review Effective
[change needed]
[obsolete]
Change Request
Document [Effective]
Archive
Document [Archived]
© uml-diagrams.org


Customer
Google
Identity Provider
Customer
Customer's Browser
Google's Application
Google's ACS Service
Authentication Service
Try to Use Google's App
User is not authenticated
Request to Google's App
Generate Auth Request
Send SSO Redirect Request
Send SAML Auth Request
Authenticate by login or cookie
Authenticate User
Return SAML Response
Forward SAML Response
Verify Authentication
[valid user]
Redirect to Destination
Redirect to Destination
Welcome to Google App
Welcome to Google App
[invalid user]
Show Error Page
Error Page
© uml-diagrams.org

COMPONENT DIAGRAMS



DEPLOYMENT DIAGRAM



ESSENTIALS OF UML

CLASS DIAGRAMS

Classes

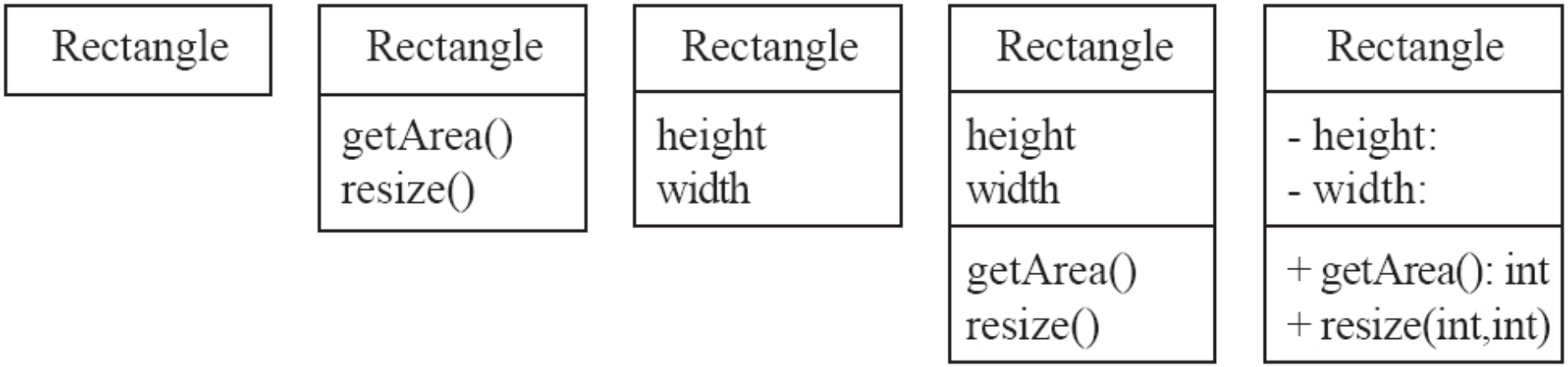
Operations

Associations

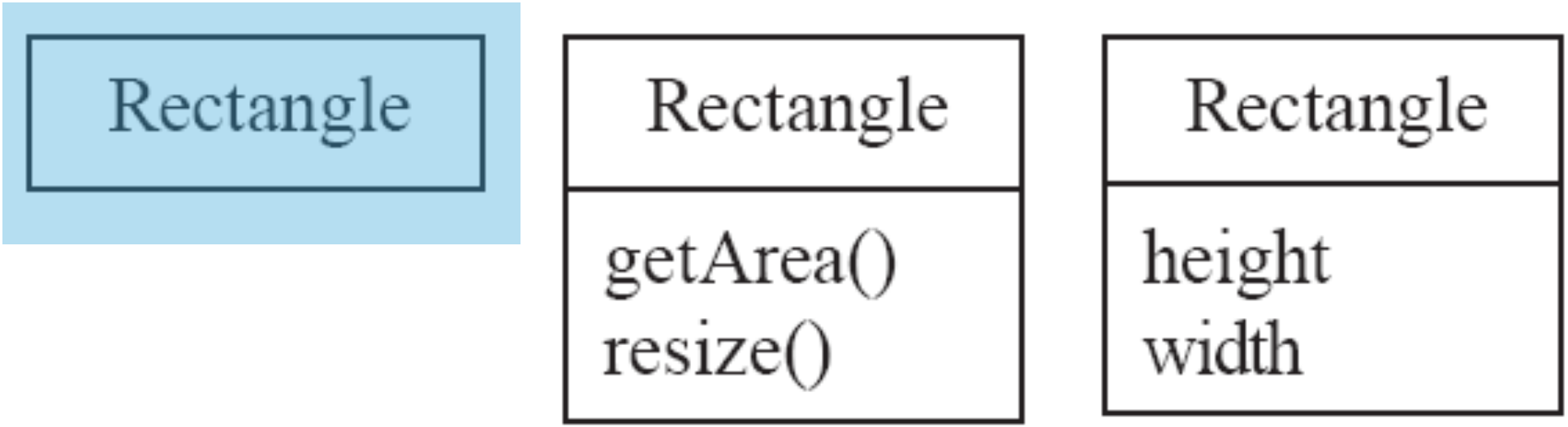
Generalizations

Attributes

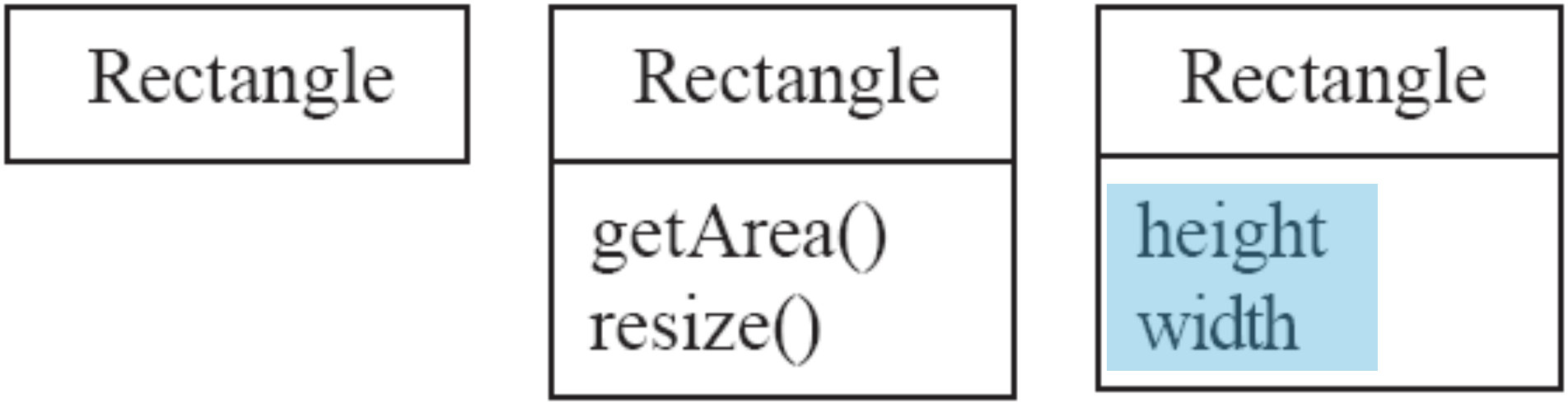
VARYING DEGREES OF SPECIFICITY



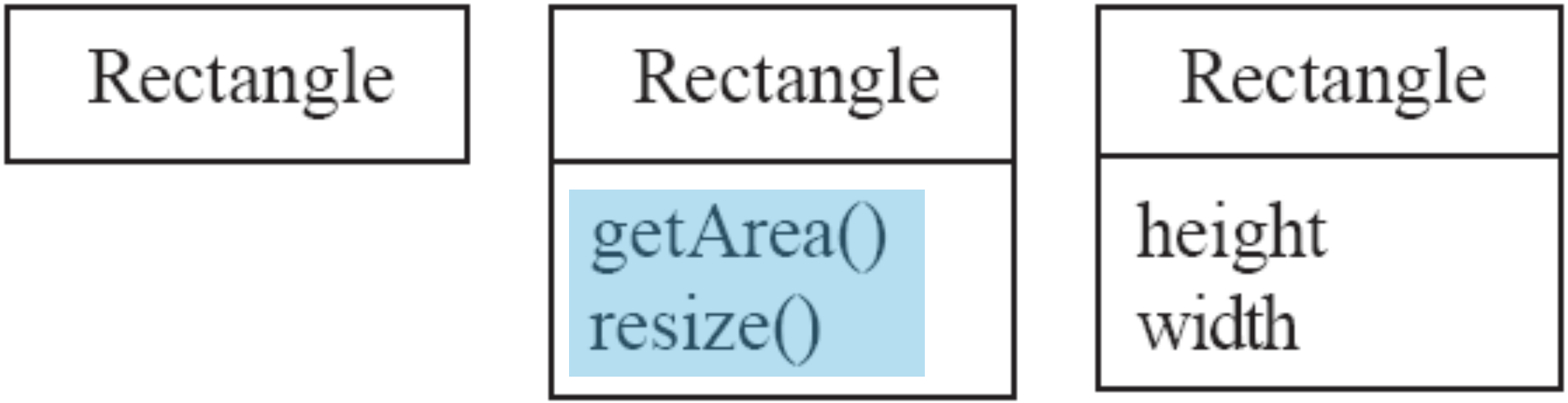
VARYING DEGREES OF SPECIFICITY

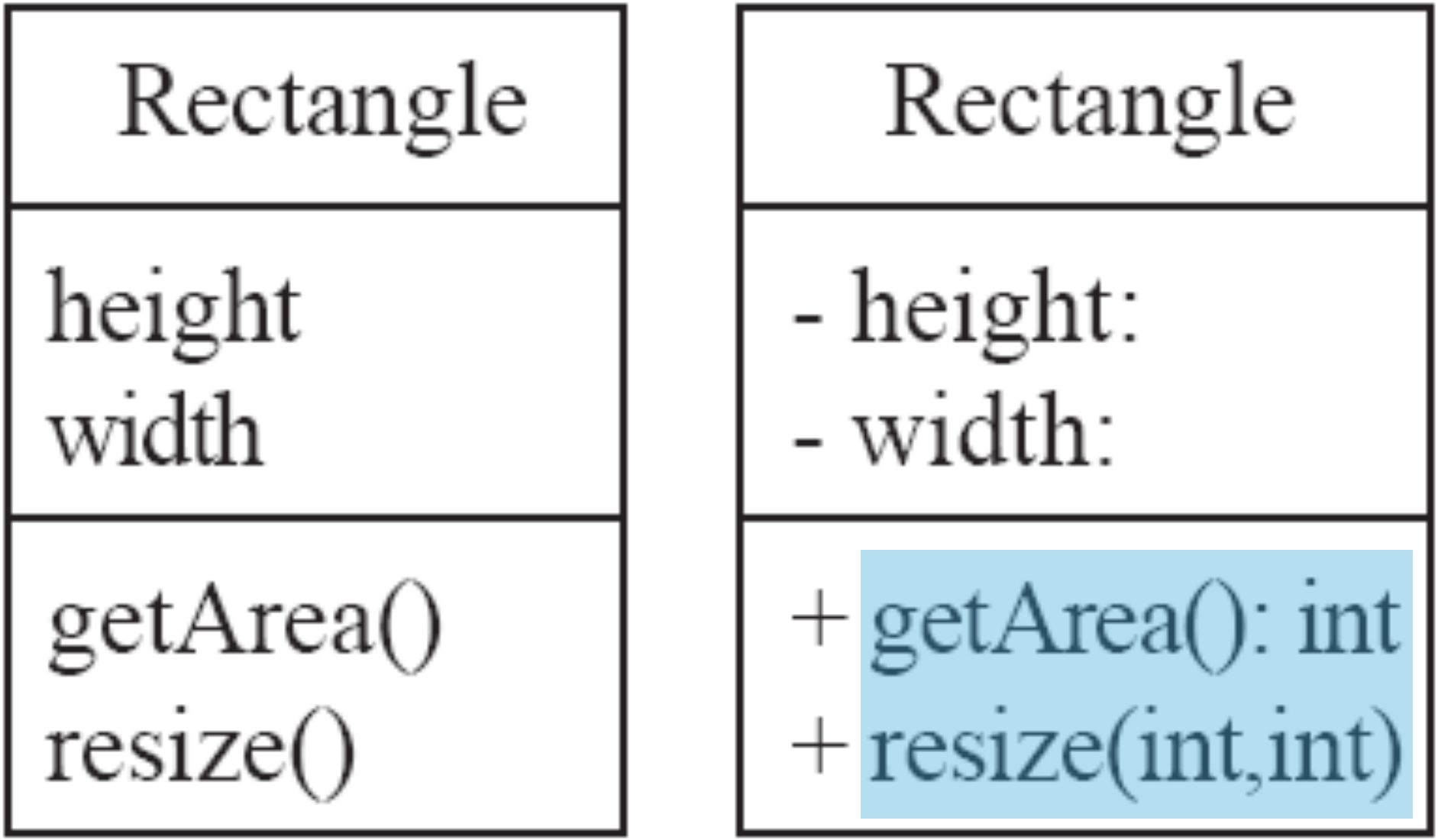


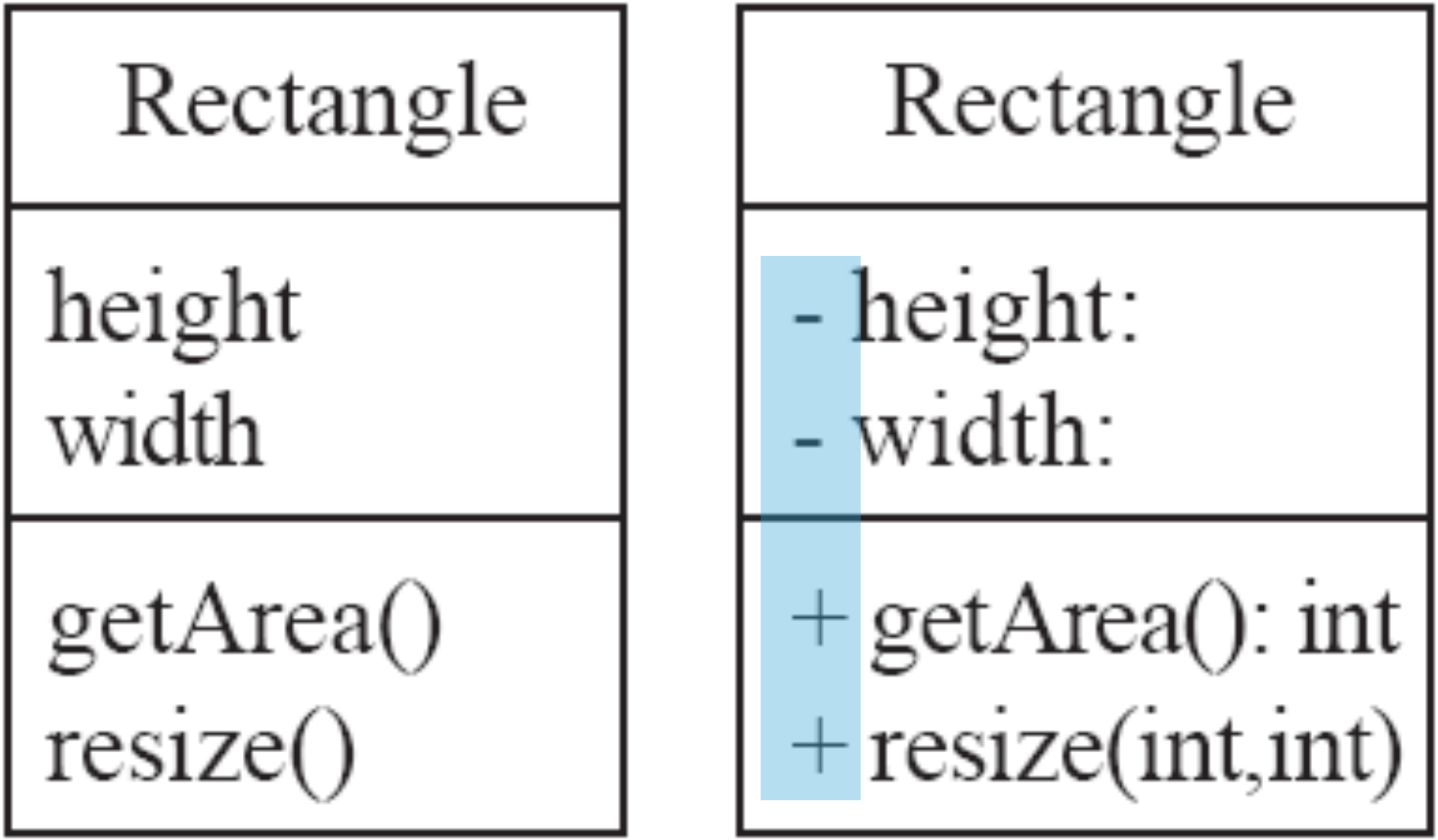
VARYING DEGREES OF SPECIFICITY



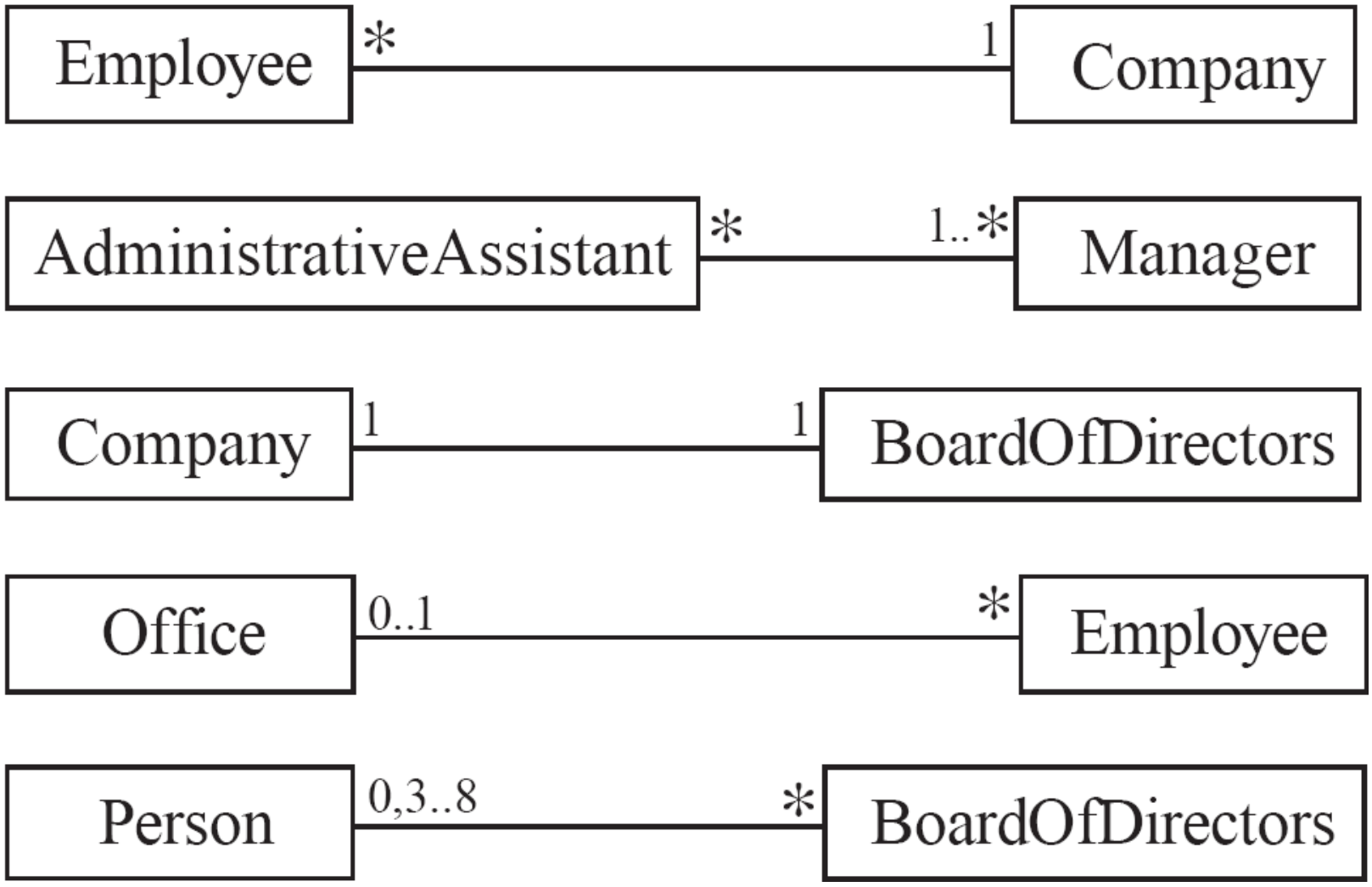
VARYING DEGREES OF SPECIFICITY



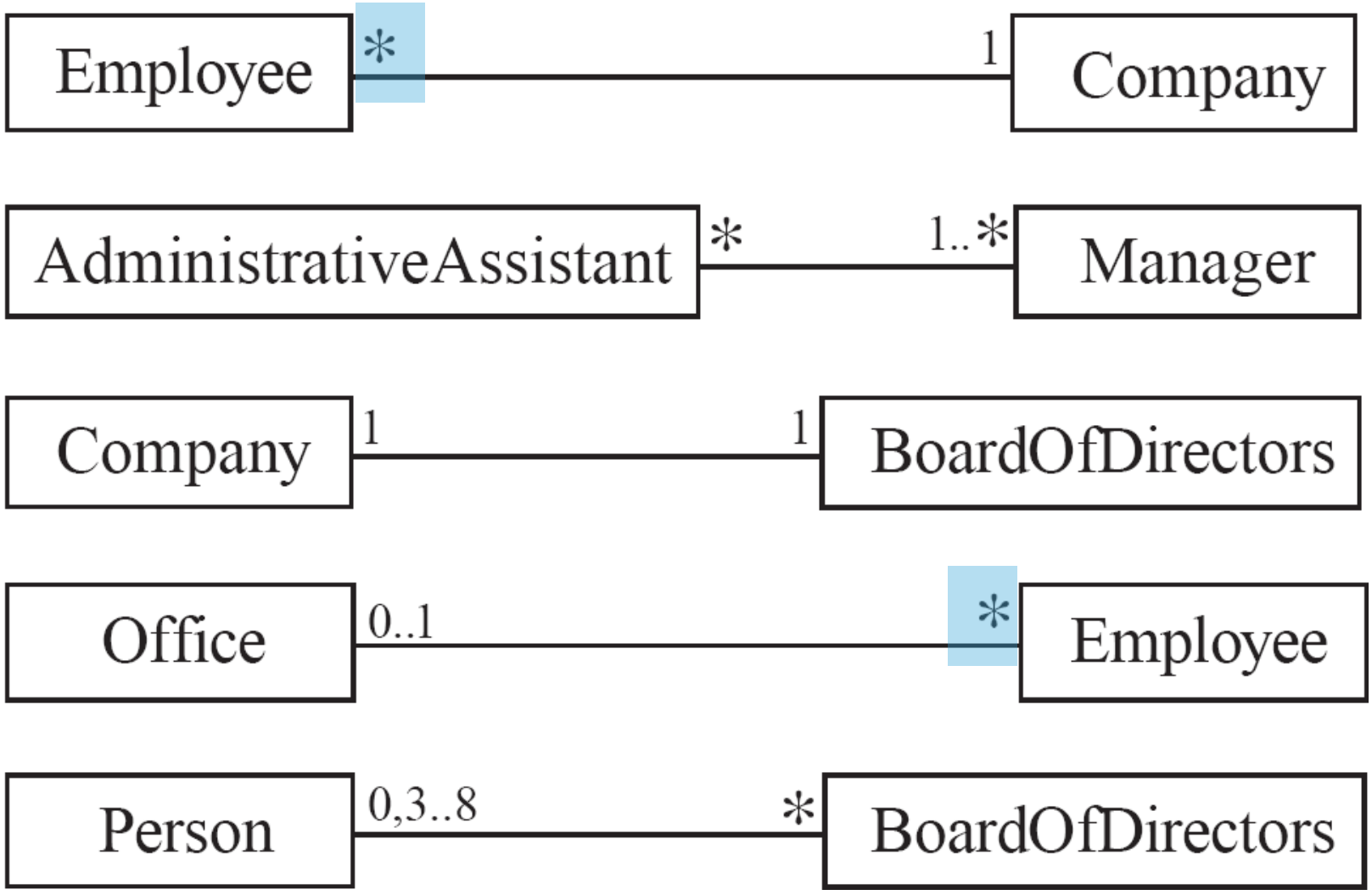

Rectangle
height
width
getArea()
resize()
Rectangle
- height:
- width:
+ getArea(): int
+ resize(int,int)


Rectangle
height
width
getArea()
resize()
Rectangle
- height:
- width:
+ getArea(): int
+ resize(int,int)

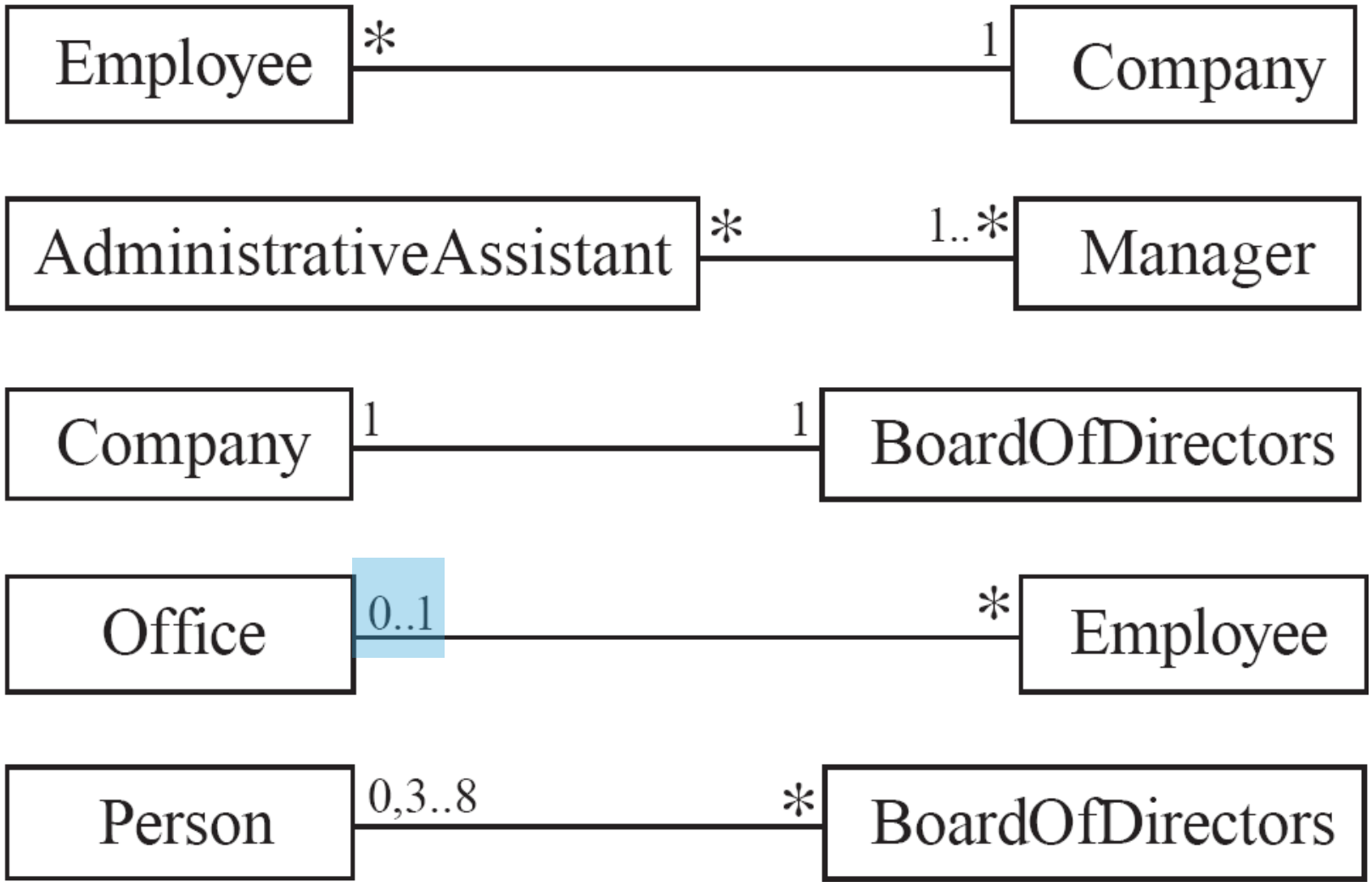
Associations and Multiplicity



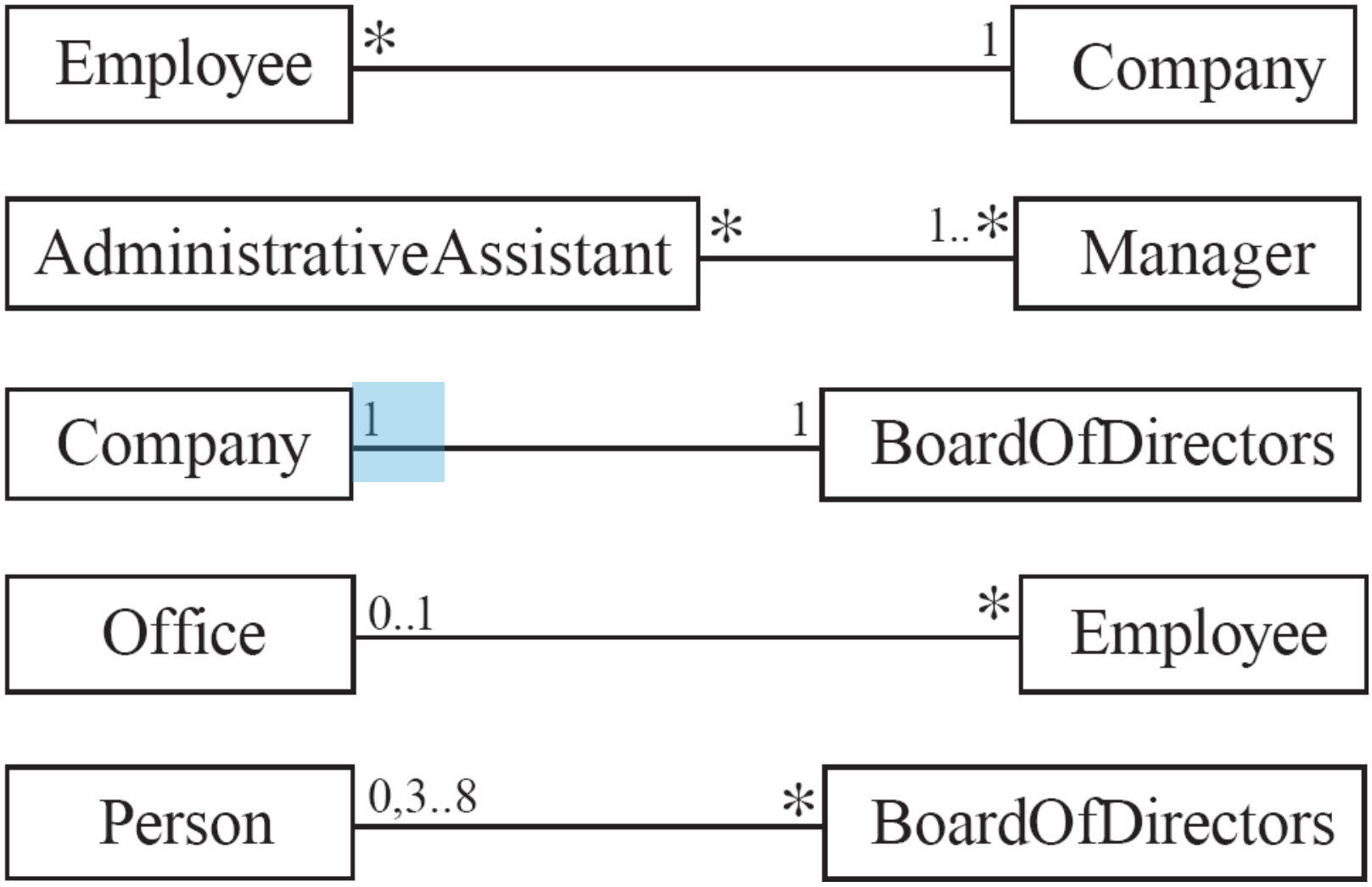
ASSOCIATIONS AND MULTIPLICITY



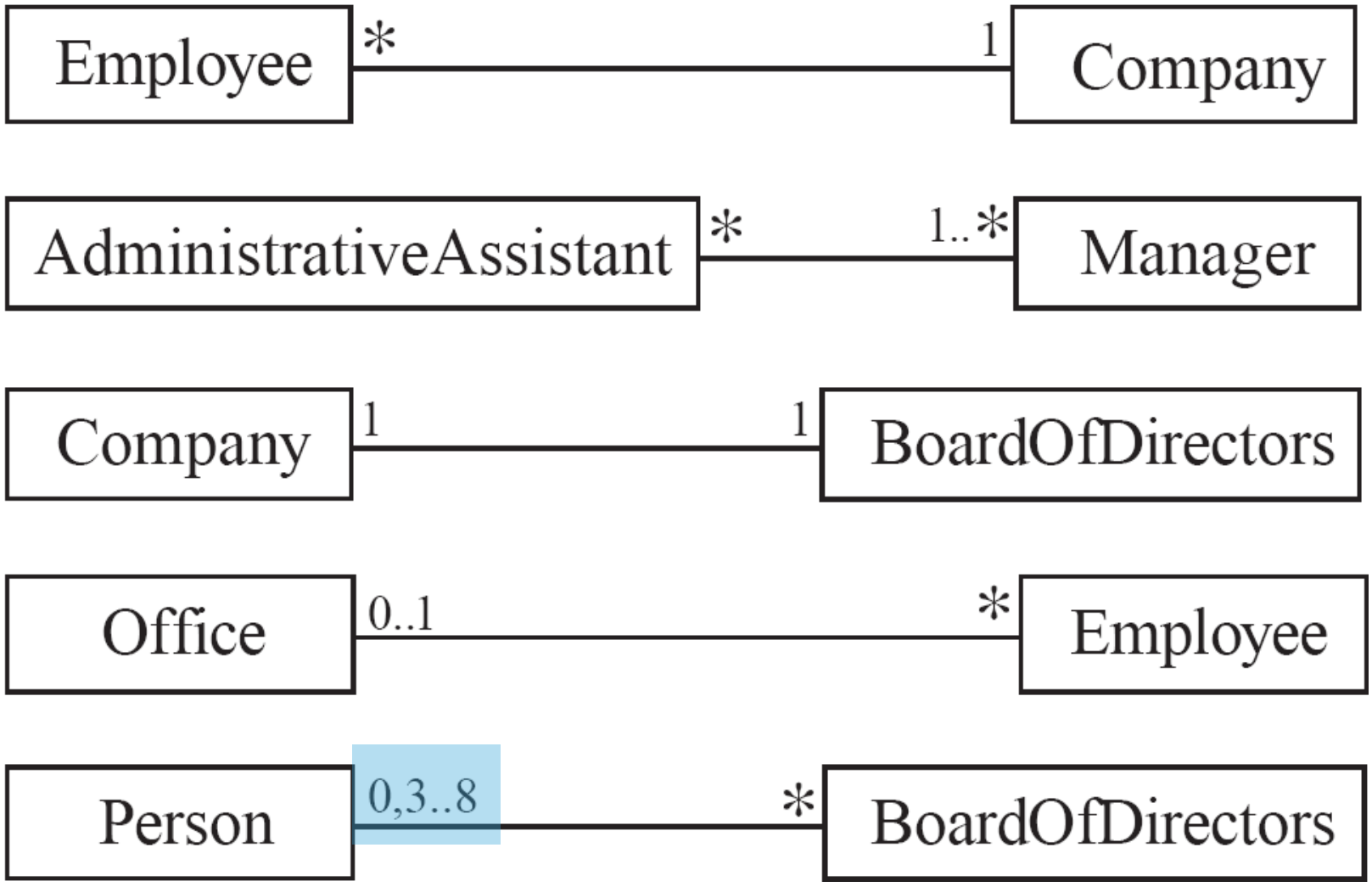
Associations and Multiplicity



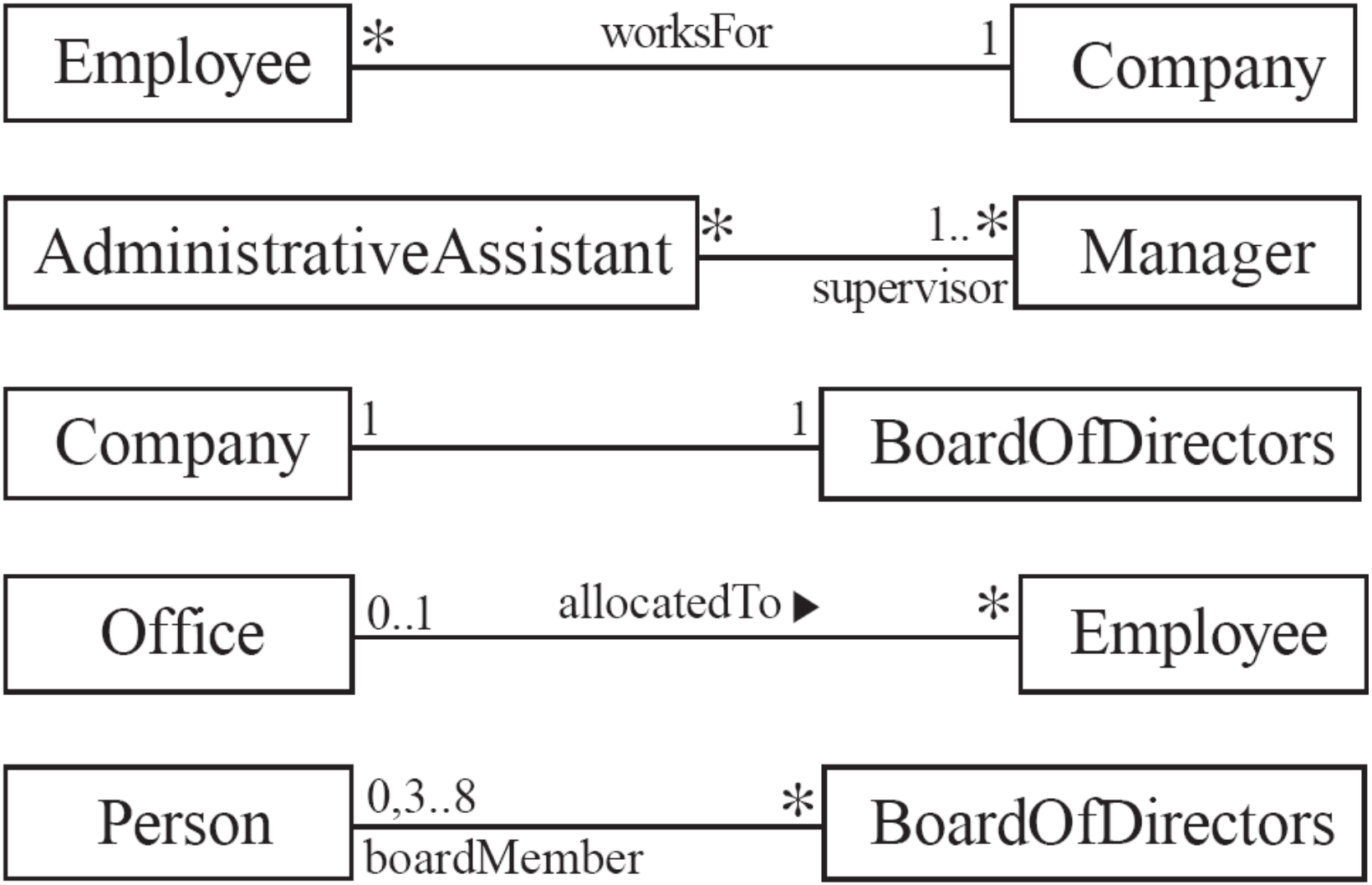
Associations and Multiplicity



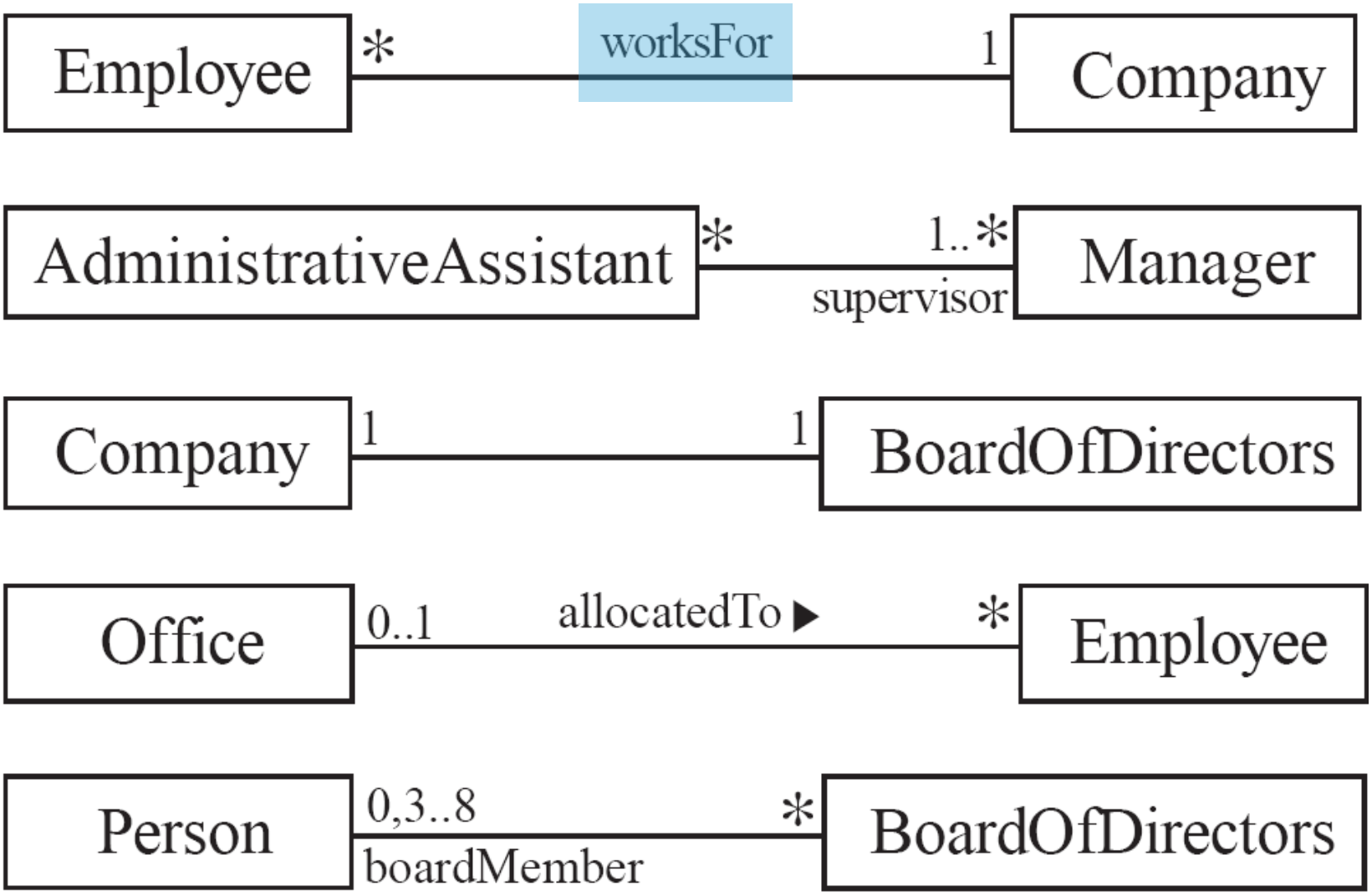
Associations and Multiplicity



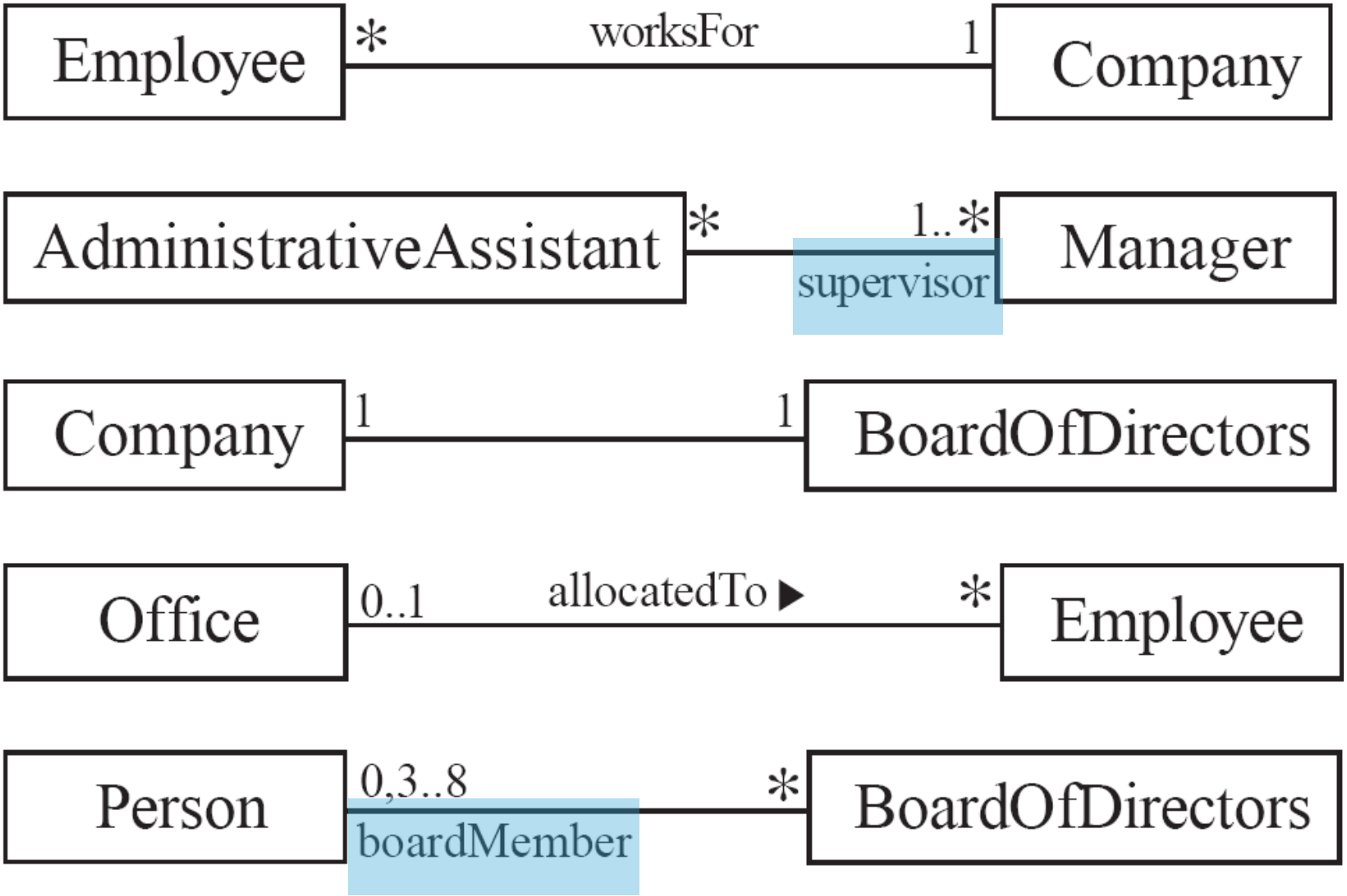
LABELLING ASSOCIATIONS



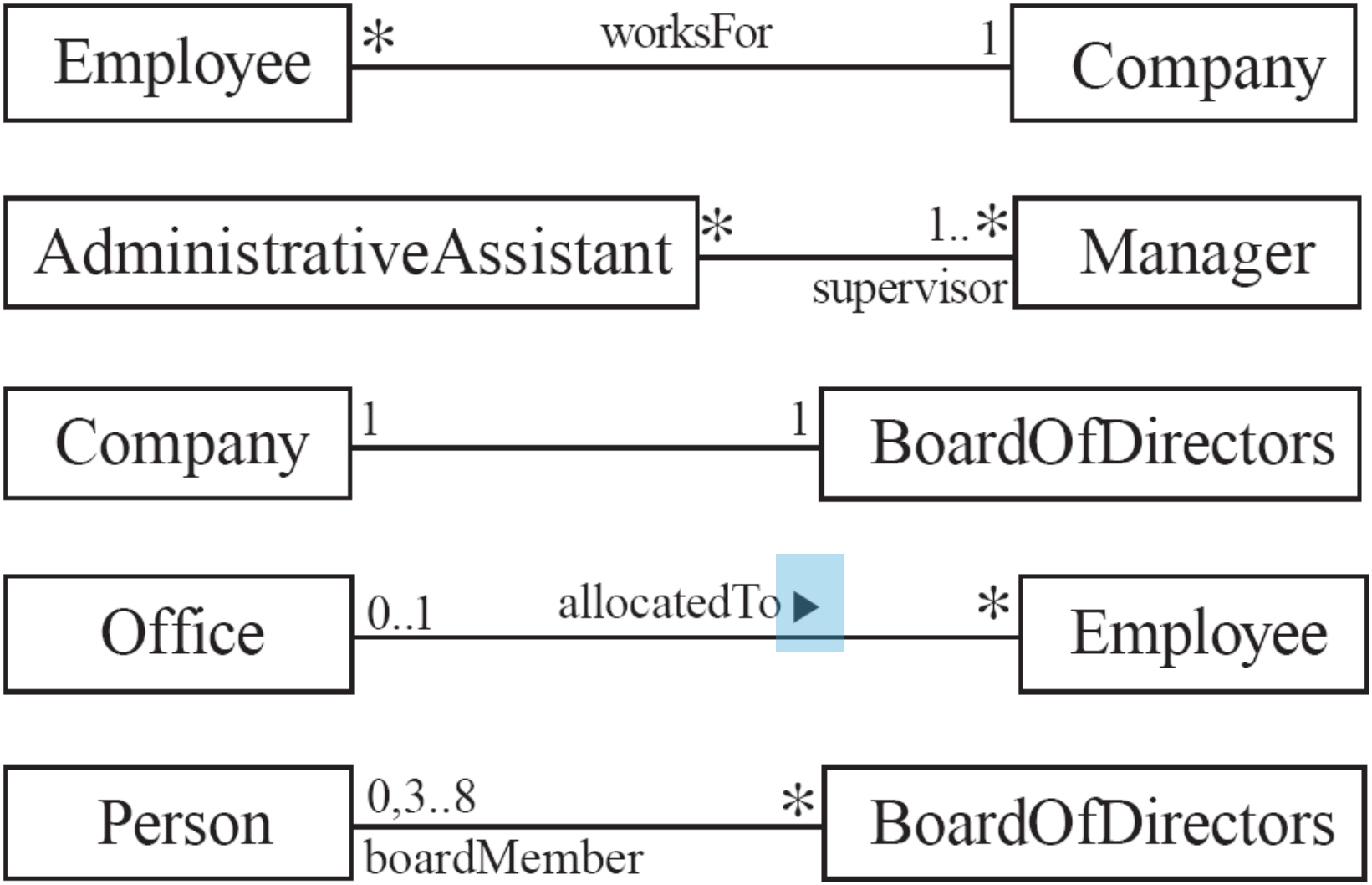
LABELLING ASSOCIATIONS



LABELLING ASSOCIATIONS



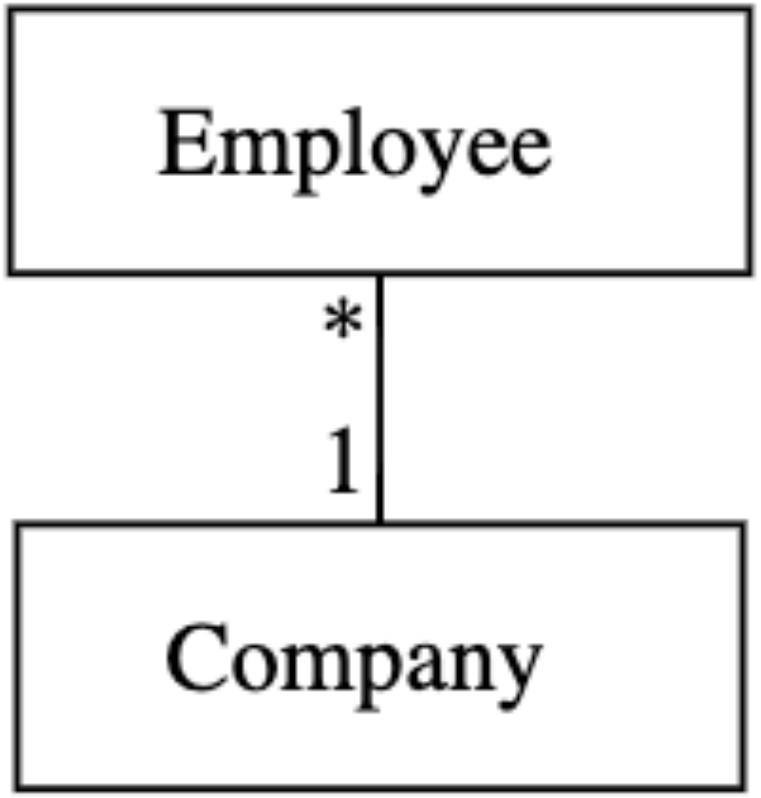
LABELLING ASSOCIATIONS



ANALYZING AND VALIDATING

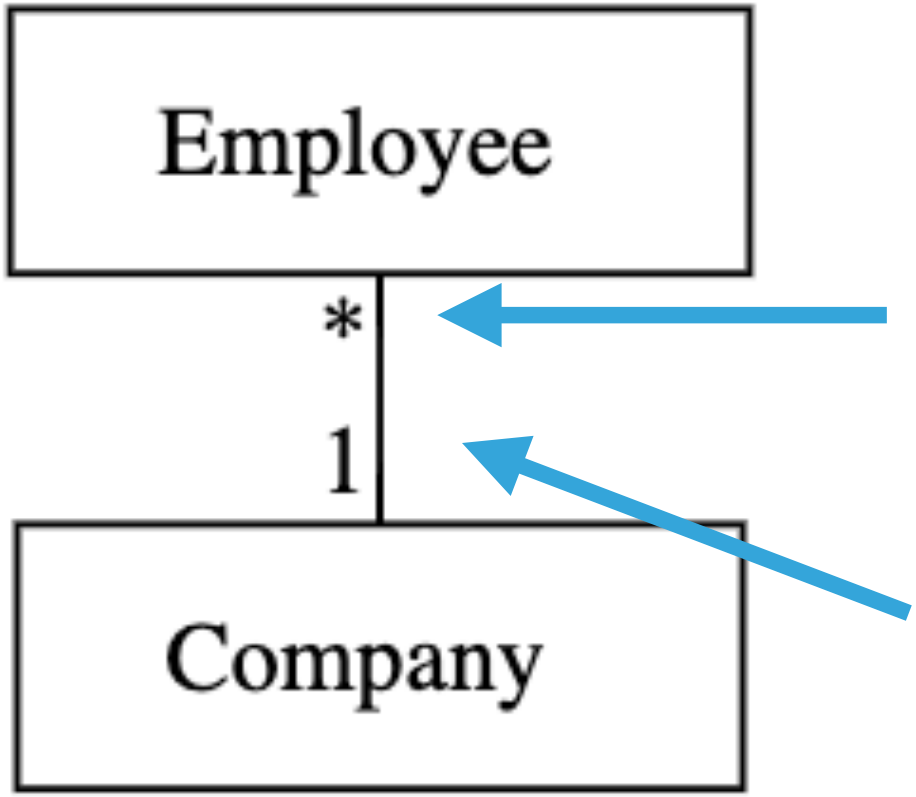
ASSOCIATIONS

Many-to-one




```
class Company {
   1 -- * Employee;
}

class Employee {}
```

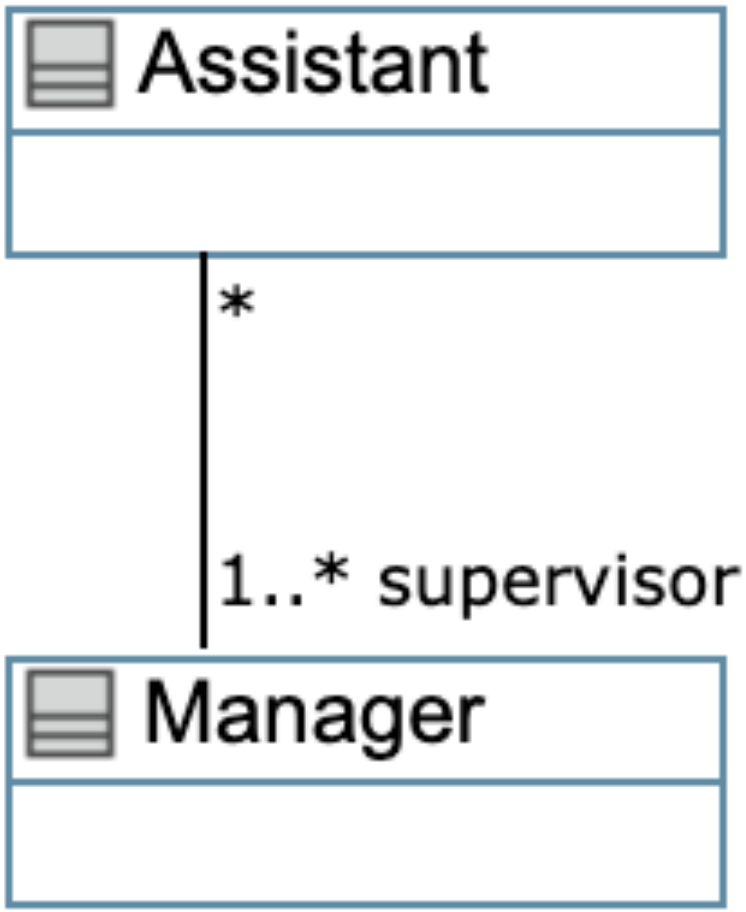
Modeling a system, not reality



Shell companies might not have any employees

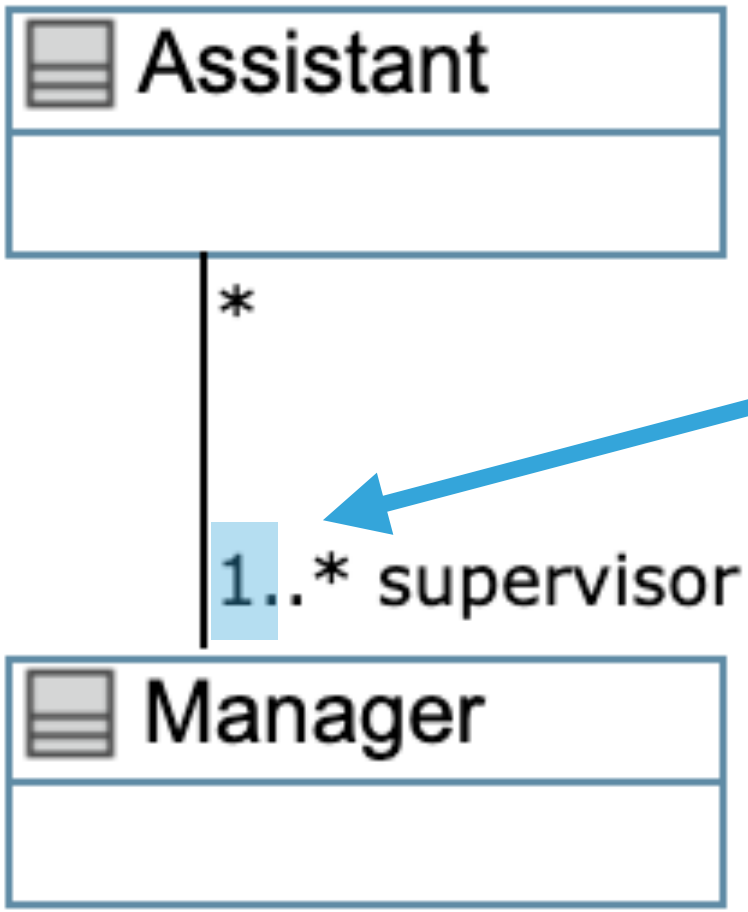
In this system, we do not care about moonlighting.

Many-to-Many




```
class Assistant {

   * -- 1..* Manager
                supervisor;

}

class Manager {}
```

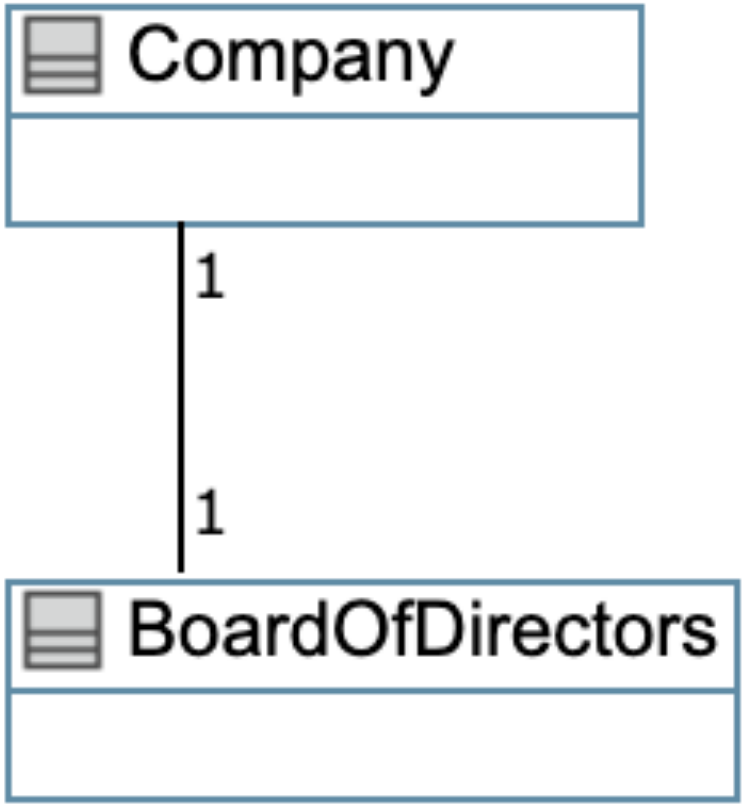
Many-to-Many



Mandatory relationships might represent reality, but can cause complications with your models.

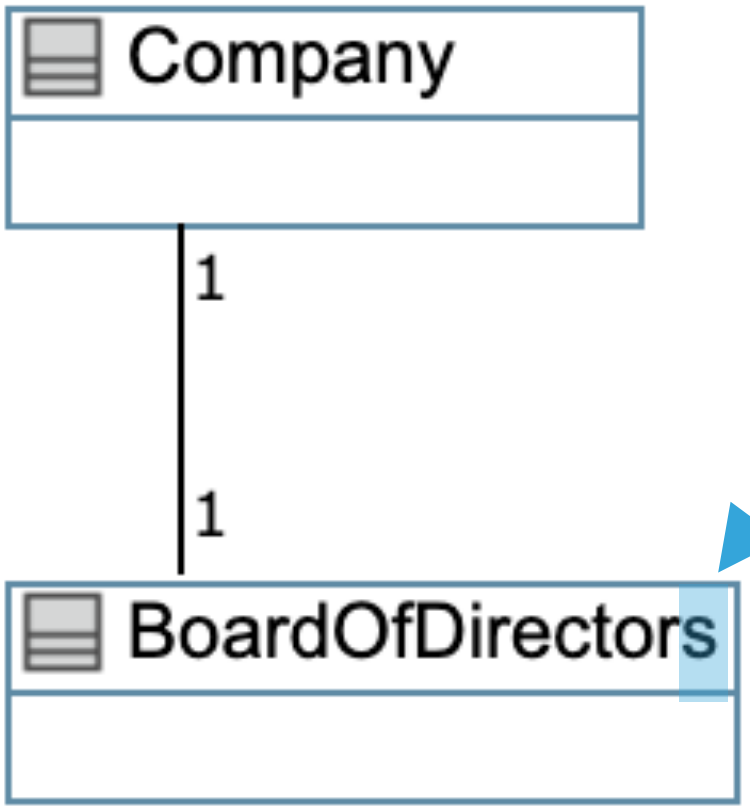
Shouldn't != Can't

One-to-One



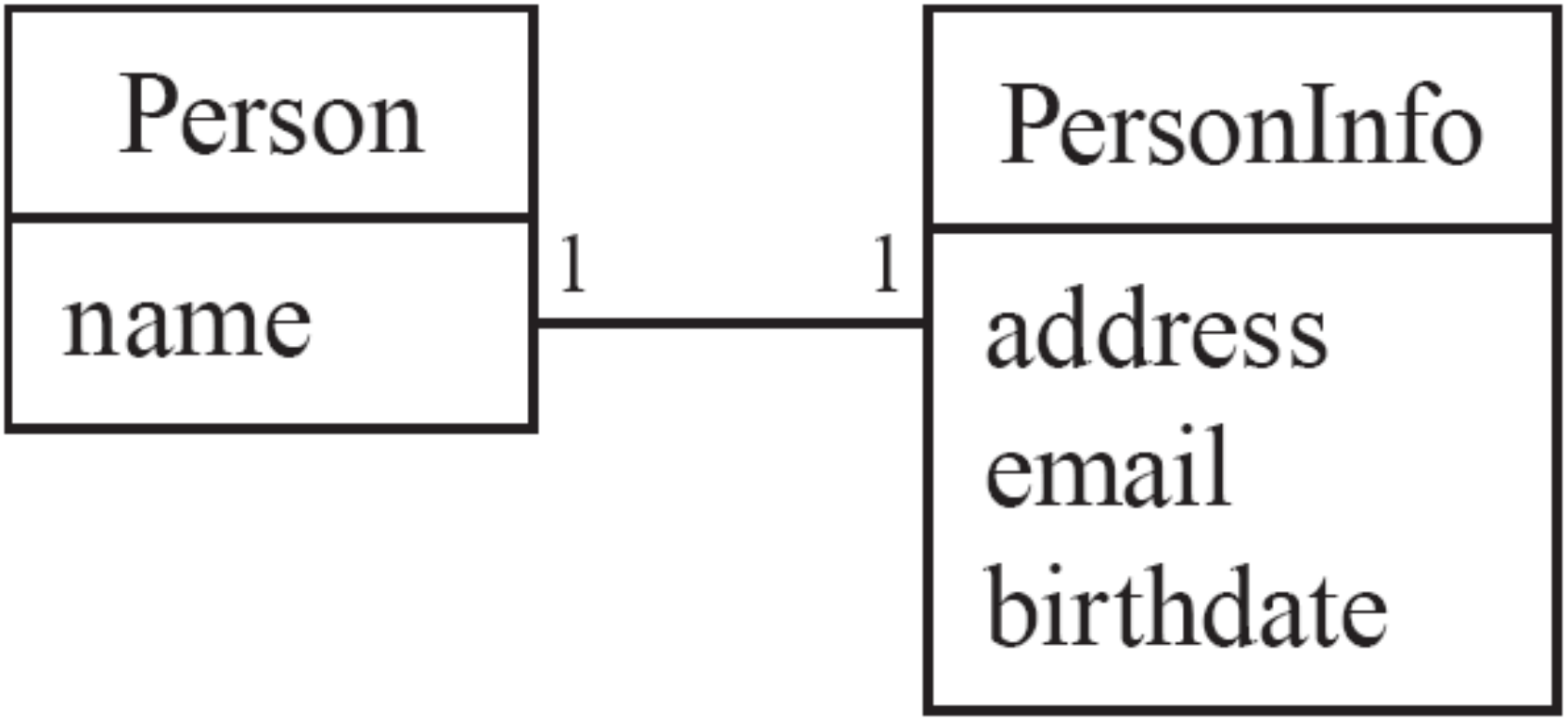

```
class Company {
   1 -- 1 BoardOfDirectors;
}

class BoardOfDirectors {}
```

One-to-One

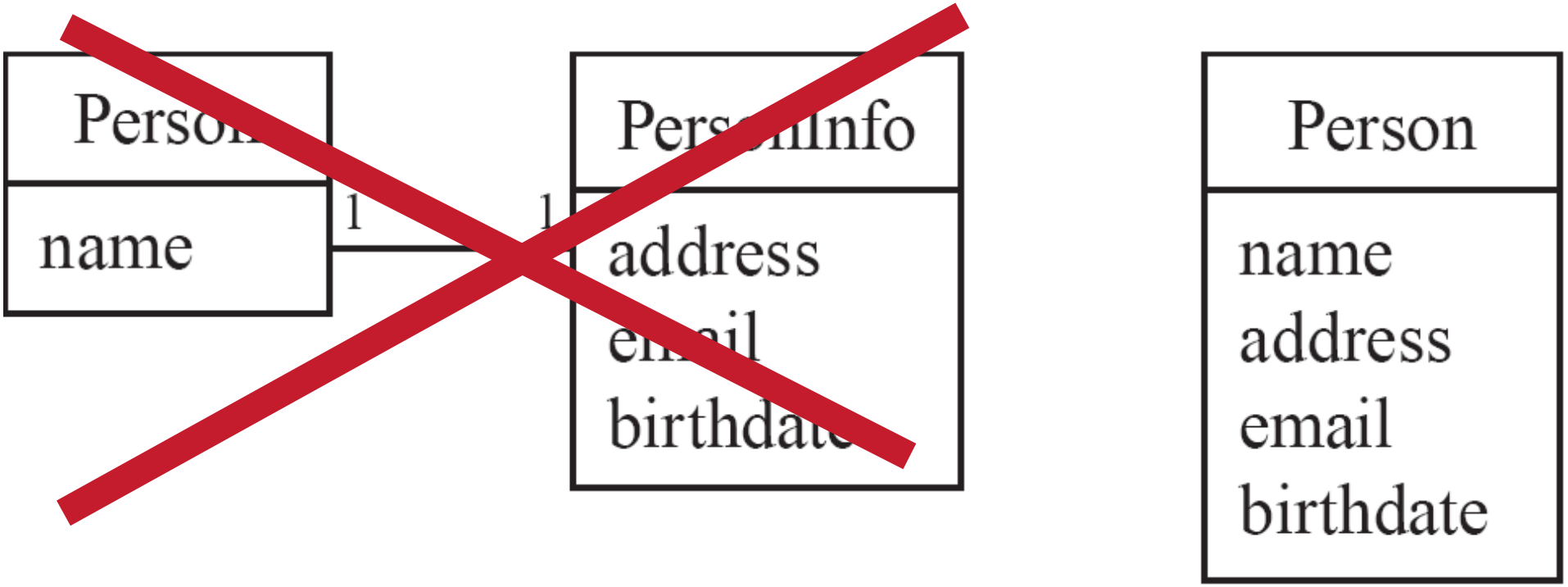


Here the plural makes sense, as it's a "board" (one) but called *the* board of directors.

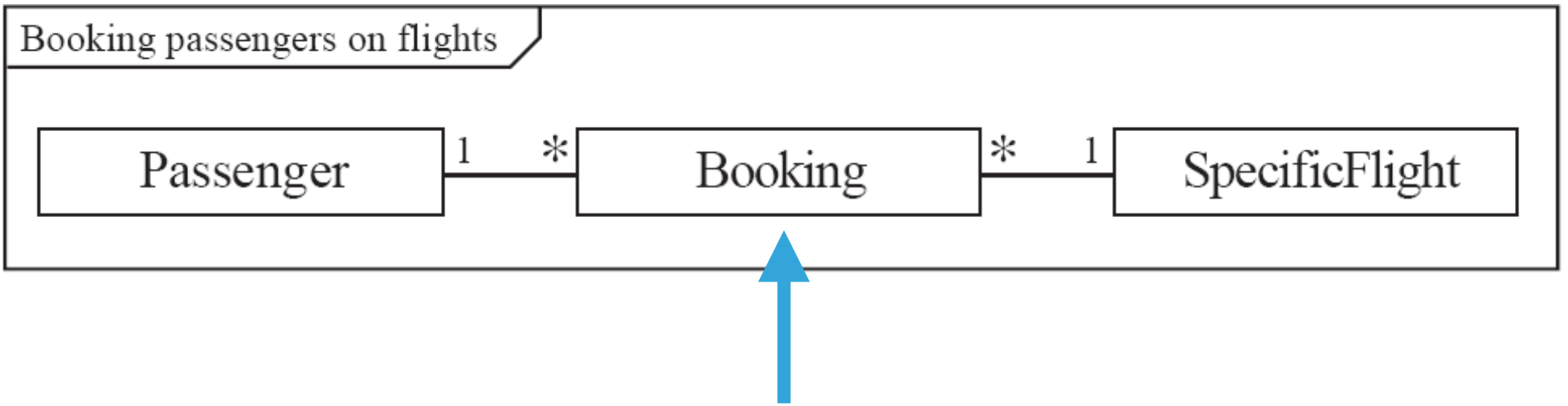
Avoid unnecessary one-to-one associations



Instead simplify

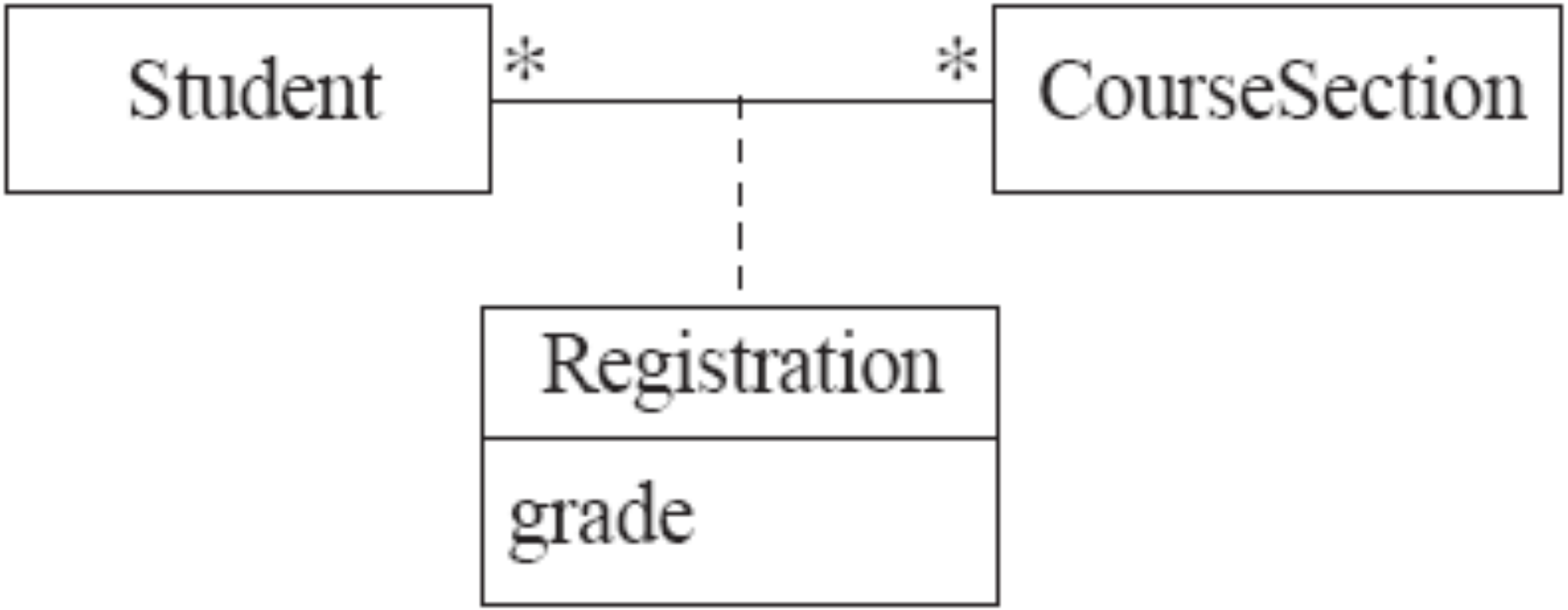


Optional Frame part of UML 2.x

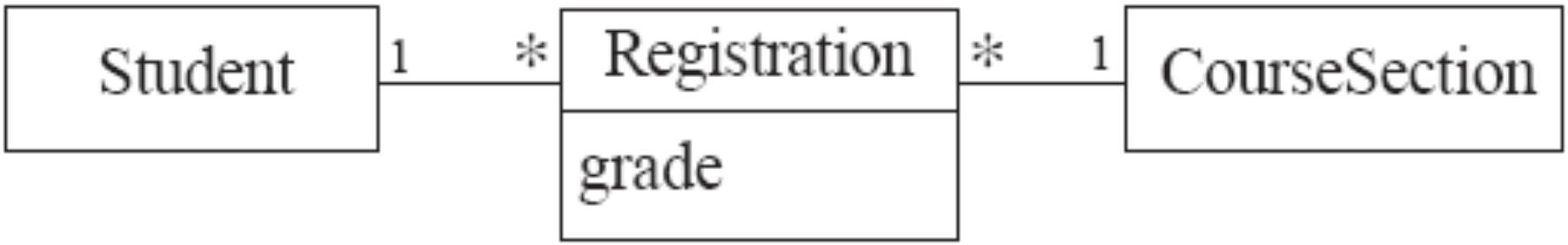


A Booking unites one passenger with one flight.

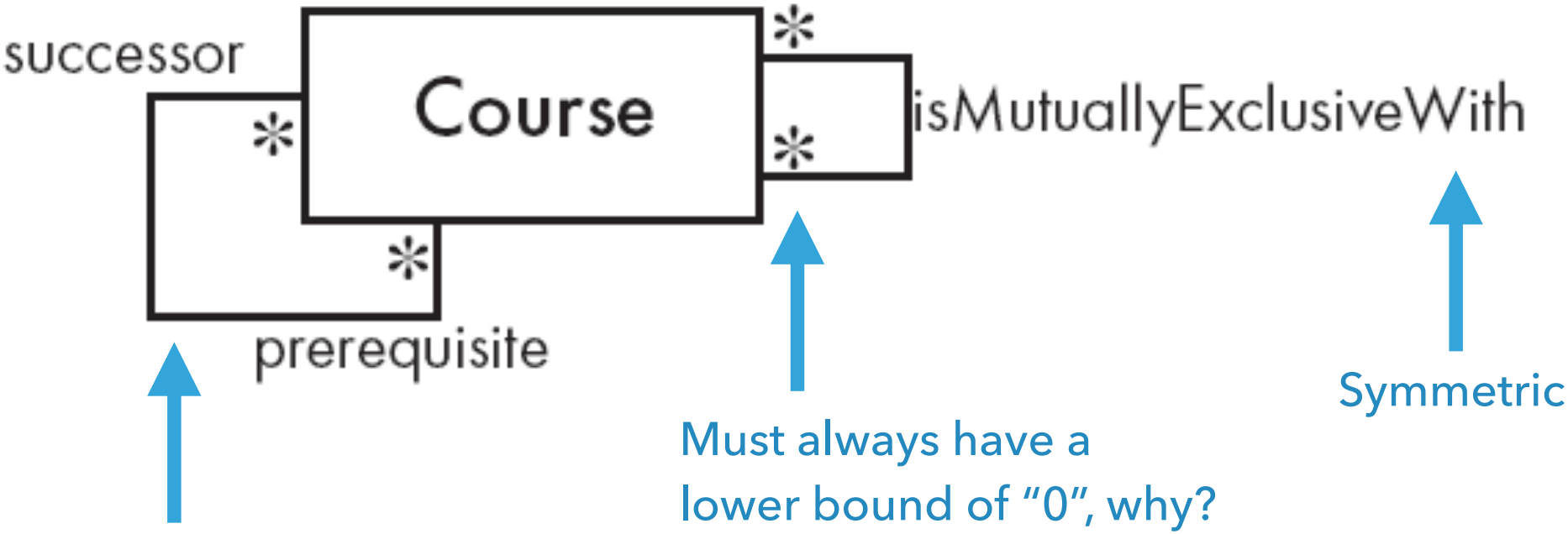
Association classes



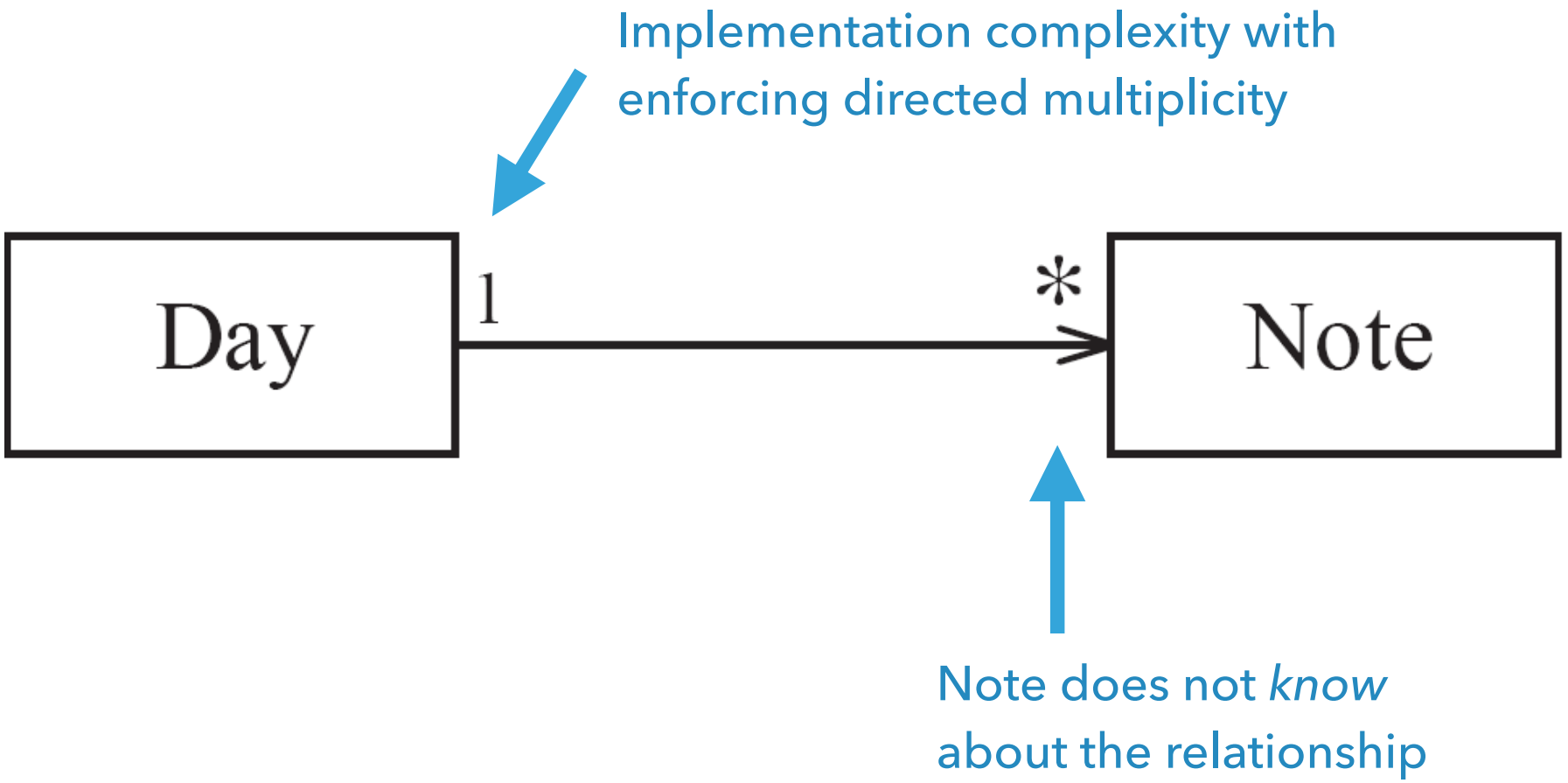
Association classes Equivalent to . . .



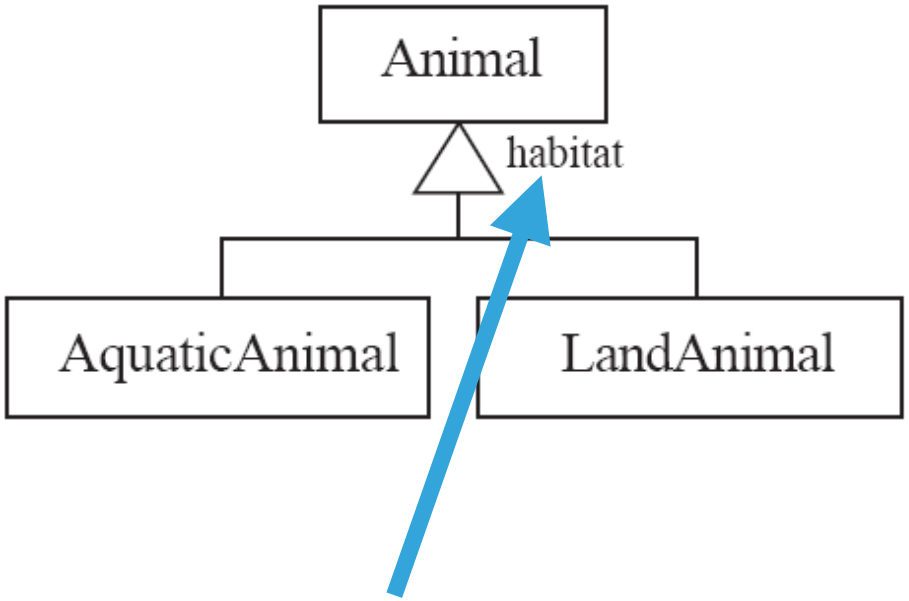
REFLEXIVE ASSOCIATIONS



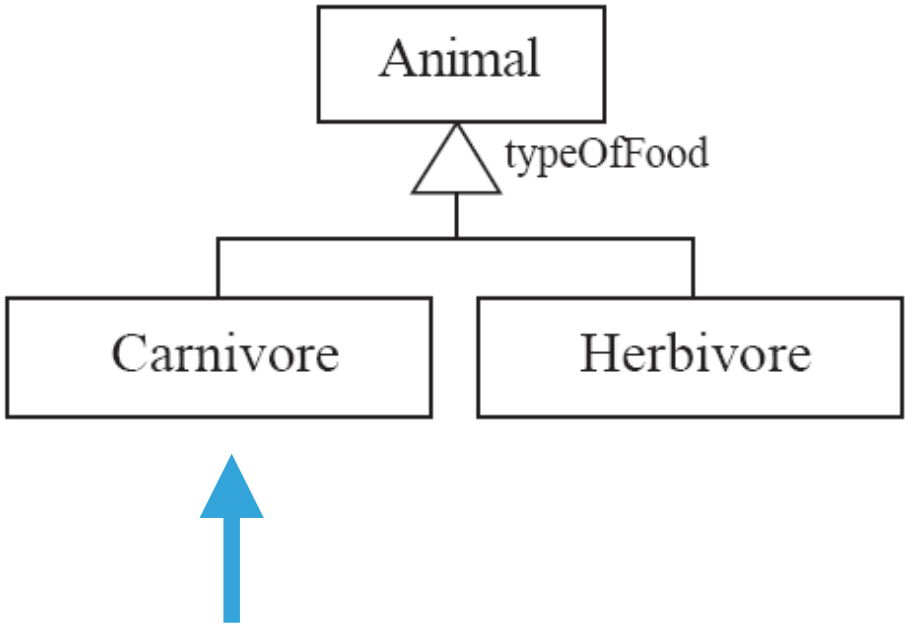
DIRECTIONALITY IN ASSOCIATIONS



GENERALIZATION

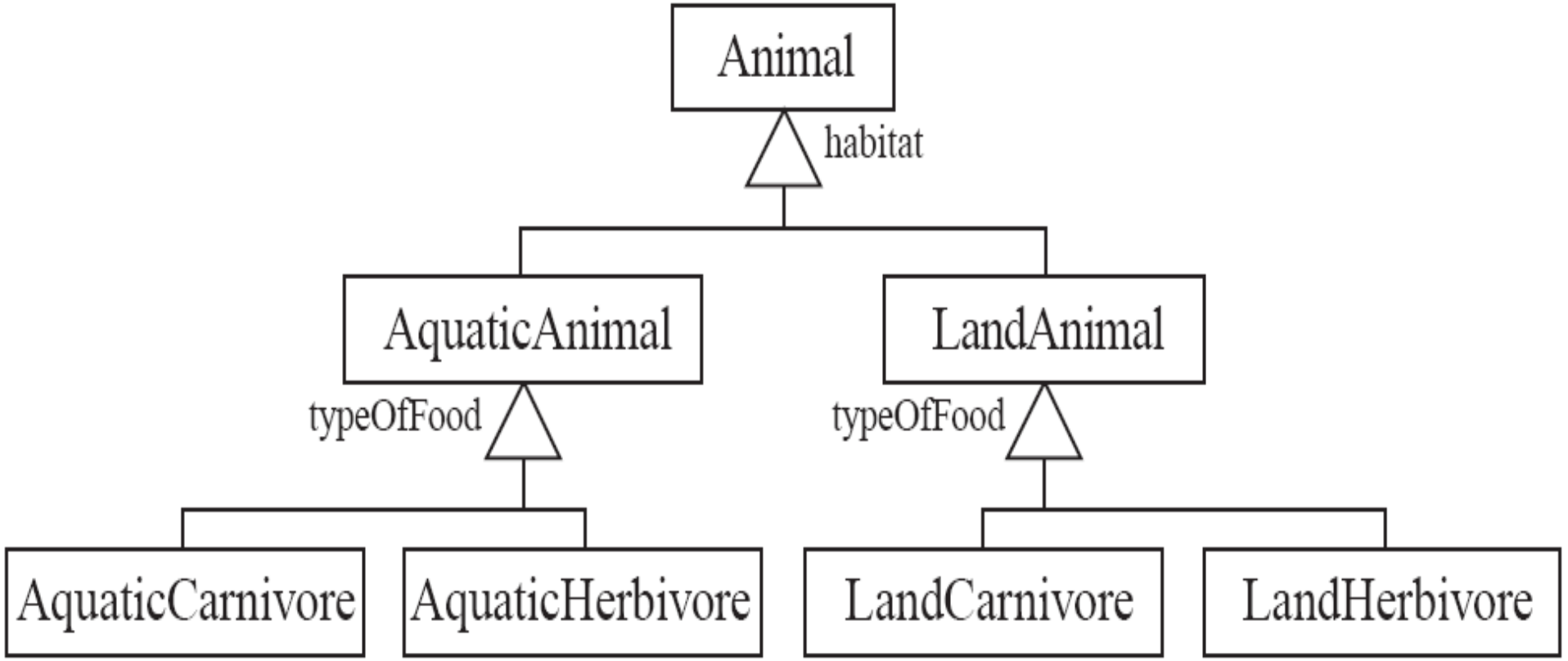


Label (or discriminator) describes the criteria

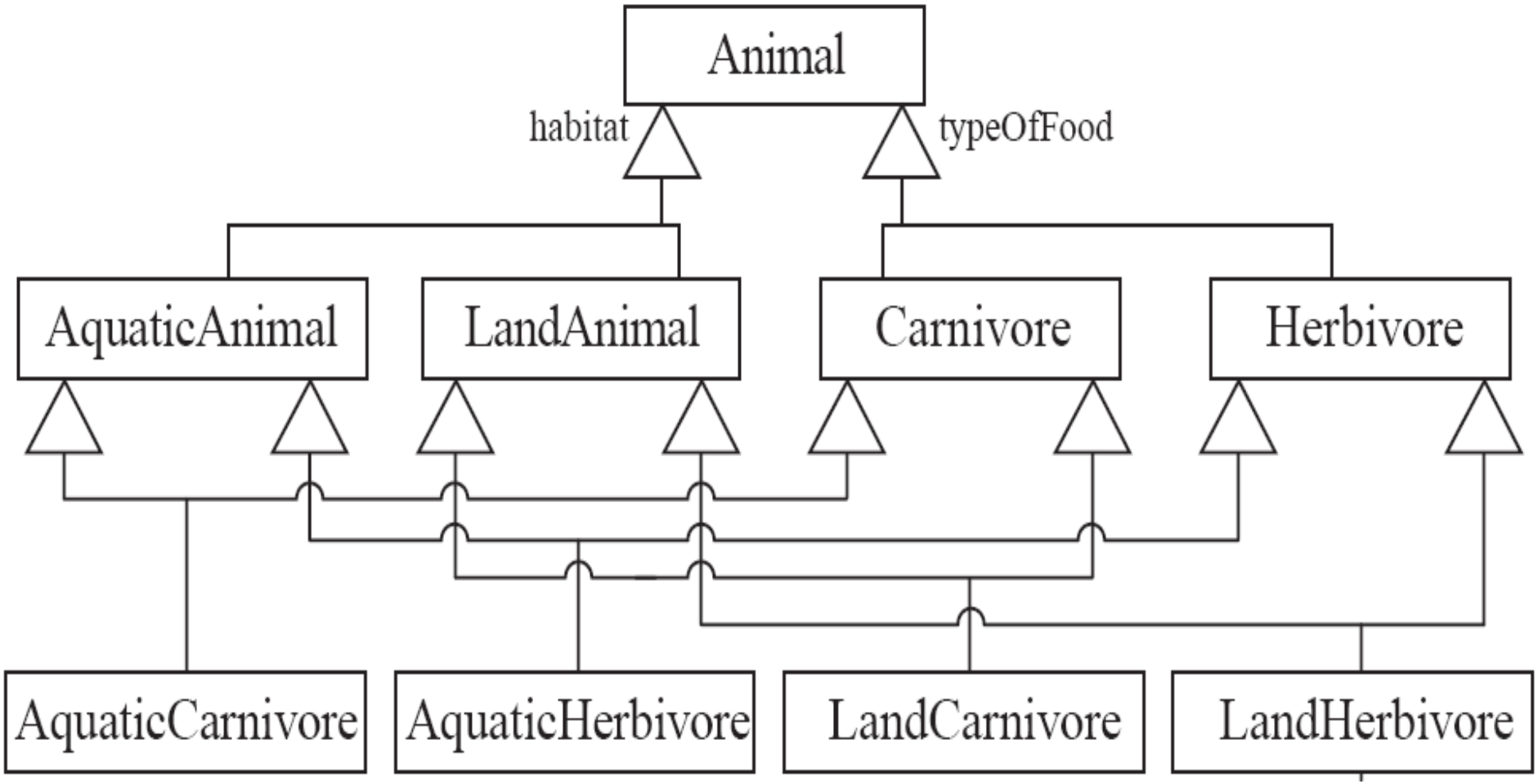


Many ways to generalize

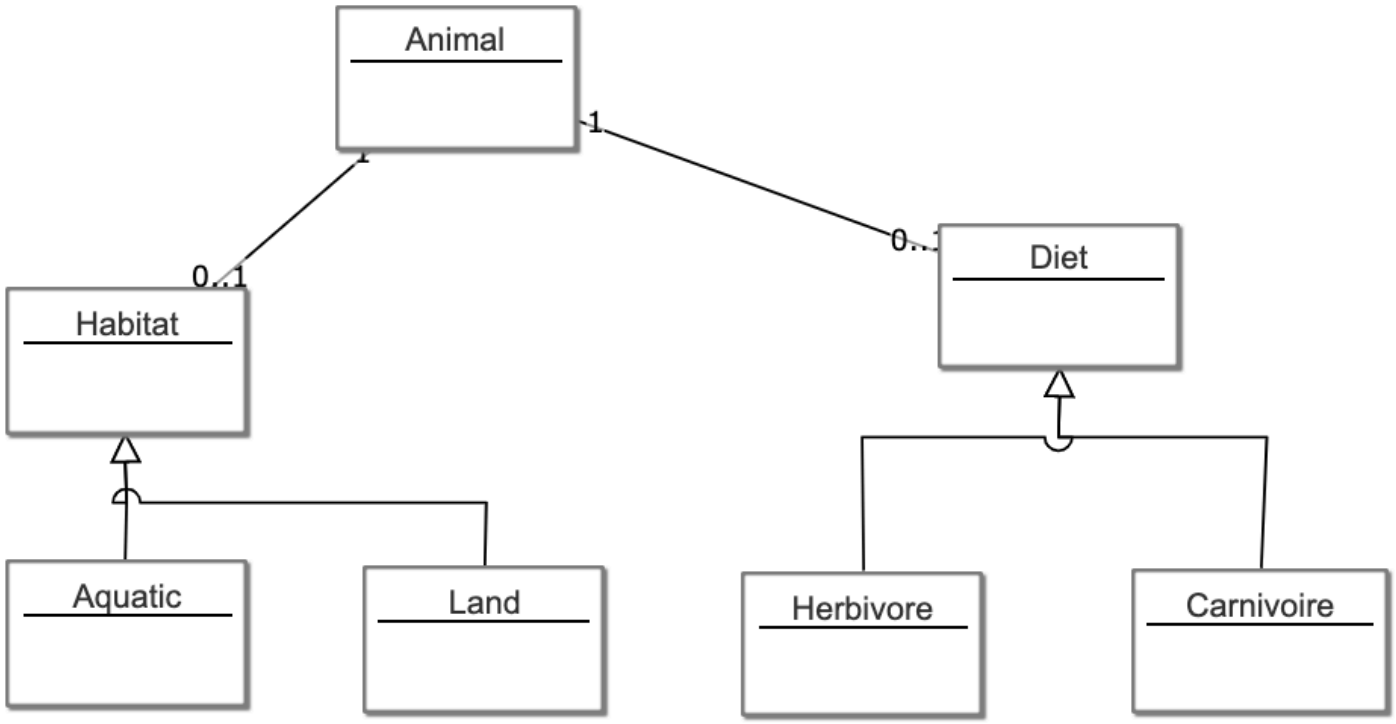
Handling multiple discriminators



Using multiple inheritance



MANY WAYS TO MODEL RELATIONSHIPS



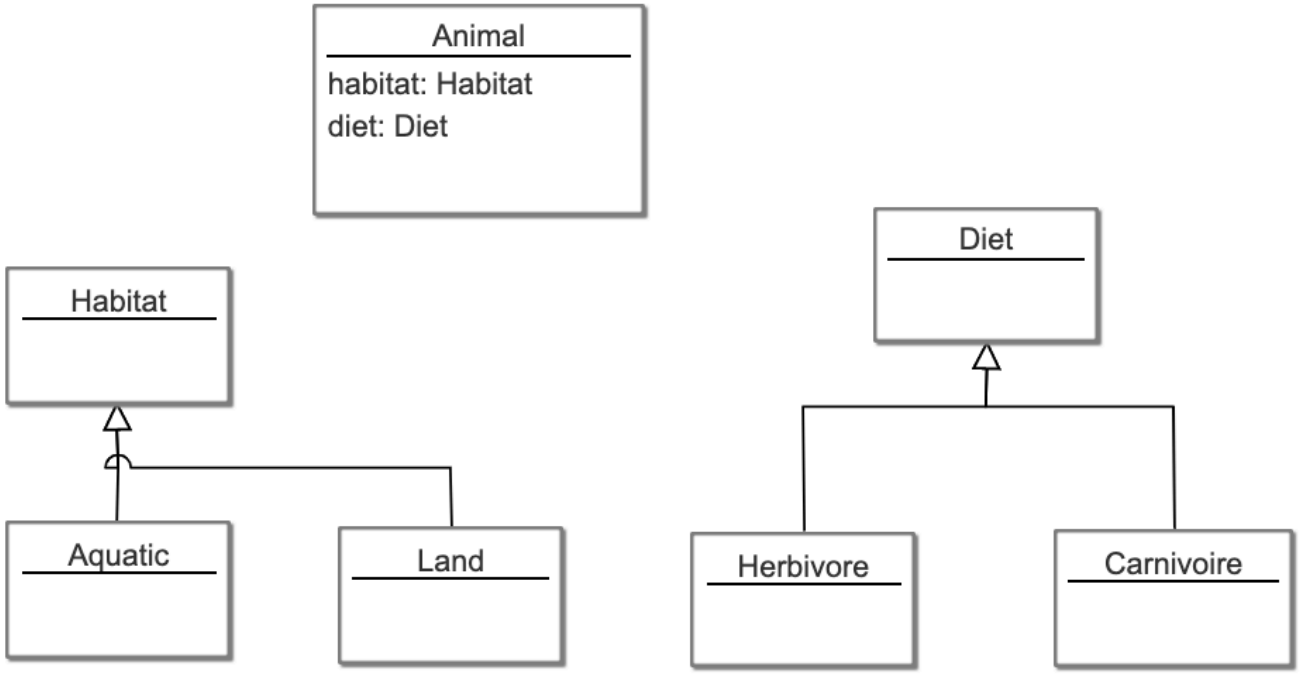

```
class Animal {
  1 -- 0..1 Habitat;
  1 -- 0..1 Diet;
}
```

```
class Habitat {}
class Diet {}
```

```
class Aquatic {
  isA Habitat;
}
class Land {
  isA Habitat;
}
```

```
class Carnivoire {
  isA Diet;
}
class Herbivore {
  isA Diet;
}
```

SOME ASSOCIATIONS CAN BE MODELED AS ATTRIBUTES



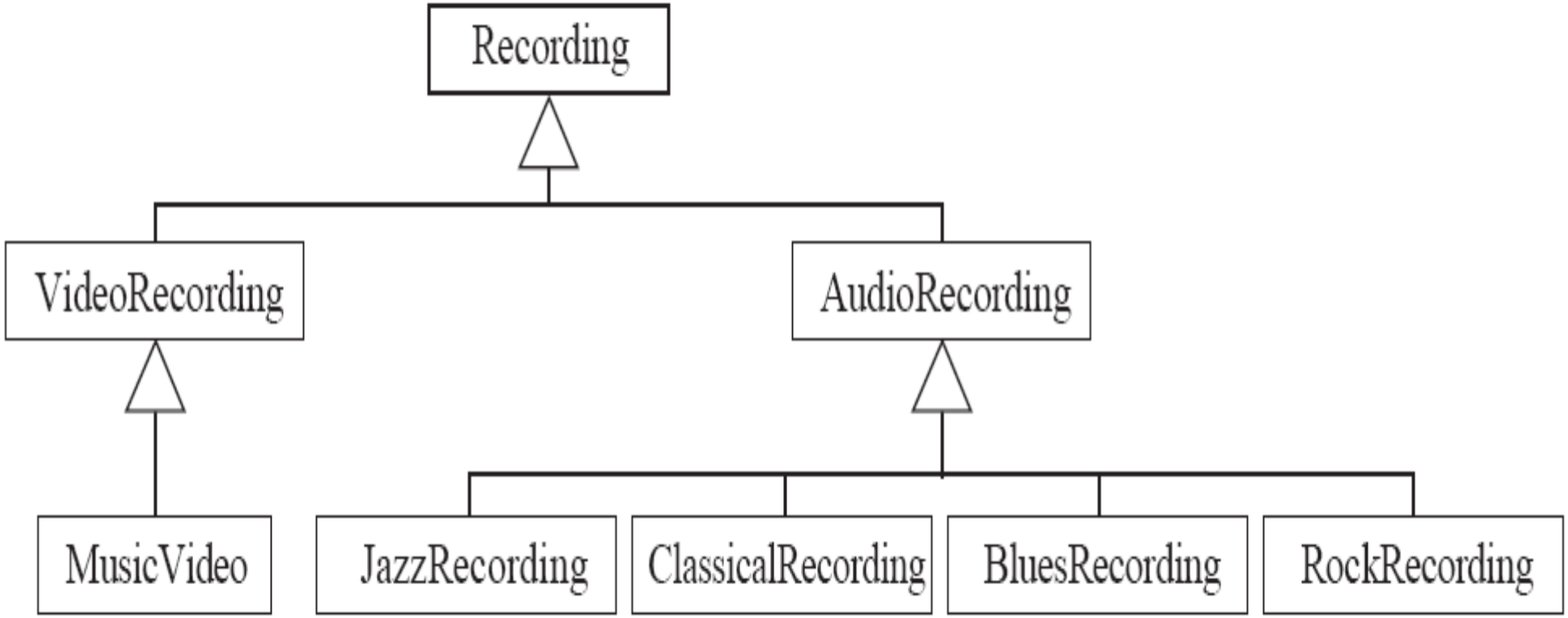

```
class Animal {
  Habitat habitat;
  Diet diet;
}
```

```
class Habitat {}
class Diet {}
```

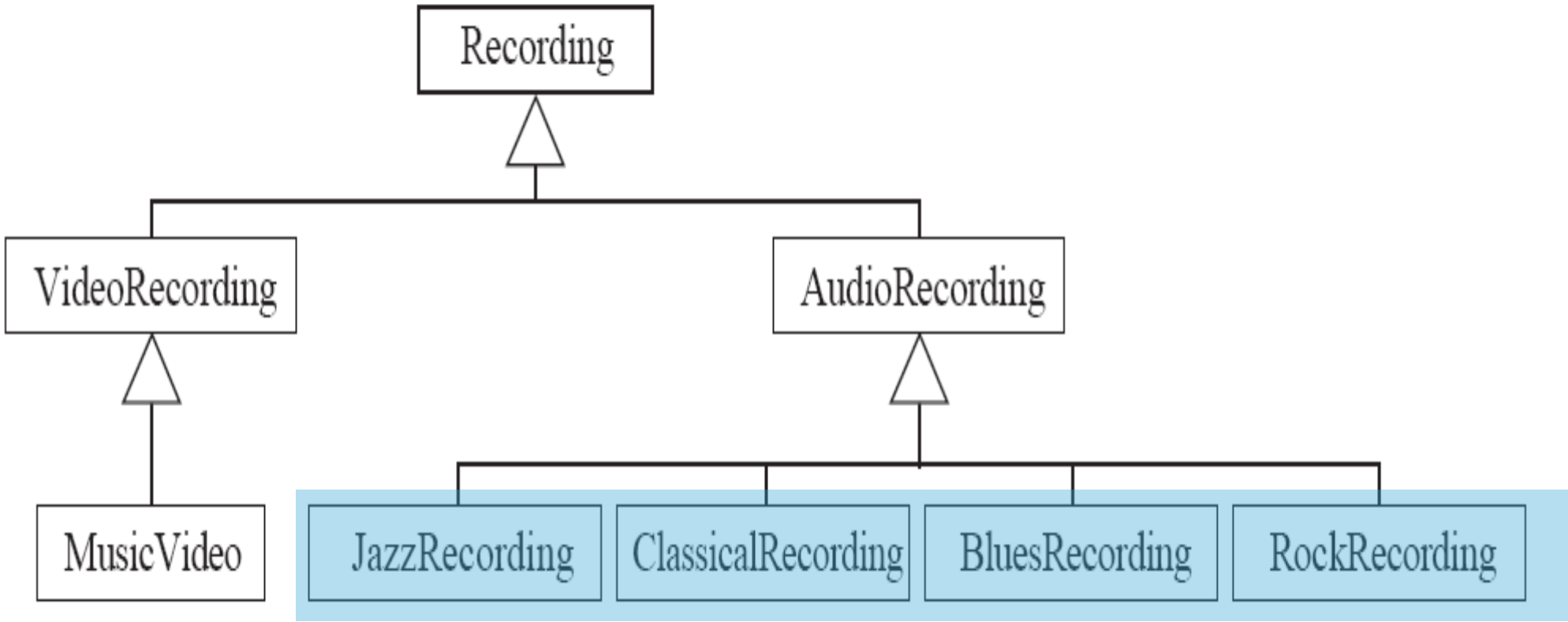
```
class Aquatic {
  isA Habitat;
}
class Land {
  isA Habitat;
}
```

```
class Carnivoire {
  isA Diet;
}
class Herbivore {
  isA Diet;
}
```

Avoiding unnecessary generalizations

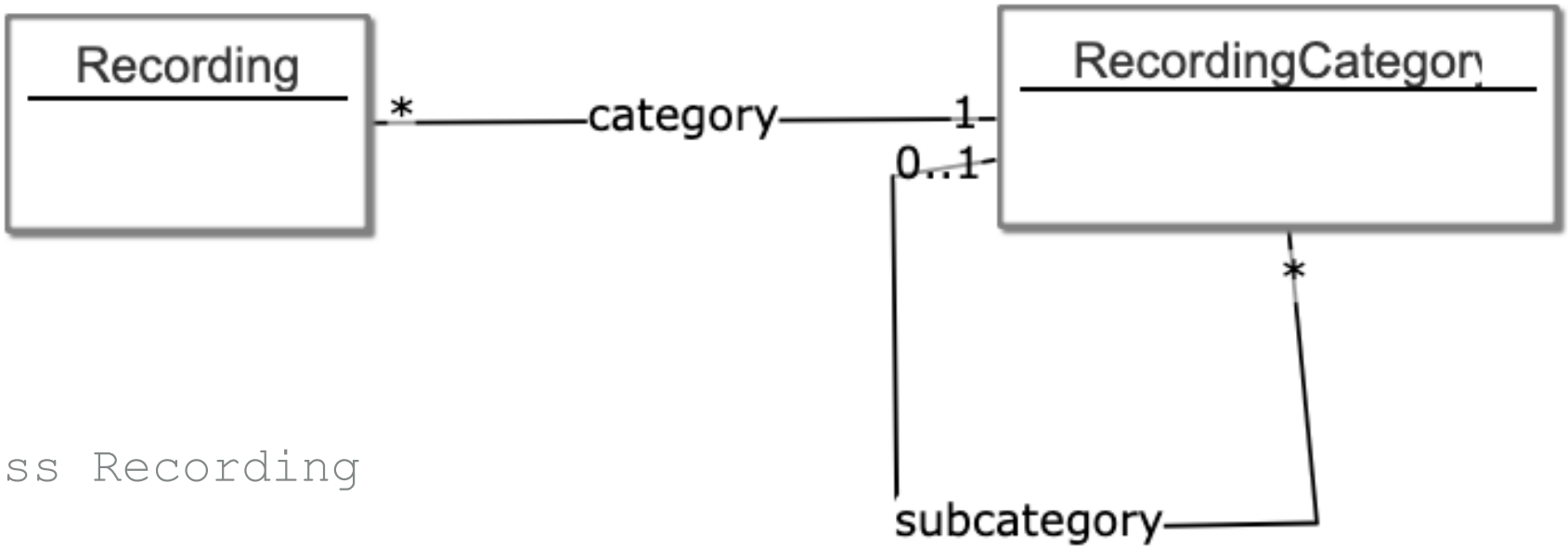


AVOIDING UNNECESSARY GENERALIZATIONS



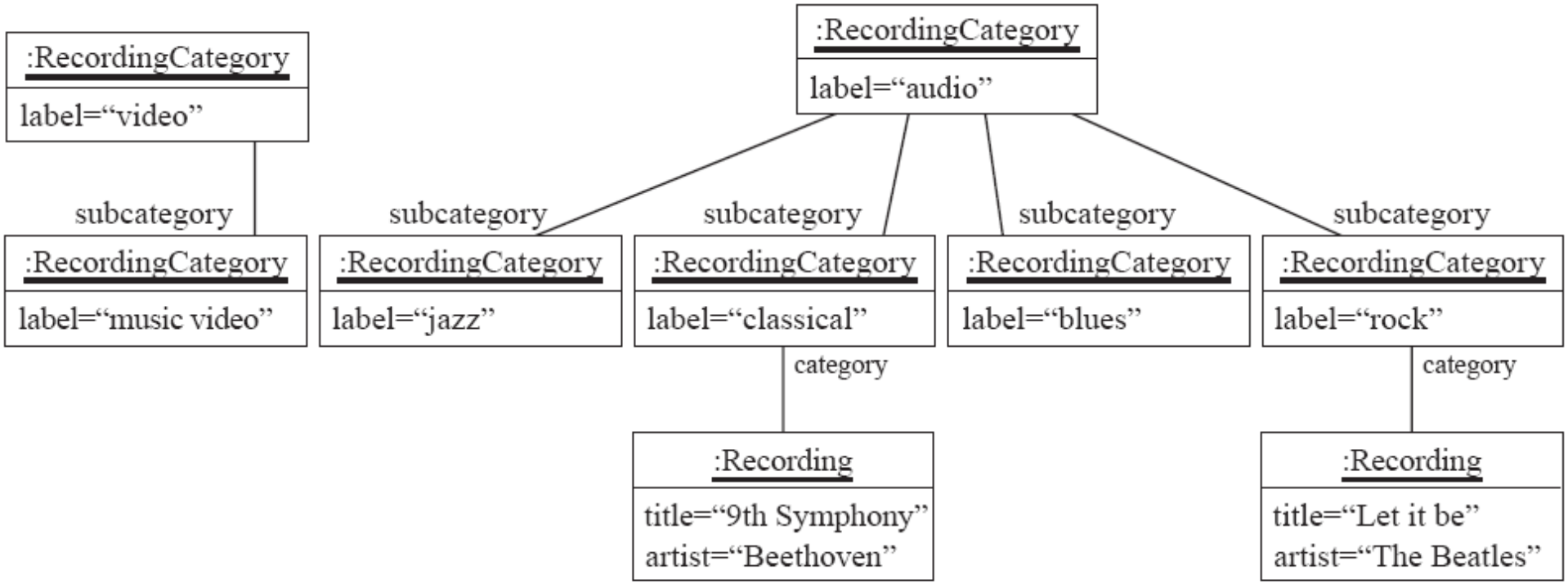
Inappropriate hierarchy of classes, which should be instances

IMPROVED DESIGN

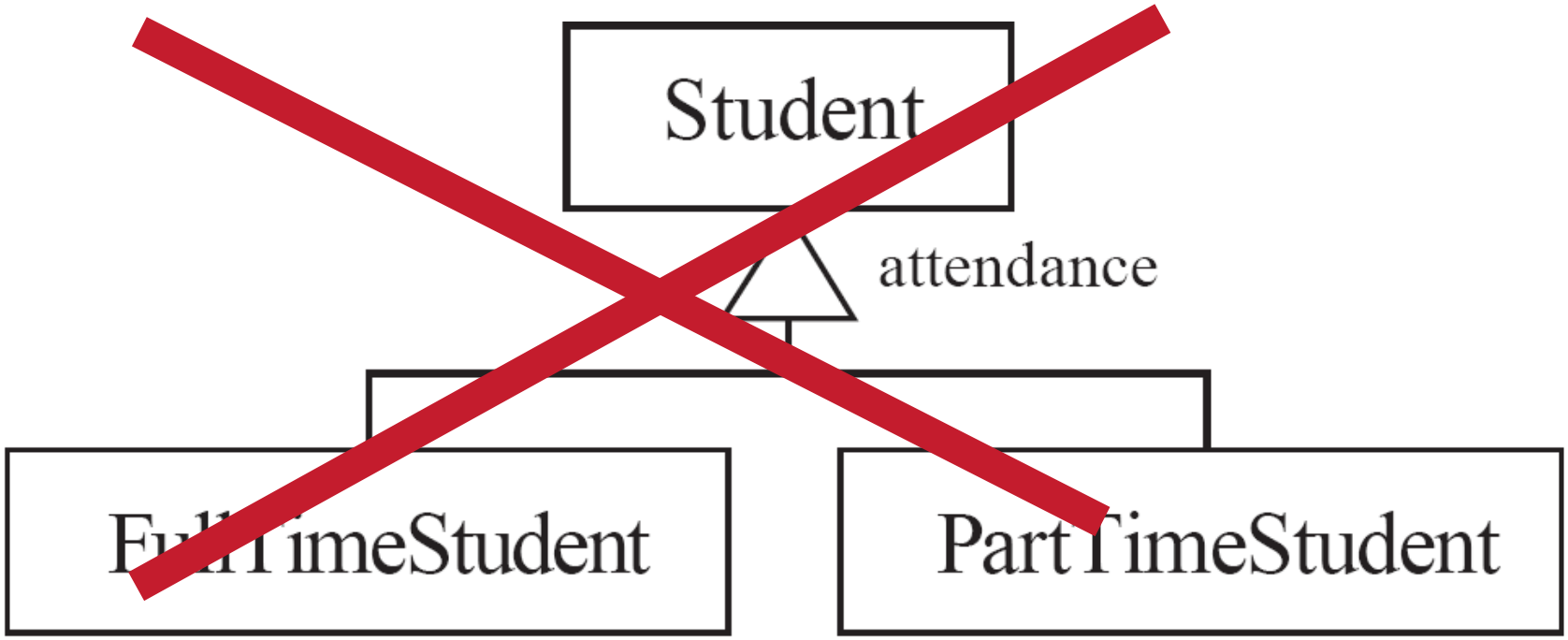



```
class Recording
{
  * -- 1 RecordingCategory category;
}

class RecordingCategory
{
  0..1 -- * RecordingCategory subcategory;
}
```

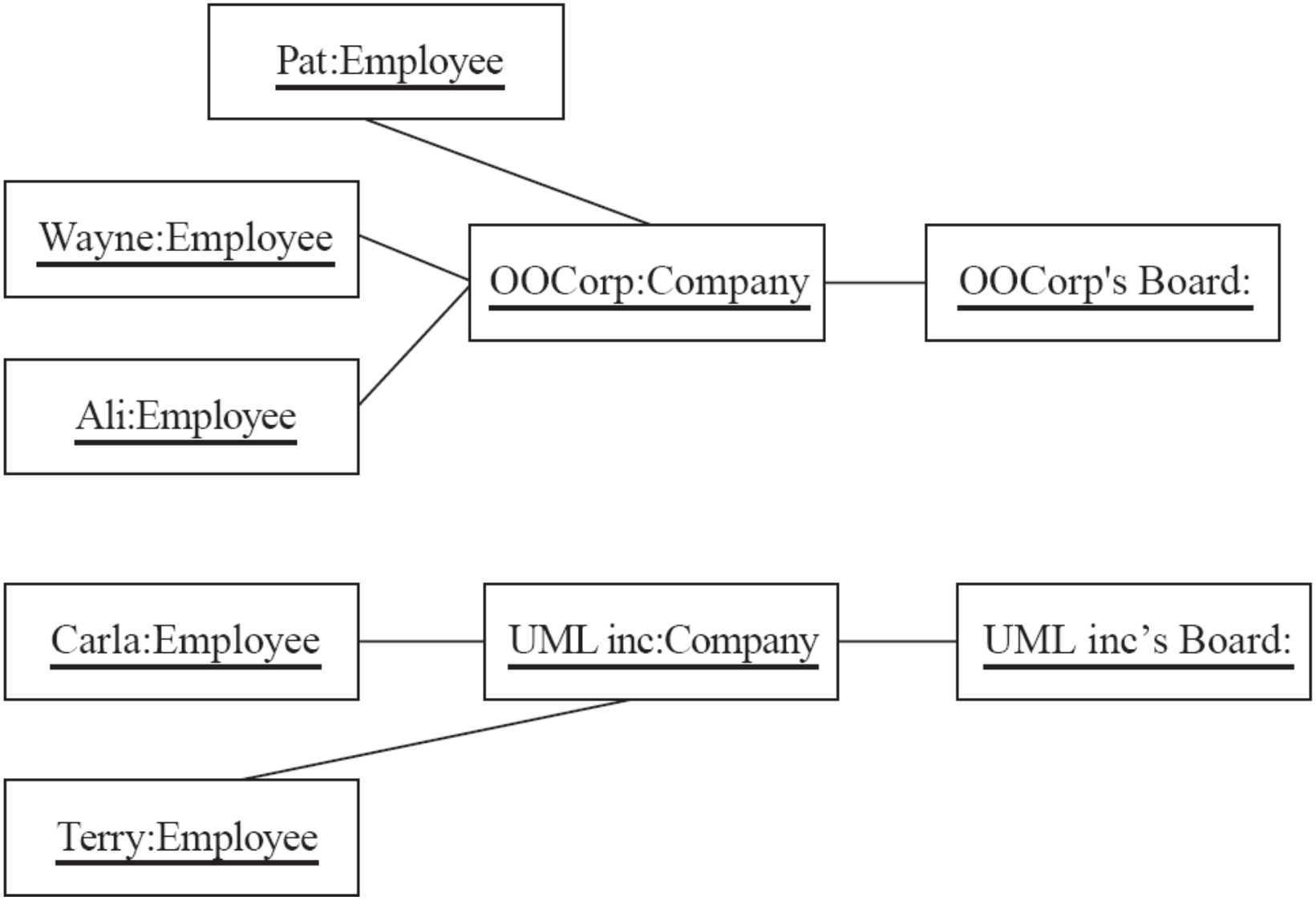
CORRESPONDING OBJECT DIAGRAM



Avoiding having instances change class



Object Diagrams

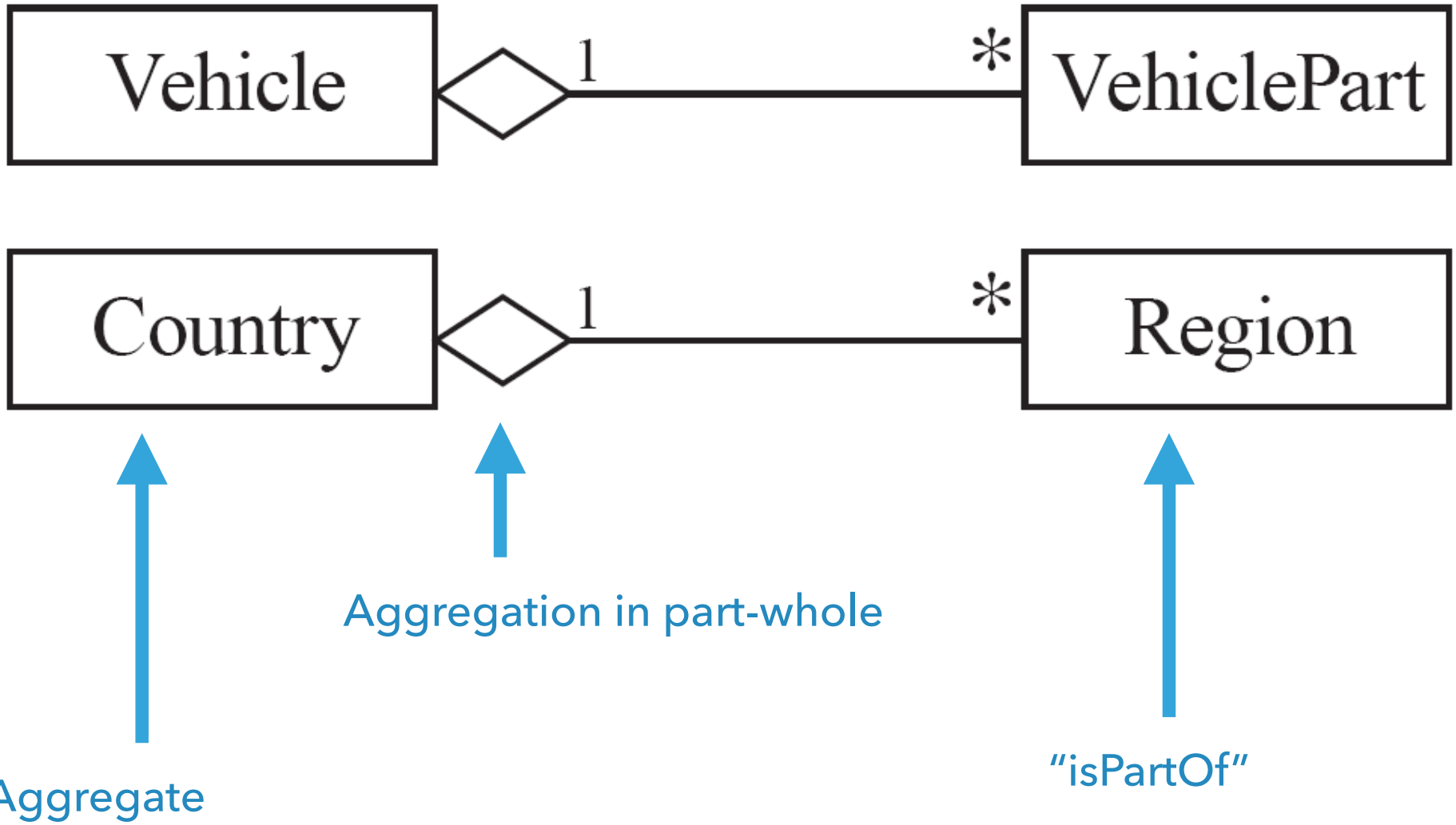


Associations versus generalizations in object diagrams

- Associations describe the relationships that will exist between instances at run time.
  - When you show an instance diagram generated from a class diagram, there will be an instance of both classes joined by an association

- Generalizations describe relationships between classes in class diagrams.
  - They do not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses

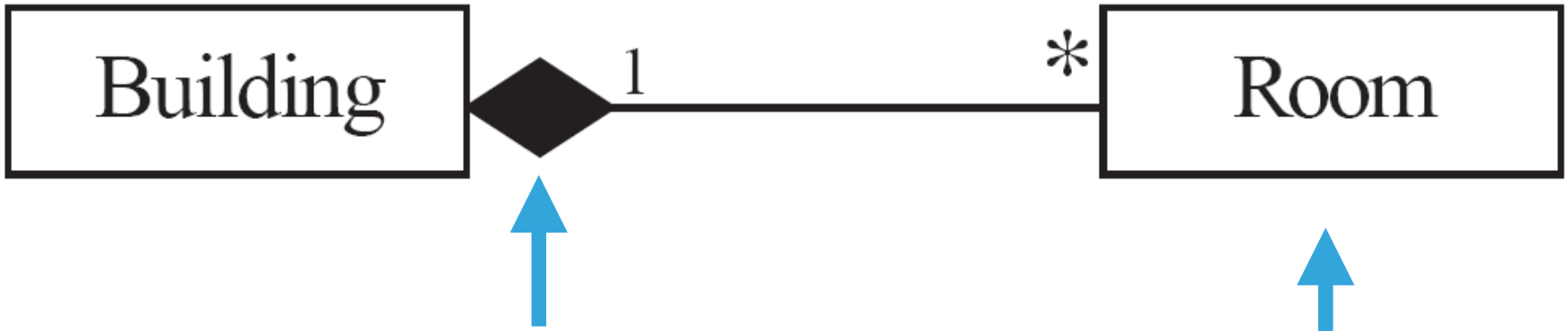
Aggregation



When to use an aggregation

- You can state that
  - the parts 'are part of' the aggregate
  - or the aggregate 'is composed of' the parts
- When something owns or controls the aggregate, then they also own or control the parts

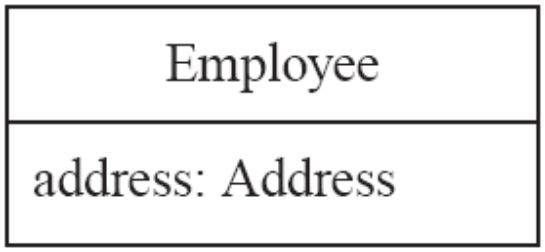
Composition

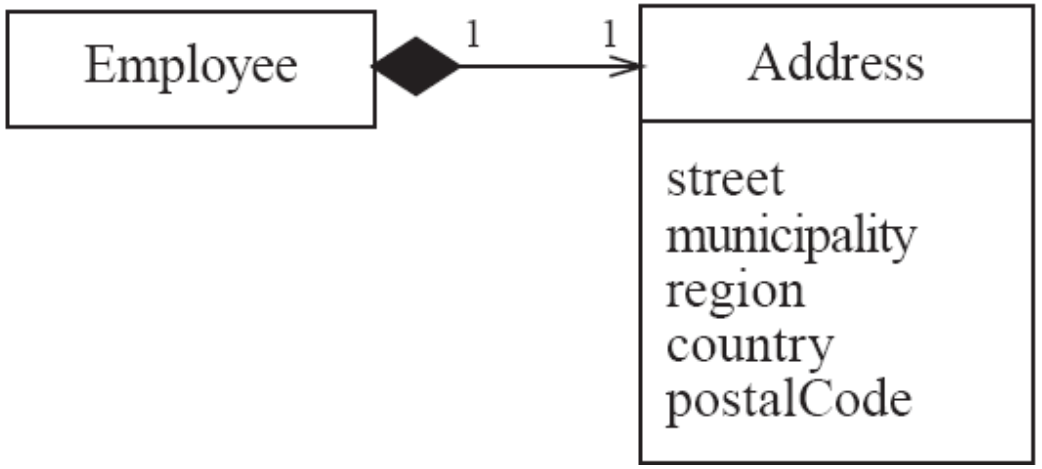


Stronger aggregate

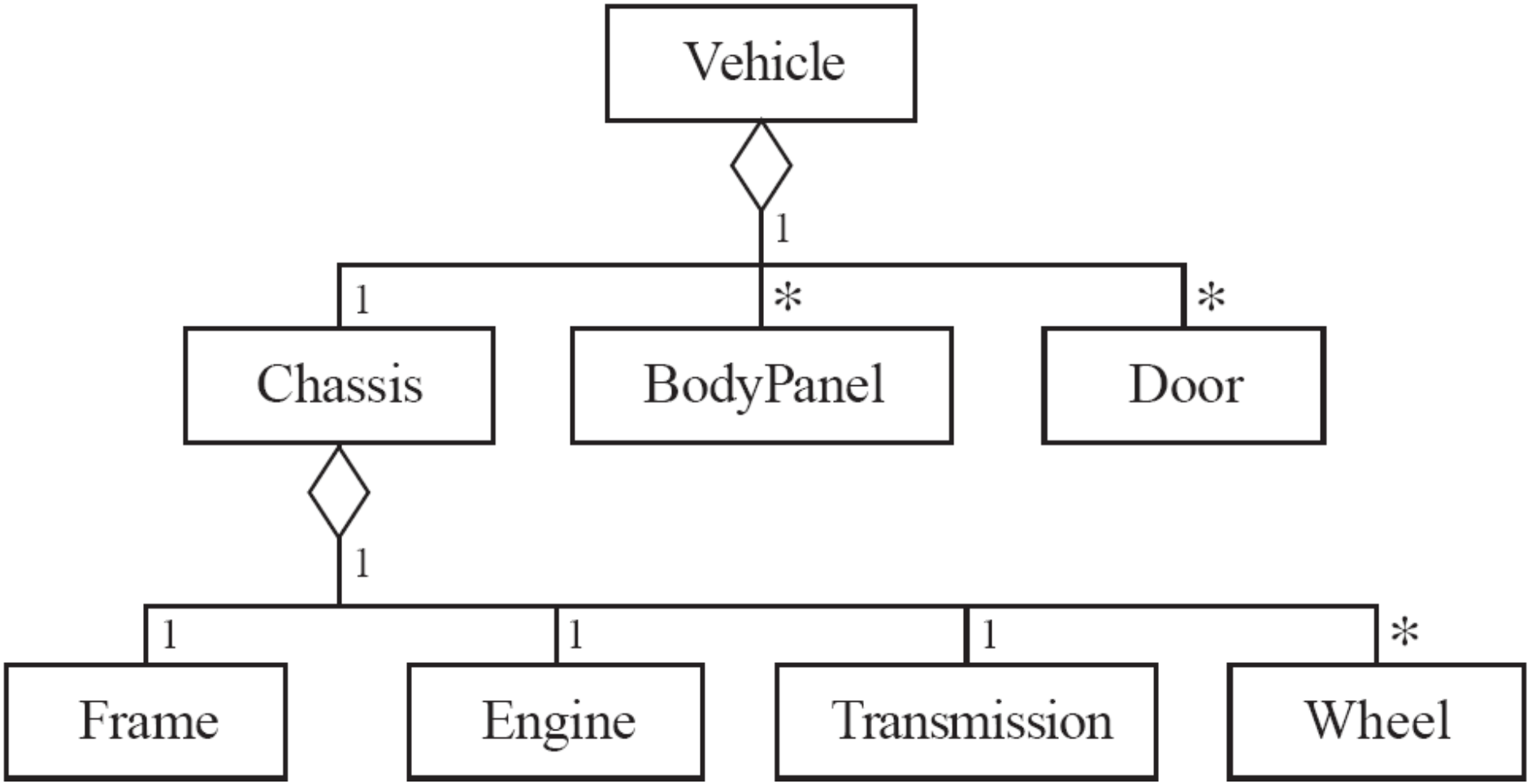
Cannot exist without the building

TWO ALTERNATIVES FOR MODELING ADDRESSES

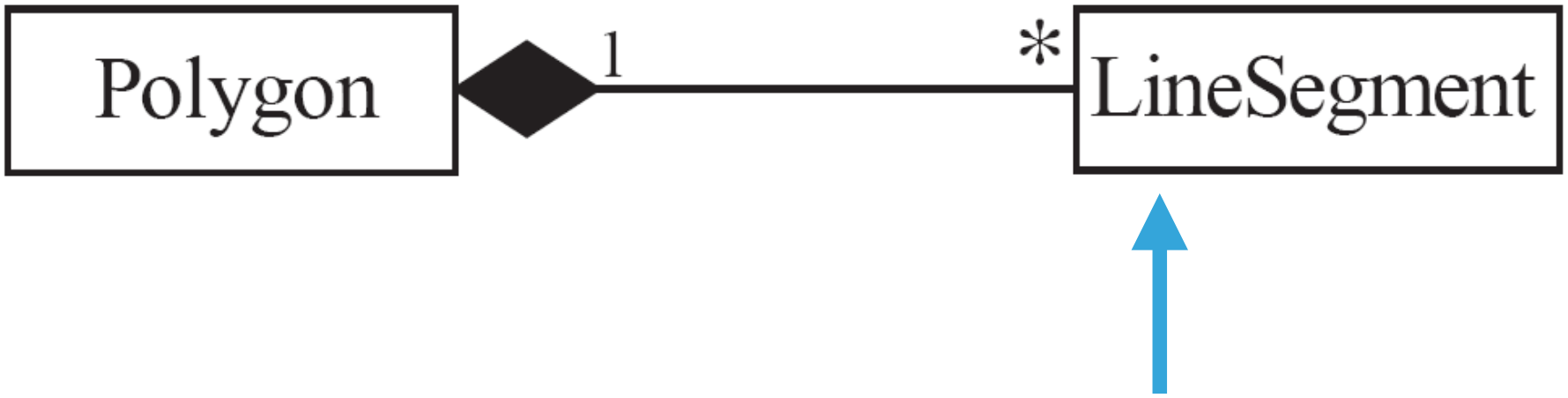




Aggregation hierarchy



Propagation

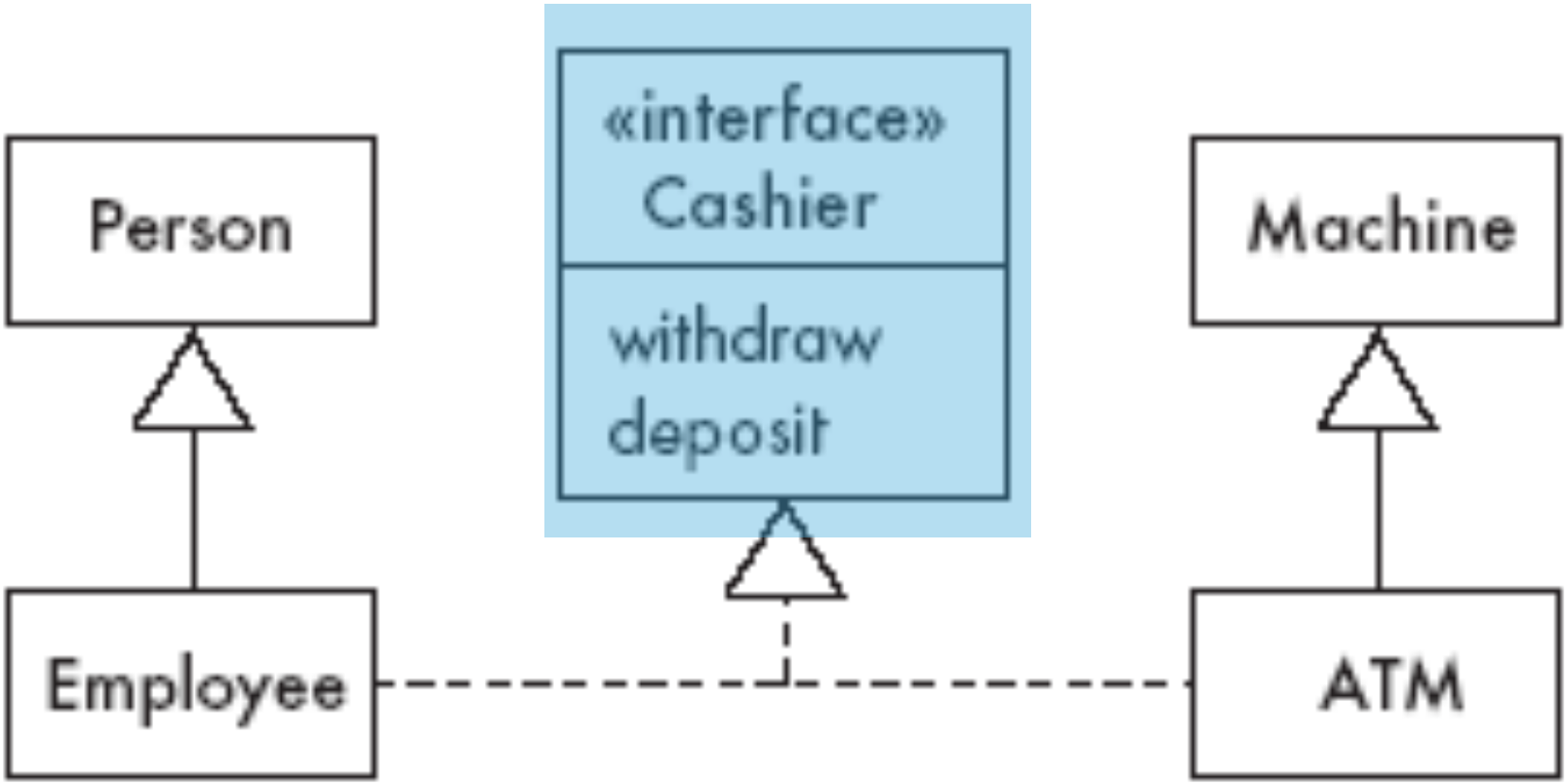


Operations and attributes delegated to its parts

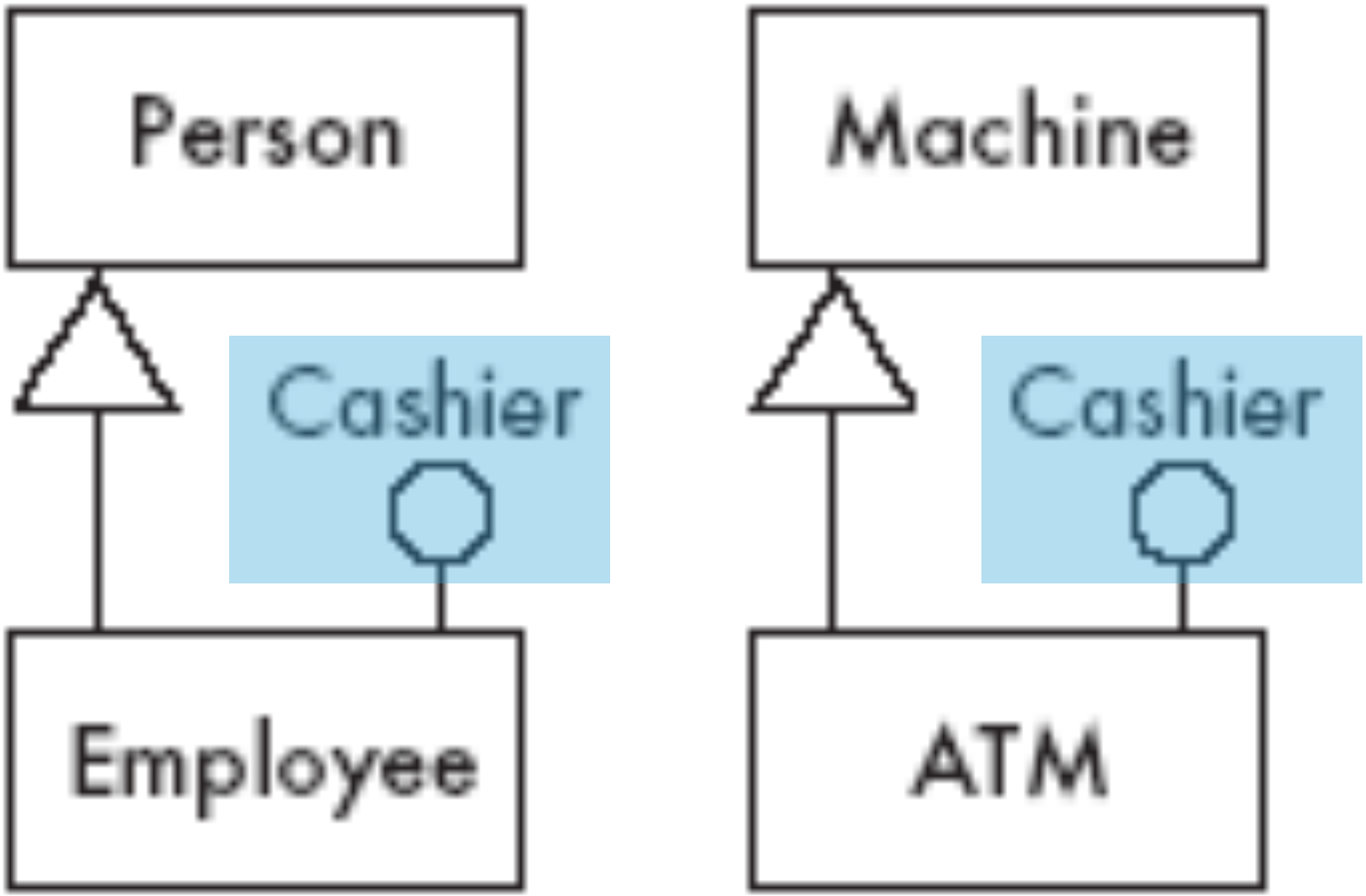
Propagation is to aggregation . . .

- as inheritance is to generalization
- The major difference is:
  - inheritance is an implicit mechanism
  - propagation has to be programmed when required

Interfaces


«interface»
Cashier
withdraw
deposit
Person
Machine
Employee
ATM

Another visualization of interfaces

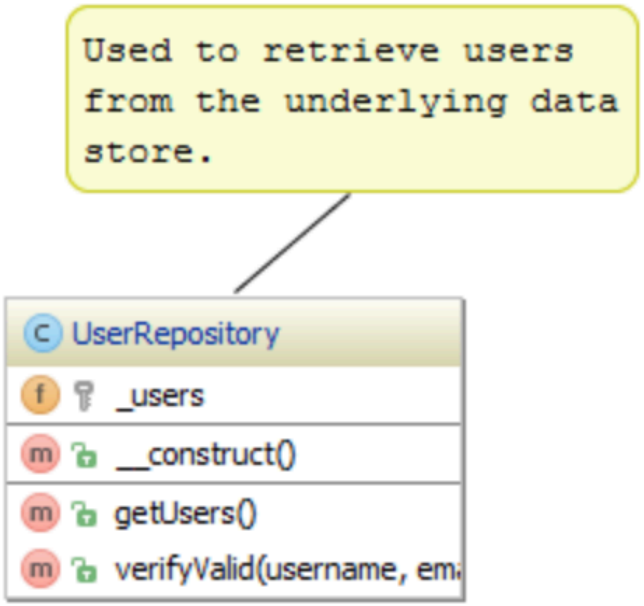


Descriptive text

- Embed your diagrams in a larger document
- Text can explain aspects of the system using any notation you like
- Highlight and expand on important features, and give rationale

NOTES

- A note is a small block of text embedded in a UML diagram
- It acts like a comment in a programming language



OBJECT CONTRAINT LANGUAGE

OCL

O
Object

C
Constraint

L
Language

By IBM in 1995, later OMG standard

W
Well

F
Formedness

R
Rules

Quality of words and conforms to grammar

https://en.wikipedia.org/wiki/Well-formedness

An **assertion** is a predicate (**a Boolean-valued function**) connected to a point in the program, that always should **evaluate to true** at that point in code execution.

Developers should fear *should*

https://en.wikipedia.org/wiki/Assertion_(software_development)

A **constraint** is a **restriction** on one or more **values** of (part of) an object-oriented model or system

- Warmer and Kleppe

OCL as a specification language for object constraints

OCL Expression

```
context Student
inv: self.age >= 0
```

Invariant must always remain true, always.

Logical fact (constraint) about the system

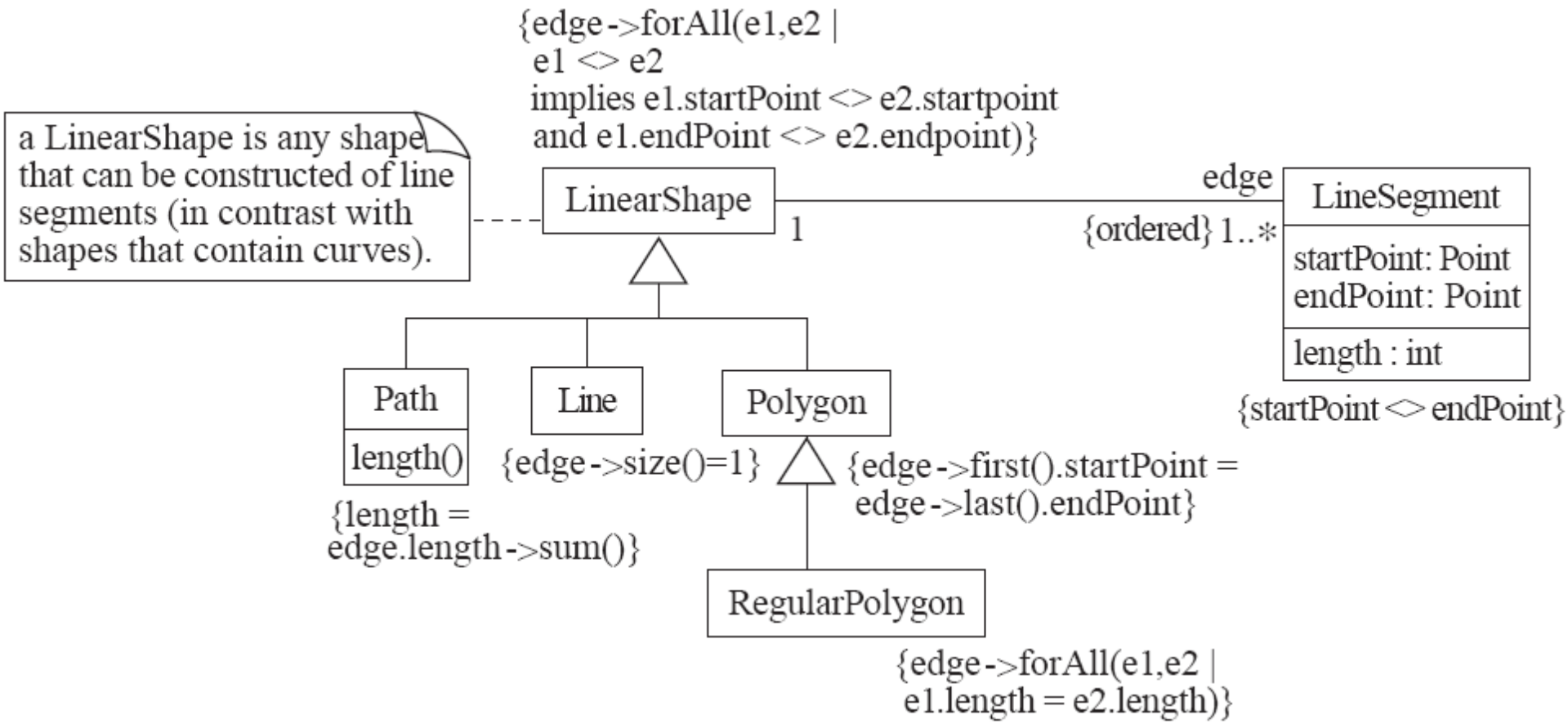
OCL CAN SPECIFY VALUES OF ATTRIBUTES AND ASSOCIATIONS

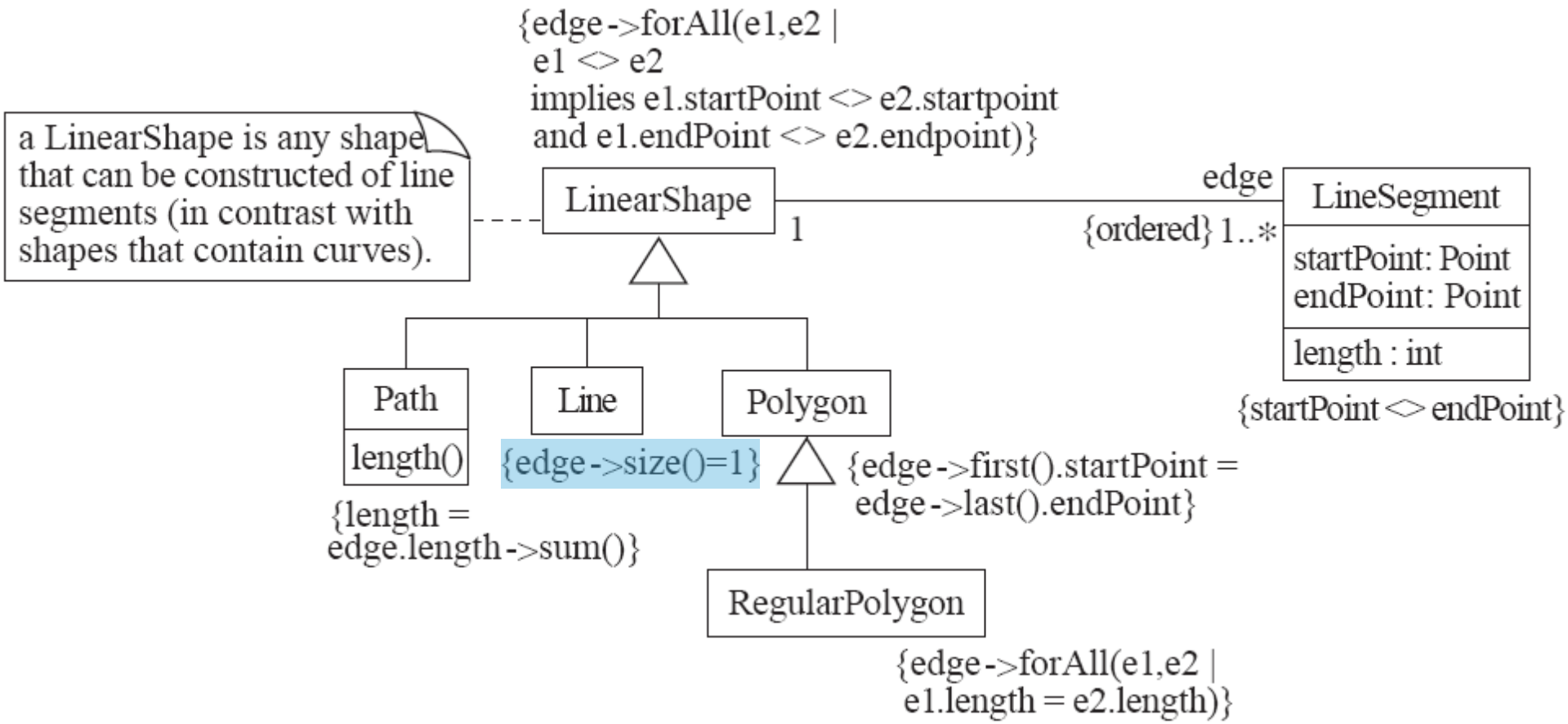
Set { 'apple' , 'orange', 'strawberry' }

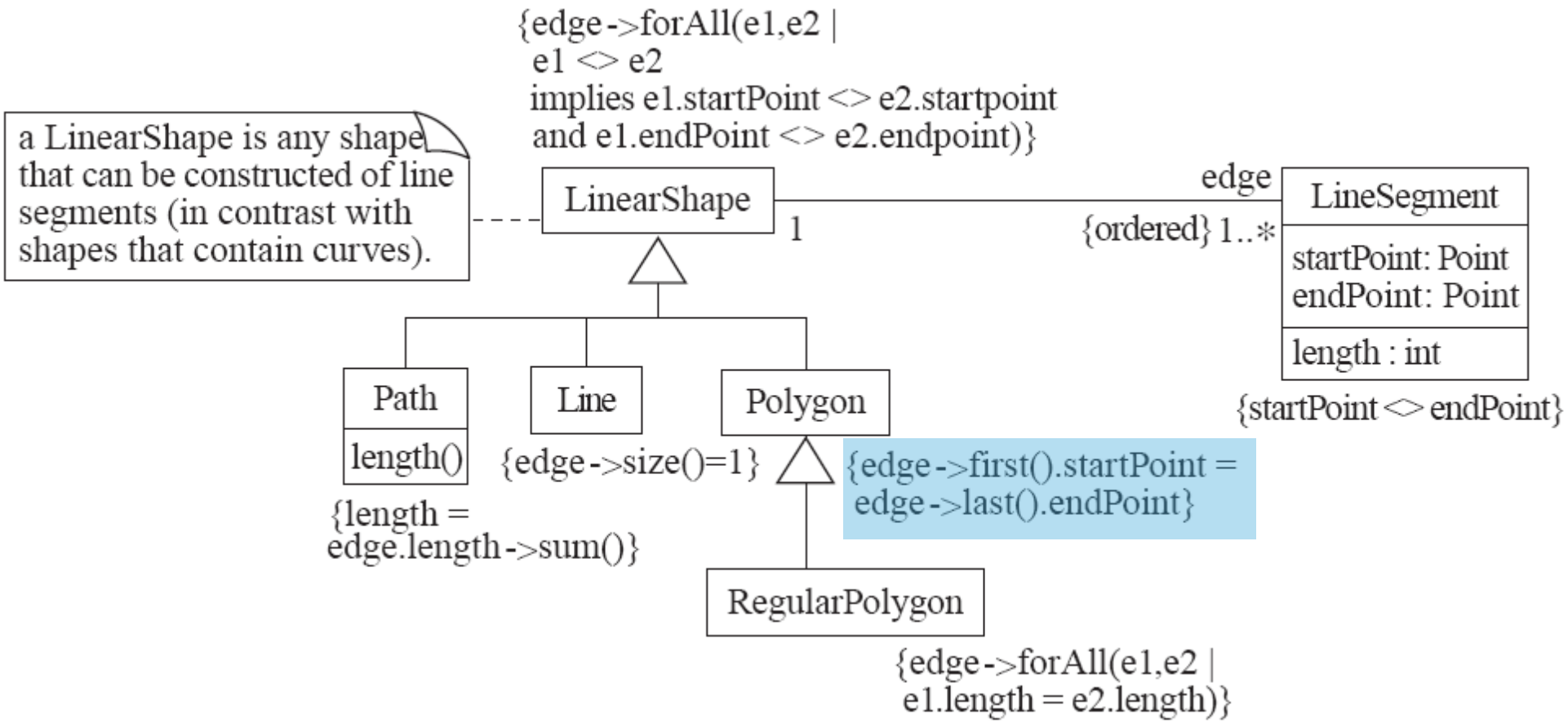
Set, sequences, bag semantics allowed

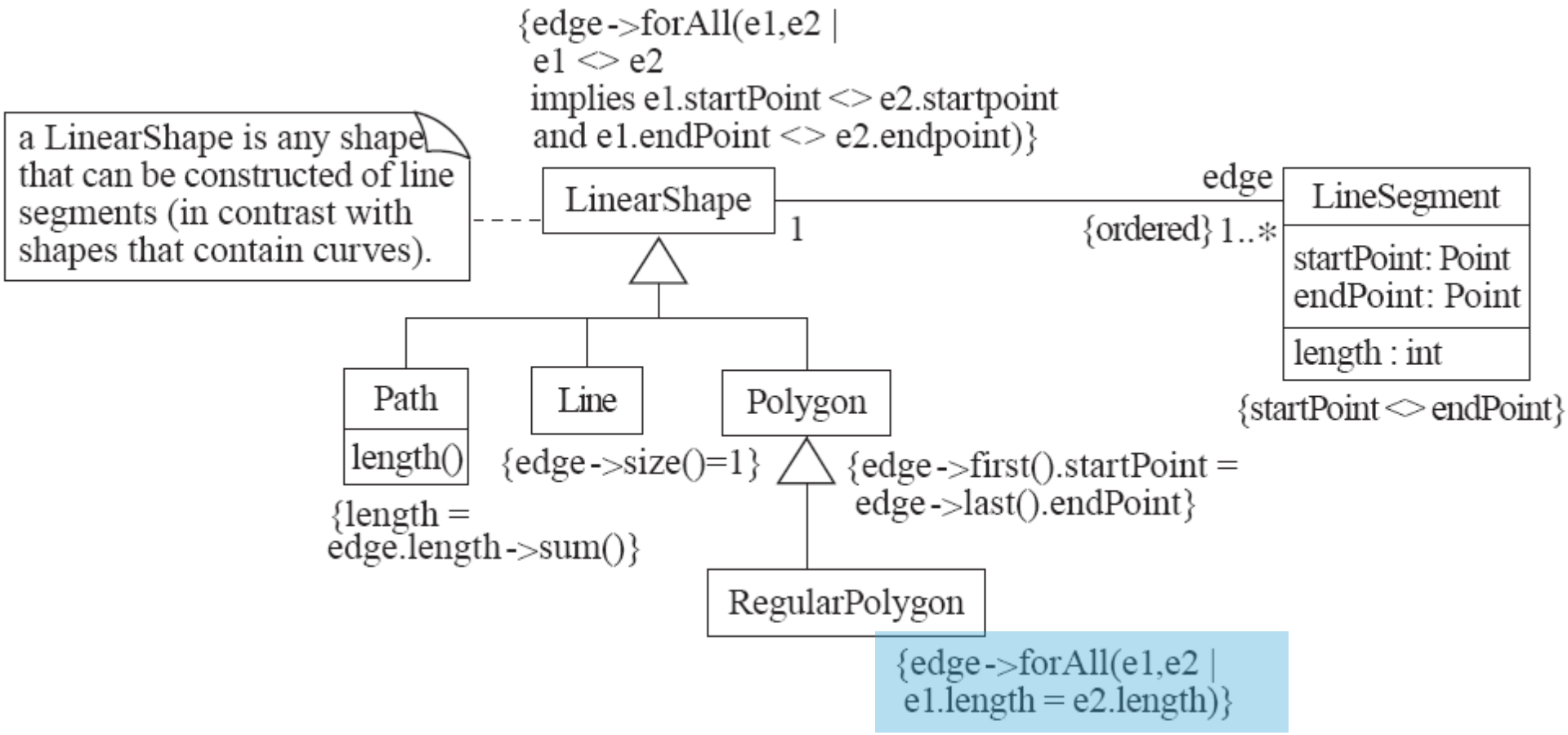
OCL Statements can be built from . . .

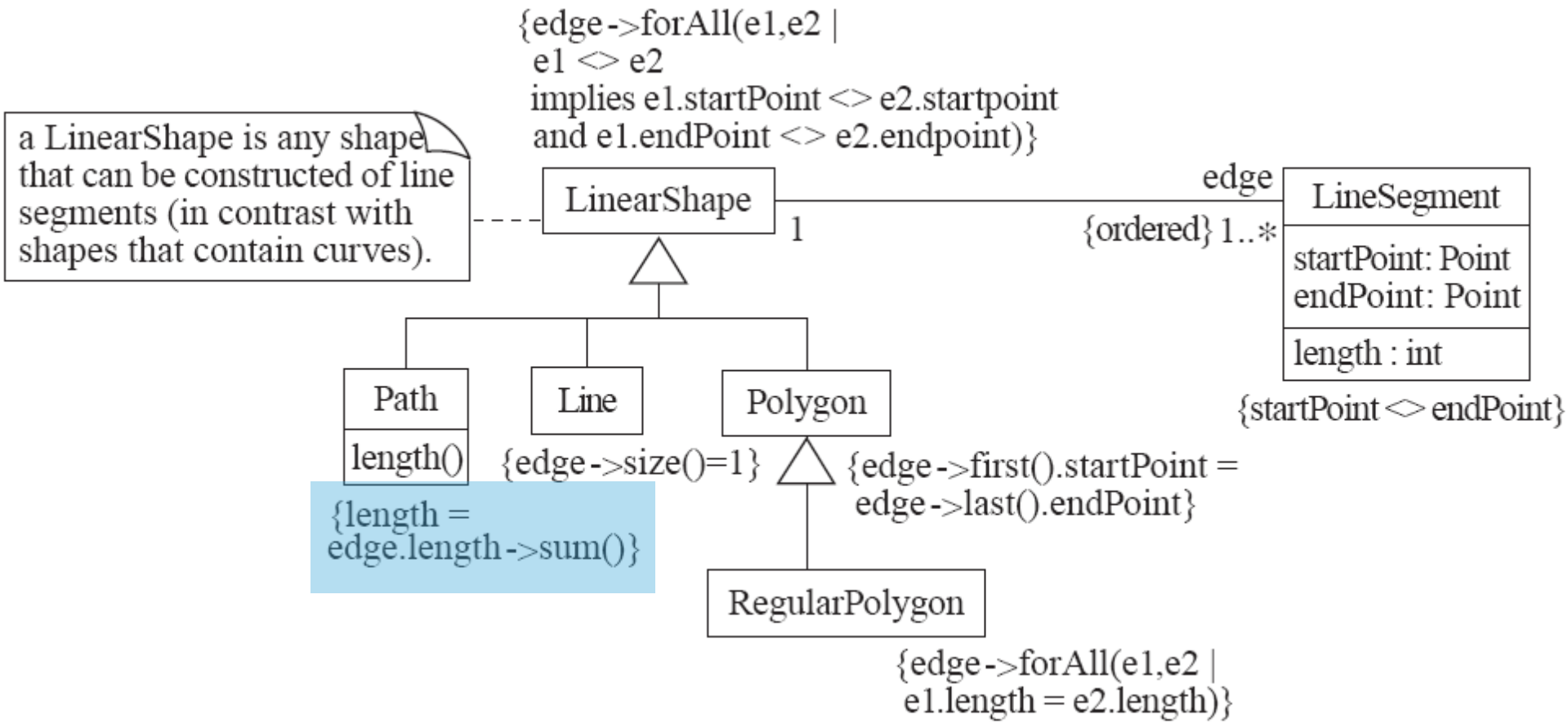
- References **names**
  - Roles,
  - Associations,
  - Attributes
  - Operation Results
- The logical values
  - **true** and
  - **false**
- **Logical** operators
  - and / or
  - = / <> (not equals)
  - > / >= / < / <=
- Strings like “O’Doyle Rules”
- Integers and real **numbers**
- **Arithmetic** operations *, /, +, -

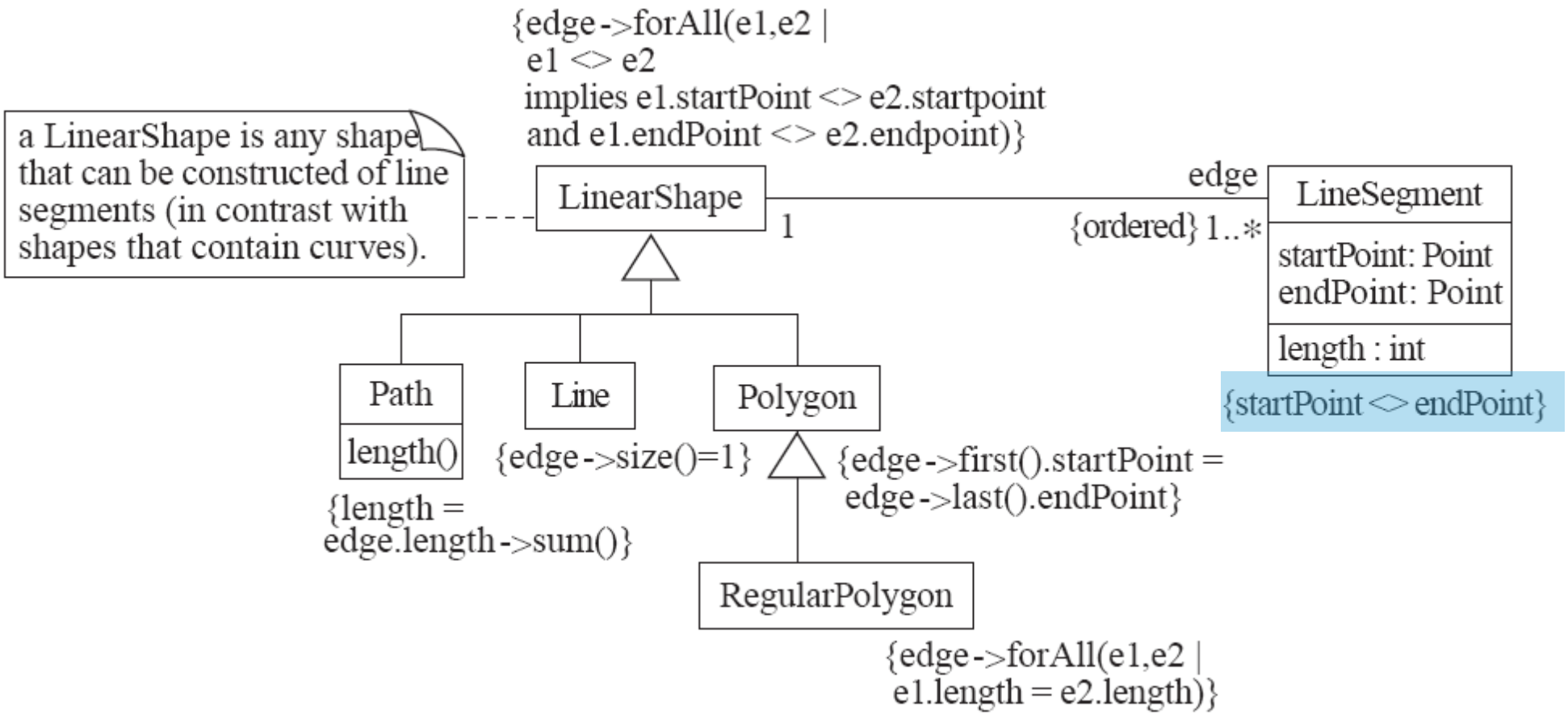

{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered} 1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}

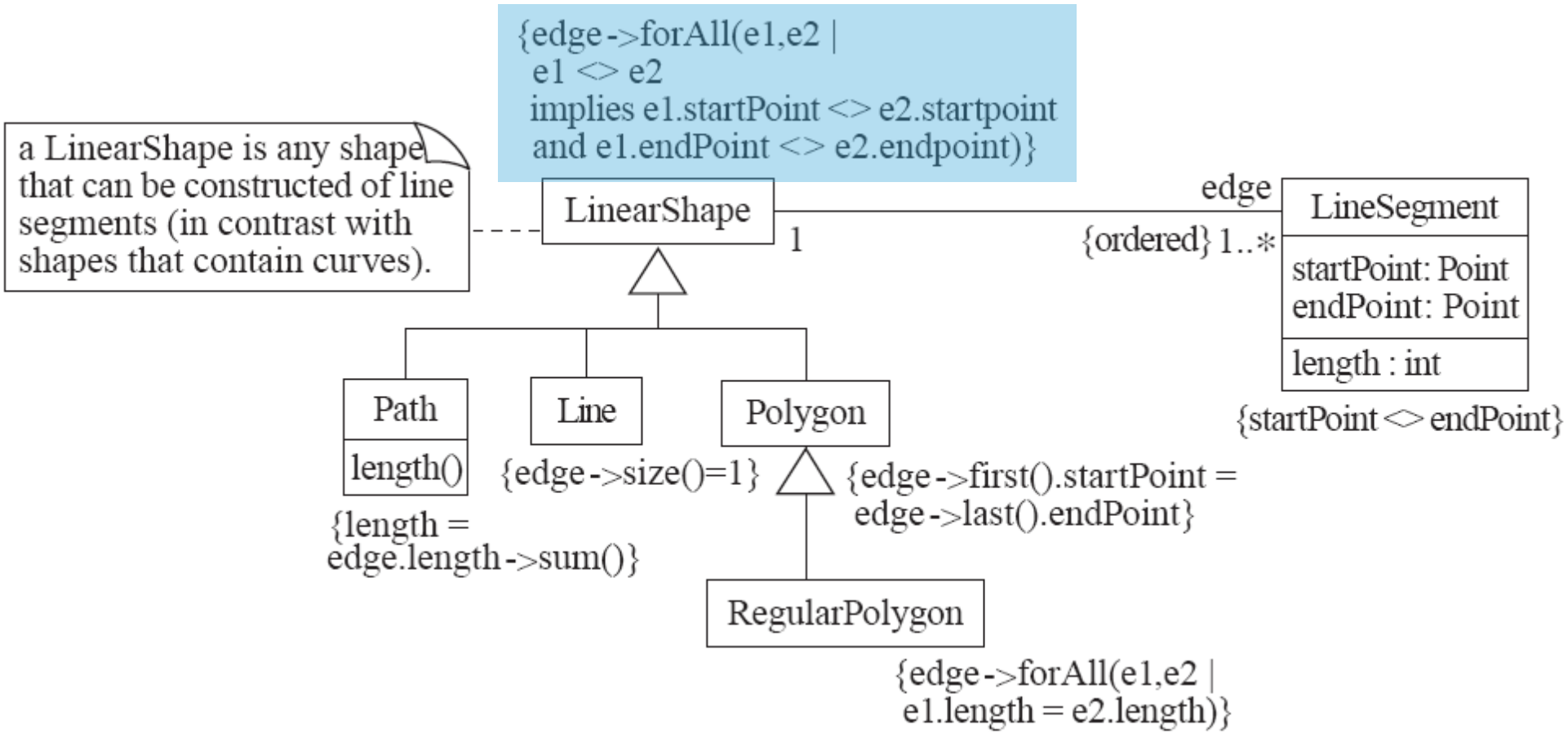

{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered} 1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}


{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered} 1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}


{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered} 1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}

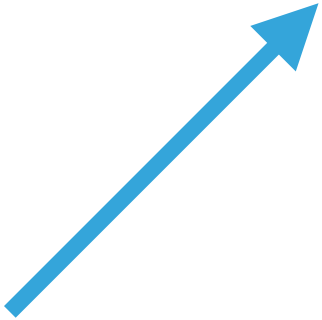

{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered} 1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}


{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered} 1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}


{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered} 1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}

A -> B equivalent to !A or B

Implies is used where
we have a precondition



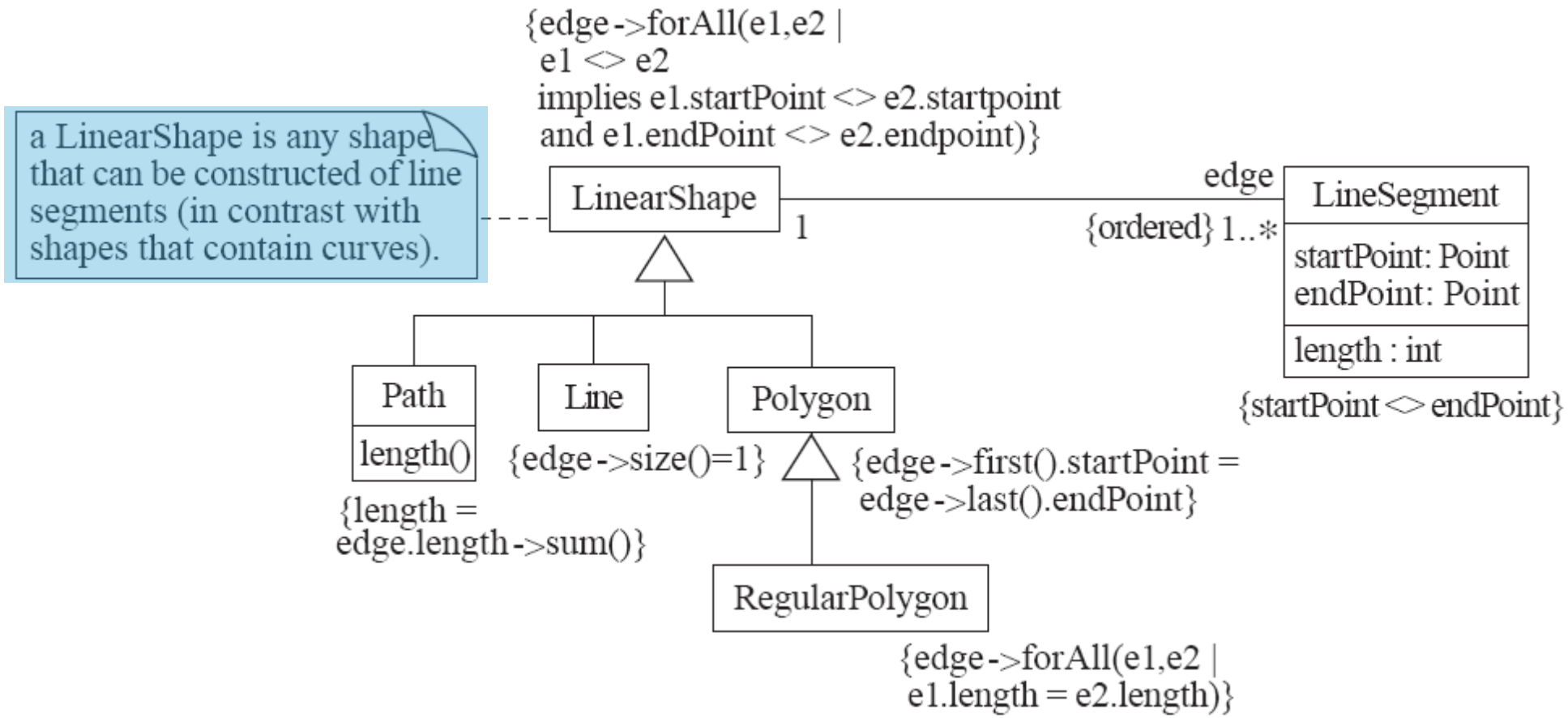
So either precondition
isn't satisfied

Or
constraint
is true

Pigs can fly implies everyone will get an A+

This is _false_ so we don't need to care about the truthiness of the next statement

Highly unlikely (unfortunately) but also highly irrelevant


{edge->forAll(e1,e2 |
e1 <> e2
implies e1.startPoint <> e2.startpoint
and e1.endPoint <> e2.endpoint)}
a LinearShape is any shape that can be constructed of line segments (in contrast with shapes that contain curves).
LinearShape
edge
1
{ordered} 1..*
LineSegment
startPoint: Point
endPoint: Point
length : int
{startPoint <> endPoint}
Path
length()
{length = edge.length->sum()}
Line
{edge->size()=1}
Polygon
{edge->first().startPoint = edge->last().endPoint}
RegularPolygon
{edge->forAll(e1,e2 |
e1.length = e2.length)}

Where to use OCL in UML

Invariants on classes and types

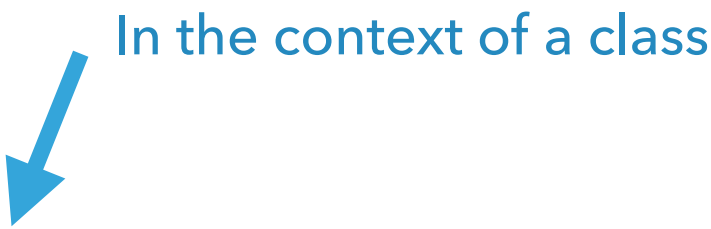
Guards

Invariants for stereotypes

Navigation Language

Pre/Post conditions

Constraints on Operations




```
context Student
self.mentor
```

self.mentor evaluates to
object stored in mentor.

If Mentor has 1-to-Many relationship to Student

context Mentor
self.students

self.students is a **Set** whose elements are of type **Student**

Or, a **Sequence** if students is **{ordered}**

context Mentor
self.students.program

Evaluates to the **Set** of all **program's** belonging to **Student**

Collection types and navigation in OCL expressions

- If self is **class C**, with attribute a
  - Then **self.a** evaluates to the **object** stored in a.
- If C has a **one-to-many** association called **assoc** to another class D
  - Then **self.assoc** evaluates to a **Set** whose elements are of **type D**.
  - If assoc is **{ordered}** then a **Sequence** results
- If D has attribute b
  - Then the expression **self.assoc.b** evaluates to the **Set** of all the **b's** belonging to D

```
context Company

inv: self.manager.isUnemployed = false

inv: self.employee->notEmpty()
```

Evaluates to **Person**

Evaluates to **Set** of **Person's**

Built in API for OCL Collections

Important OCL Collection Functions

`isEmpty()`

`notEmpty()`

`size()`

`includes(anObj)`

`sum()`

`exists(expr)`
i.e. ∃ in math

COLLECTION FILTERS

select()

```
self.employee
->select
   (age > 50)
->notEmpty()
```

Subset of anyone over 50

reject()

```
self.employee
->reject(p |
   p.age<=50)
->notEmpty()
```

Named object **"p"**, subset to remove anyone 50 or under

FORALL (∀ IN LOGIC)

```
inv: self.employee
->forAll( name = 'Robert Paulson' )
```

Every employee is named
'Robert Paulson'

FORALL (∀ IN LOGIC)

```
inv: self.employee
->forAll(p | p.name = 'Robert Paulson' )
```

Represent each employee as
interim variable **p** (for person)

FORALL (∀ IN LOGIC)

```
inv: self.employee
->forAll(p : Person | p.name = 'Robert Paulson' )
```

Further specify that **p** is indeed a **Person**

Syntax forAll (∀ in Logic)

collection->forAll(v : Type | bool-expr-with-v)

collection->forAll(v | boolean-expression-with-v)

collection->forAll(boolean-expression)

Important OCL Collection Functions

```
c1->
includesAll(c2)
```

```
c1->
excludesAll(c2)
```

```
s1->
intersection(s2)
```

```
s1->
union(s2)
```

```
s1->
excluding(x)
```

```
seq->first()
```

FORALL ON PAIRS

```
self.employee->forAll( e1, e2 : Person |
  e1 <> e2 implies e1.ssn <> e2.ssn)
```

Must include “yourself”

Employees cannot have the same social security number (SSN)

SIMPLER THAN

```
self.employee->forAll(e1 | self.employee->forAll (e2 |
    e1 <> e2 implies e1.forename <> e2.forename))
```

Must include “yourself”

Employees cannot have the same social security number (SSN)

Class invariants

numberOfEmployees > 50

Must have more than 50 employees

CLASS INVARIANTS

self.numberOfEmployees > 50

Explicitly refer to *self*

CLASS INVARIANTS

Explicitly state the class

```
context Company
inv: self.numberOfEmployees > 50
```

Explicitly state it's an invariant

CLASS INVARIANTS

Give a local variable name to context

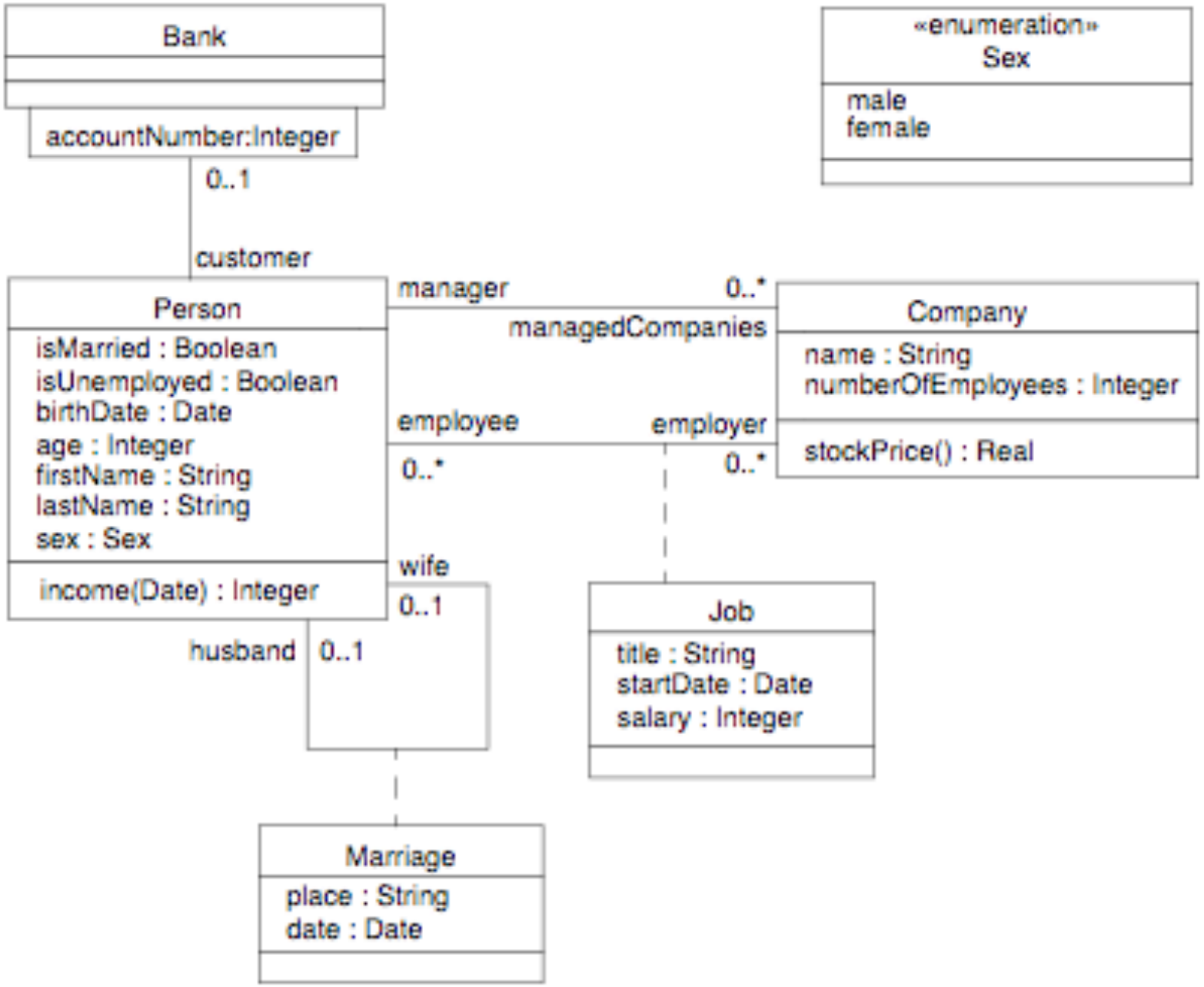
```
context c : Company
inv: c.numberOfEmployees > 50
```

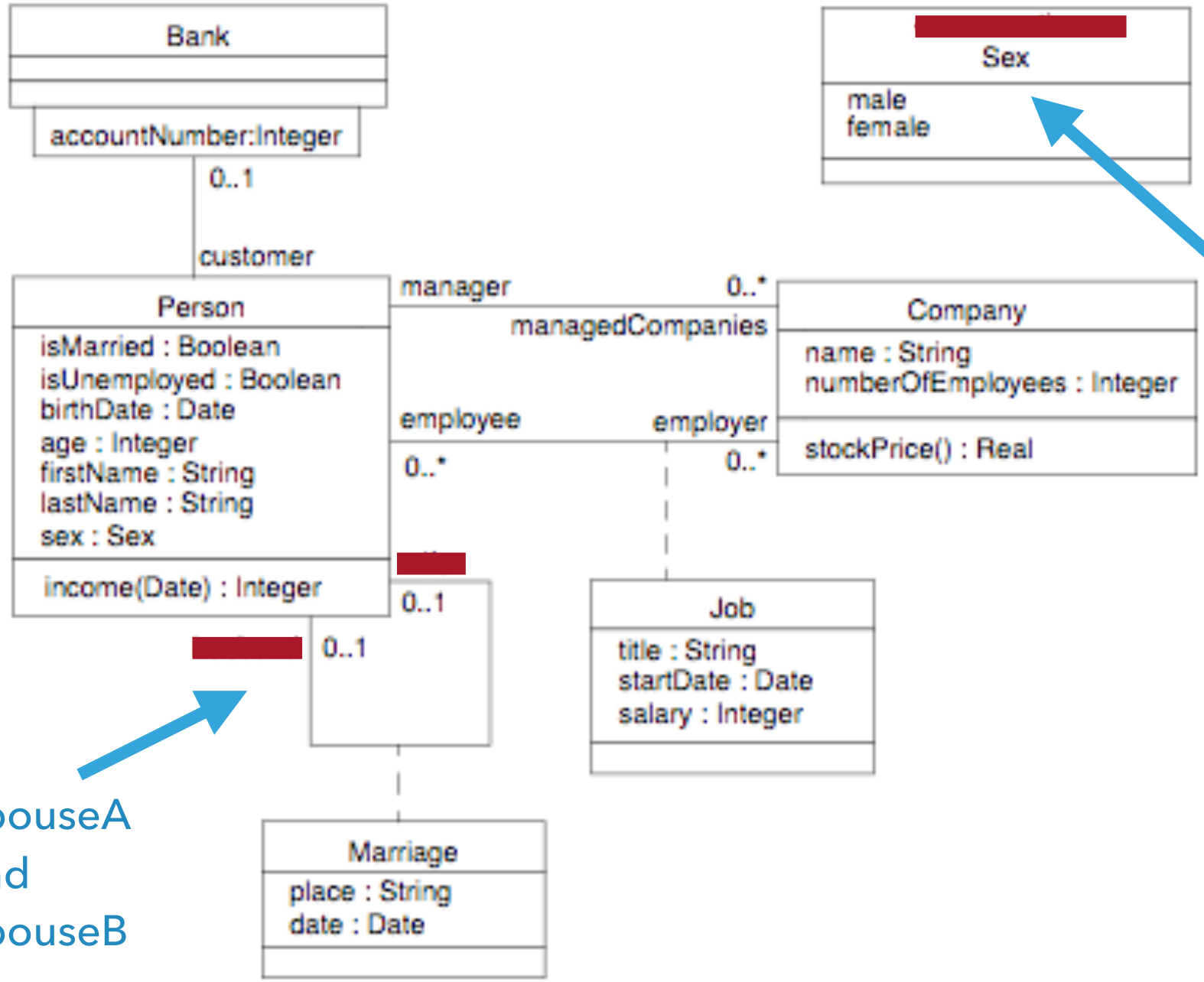
Use variable instead of self

CLASS INVARIANTS

Give invariant a name

```
context c : Company
inv enoughEmployees:
  c.numberOfEmployees > 50
```


Bank
accountNumber:Integer
0..1
customer
«enumeration»
Sex
male
female
Person
isMarried : Boolean
isUnemployed : Boolean
birthDate : Date
age : Integer
firstName : String
lastName : String
sex : Sex
income(Date) : Integer
manager
0..*
managedCompanies
Company
name : String
numberOfEmployees : Integer
stockPrice() : Real
employee
0..*
employer
0..*
wife
0..1
husband
0..1
Job
title : String
startDate : Date
salary : Integer
Marriage
place : String
date : Date


Bank
accountNumber:Integer
0..1
customer
Person
isMarried : Boolean
isUnemployed : Boolean
birthDate : Date
age : Integer
firstName : String
lastName : String
sex : Sex
income(Date) : Integer
manager
0..*
managedCompanies
employee
0..*
employer
0..*
Company
name : String
numberOfEmployees : Integer
stockPrice() : Real
Job
title : String
startDate : Date
salary : Integer
0..1
0..1
Marriage
place : String
date : Date
Sex
male
female
Gender ! = Sex,
But, binary still too rigid
SpouseA and SpouseB

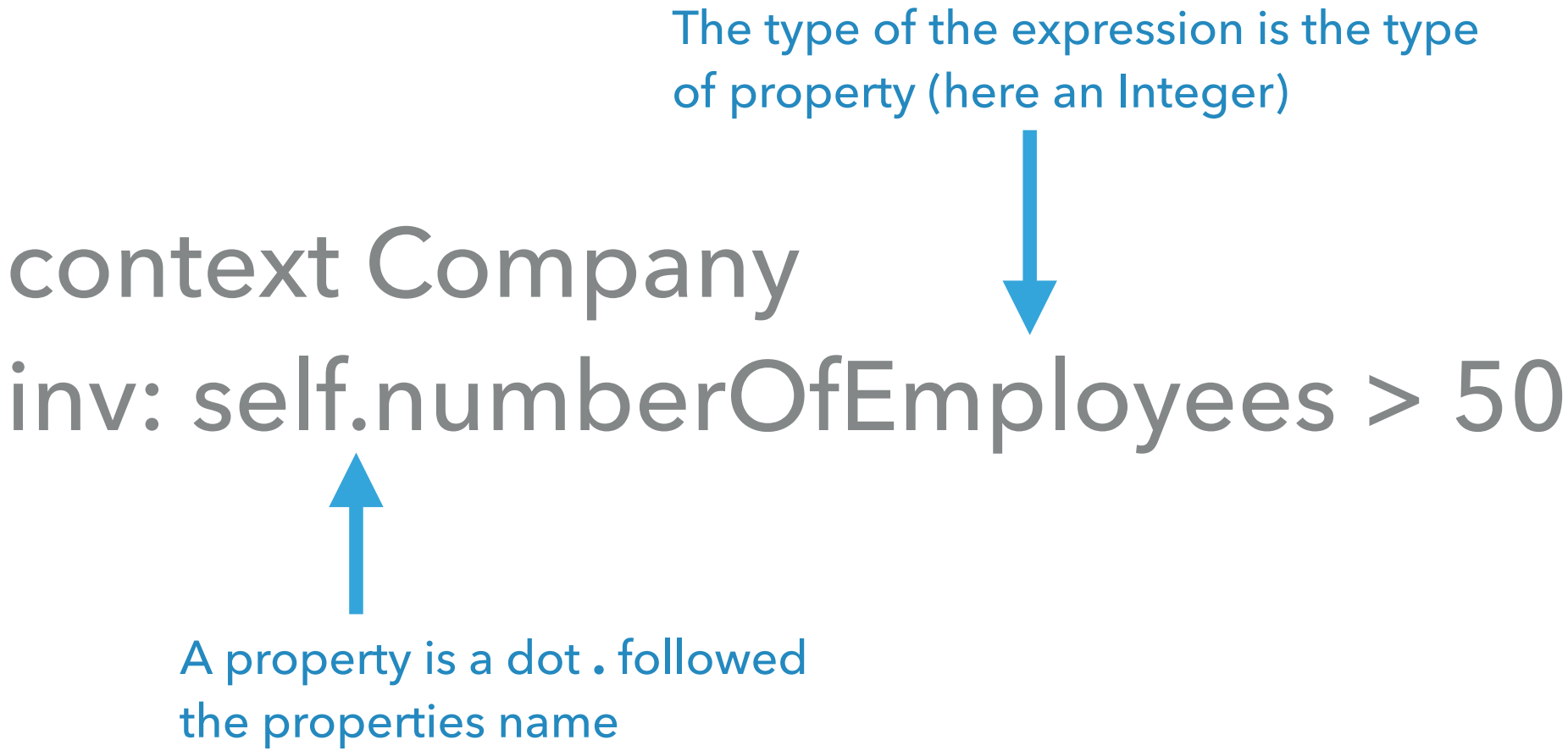
CONTEXT AND SELF

All classifiers (types, classes, interfaces, associations, datatypes, . . .) from an UML model are OCL types

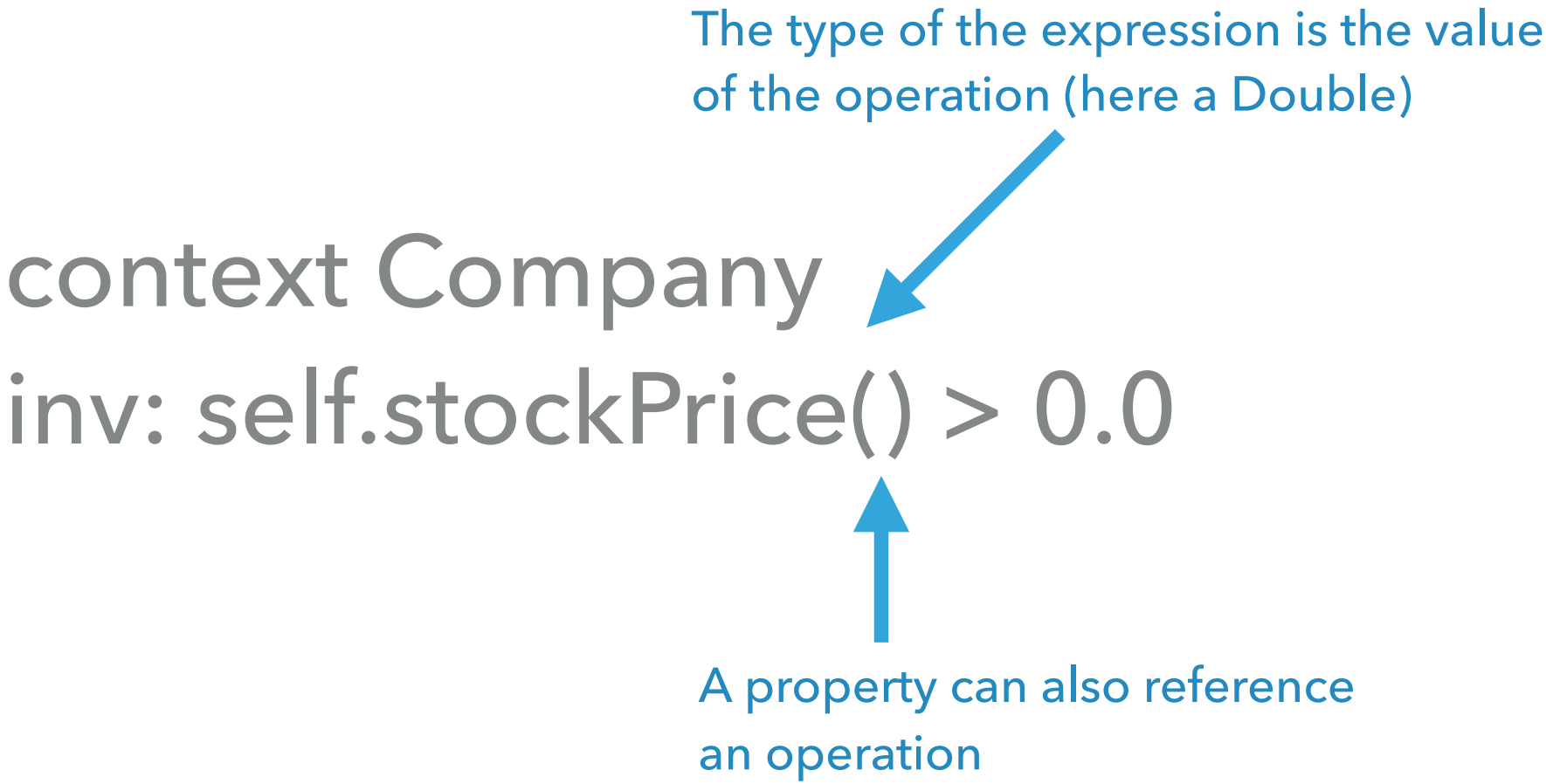
```
context Company
inv: self.numberOfEmployees > 50
```

Here **self** refers to an instance of **Company**

Object and Properties



Object and Operations



INVARIANTS

```
context Company
inv: self.numberOfEmployees > 50
```

A constraint that must always be true for all instances of a type

Pre and Post Conditions

Context is the value of a Person's income

```
context Person::income(): Integer

pre: self.age >= 18
post: result < 5000
```

Guarantee from operation *after* the call performed

Assumed to be true (and tested) before executing operations

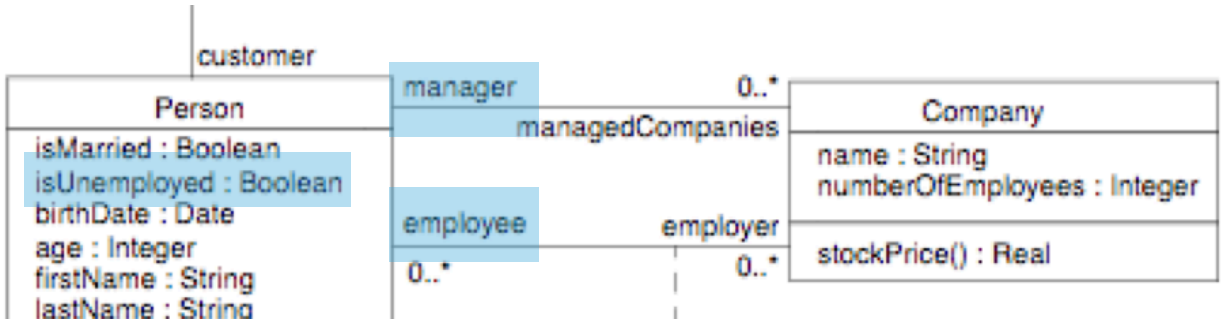
Reserved word **result** for the returned value of the operation

NAVIGATING ASSOCIATIONS

Navigate associations using *opposite* role name.

```
context Company

  inv: self.manager.isUnemployed = false

  inv: self.employee->notEmpty()
```

Collection semantics

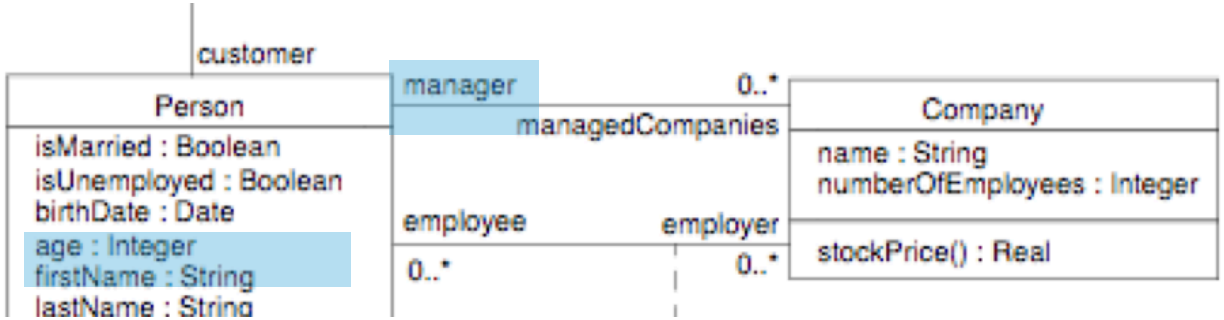


NAVIGATING ASSOCIATIONS

Simple object can be returned if multiplicity at most 1

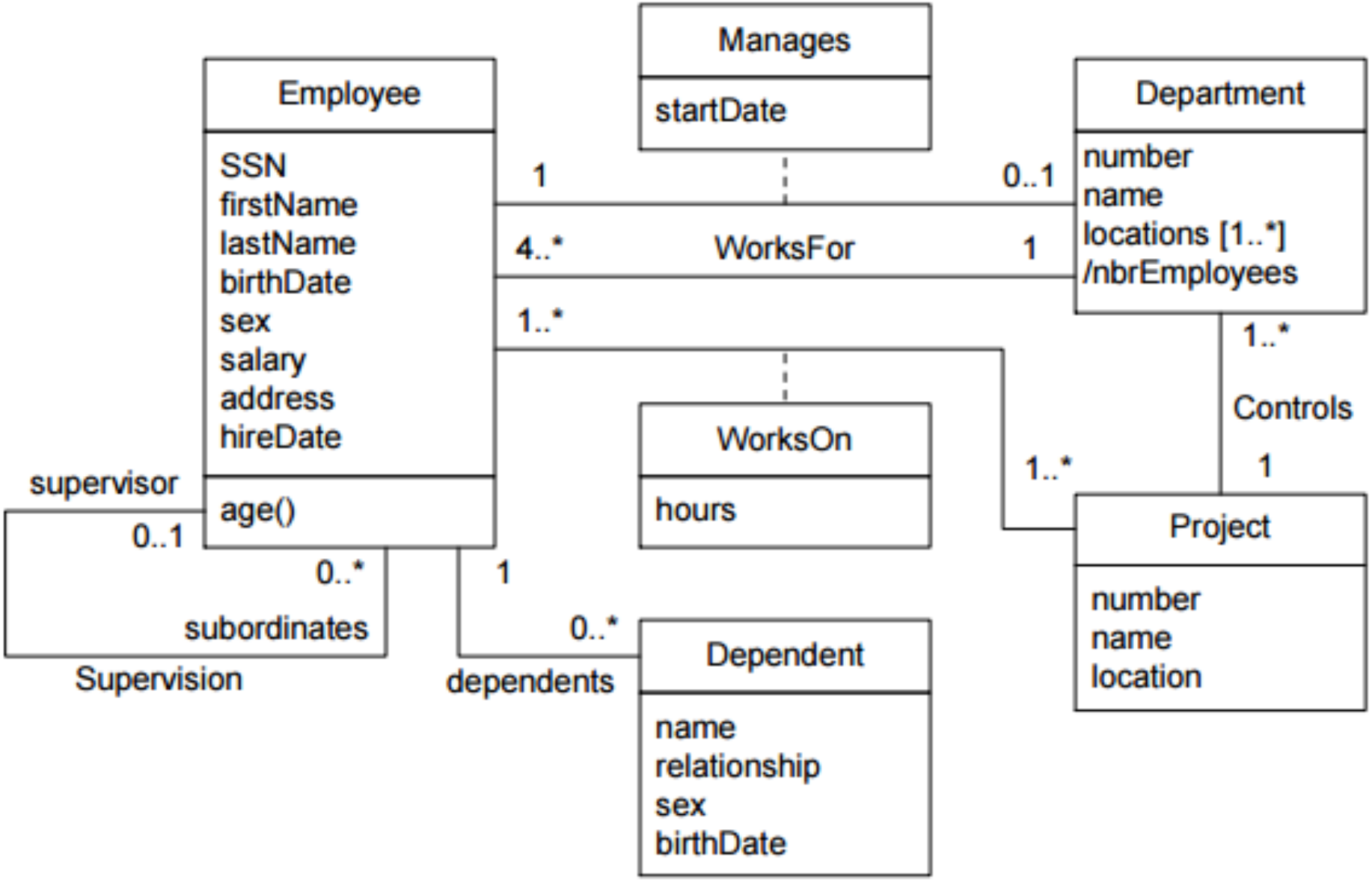
```
context Company

  inv: self.manager.age > 40

  inv: self.manager->size() = 1
```

Collection semantics even when at most 1



Type	Operations
Boolean	and, or, xor, not, implies, if-then-else-endif
Integer	+, -, *, /, abs, div, mod, max, min
Real	+, -, *, /, abs, div, mod, max, min, floor, round
String	size, concat, substring, toInteger, toReal

Operation	Usage
isEmpty	aCollection->isEmpty()
notEmpty	aCollection-> notEmpty ()
includes	aCollection->includes(anObject)
excludes	aCollection->excludes(anObject)
sum	aCollectionOfNumbers->sum()
exists	aCollection->exists(booleanExpression)
size	aCollection->size()


Manages
startDate
Employee
SSN
firstName
lastName
birthDate
sex
salary
address
hireDate
age()
Department
number
name
locations [1..*]
/nbrEmployees
1
0..1
4..*
WorksFor
1
1..*
1..*
Controls
1
WorksOn
hours
1..*
supervisor
0..1
0..*
subordinates
Supervision
1
0..*
dependents
Dependent
name
relationship
sex
birthDate
Project
number
name
location

Examples for home...

- The supervisor of an employee must be older than the employee
- The salary of an employee cannot be greater than the salary of his/her supervisor
- The manager of a department must be an employee of the department
- The SSN of employees is an identifier (must be unique)
- The location of a project must be one of the locations of its department
- A project can have at most 2 employees working on the project less than 10 hours
- An employee cannot supervise him/herself

THE PROCESS OF DEVELOPING

CLASS DIAGRAMS

Different stages, different purposes, different details

Exploratory Domain Model

- During Domain Analysis to understand and explore

System Domain Model

- Models aspects of the domain for the system

System Model

- More detailed with UI and Architecture details

Type of model	*Contains elements that represent things in the domain*	*Models only things that will actually be implemented*	*Contains elements that do not represent things in the domain, but are needed to build a complete system*
Exploratory domain model: developed in domain analysis to learn about the domain	Yes	No	No
System domain model: models those aspects of the domain represented by the system	Yes	Yes	No
System model: includes classes used to build the user interface and system architecture	Yes	Yes	Yes

Type of model	*Contains elements that represent things in the domain*	*Models only things that will actually be implemented*	*Contains elements that do not represent things in the domain, but are needed to build a complete system*
Exploratory domain model: developed in domain analysis to learn about the domain	Yes	No	No
System domain model: models those aspects of the domain represented by the system	Yes	Yes	No
System model: includes classes used to build the user interface and system architecture	Yes	Yes	Yes

Type of model	*Contains elements that represent things in the domain*	*Models only things that will actually be implemented*	*Contains elements that do not represent things in the domain, but are needed to build a complete system*
Exploratory domain model: developed in domain analysis to learn about the domain	Yes	No	No
System domain model: models those aspects of the domain represented by the system	Yes	Yes	No
System model: includes classes used to build the user interface and system architecture	Yes	Yes	Yes

Type of model	*Contains elements that represent things in the domain*	*Models only things that will actually be implemented*	*Contains elements that do not represent things in the domain, but are needed to build a complete system*
Exploratory domain model: developed in domain analysis to learn about the domain	Yes	No	No
System domain model: models those aspects of the domain represented by the system	Yes	Yes	No
System model: includes classes used to build the user interface and system architecture	Yes	Yes	Yes

Type of model	*Contains elements that represent things in the domain*	*Models only things that will actually be implemented*	*Contains elements that do not represent things in the domain, but are needed to build a complete system*
Exploratory domain model: developed in domain analysis to learn about the domain	Yes	No	No
System domain model: models those aspects of the domain represented by the system	Yes	Yes	No
System model: includes classes used to build the user interface and system architecture	Yes	Yes	Yes

Sytem Domain Model	System Model
Omits many classes	Includes all System Domain
Independent of UI / Architecture	Models UI, Architecture and Utilities

Suggested sequence of activities

- Identify a first set of **candidate classes**
- Focus on the on **core classes**
  - Add **associations** and **attributes**
  - Find **generalizations**
  - **Iterate** for the other classes
- List the main **responsibilities**
- Decide on specific **operations**

Iterate until the model is satisfactory

- Add or delete things
- Identify interfaces
- Apply design patterns (Chapter 6)

Don't be too disorganized.
Don't be too rigid either.

Identifying classes

- You discover classes during domain modeling
- You invent classes when you work on the interface
  - Or do system architecture
  - Needed to solve design problem
  - Inventing can occur during domain modeling

Reuse should always be a concern

- Frameworks
- System extensions
- Similar systems

Discovering Domain Classes

- Source material like description of requirements
- Extract nouns
- Eliminate nouns that:
  - Redundant
  - Too specific (represent instances)
  - Too vague (or highly general)
  - Outside application scope
- Pay attention to classes that represent types of actors

Identifying associations and attributes

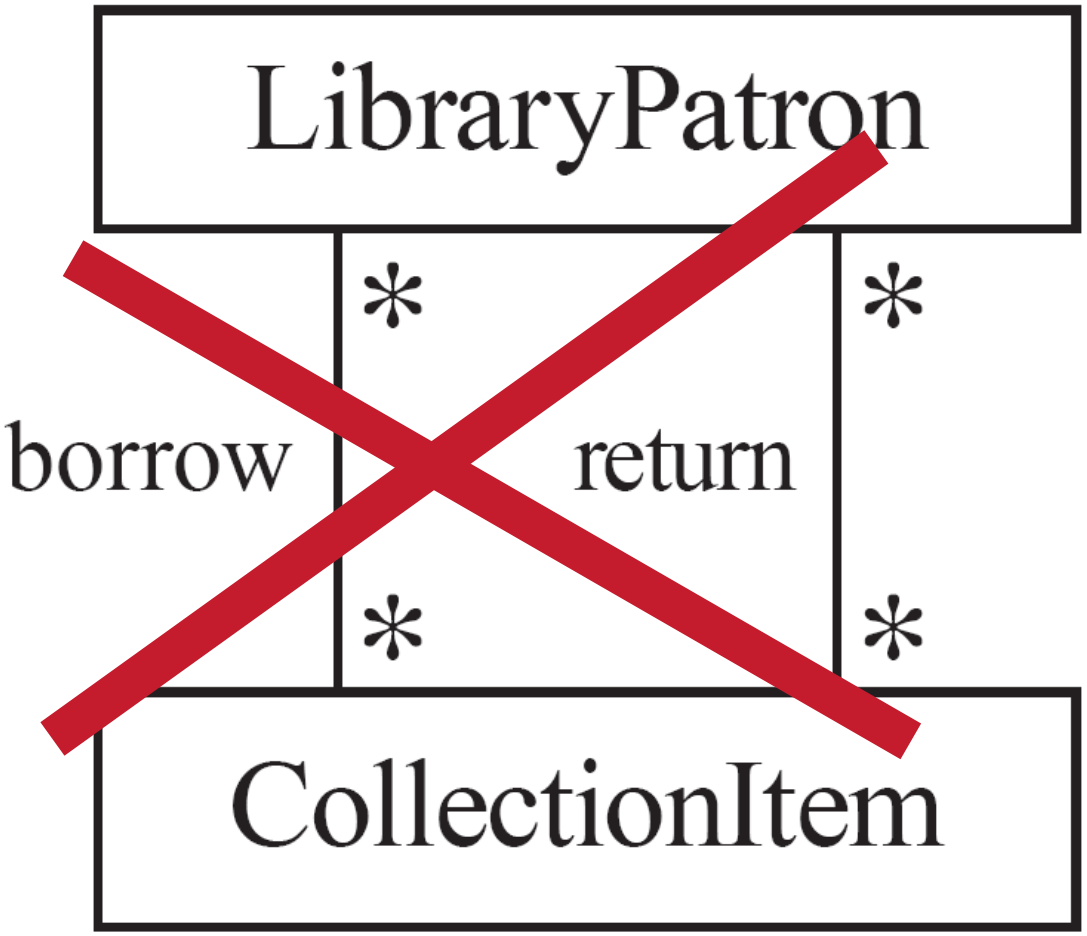
- Start in the middle (most central classes)
- Decide on obvious
  - data those classes must contain
  - Relationships to other classes
- Work outwards
- Avoid over-doing it

An association should exist if a class . . . some other class

- possesses
- controls
- is connected to
- is related to
- is a part of
- has as parts
- is a member of, or
- has as members

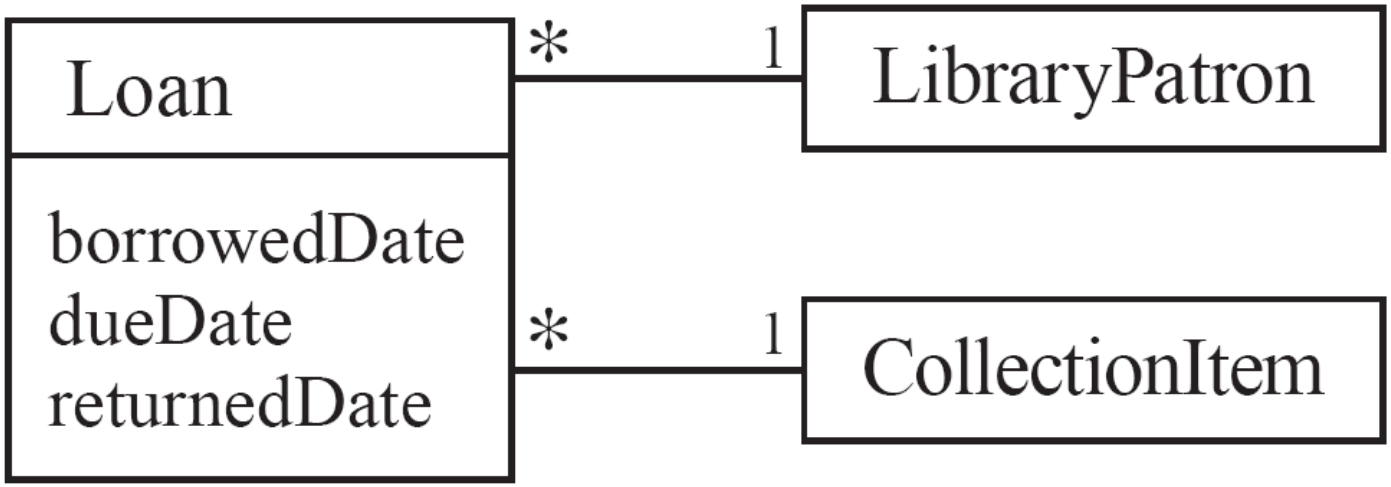
- Specify the multiplicity at both ends
- Label it clearly

Actions versus associations



BAD Associations:
These are actions to borrow and return, not associations

Actions versus associations

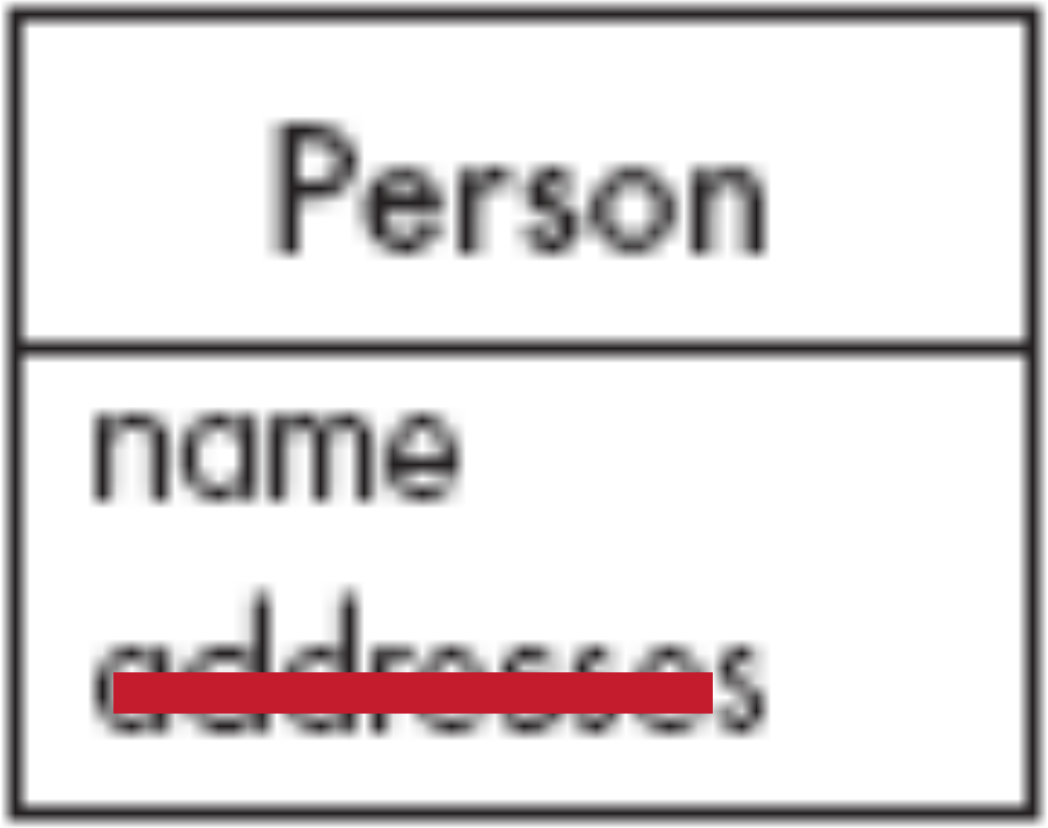


Better Associations: The borrow operation creates a Loan and the return operation sets the returnedDate attribute

Identifying attributes

- Information that must be maintained about each class
- Nouns rejected as classes
- Generally contain simple values, or a set of simple values
  - String, Number, Boolean
  - Address (several strings)

Invalidating attributes



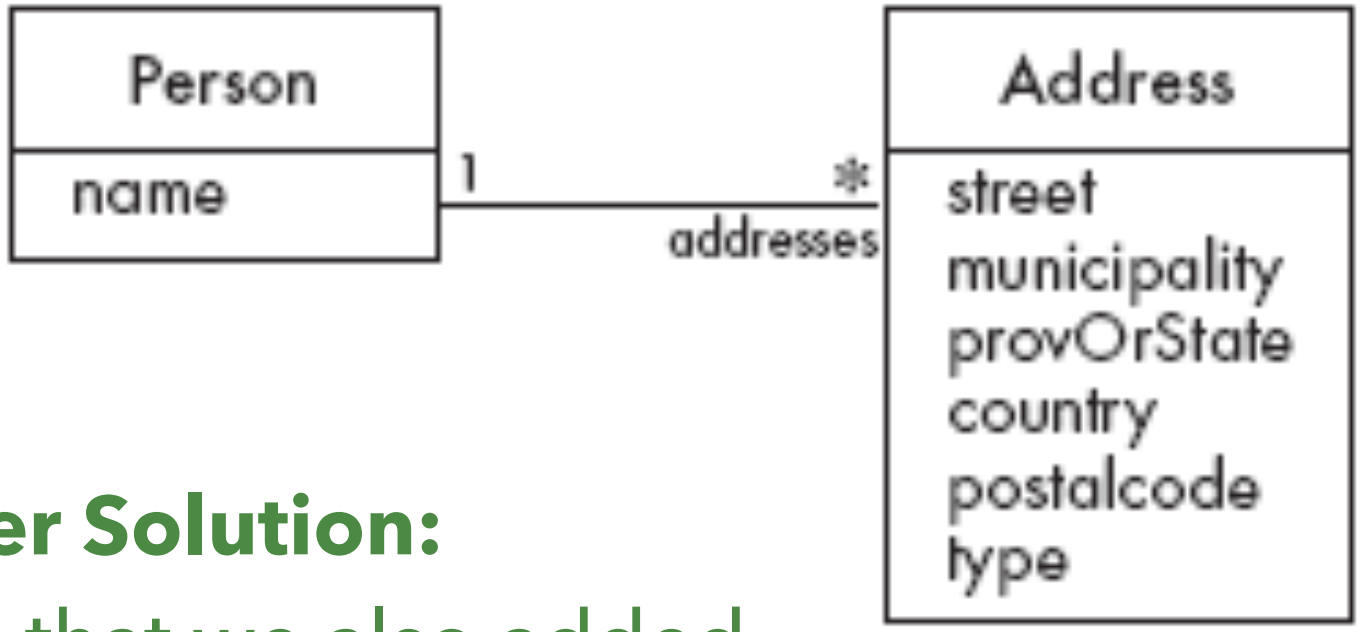
BAD Attribute:
Avoid plurals
(implicit multiplicity)

Invalidating attributes

BAD Attribute:

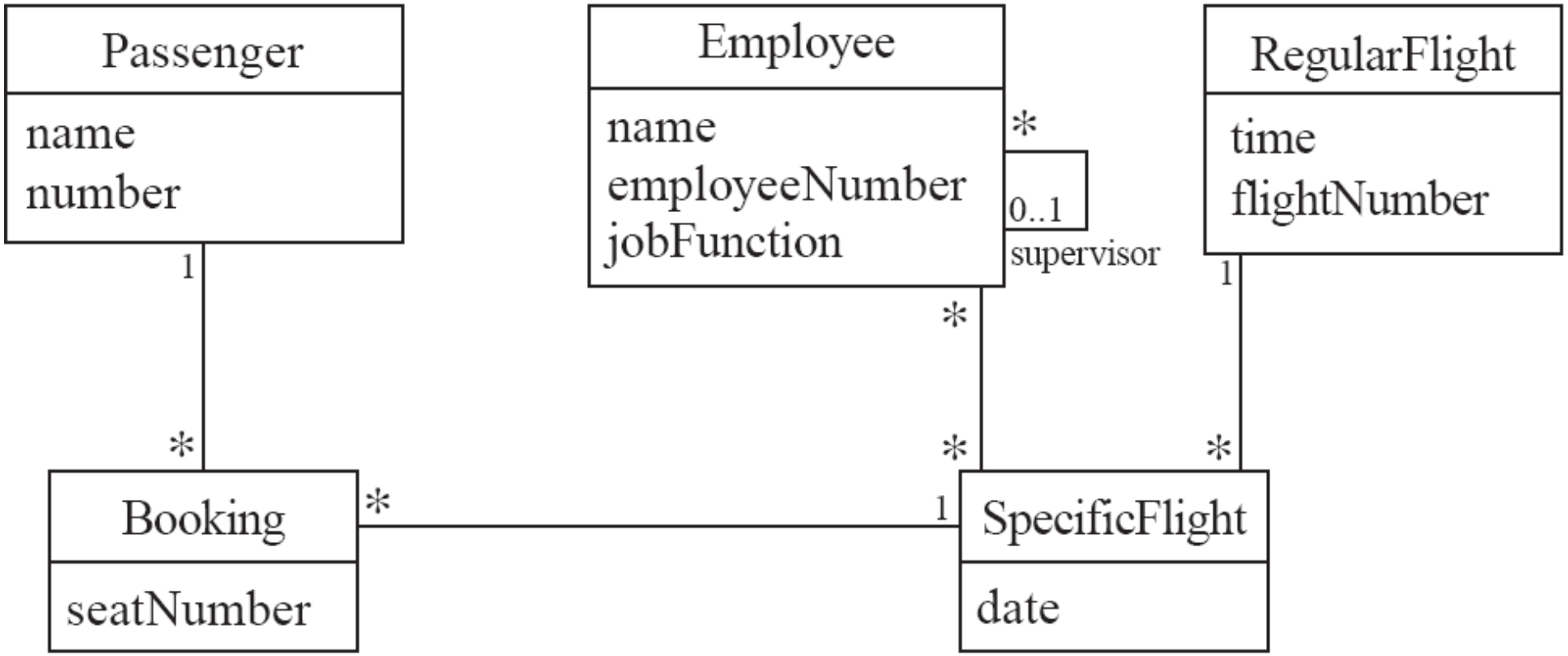
Too many, and inability to add more

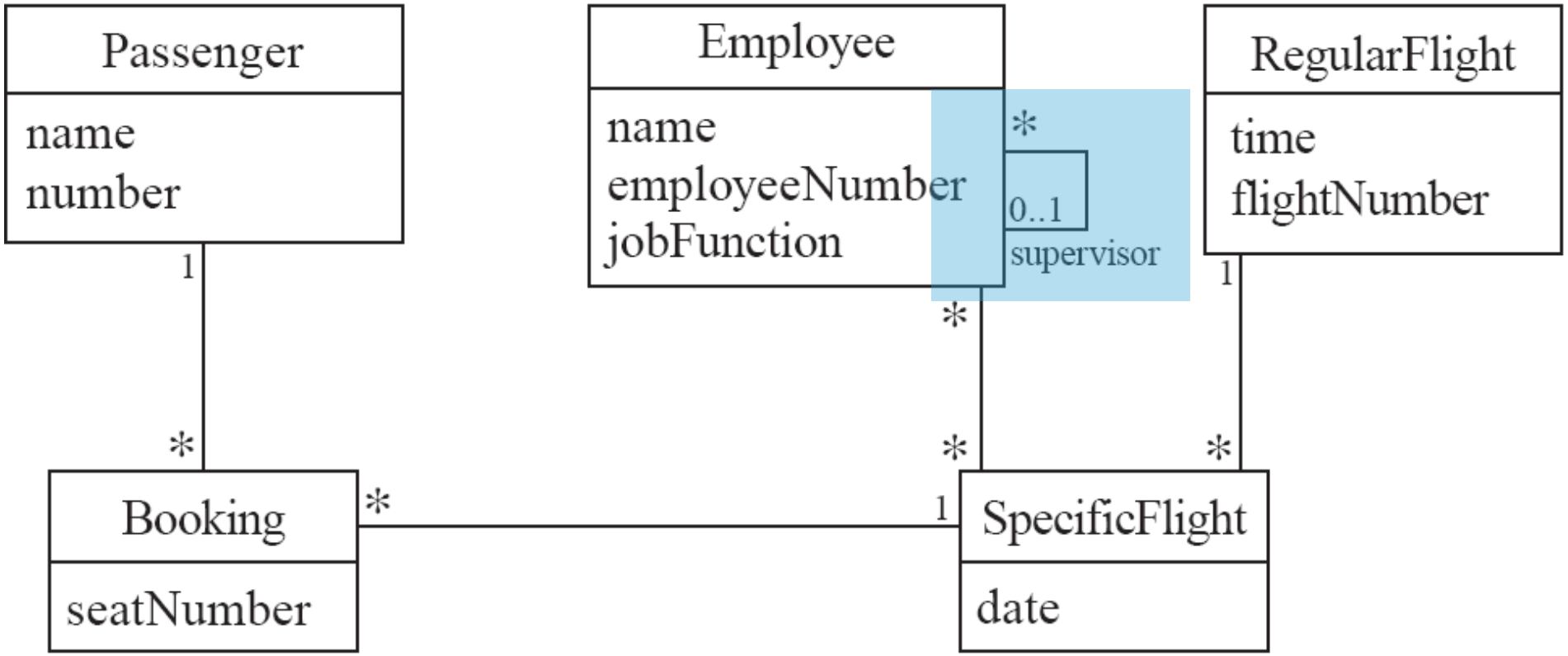
ALIDATING ATTRIBUTES (POSSIBLY AS AN ASSOCIATION)

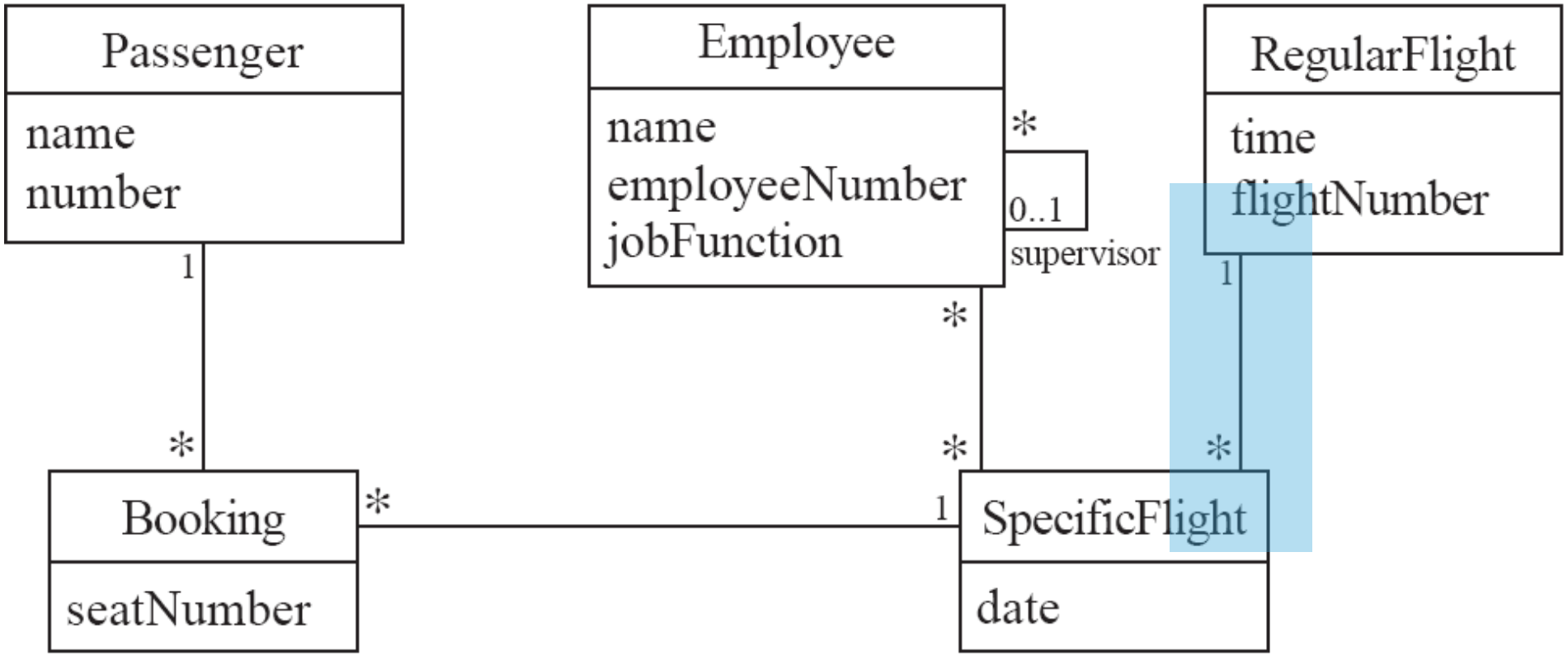


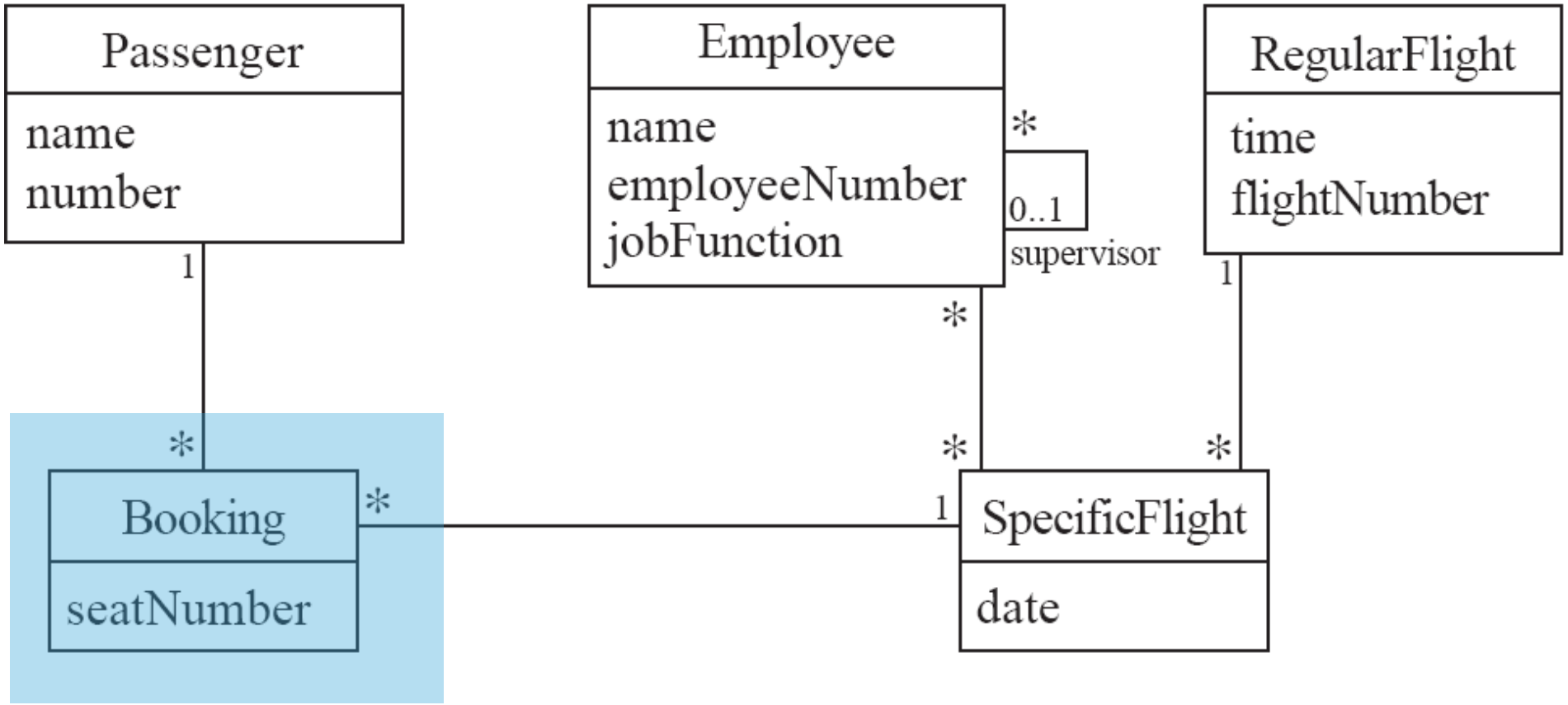
Better Solution:

Note that we also added *type* to distinguish (for example) home and business


Passenger
name
number
Employee
name
employeeNumber
jobFunction
RegularFlight
time
flightNumber
Booking
seatNumber
SpecificFlight
date
supervisor
0..1
1
*


Passenger
name
number
Employee
name
employeeNumber
jobFunction
RegularFlight
time
flightNumber
Booking
seatNumber
SpecificFlight
date
supervisor
0..1
1
*


Passenger
name
number
Employee
name
employeeNumber
jobFunction
RegularFlight
time
flightNumber
Booking
seatNumber
SpecificFlight
date
supervisor
0..1
1
*


Passenger
name
number
Employee
name
employeeNumber
jobFunction
RegularFlight
time
flightNumber
Booking
seatNumber
SpecificFlight
date
supervisor
0..1
1
*

Identifying generalizations

Bottom-up

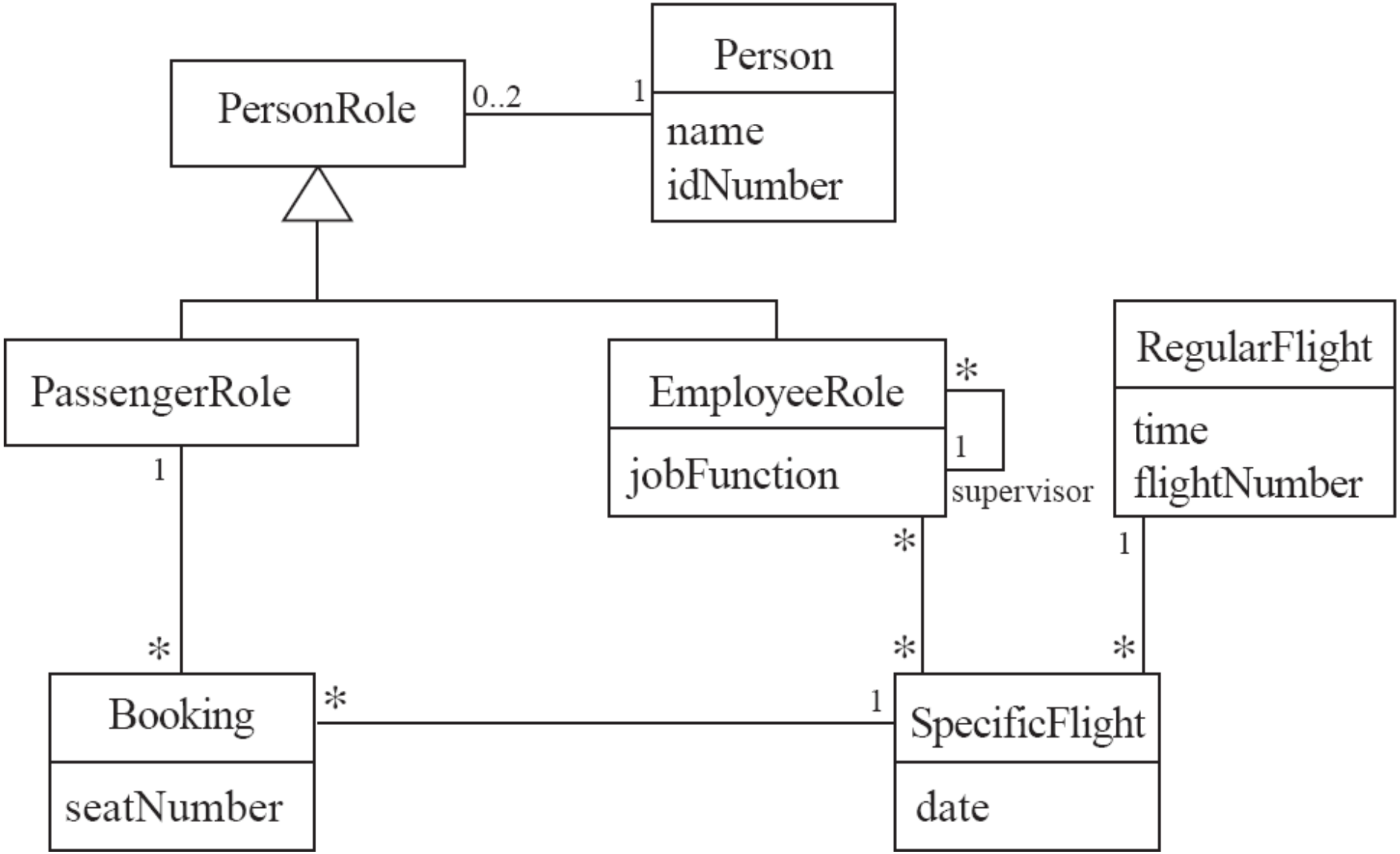
- Group together similar classes creating a new superclass

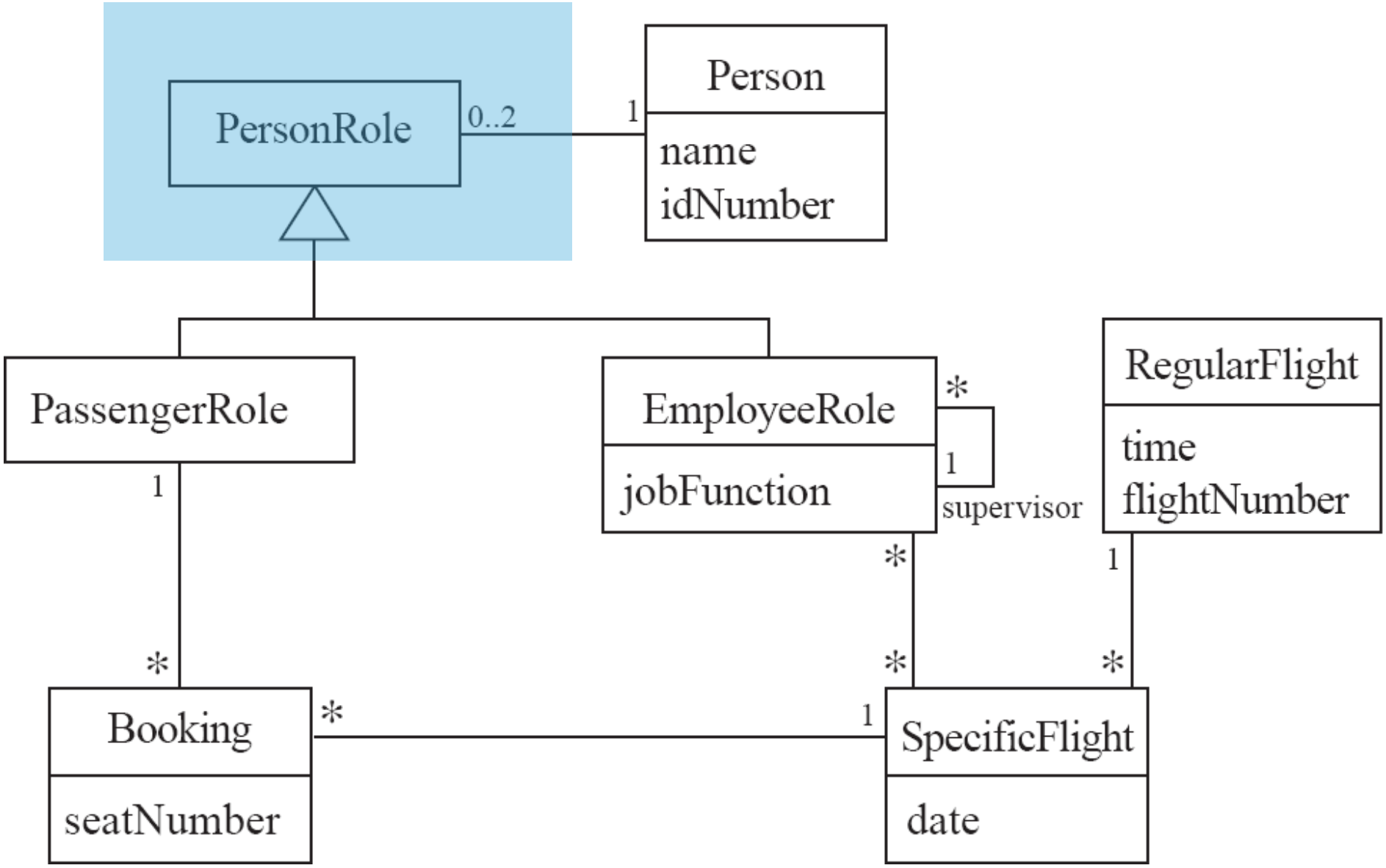
Top-down

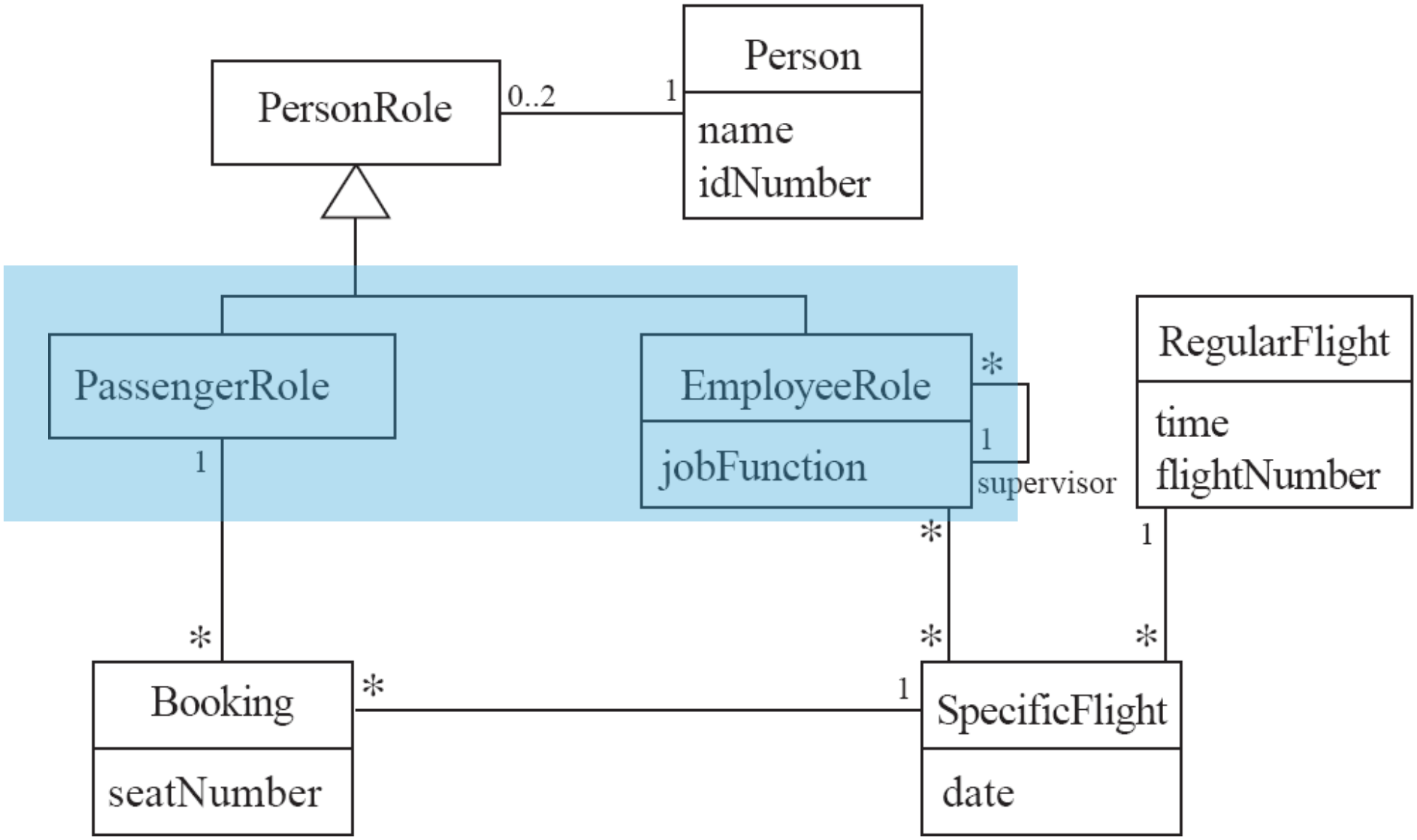
- Look for more general classes first, specialize them if needed

Create an interface, instead of a superclass if

- Classes are dissimilar except a few common operations
- Class already has a superclass
- Different implementations of same class


PersonRole
0..2
1
Person
name
idNumber
PassengerRole
EmployeeRole
jobFunction
*
1
supervisor
RegularFlight
time
flightNumber
1
*
*
*
1
*
*
Booking
seatNumber
*
1
SpecificFlight
date


PersonRole
0..2
1
Person
name
idNumber
PassengerRole
EmployeeRole
jobFunction
*
1
supervisor
RegularFlight
time
flightNumber
1
*
*
*
1
*
Booking
seatNumber
1
SpecificFlight
date
*
*


PersonRole
0..2
1
Person
name
idNumber
PassengerRole
EmployeeRole
jobFunction
*
1
supervisor
RegularFlight
time
flightNumber
1
*
*
*
*
1
Booking
seatNumber
*
1
SpecificFlight
date

Allocating responsibilities to classes

A responsibility is something that the system is required to do.

Each functional requirement must be attributed to one of the classes

- Responsibilities should be clearly related.
- Split the class if there are too many responsibilities
- If no responsibilities attached to it, needed?
- If responsibility has no class, missing class?

To determine responsibilities

- Perform use case analysis
- Look for … describing actions in the system description
  - verbs and
  - nouns

Categories of responsibilities

Setter / Getter

Managing Associations

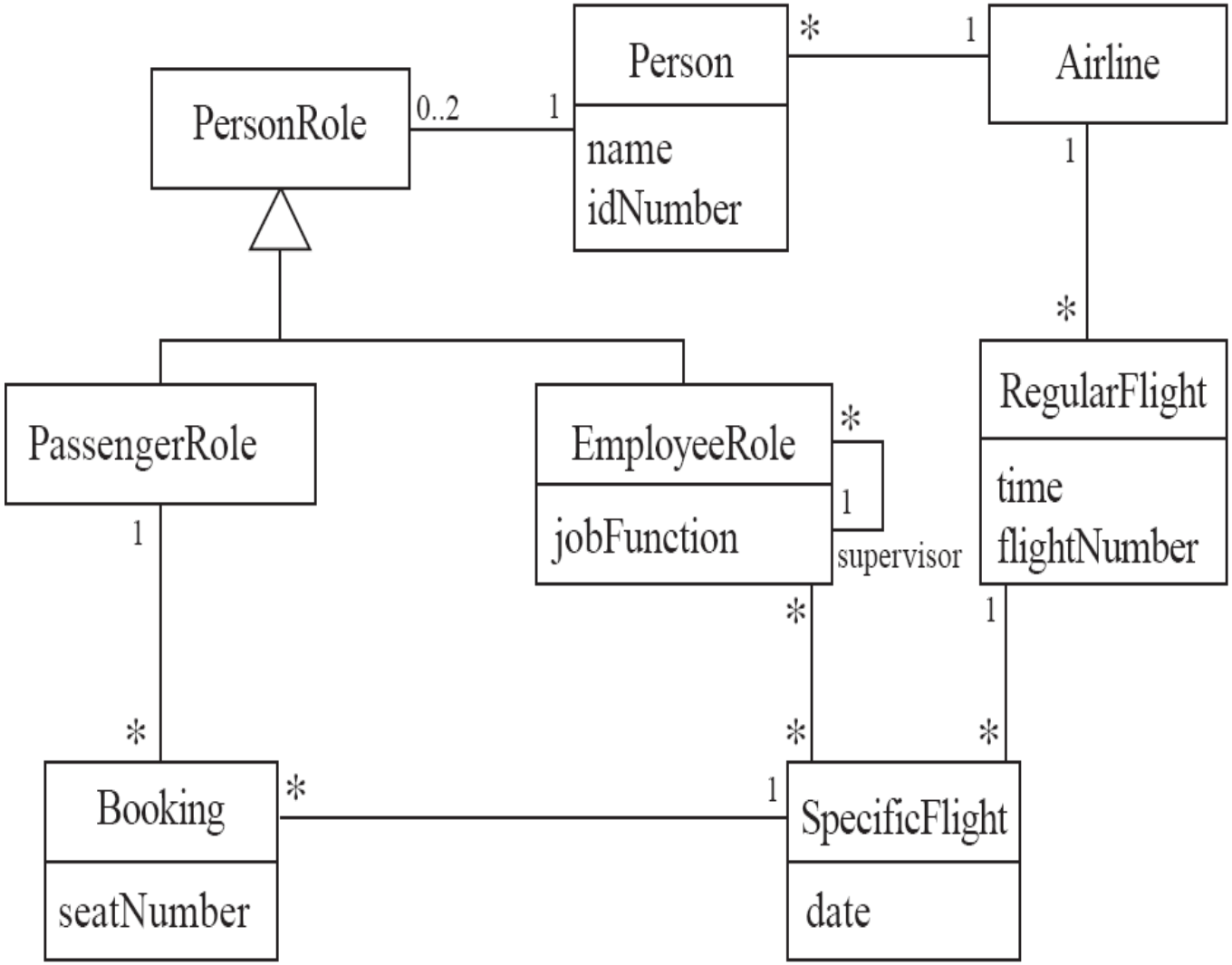
Initializing Objects

Data Transformation

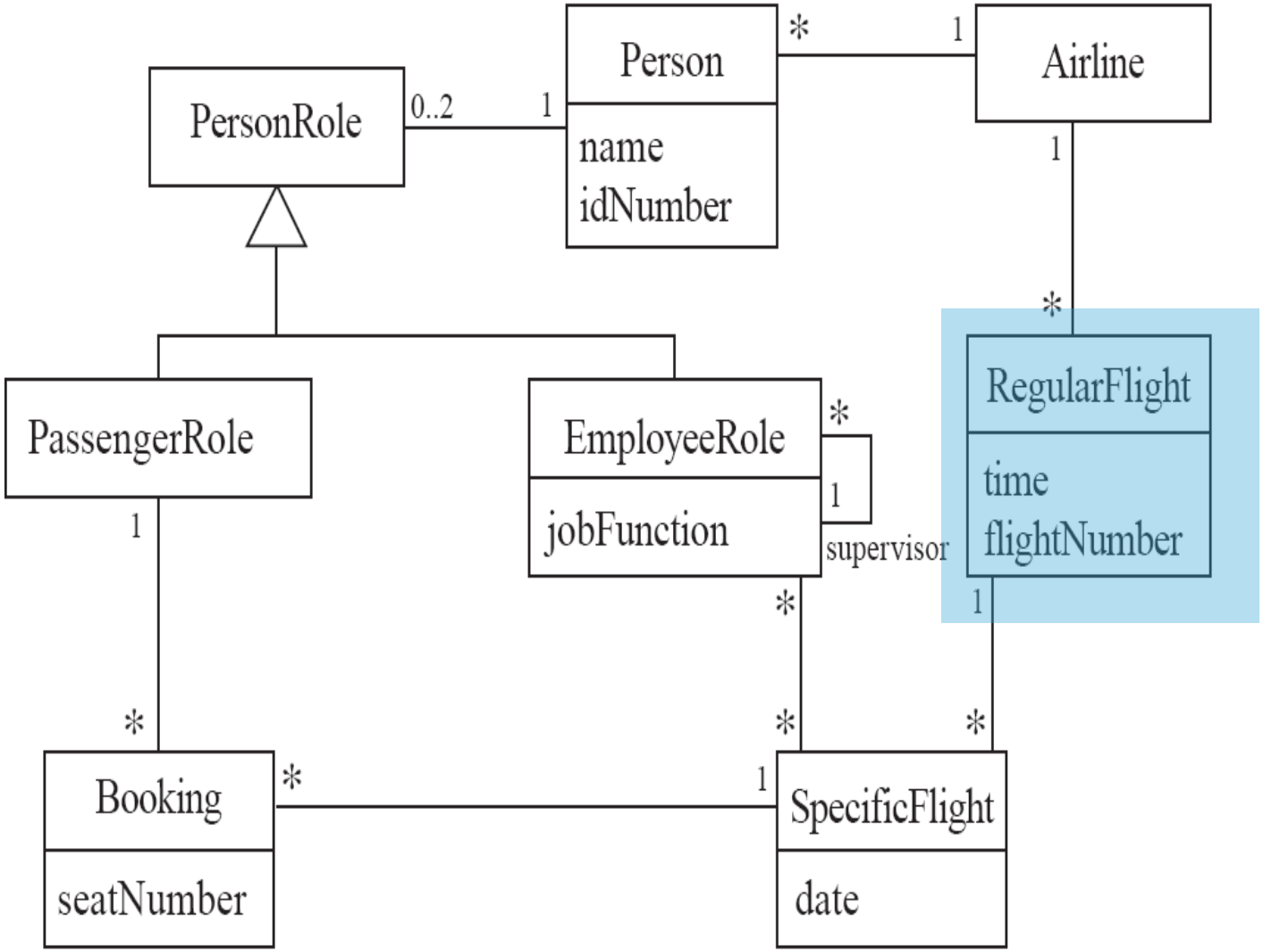
CRUD

Navigation and Searching

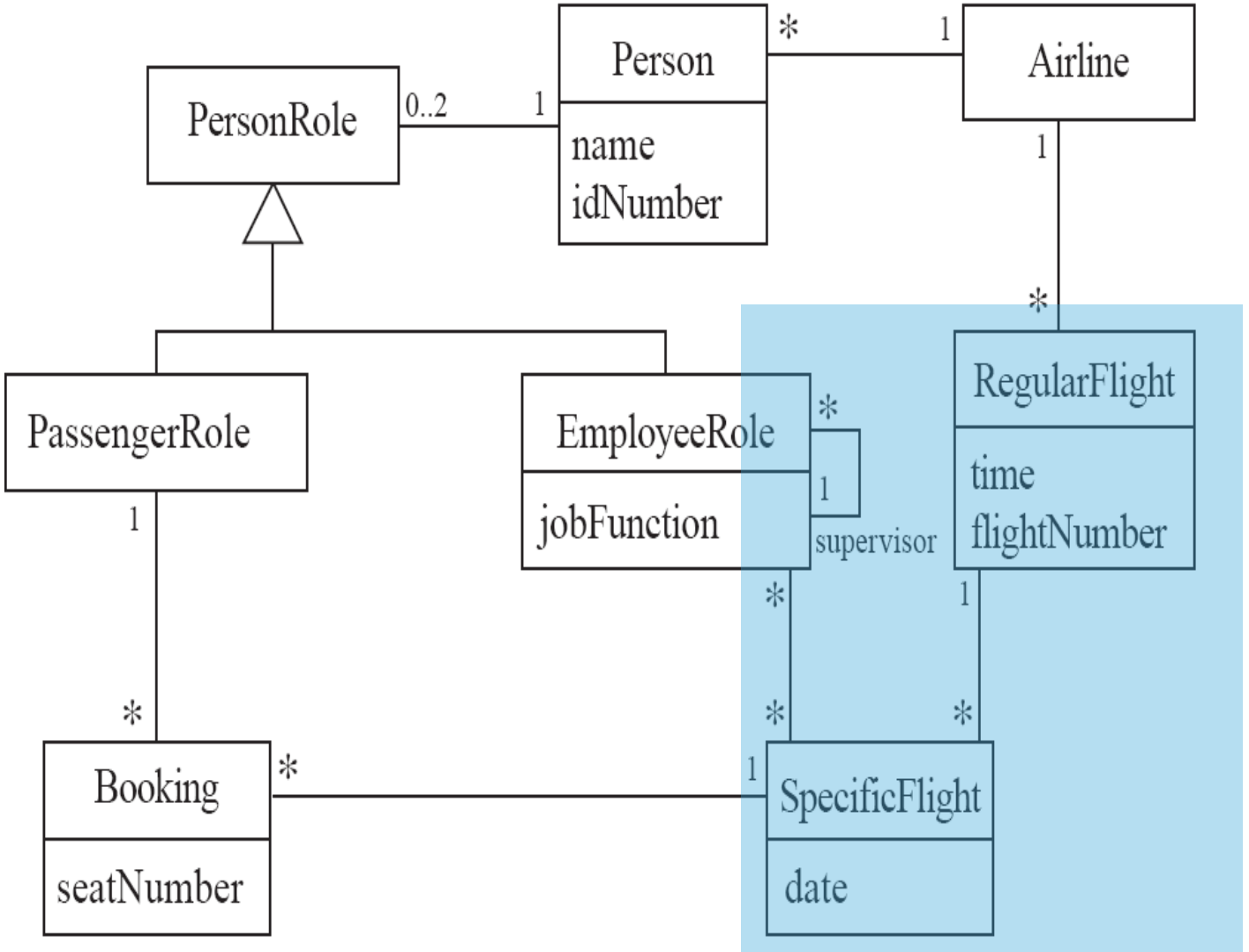
AN EXAMPLE (RESPONSIBILITIES)



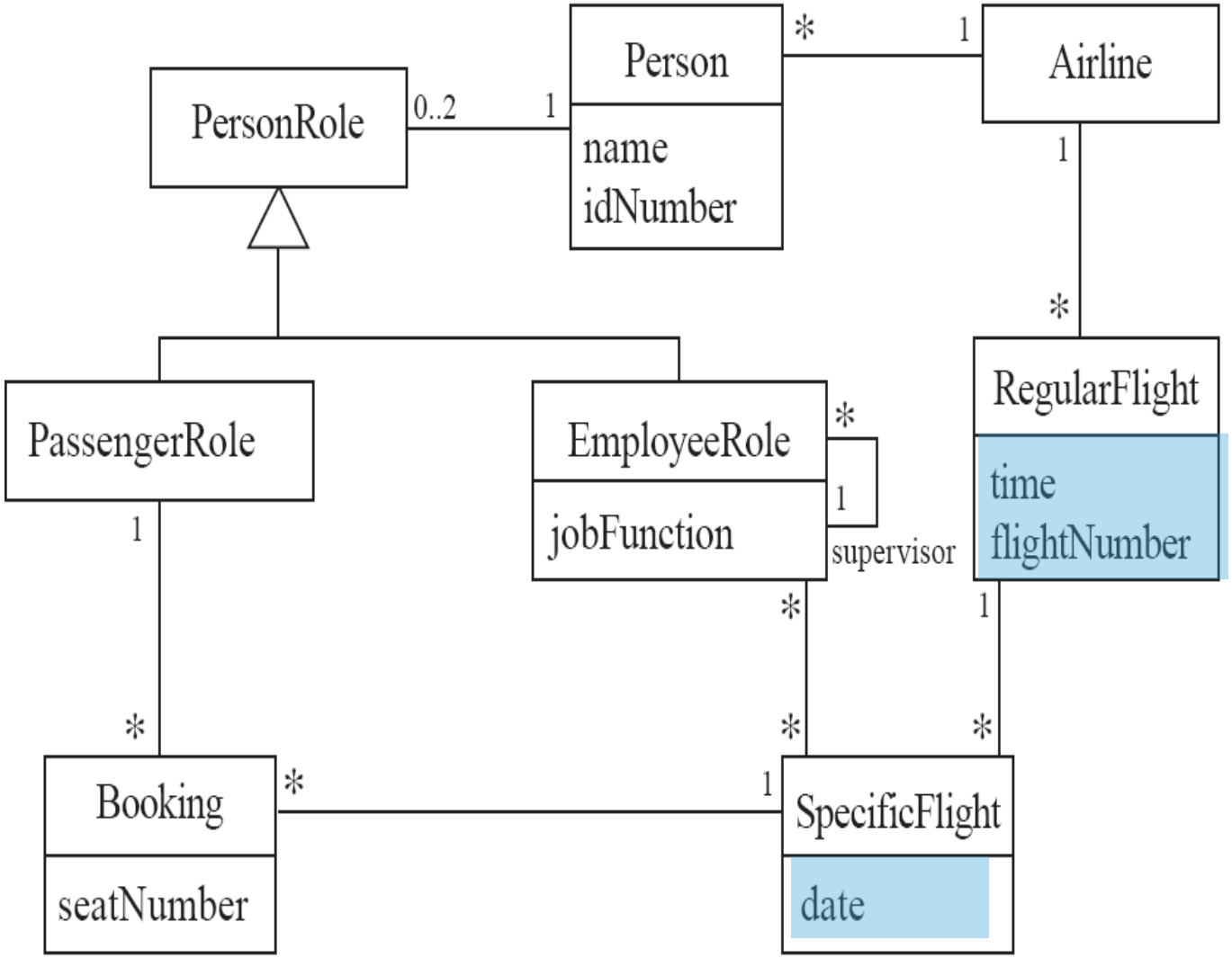
CREATE A NEW REGULAR FLIGHT



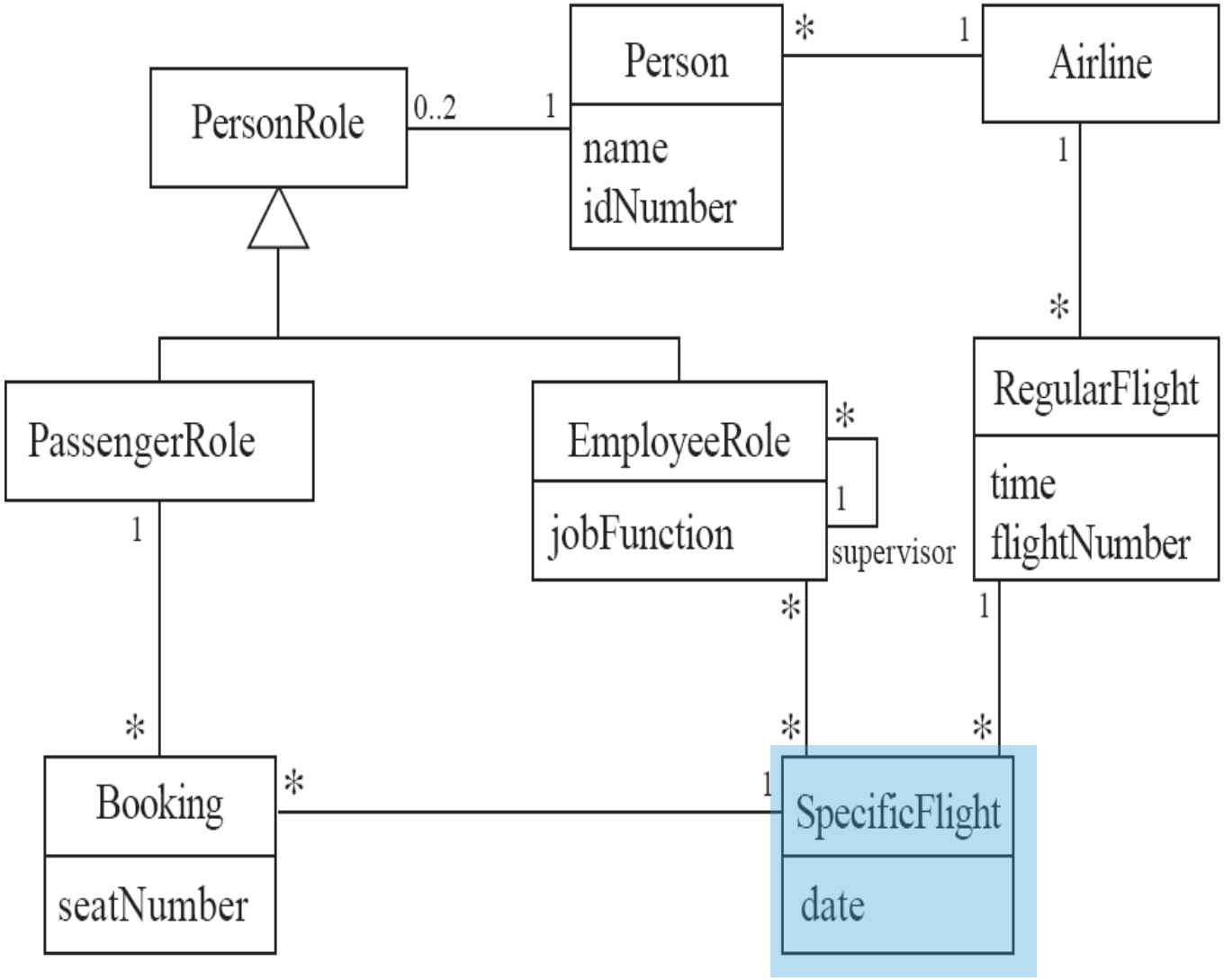
SEARCHING FOR A FLIGHT



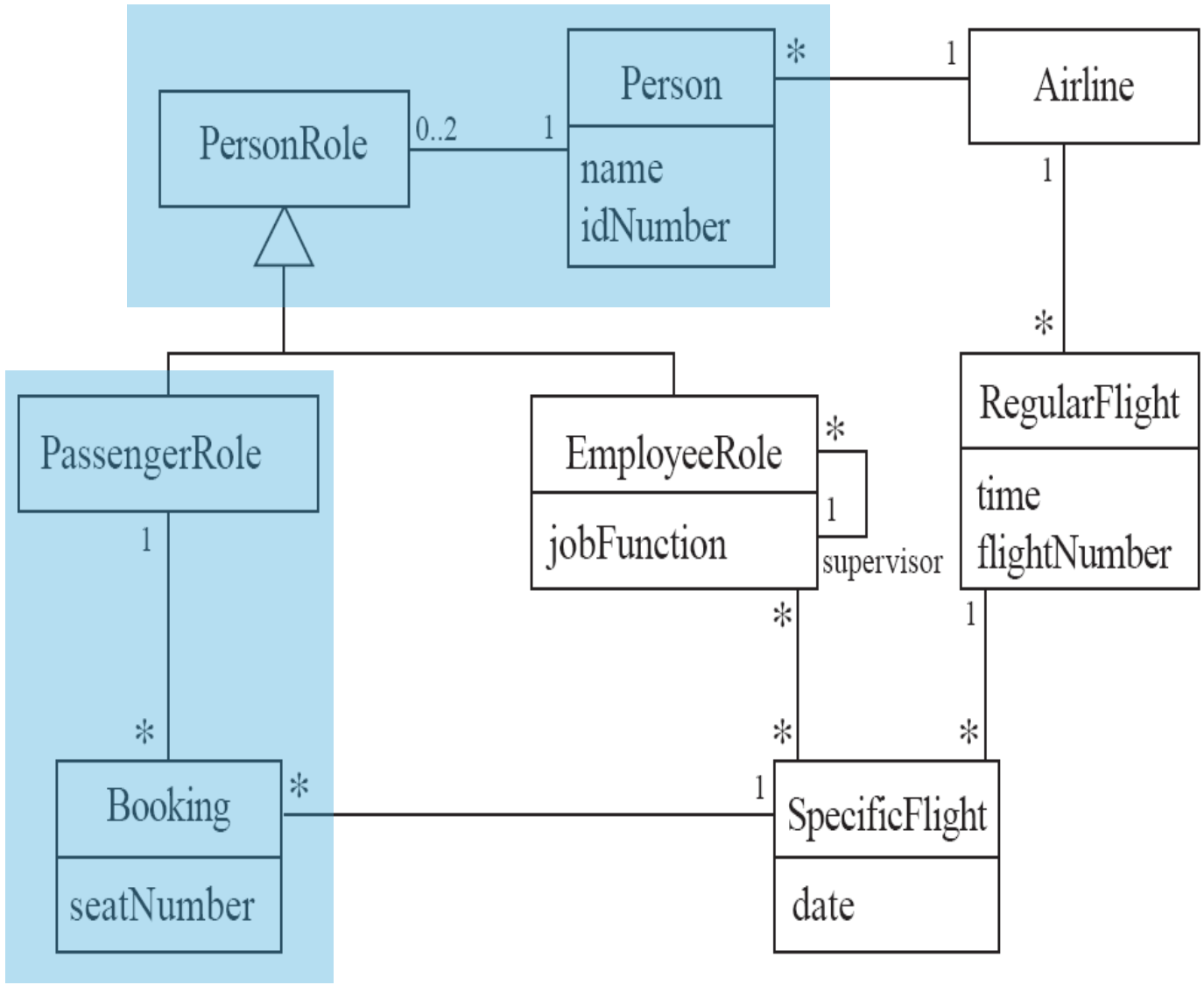
MODIFYING ATTRIBUTES OF A FLIGHT



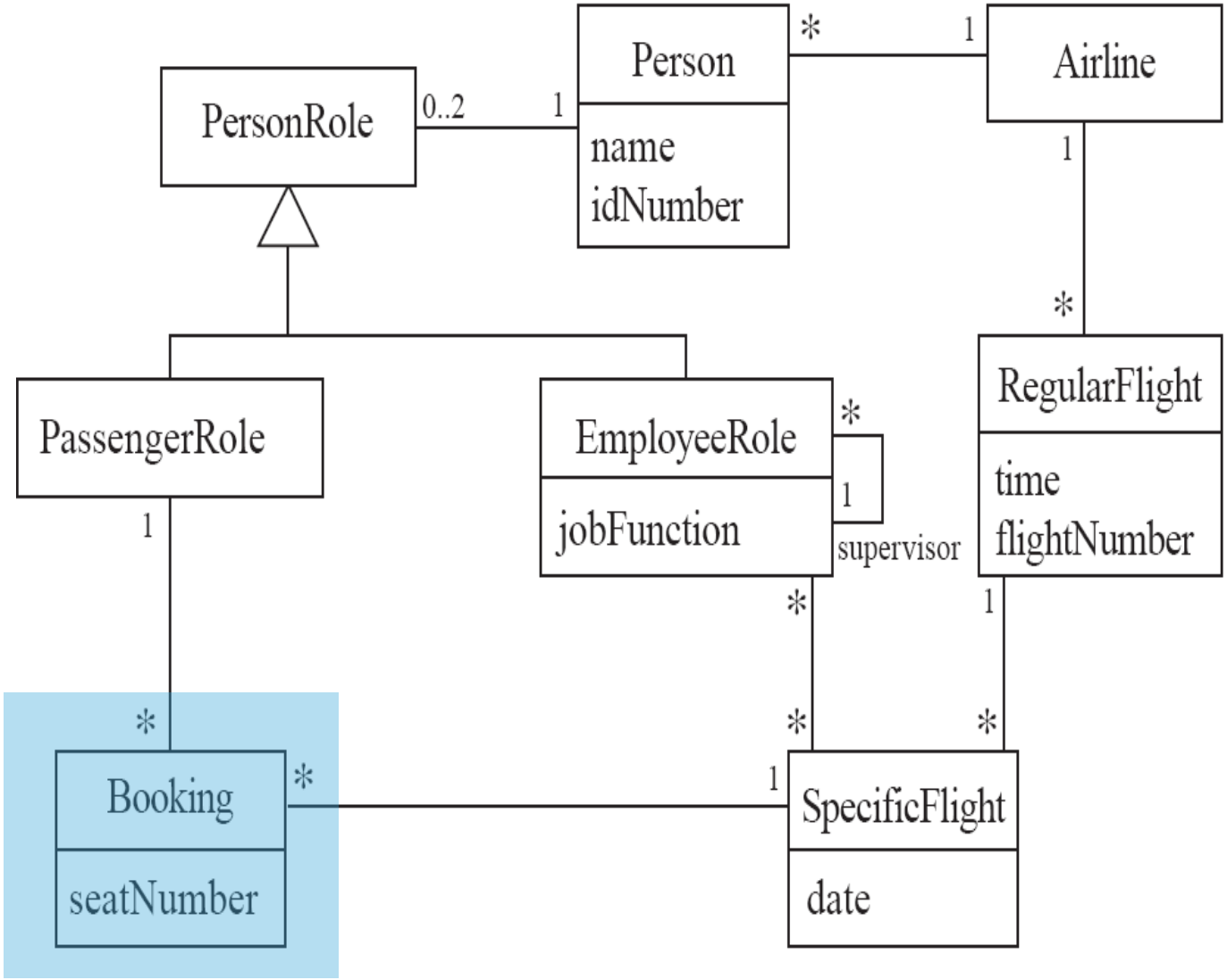
CREATING A SPECIFIC FLIGHT



BOOKING A PASSENGER



CANCELING A BOOKING



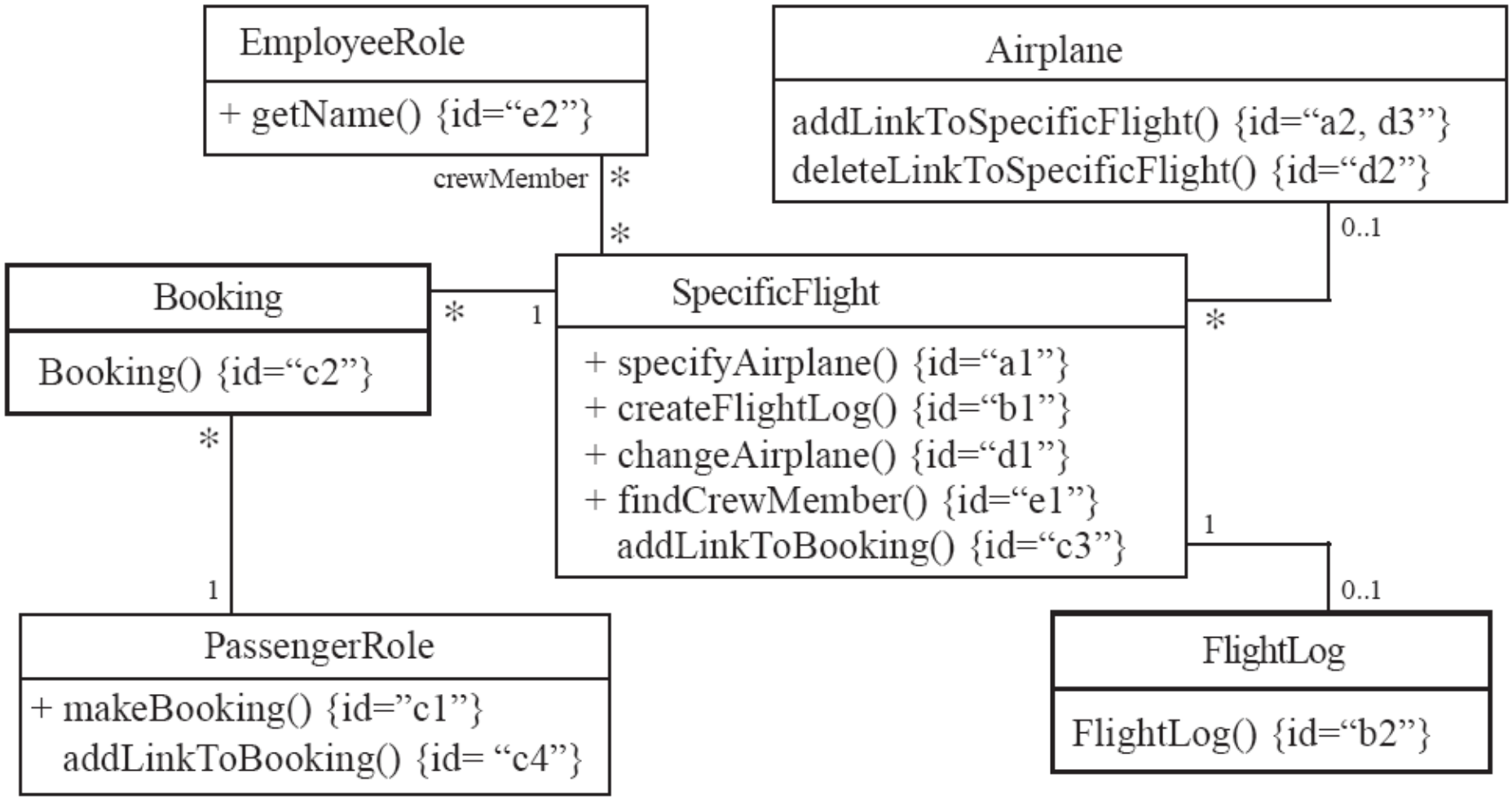
PROTOTYPING A CLASS DIAGRAM ON PAPER

- Write names of classes on small cards
- List the attributes and responsibilities on the cards
- If *too many* then split into multiple cards (classes)
- Arrange cards into a class diagram
- Lines among cards for associations and generalizations

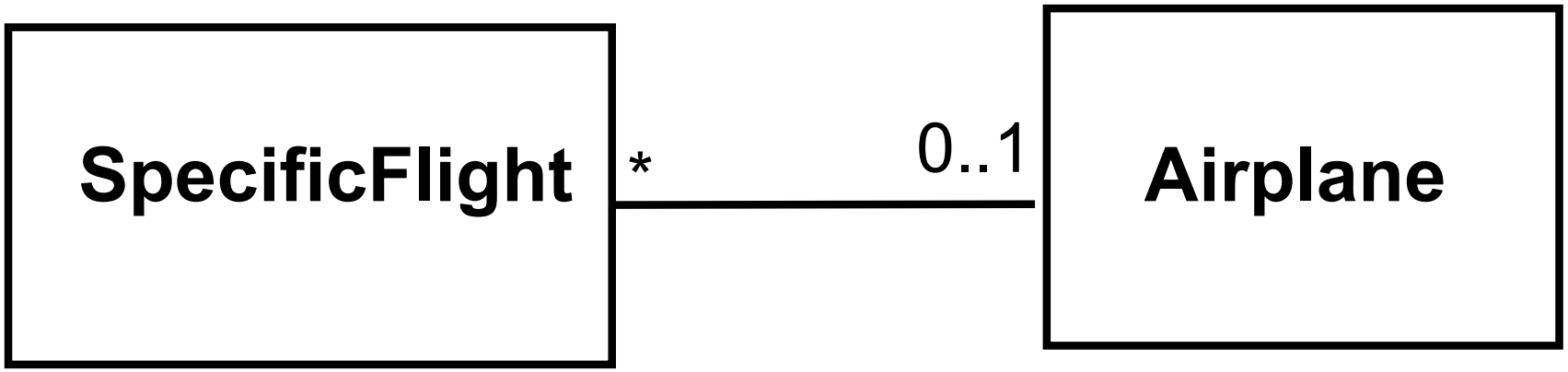
Operations are needed to realize the responsibilities of each class

- Possibly several operations per responsibility
- Declare main operations as public
- Collaborating methods as private (if possible)

AN EXAMPLE (CLASS COLLABORATION)



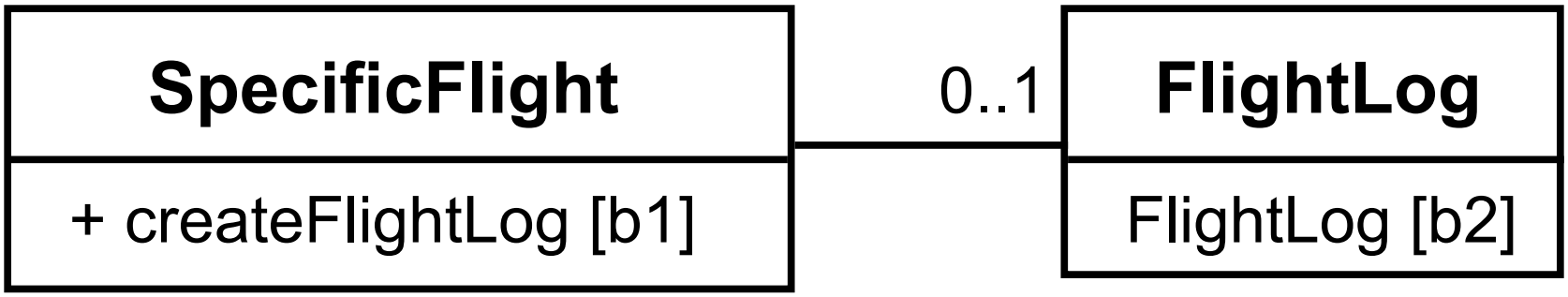
Making a bi-directional link between two existing objects



SpecificFlight makes a **one-directional link (public)** to Airplane then calls **(non-public) link from Airplane back** to instance of SpecificFlight

```
}
/* Code from template association_SetOneToMany */
public boolean setAirline(Airline aAirline)
{
  boolean wasSet = false;
  if (aAirline == null)
  {
    return wasSet;
  }

  Airline existingAirline = airline;
  airline = aAirline;
  if (existingAirline != null && !existingAirline.equals(aAirline))
  {
    existingAirline.removeRegularFlight(this);
  }
  airline.addRegularFlight(this);
  wasSet = true;
  return wasSet;
}
```

CREATING AN OBJECT AND LINKING IT TO AN EXISTING OBJECT

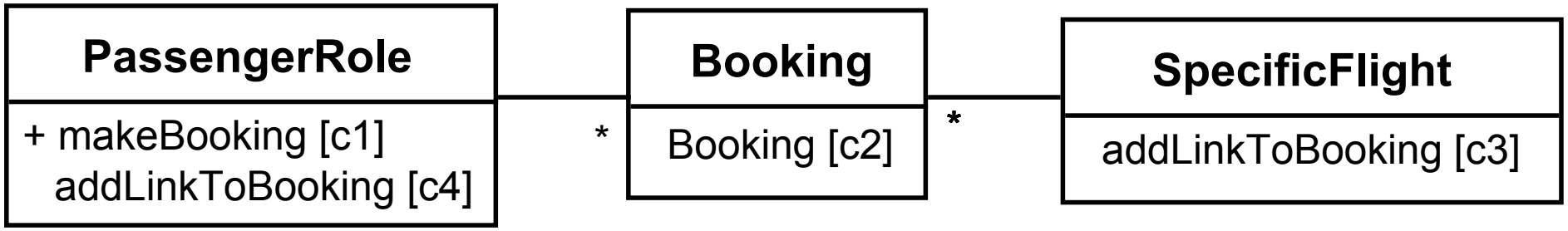


SpecificFlight calls the **constructor of FlightLog**, and then makes a **one-directional link to that instance**. The FlightLog's **constructor** makes a **one-directional link back** to SpecificFlight.

```
  //------------------------
  // CONSTRUCTOR
  //------------------------

  public FlightLog(SpecificFlight aSpecificFlight)
  {
    boolean didAddSpecificFlight = setSpecificFlight(aSpecificFlight);
```

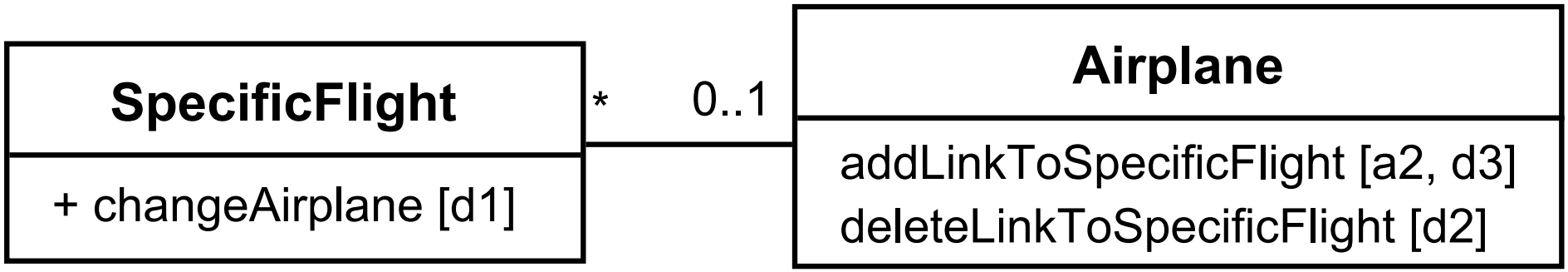
```
/* Code from template association_SetOneToOptionalOne */
public boolean setSpecificFlight(SpecificFlight aNewSpecificFlight)
{
  boolean wasSet = false;
  if (aNewSpecificFlight == null)
  {
    //Unable to setSpecificFlight to null, as flightLog must always be associated to a specificFlight
    return wasSet;
  }

  FlightLog existingFlightLog = aNewSpecificFlight.getFlightLog();
  if (existingFlightLog != null && !equals(existingFlightLog))
  {
    //Unable to setSpecificFlight, the current specificFlight already has a flightLog, which would be orphaned if it were re-assigned
    return wasSet;
  }

  SpecificFlight anOldSpecificFlight = specificFlight;
  specificFlight = aNewSpecificFlight;
  specificFlight.setFlightLog(this);

  if (anOldSpecificFlight != null)
  {
    anOldSpecificFlight.setFlightLog(null);
  }
  wasSet = true;
  return wasSet;
}
```

CREATING AN ASSOCIATION CLASS, GIVEN TWO EXISTING OBJECTS



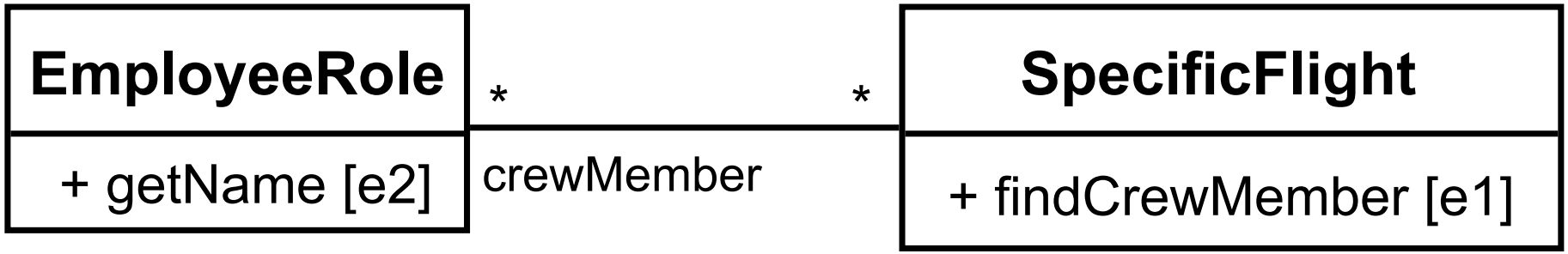
1. (public) The instance of PassengerRole
   - calls the constructor of Booking (operation 2).
2. (non-public) Class Booking's constructor, among its other actions
   - makes a one-directional link back to the instance of PassengerRole
   - makes a one-directional link to the instance of SpecificFlight
   - calls operations 3 and 4.
3. (non-public) The instance of SpecificFlight
   - makes a one-directional link to the instance of Booking.
4. (non-public) The instance of PassengerRole
   - makes a one-directional link to the instance of Booking.

CHANGING THE DESTINATION OF A LINK



1. (public) The instance of SpecificFlight
   - deletes the link to airplane1
   - makes a one-directional link to airplane2
   - calls operation 2
   - then calls operation 3.
2. (non-public) airplane1
   - deletes its one-directional link to the instance of SpecificFlight.
3. (non-public) airplane2
   - makes a one-directional link to the instance of SpecificFlight.

SEARCHING FOR AN ASSOCIATED INSTANCE



1. (public) The instance of SpecificFlight
   - creates an Iterator over all the crewMember links of the SpecificFlight
   - for each of them call operation 2, until it finds a match.
2. (may be public) The instance of EmployeeRole returns its name.

Implementing Class Diagrams in Java

Model	Java
Attributes	Instance variables
Generalizations	Using **extends**
Intefaces	Using **implements**
Associations	Instance variables

Implementing Associations

- Two-way association into 2 one-way associations
  - Each class has an instance variable
- One-way association when other end is ‘one’ or ‘optional’
  - Declare a variable of that class (a reference)
- One-way when multiplicity is ‘many’
  - use a collection class implementing List, such as Vector

```
class RegularFlight
{

    private List specificFlights;
    ...
    // Method that has primary responsibility
    public void addSpecificFlight(Calendar aDate)  {
        SpecificFlight newSpecificFlight;
        newSpecificFlight = new SpecificFlight(aDate, this);
        specificFlights.add(newSpecificFlight);
    }
    ...
}
```

```
class SpecificFlight
{

    private Calendar date;
    private RegularFlight regularFlight;

    ...

   // Constructor that should only be called from
   // addSpecificFlight
   SpecificFlight( Calendar aDate, RegularFlight aRegularFlight) {
       date = aDate;
       regularFlight = aRegularFlight;
    }
}
```

DIFFICULTIES AND RISKS WHEN

CREATING CLASS DIAGRAMS

Modeling is particularly difficult skill

- Even excellent programmers have difficulty thinking at the appropriate level of abstraction
- Education traditionally focus more on design and programming than modeling

Resolution

- Adequate training
- Experienced modeler as team member
- Review all models thoroughly

References

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