



# Object-Oriented Software Engineering

## Practical Software Development using UML and Java

### Chapter 5: Modelling with Classes

#### Lecture 5

[www.lloseng.com](http://www.lloseng.com)

## 5.1 What is UML?

**The Unified Modelling Language is a standard graphical language for modelling object oriented software**

- At the end of the 1980s and the beginning of 1990s, the first object-oriented development processes appeared
- The proliferation of methods and notations tended to cause considerable confusion
- Two important methodologists Rumbaugh and Booch decided to merge their approaches in 1994.
  - They worked together at the Rational Software Corporation
- In 1995, another methodologist, Jacobson, joined the team
  - His work focused on use cases
- In 1997 the Object Management Group (OMG) started the process of UML standardization

# UML diagrams

- Class diagrams
  - describe classes and their relationships
- Interaction diagrams
  - show the behaviour of systems in terms of how objects interact with each other
- State diagrams and activity diagrams
  - show how systems behave internally
- Component and deployment diagrams
  - show how the various components of systems are arranged logically and physically

227

# UML features

- It has detailed *semantics*
- It has *extension* mechanisms
- It has an associated textual language
  - *Object Constraint Language* (OCL)

**The objective of UML is to assist in software development**

— It is not a *methodology*

228

# What constitutes a good model?

## **A model should**

- use a standard notation
- be understandable by clients and users
- lead software engineers to have insights about the system
- provide abstraction

Models are used:

- to help create designs
- to permit analysis and review of those designs.
- as the core documentation describing the system.

229

## 5.2 Essentials of UML Class Diagrams

*The main symbols shown on class diagrams are:*

- *Classes*
  - represent the types of data themselves
- *Associations*
  - represent linkages between instances of classes
- *Attributes*
  - are simple data found in classes and their instances
- *Operations*
  - represent the functions performed by the classes and their instances
- *Generalizations*
  - group classes into inheritance hierarchies

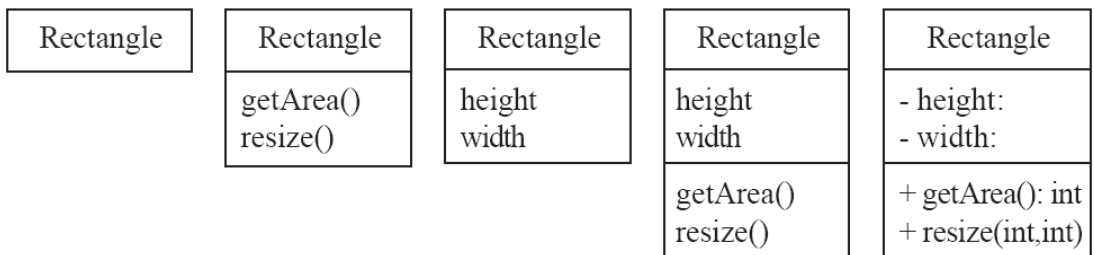
230

# Classes

**A class is simply represented as a box with the name of the class inside**

- The diagram may also show the attributes and operations
- The complete signature of an operation is:

operationName(parameterName: parameterType ...): returnType

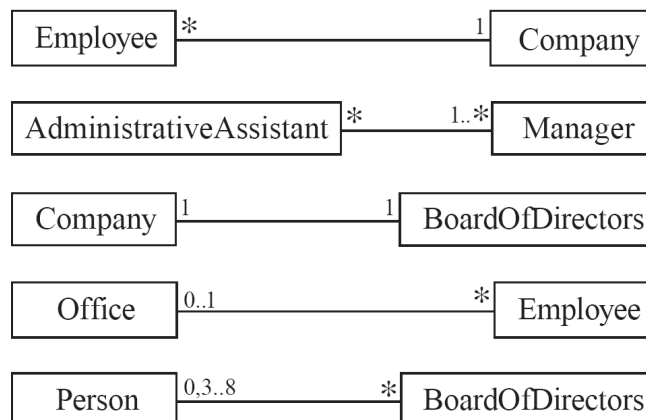


231

## 5.3 Associations and Multiplicity

**An association is used to show how two classes are related to each other**

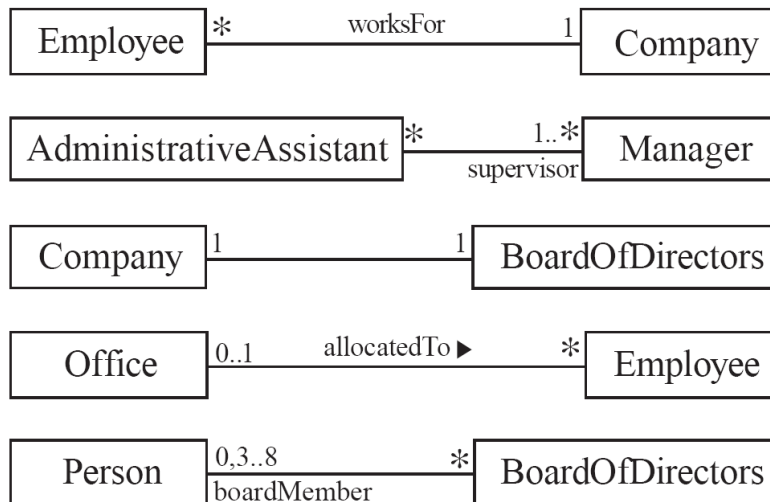
- Symbols indicating *multiplicity* are shown at each end of the association



232

# Labelling associations

- Each association can be labelled, to make explicit the nature of the association

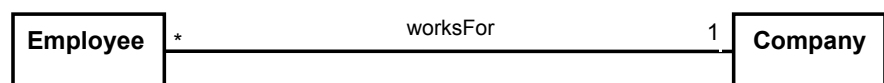


233

## Analyzing and validating associations

- **Many-to-one**

- A company has many employees,
- An employee can only work for one company.
  - This company will not store data about the moonlighting activities of employees!
- A company can have zero employees
  - E.g. a 'shell' company
- It is not possible to be an employee unless you work for a company

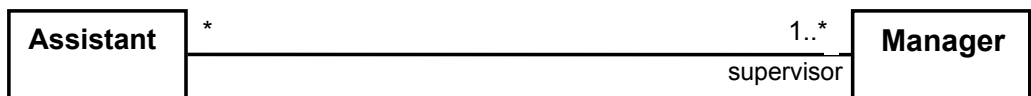


234

# Analyzing and validating associations

- **Many-to-many**

- An assistant can work for many managers
- A manager can have many assistants
- Assistants can work in pools
- Managers can have a group of assistants
- Some managers might have zero assistants.
- Is it possible for an assistant to have, perhaps temporarily, zero managers?

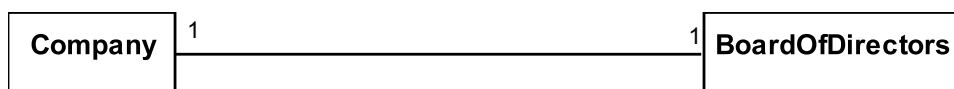


235

# Analyzing and validating associations

- **One-to-one**

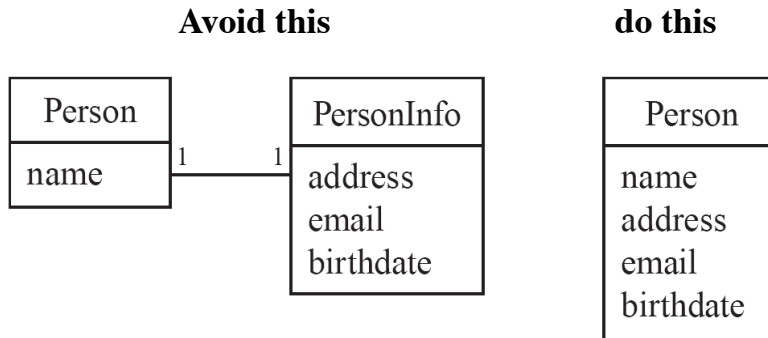
- For each company, there is exactly one board of directors
- A board is the board of only one company
- A company must always have a board
- A board must always be of some company



236

# Analyzing and validating associations

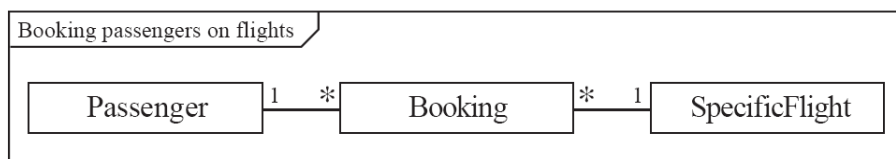
## Avoid unnecessary one-to-one associations



237

## A more complex example

- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could *never* involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking

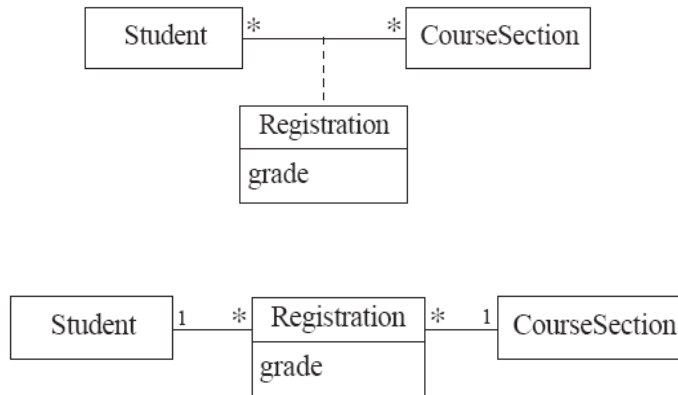


- The *frame* around this diagram is an optional feature that any UML 2.0 may possess.

238

# Association classes

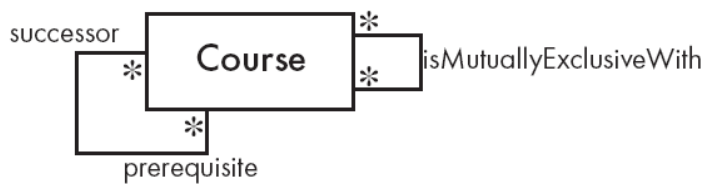
- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent



239

# Reflexive associations

- It is possible for an association to connect a class to itself



240



# Directionality in associations

- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end



241

## 5.4 Generalization

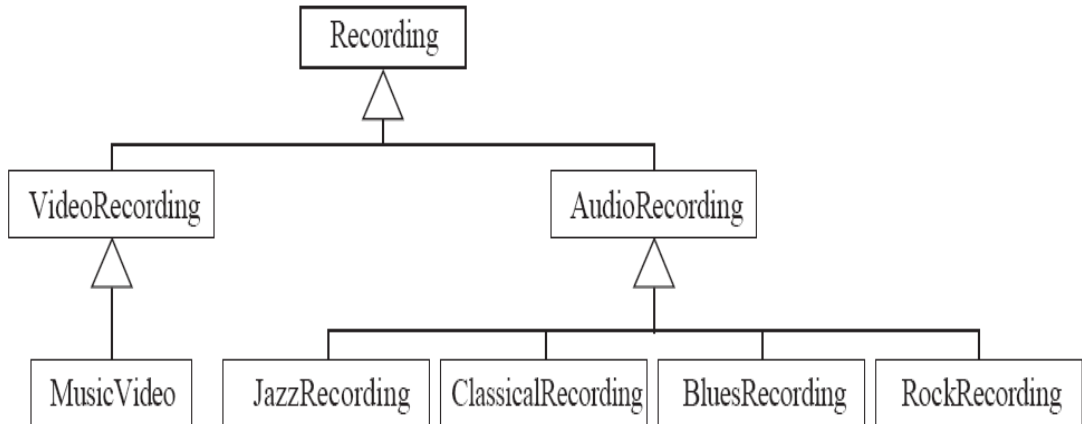
### Specializing a superclass into two or more subclasses

- A *generalization set* is a labeled group of generalizations with a common superclass
- The label (sometimes called the *discriminator*) describes the criteria used in the specialization



242

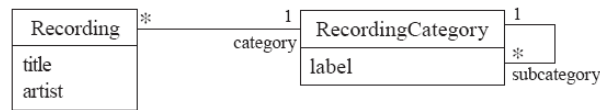
# Avoiding unnecessary generalizations



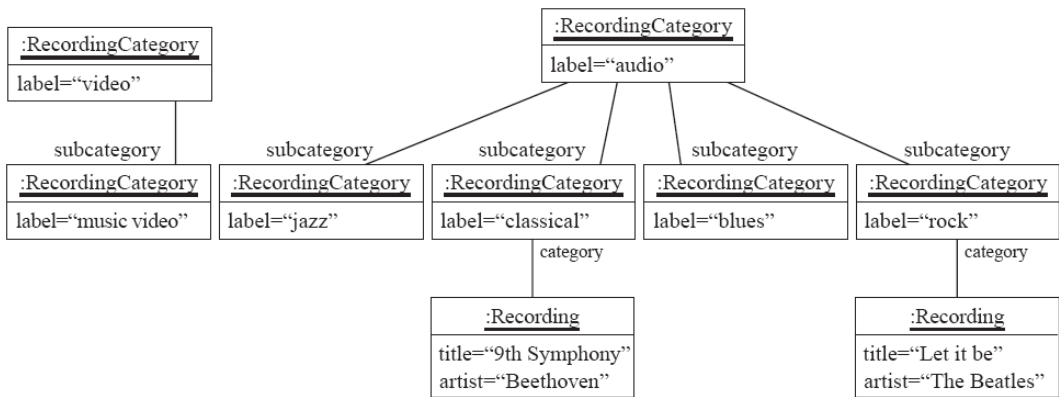
Inappropriate hierarchy of classes, which should be instances

243

# Avoiding unnecessary generalizations (cont)



(a)



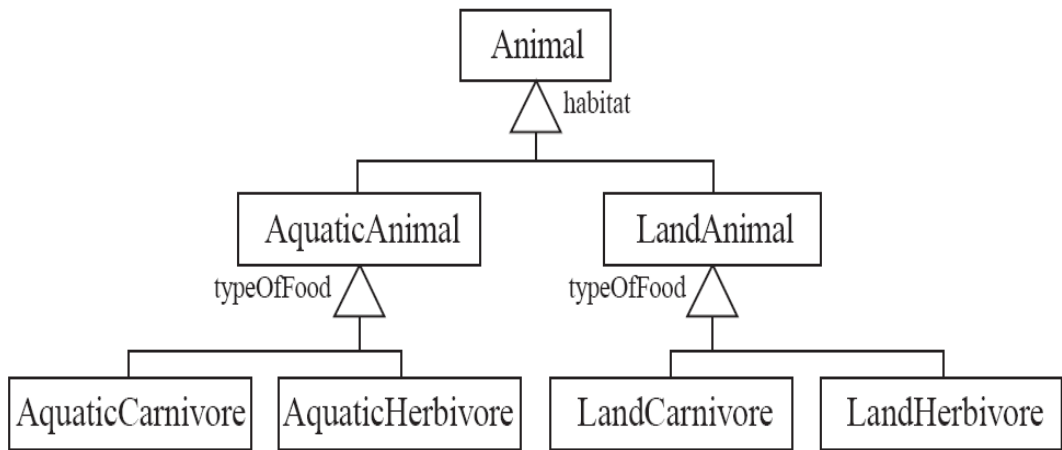
(b)

Improved class diagram, with its corresponding instance diagram

244

# Handling multiple discriminators

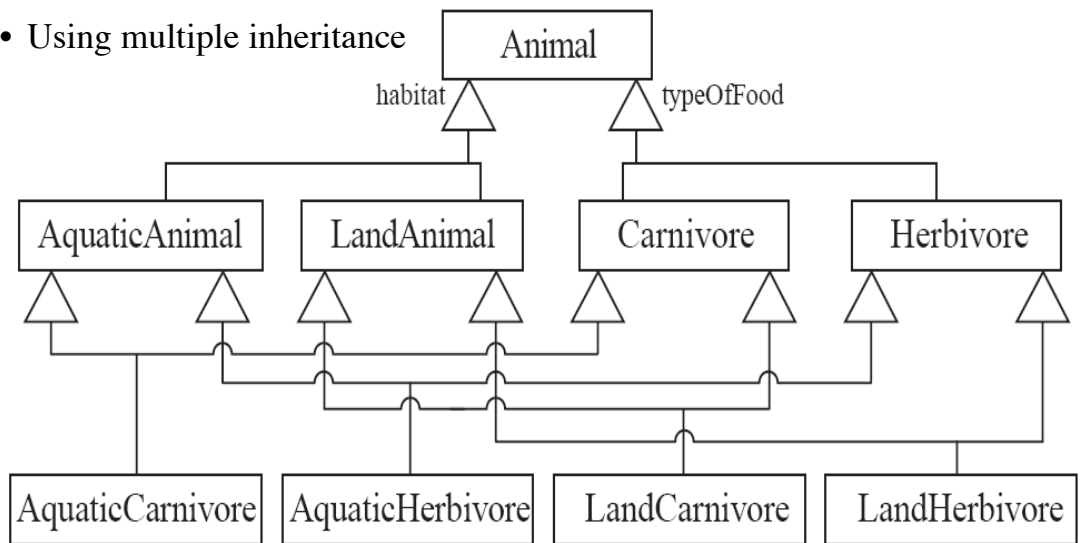
- Creating higher-level generalization



245

# Handling multiple discriminators

- Using multiple inheritance

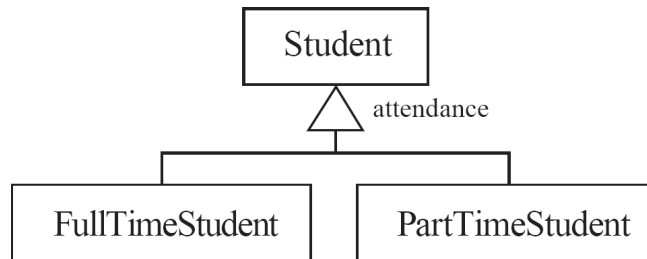


- Using the Player-Role pattern (in Chapter 6)

246

# Avoiding having instances change class

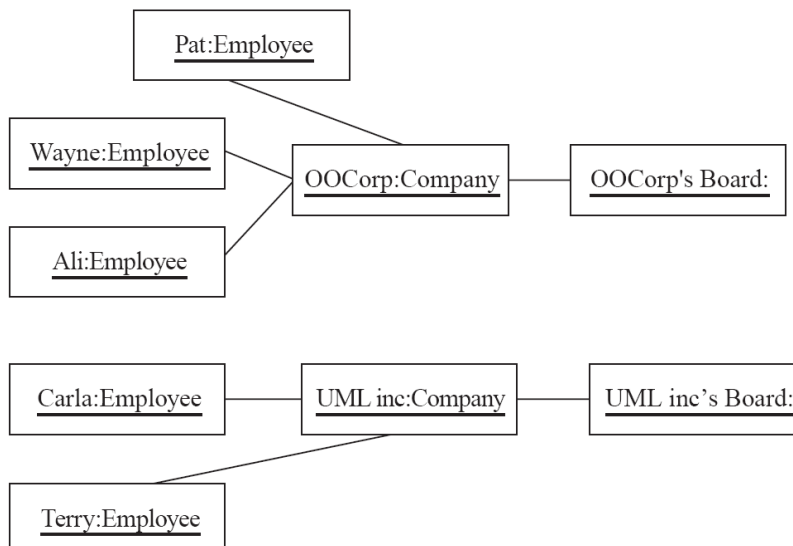
- An instance should never need to change class



247

## 5.5 Object Diagrams

- A *link* is an instance of an association
  - In the same way that we say an object is an instance of a class



248

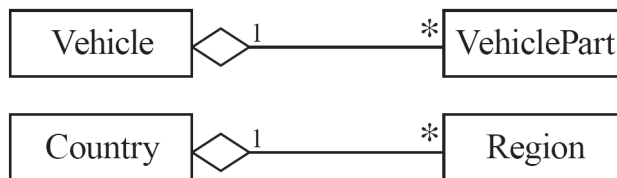
# Associations versus generalizations in object diagrams

- 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

249

## 5.6 More Advanced Features: Aggregation

- Aggregations are special associations that represent 'part-whole' relationships.
  - The 'whole' side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named `isPartOf`



250

# When to use an aggregation

**As a general rule, you can mark an association as an aggregation if the following are true:**

- 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

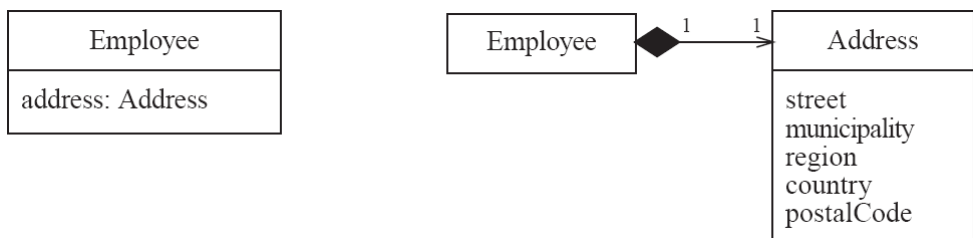
251

## Composition

- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well

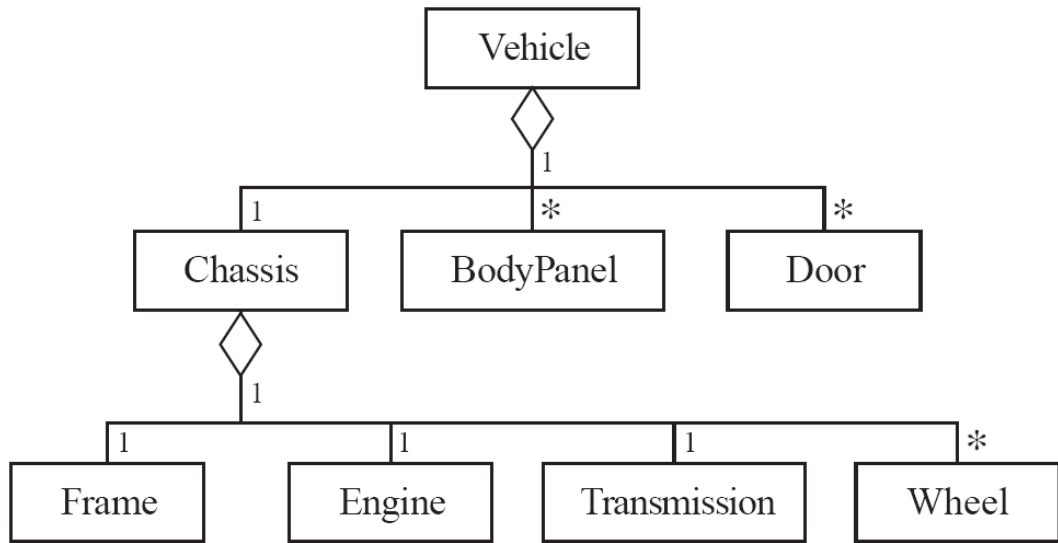


- Two alternatives for addresses



252

# Aggregation hierarchy



253

## Propagation

- A mechanism where an operation in an aggregate is implemented by having the aggregate perform that operation on its parts
- At the same time, properties of the parts are often propagated back to the aggregate
- 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

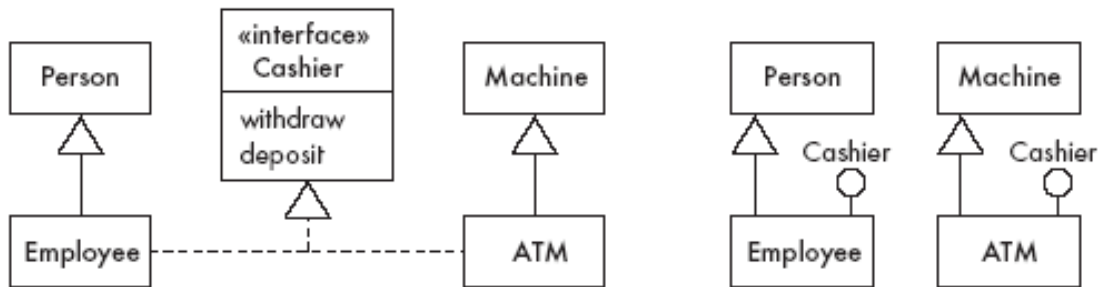


254

# Interfaces

**An interface describes a *portion of the visible behaviour* of a set of objects.**

- An *interface* is similar to a class, except it lacks instance variables and implemented methods



255

## Notes and descriptive text

- **Descriptive text and other diagrams**
  - 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

256



## 5.7 Object Constraint Language (OCL)

**OCL is a *specification* language designed to formally specify constraints in software modules**

- An OCL expression simply specifies a logical fact (a constraint) about the system that must remain **true**
- A constraint cannot have any side-effects
  - it cannot compute a non-Boolean result nor modify any data.
- OCL statements in class diagrams can specify what the values of attributes and associations must be

257

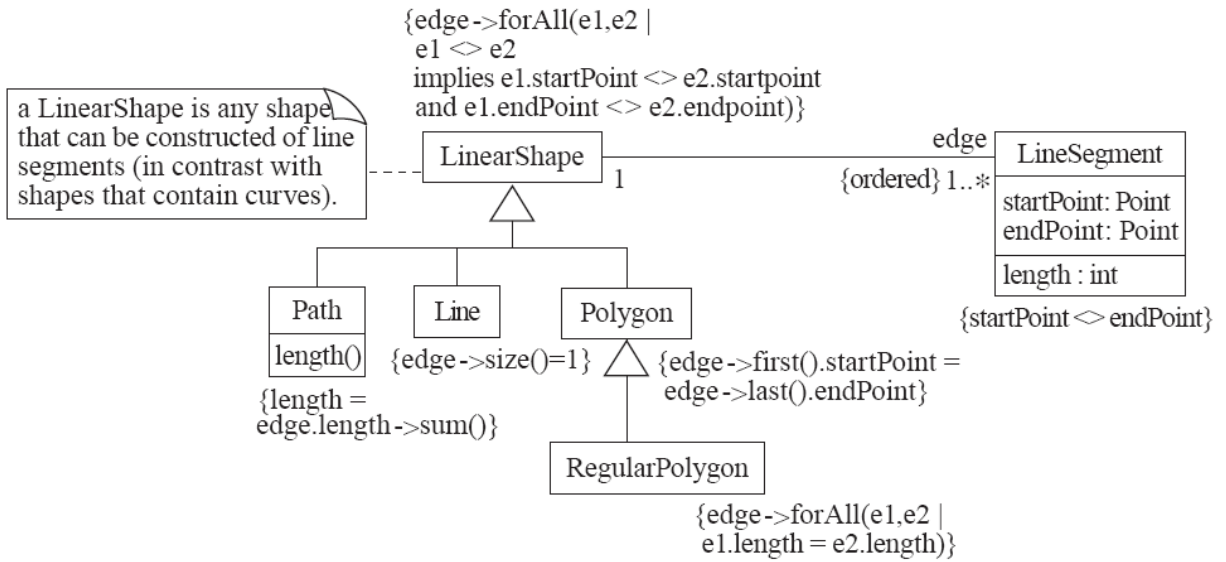
### OCL statements

**OCL statements can be built from:**

- References to role names, association names, attributes and the results of operations
- The logical values **true** and **false**
- Logical operators such as **and**, **or**, **=**, **>**, **<** or **<>** (not equals)
- String values such as: **'a string'**
- Integers and real numbers
- Arithmetic operations **\***, **/**, **+**, **-**

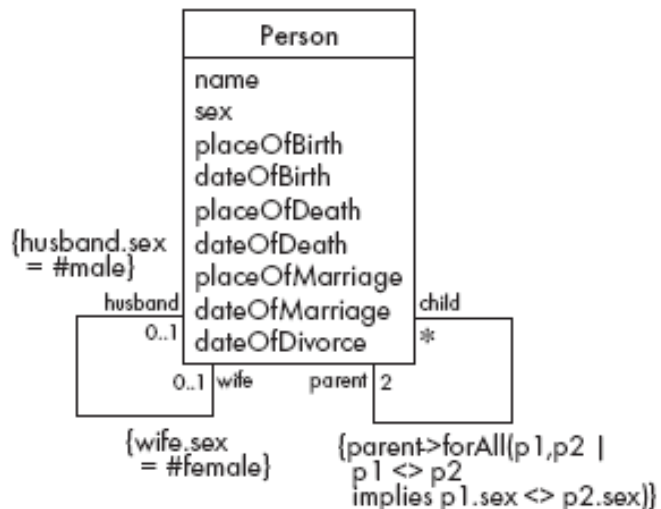
258

# An example: constraints on Polygons



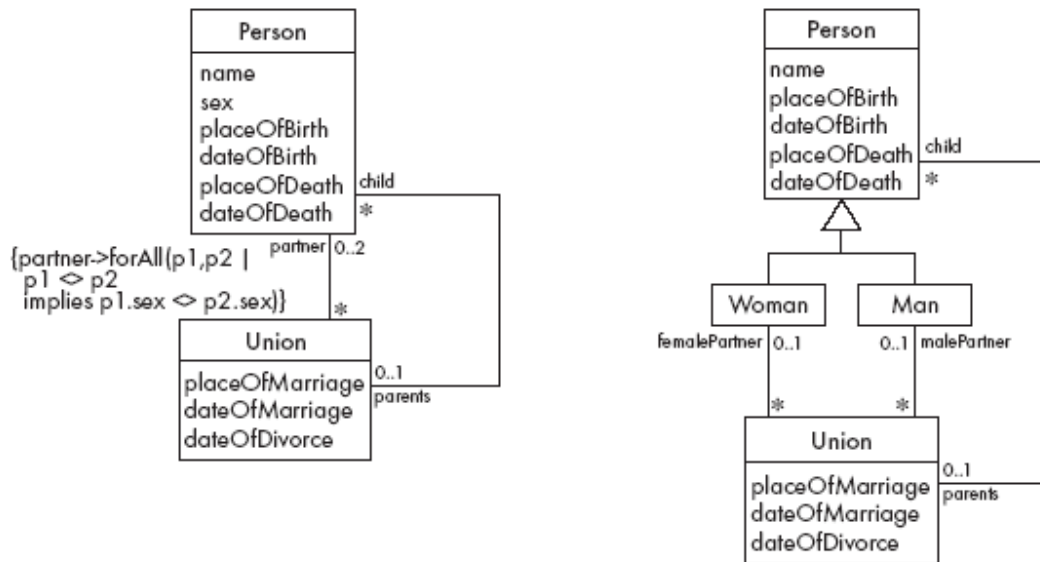
259

## 5.8 Detailed Example: A Class Diagram for Genealogy



260

# Genealogy example: Possible solutions



261

## 5.9 The Process of Developing Class Diagrams

**You can create UML models at different stages and with different purposes and levels of details**

- **Exploratory domain model:**
  - Developed in domain analysis to learn about the domain
- **System domain model:**
  - Models aspects of the domain represented by the system
- **System model:**
  - Includes also classes used to build the user interface and system architecture

262

# System domain model vs System model

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

263

# System domain model vs System model

- The *system domain model* omits many classes that are needed to build a complete system
  - Can contain less than half the classes of the system.
  - Should be developed to be used independently of particular sets of
    - user interface classes
    - architectural classes
- The complete *system model* includes
  - The system domain model
  - User interface classes
  - Architectural classes
  - Utility classes

264

# Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the entire process until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces
  - Apply design patterns (Chapter 6)

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

265

# Identifying classes

- When developing a domain model you tend to *discover* classes
- When you work on the user interface or the system architecture, you tend to *invent* classes
  - Needed to solve a particular design problem
  - (Inventing may also occur when creating a domain model)
- Reuse should always be a concern
  - Frameworks
  - System extensions
  - Similar systems

266

# A simple technique for discovering domain classes

- Look at a source material such as a description of requirements
- Extract the *nouns* and *noun phrases*
- Eliminate nouns that:
  - are redundant
  - represent instances
  - are vague or highly general
  - not needed in the application
- Pay attention to classes in a domain model that represent *types of users* or other actors

267

## Identifying associations and attributes

- Start with classes you think are most **central** and important
- Decide on the clear and obvious data it must contain and its relationships to other classes.
- Work outwards towards the classes that are less important.
- Avoid adding many associations and attributes to a class
  - A system is simpler if it manipulates less information

268

# Tips about identifying and specifying valid associations

- An association should exist if a class
  - *possesses*
  - *controls*
  - *is connected to*
  - *is related to*
  - *is a part of*
  - *has as parts*
  - *is a member of*, or
  - *has as members*

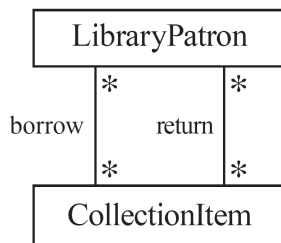
some other class in your model

- Specify the multiplicity at both ends
- Label it clearly.

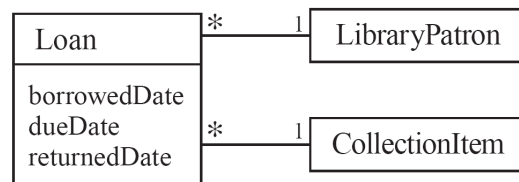
269

## Actions versus 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.

270

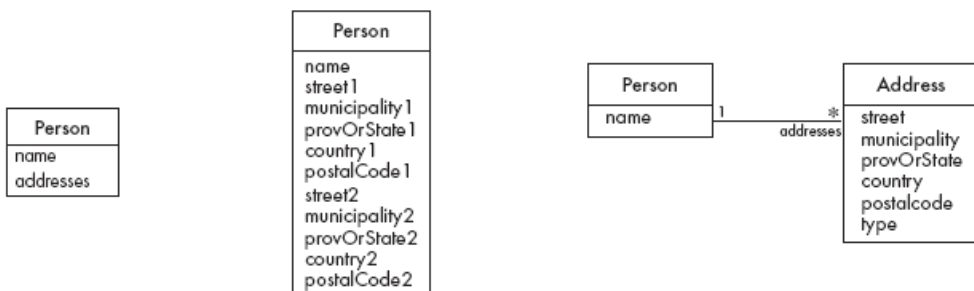
# Identifying attributes

- Look for information that must be maintained about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
  - E.g. string, number

271

## Tips about identifying and specifying valid attributes

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



Bad, due to a plural attribute

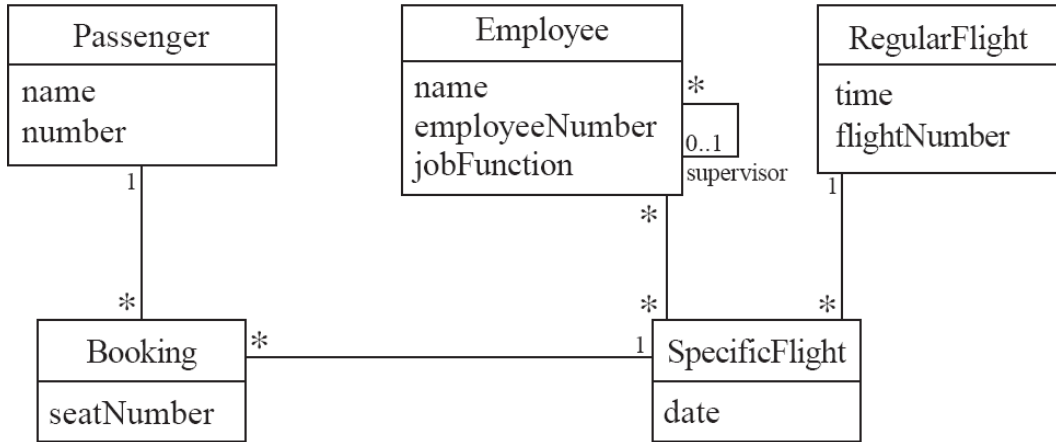
Bad, due to too many attributes, and the inability to add more addresses

Good solution. The type indicates whether it is a home address, business address etc.

272



# An example (attributes and associations)



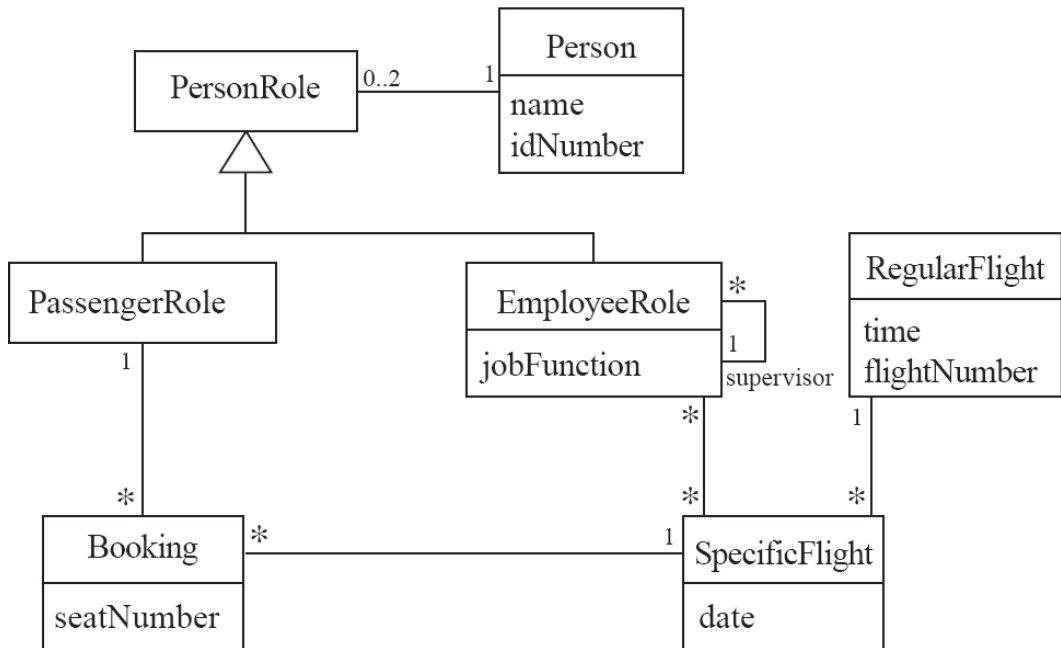
273

## Identifying generalizations and interfaces

- There are two ways to identify 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
  - The classes are very dissimilar except for having a few operations in common
  - One or more of the classes already have their own superclasses
  - Different implementations of the same class might be available

274

# An example (generalization)



275

## 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
  - All the responsibilities of a given class should be *clearly related*.
  - If a class has too many responsibilities, consider *splitting* it into distinct classes
  - If a class has no responsibilities attached to it, then it is probably *useless*
  - When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created
- To determine responsibilities
  - Perform use case analysis
  - Look for verbs and nouns describing *actions* in the system description

276

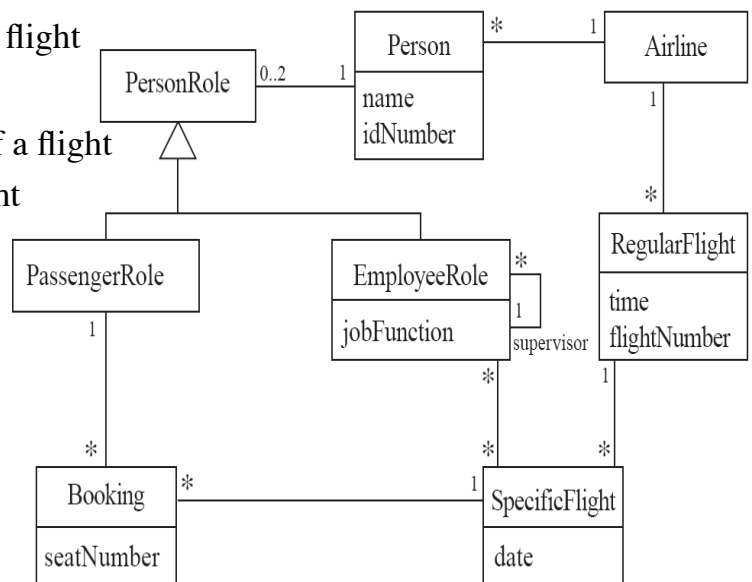
# Categories of responsibilities

- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work

277

## An example (responsibilities)

- Creating a new regular flight
- Searching for a flight
- Modifying attributes of a flight
- Creating a specific flight
- Booking a passenger
- Canceling a booking



278

# Prototyping a class diagram on paper

- As you identify classes, you write their names on small cards
- As you identify attributes and responsibilities, you list them on the cards
  - If you cannot fit all the responsibilities on one card:
    - this suggests you should split the class into two related classes.
- Move the cards around on a whiteboard to arrange them into a class diagram.
- Draw lines among the cards to represent associations and generalizations.

279

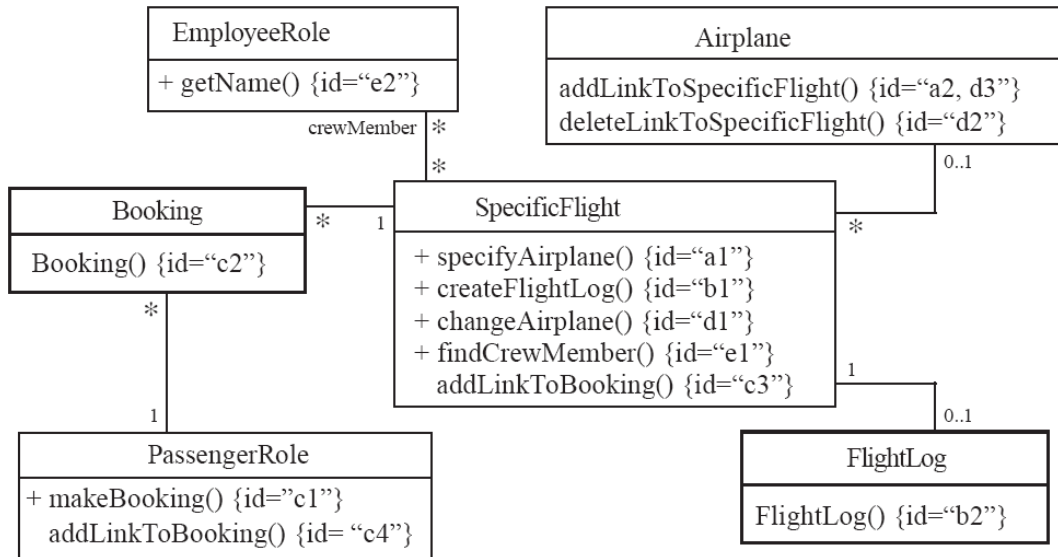
## Identifying operations

**Operations are needed to realize the responsibilities of each class**

- There may be several operations per responsibility
- The main operations that implement a responsibility are normally declared **public**
- Other methods that collaborate to perform the responsibility must be as private as possible

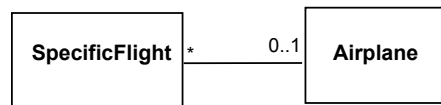
280

# An example (class collaboration)



281

## Class collaboration 'a'

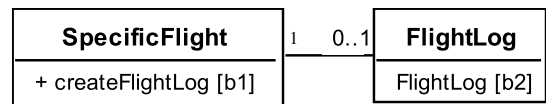


*Making a bi-directional link between two existing objects;*  
 e.g. adding a link between an instance of SpecificFlight and  
 an instance of Airplane.

1. (public) The instance of SpecificFlight
  - **makes a one-directional link to the instance of Airplane**
  - **then calls operation 2.**
2. (non-public) The instance of Airplane
  - **makes a one-directional link back to the instance of SpecificFlight**

282

## Class collaboration 'b'



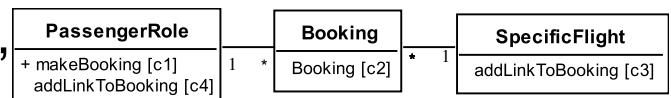
### *Creating an object and linking it to an existing object*

e.g. creating a **FlightLog**, and linking it to a **SpecificFlight**.

1. (public) The instance of **SpecificFlight**
  - **calls the constructor of FlightLog (operation 2)**
  - **then makes a one-directional link to the new instance of FlightLog.**
2. (non-public) Class **FlightLog**'s constructor
  - **makes a one-directional link back to the instance of SpecificFlight.**

283

## Class collaboration 'c'



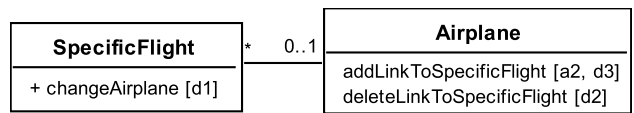
### *Creating an association class, given two existing objects*

e.g. creating an instance of **Booking**, which will link a **SpecificFlight** to a **PassengerRole**.

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**.

284

## Class collaboration 'd'



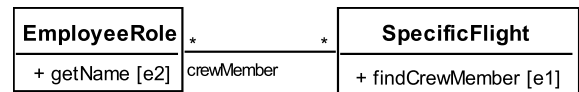
### *Changing the destination of a link*

e.g. changing the **Airplane** of to a **SpecificFlight**, from **airplane1** to **airplane2**

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**.

285

## Class collaboration 'e'



### *Searching for an associated instance*

e.g. searching for a crew member associated with a **SpecificFlight** that has a certain name.

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.

286

## 5.10 Implementing Class Diagrams in Java

- Attributes are implemented as instance variables
- Generalizations are implemented using extends
- Interfaces are implemented using implements
- Associations are normally implemented using instance variables
  - Divide each two-way association into two one-way associations
    - so each associated class has an instance variable.
  - For a one-way association where the multiplicity at the other end is ‘one’ or ‘optional’
    - declare a variable of that class (a reference)
  - For a one-way association where the multiplicity at the other end is ‘many’:
    - use a collection class implementing List, such as Vector

287

## 5.11 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:
  - Ensure that team members have adequate training*
  - Have experienced modeler as part of the team*
  - Review all models thoroughly*

288