Object-Oriented Software Engineering

Practical Software Development using UML and Java

Modelling with Classes

2.1 What is Object Orientation?

Procedural paradigm:

- Software is organized around the notion of *procedures/functions*
- *Procedural abstraction*

  —Works as long as the data is simple, self-contained
- *Adding data abstractions*

  —Group the pieces of data in *records* and *structures*

Object-Oriented paradigm:

- Organizing procedural abstractions in the context of data abstractions. Helps reduce complexity.

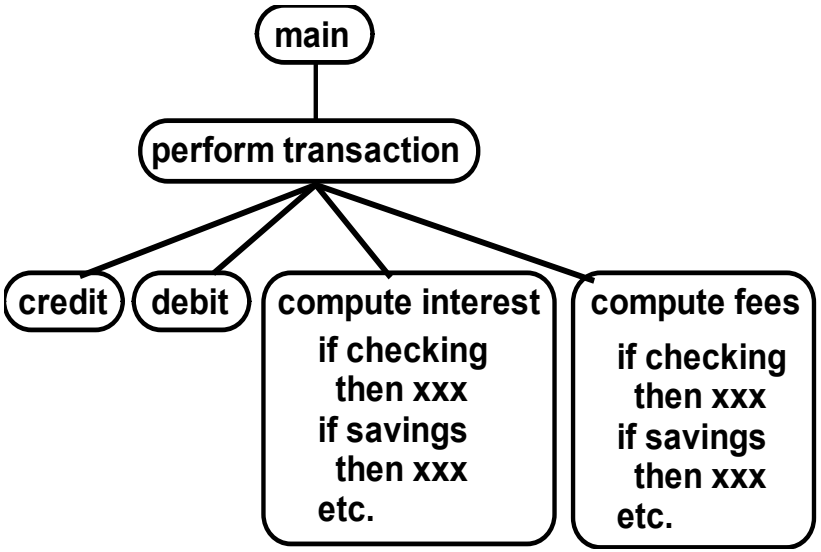
  —Group the pieces of data that describes an entity in *classes* and *objects*.

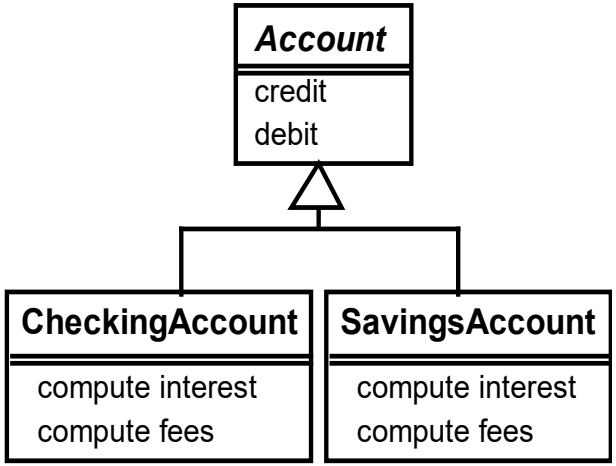
Object-Oriented (OO or O.O.) paradigm

An approach to the solution of problems in which all computations are performed in the context of *objects*.

- The objects are instances of classes, which:
  - —are data abstractions
  - —contain procedural abstractions that operates on the objects
- A running program can be seen as a collection of objects collaborating to perform a given task, provide a service/funtionality.

Paradigms: Procedural X Object-Oriented





Classes

A class:

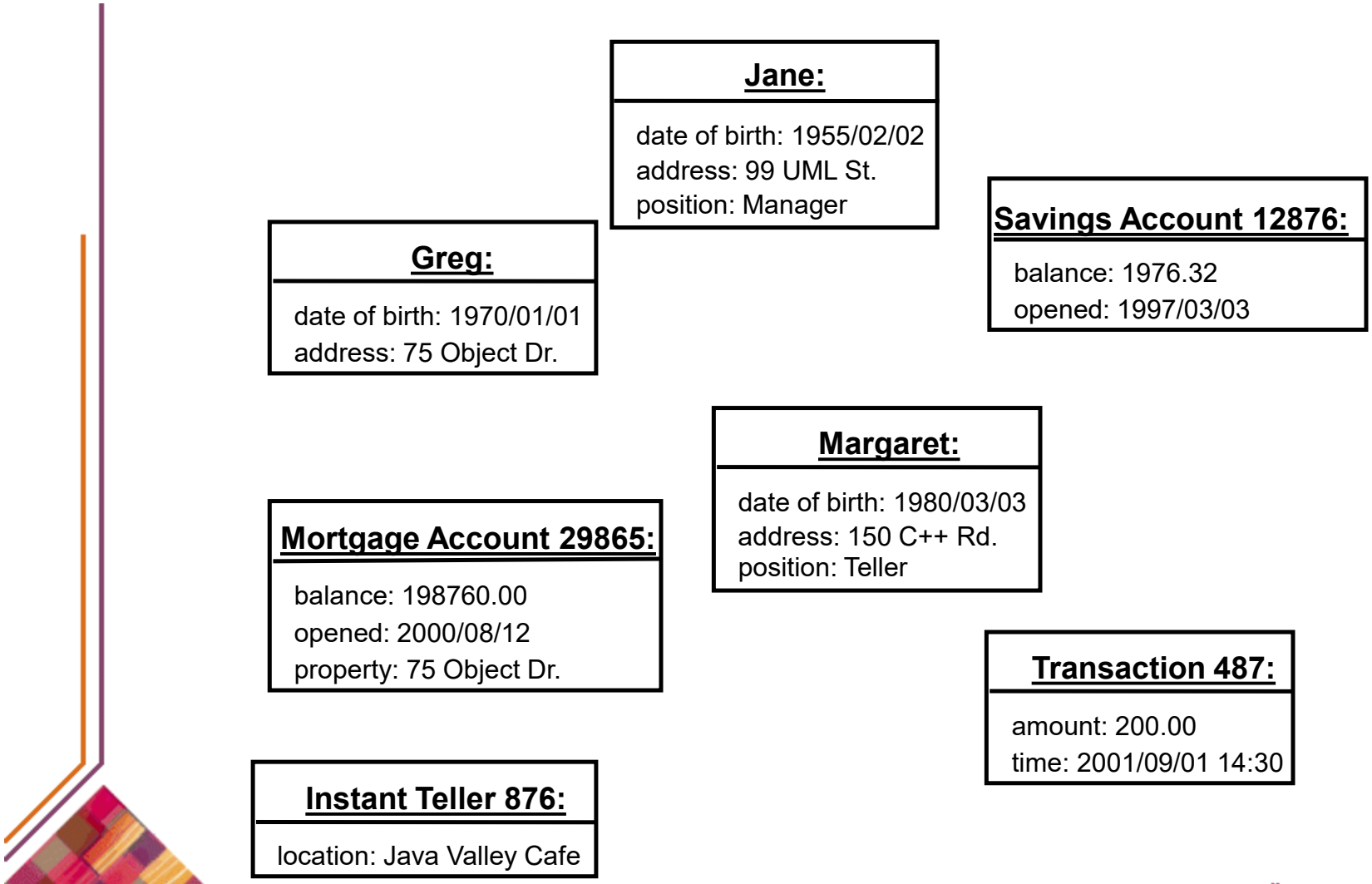
- Is a unit of abstraction in an object-oriented (OO) program
- Represents similar objects
  —Its *instances*
- Is a kind of software module
  —Describes its instances' structure (properties/attributes)
  —Contains *methods* to implement their behaviour.

Objects

Object

- A chunk of structured data in a running software system
- Has *properties/attributes*
  - —Represent its state
- Has *behaviour*
  - —How it acts and reacts
  - —May simulate the behaviour of an object in the real world.
    - For instance, a file can be deleted, opened, closed, renamed, moved, etc.

Objects



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
- Three important methodologists Booch, Rumbaugh and Jacobson decided to merge their approaches in 1995
  - —They worked together at the Rational Software Corporation
- In 1997 the Object Management Group (OMG –www.omg.org) started the process of UML standardization.

At that time, but we did not become millionaire!

UML Diagrams

- Class diagrams
  —describe classes and their relationships
- Object diagrams
  —describe interaction about objects
- 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.

What constitutes a good model?

Modelling is one of the most difficult tasks of software development. A model should:

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

Models are used:

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

5.2 Essentials of UML Class Diagrams

The main symbols shown on <u>class diagrams</u> are:

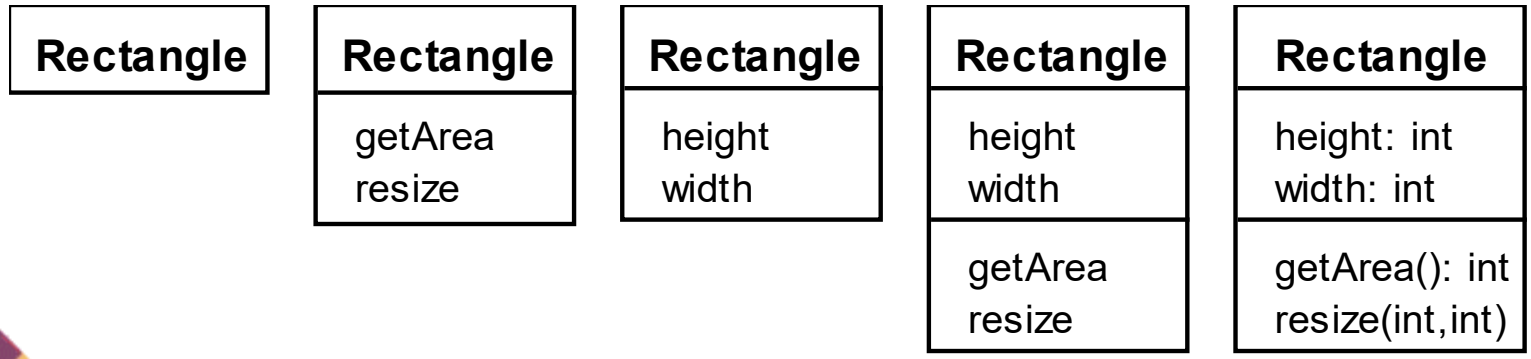
- *Classes*
  - represent the types of data themselves
- *Associations*
  - represent linkages between instances of classes
- *Attributes*
  - are simple data found in classes and their instances
- *Operations*
  - represent the functions performed by the classes instances. i.e. objects
- *Generalizations*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

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:

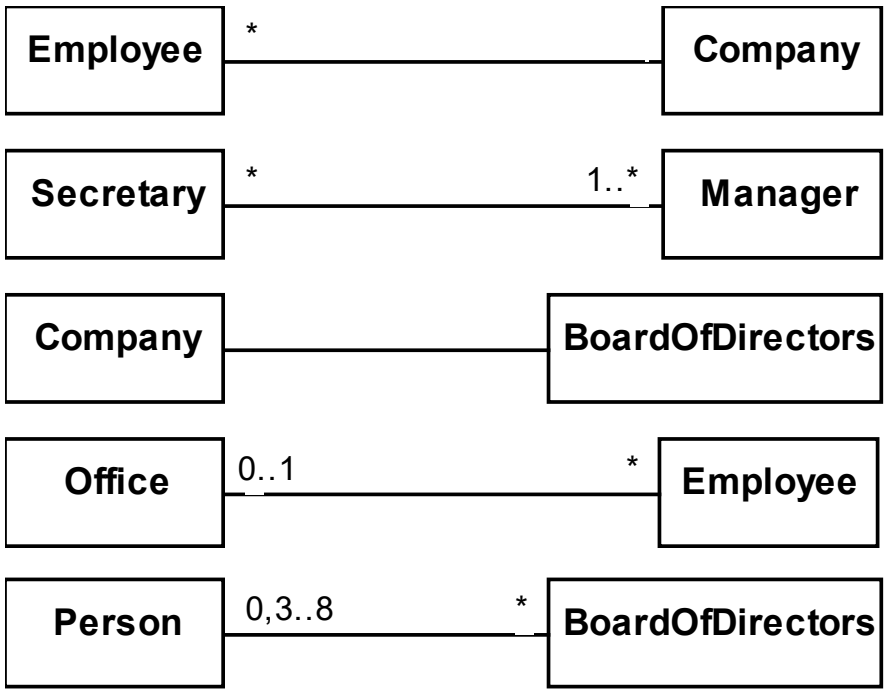
  operationName(parameterName: parameterType …): returnType



Associations (Relationships) and Multiplicity

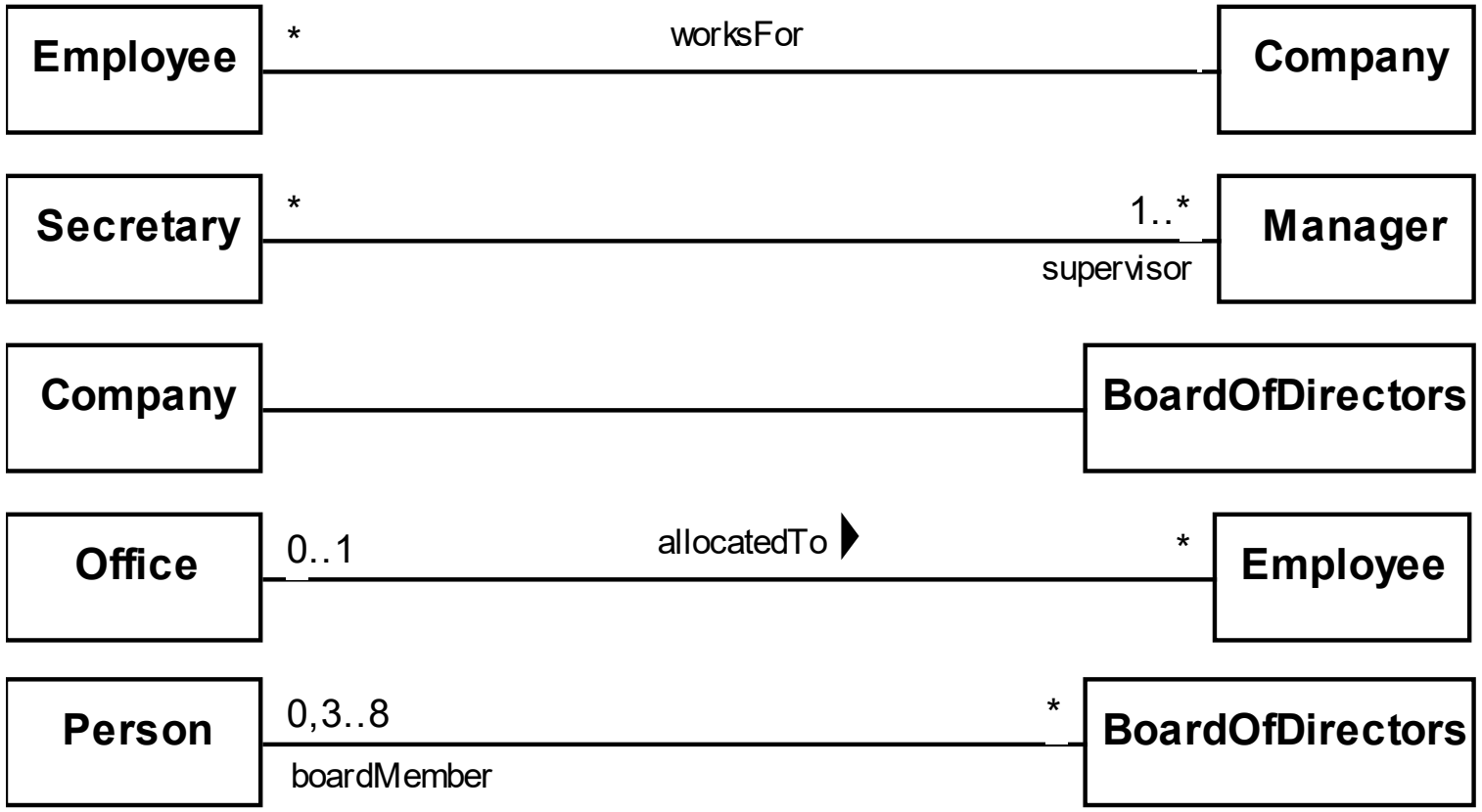
An *association* is used to show how two classes are related to each other – reads: *is associated with*

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



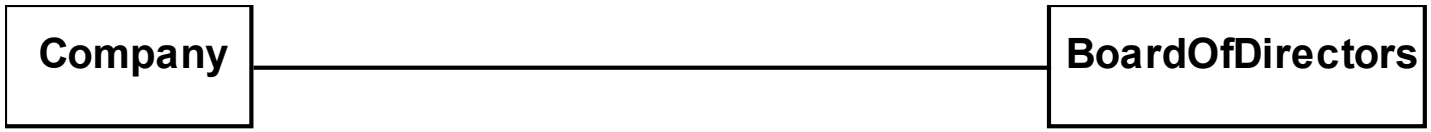
Labelling associations (is associated with)

- Each association can be labelled, to make explicit the nature of the association or role of the entities.



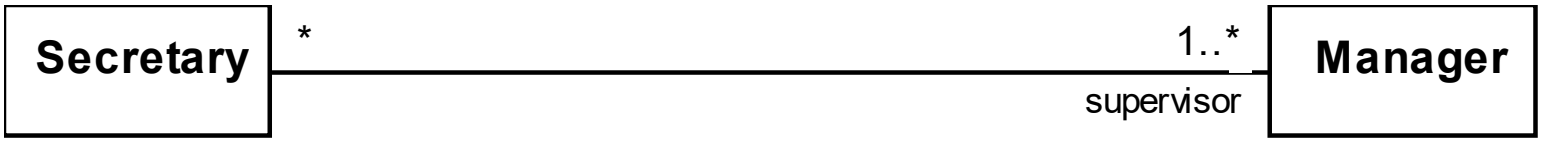
Analyzing and validating associations

- **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



Analyzing and validating associations

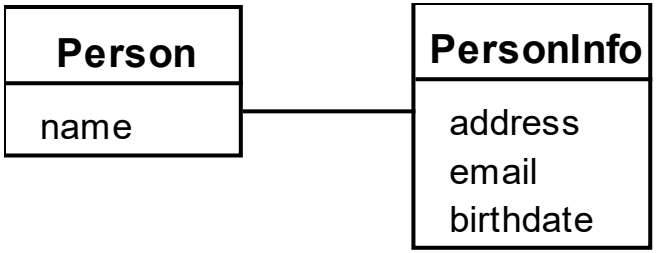
- **Many-to-many**
  - —A secretary can work for many managers
  - —A manager can have many secretaries
  - —Secretaries can work in pools
  - —Managers can have a group of secretaries
  - —Some managers might have zero secretaries.
  - —Is it possible for a secretary to have, perhaps temporarily, zero managers?



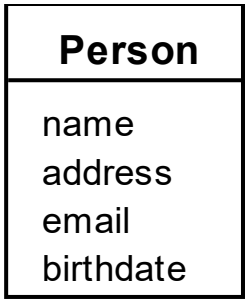
Analyzing and validating associations

Avoid unnecessary one-to-one associations

Avoid this



do this

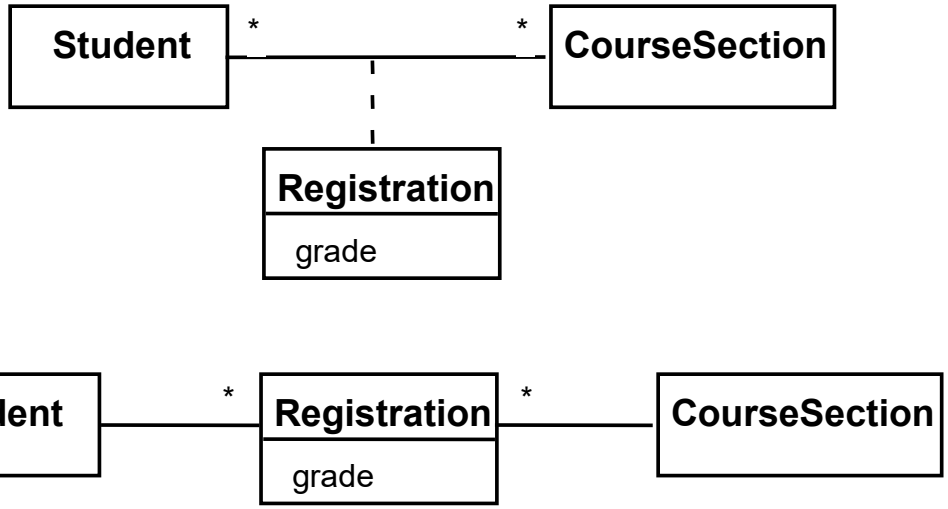


A more complex example

- 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

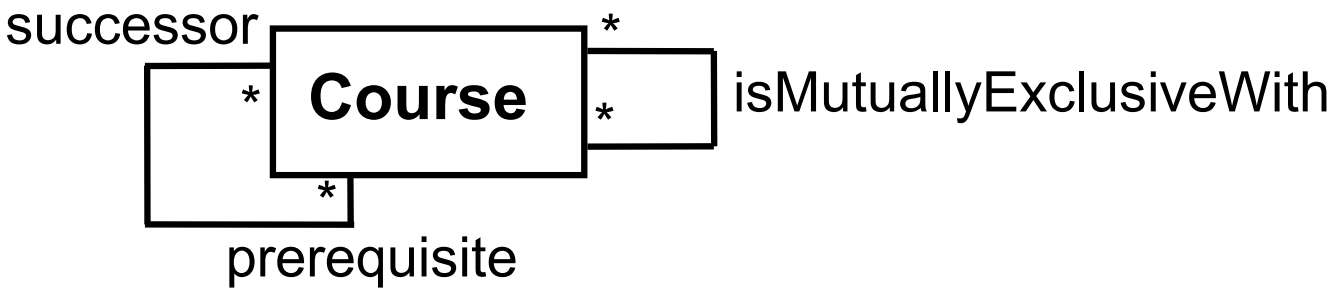
Association classes

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



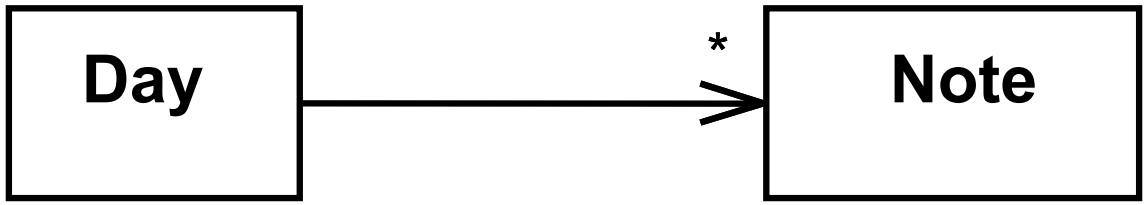
Reflexive associations

- It is possible for an association to connect a class to itself
- Example
  - —Course pre-reqs
  - —Course precludes



Directionality in associations

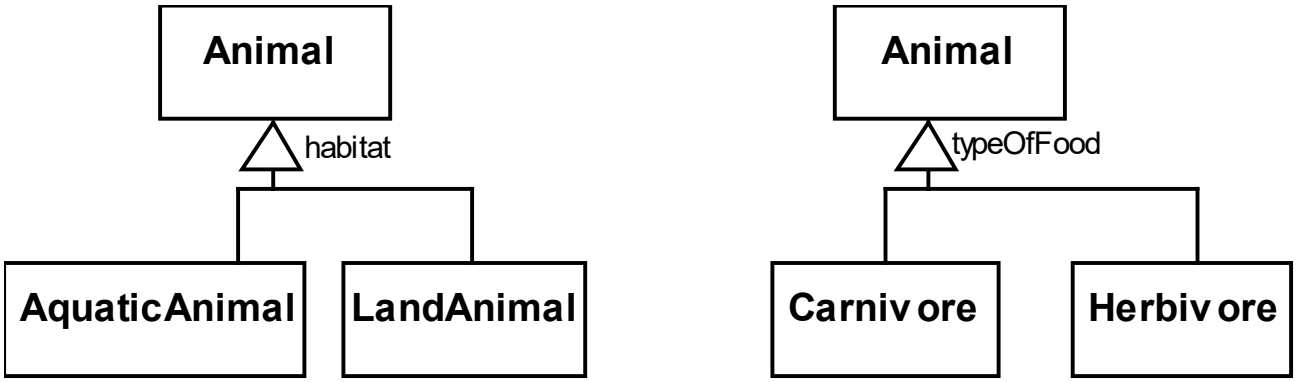
- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end



5.4 Generalization

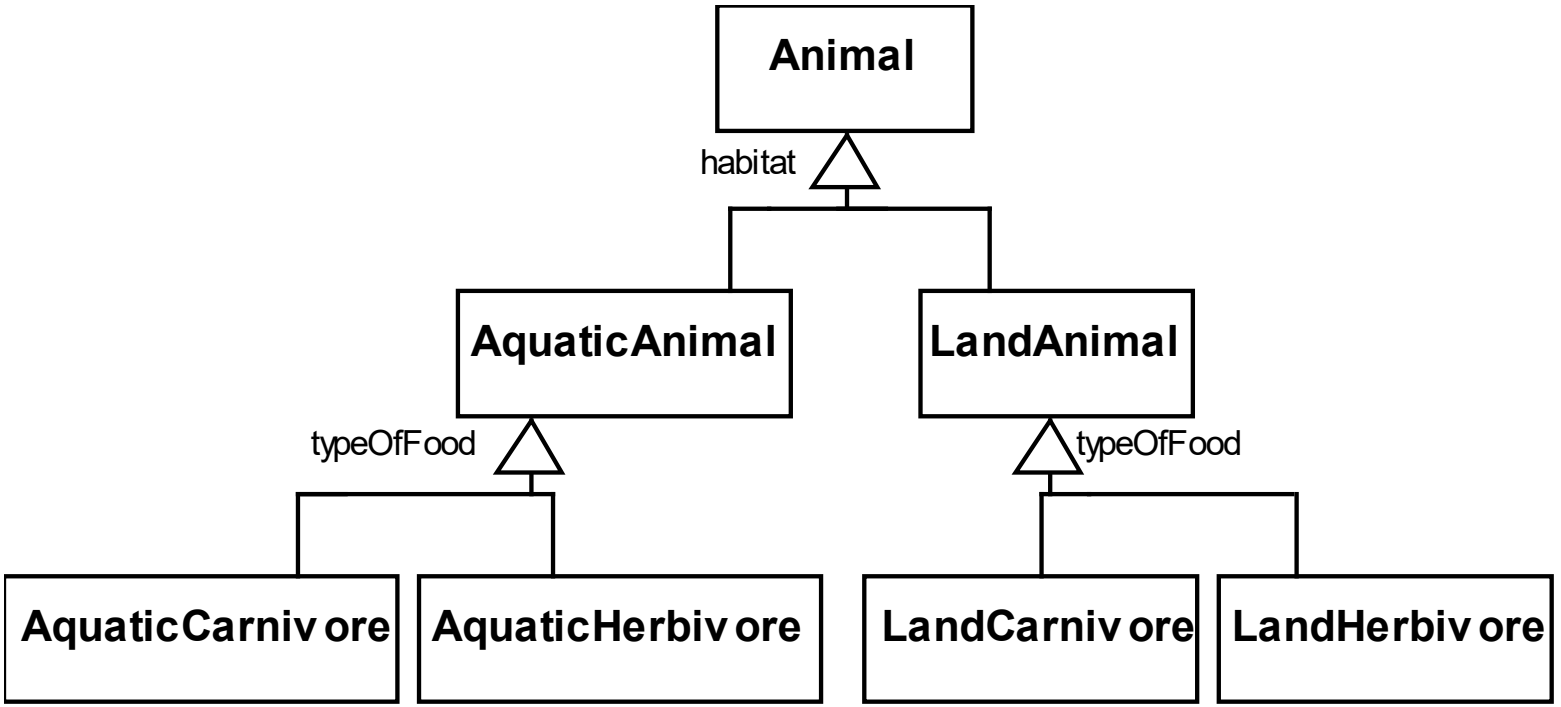
Specializing a superclass into two or more subclasses

- The *discriminator/label* is a label that describes the criteria used in the specialization
- Where do you put sharks?



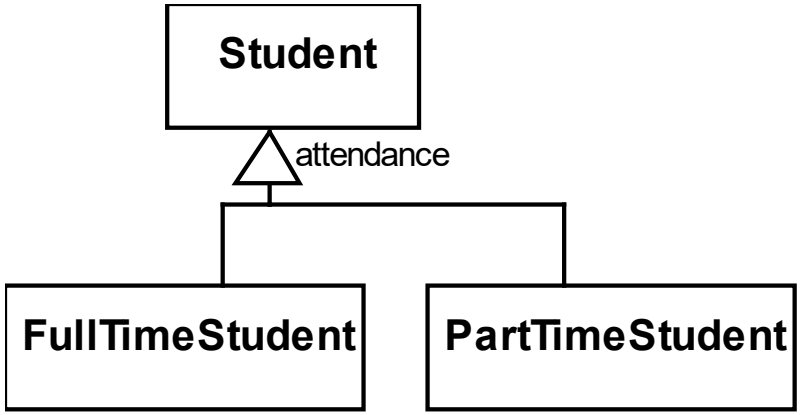
Handling multiple discriminators (labels)

- Creating higher-level generalization



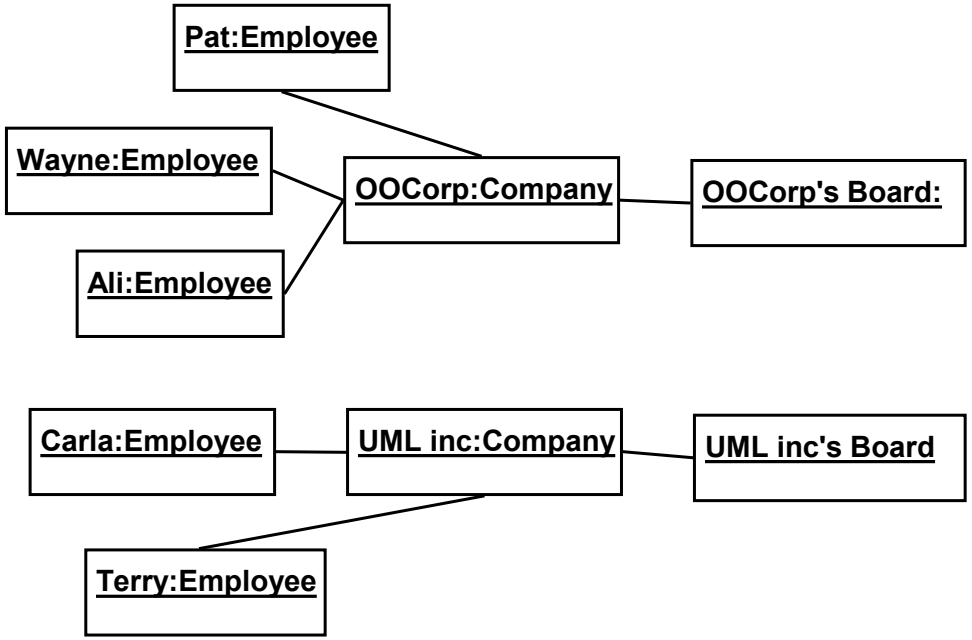
Avoiding having instances change class

- An instance should never need to change class



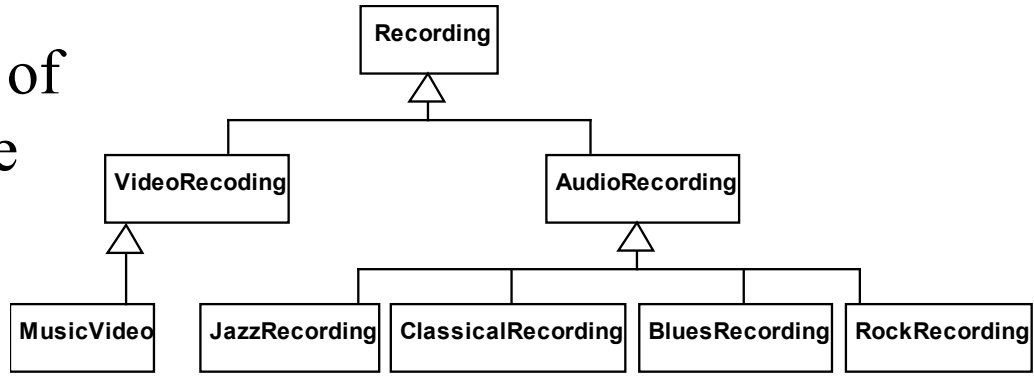
Object Diagrams – It helps abstract a class

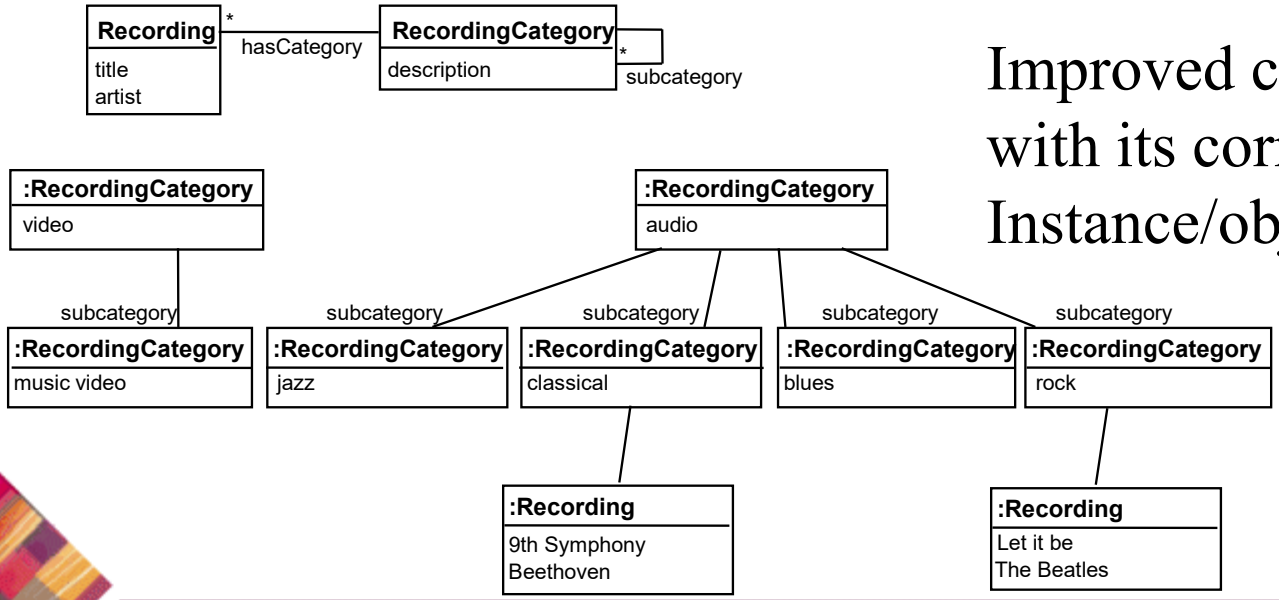
- A *link* is an instance of an association
- In the same way that we say an object is an instance of a class



Avoiding unnecessary generalizations

Inappropriate hierarchy of classes, which should be instances

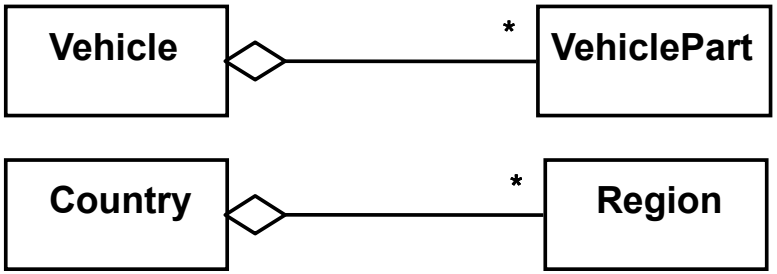




Improved class diagram, with its corresponding Instance/object diagram

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`



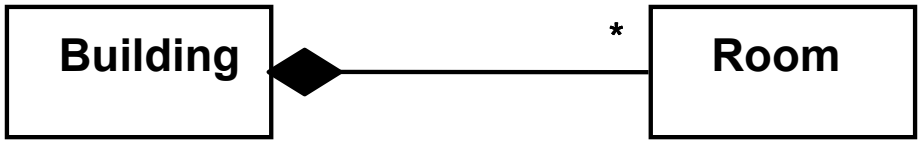
When to use an aggregation

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

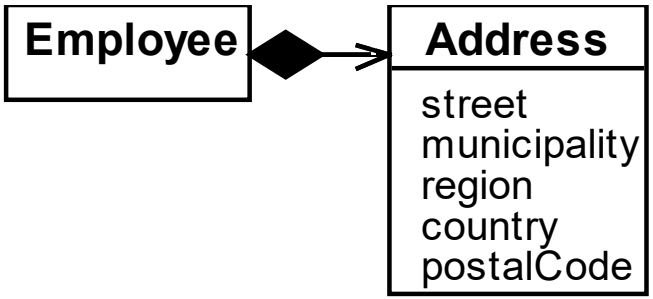
- You can state that
  - —the parts ‘are part of’ the aggregate
  - —or the aggregate ‘is composed of’ the parts
- When something owns or controls the aggregate, then they also own or control the parts

Composition

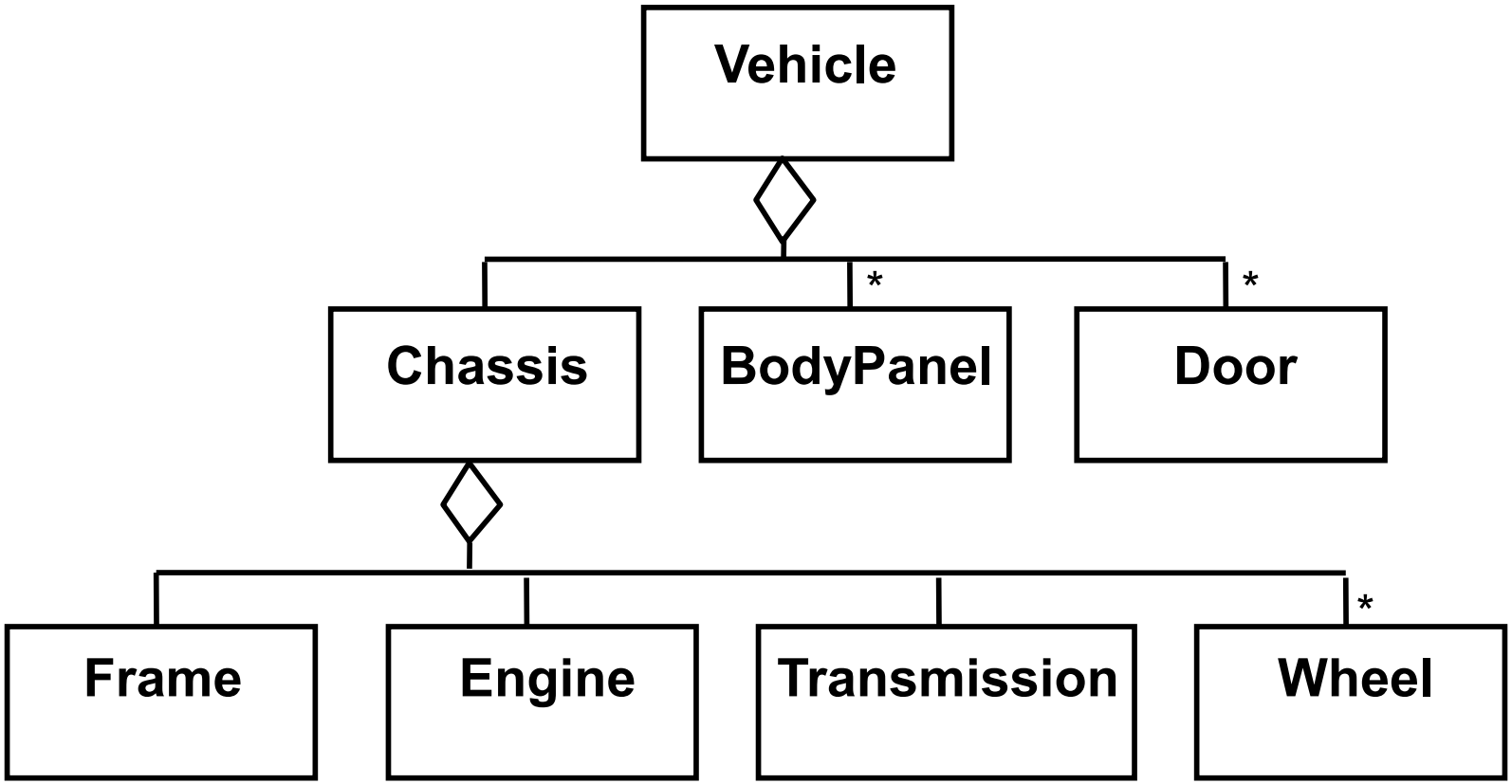
- A *composition* is a strong kind of aggregation
  —if the aggregate is destroyed, then the parts are destroyed as well



- Two alternatives for addresses



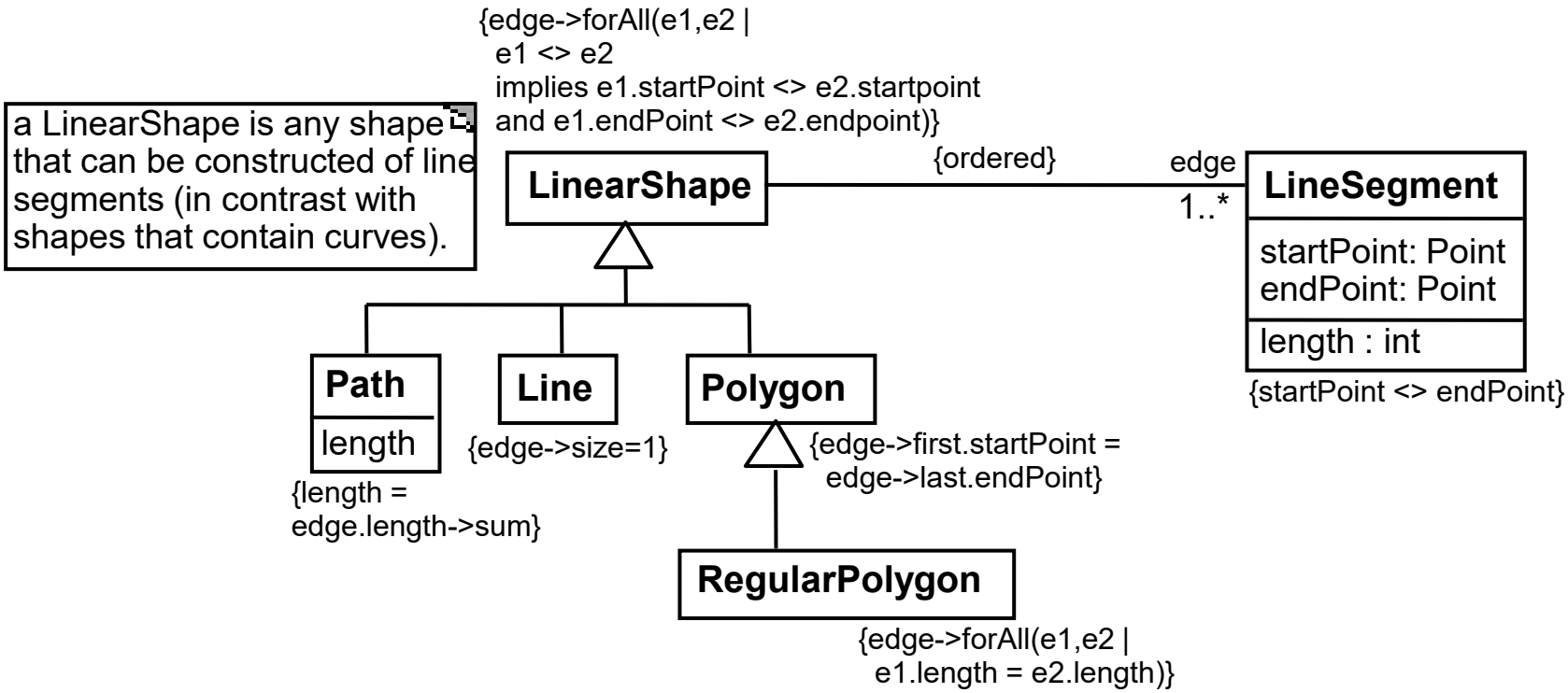
Aggregation hierarchy



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

An example with Polygons



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

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

Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities (functionalities)** 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

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

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
- Reuse should always be a concern
  - —Frameworks
  - —System extensions
  - —Similar systems

A simple technique for discovering domain classes - Hand-out of Airline system

- 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.

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
- Avoid adding many associations and attributes to a class
  - —A system is simpler if it manipulates less information.

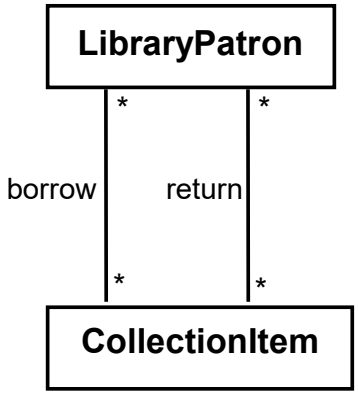
Tips about identifying and specifying valid associations

- 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*

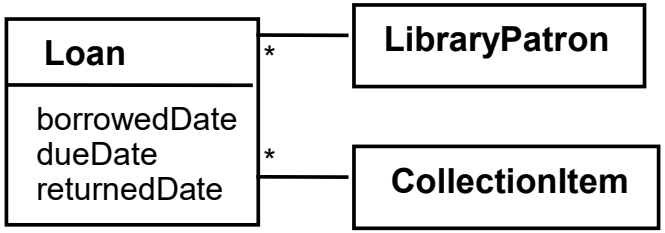
  some other class in your model
- Specify the multiplicity at both ends
- Label it clearly.

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



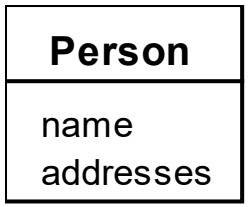
Better: The **`borrow`** operation creates a **`Loan`**, and the **`return`** operation sets the **`returnedDate`** attribute.

Identifying attributes

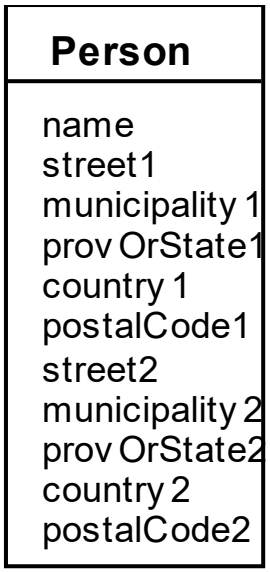
- Look for information that must be maintained about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
  —E.g. string, number

Tips about identifying and specifying valid attributes

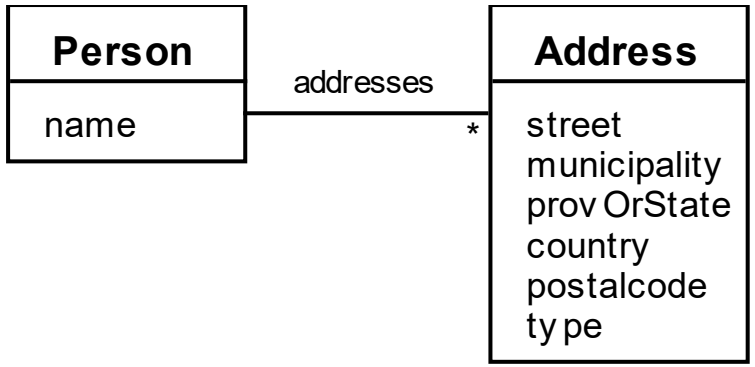
- It is not good to have many duplicate attributes
- If a subset of a class's attributes form a coherent group, then create a distinct class containing these attributes



Bad due to a plural attribute

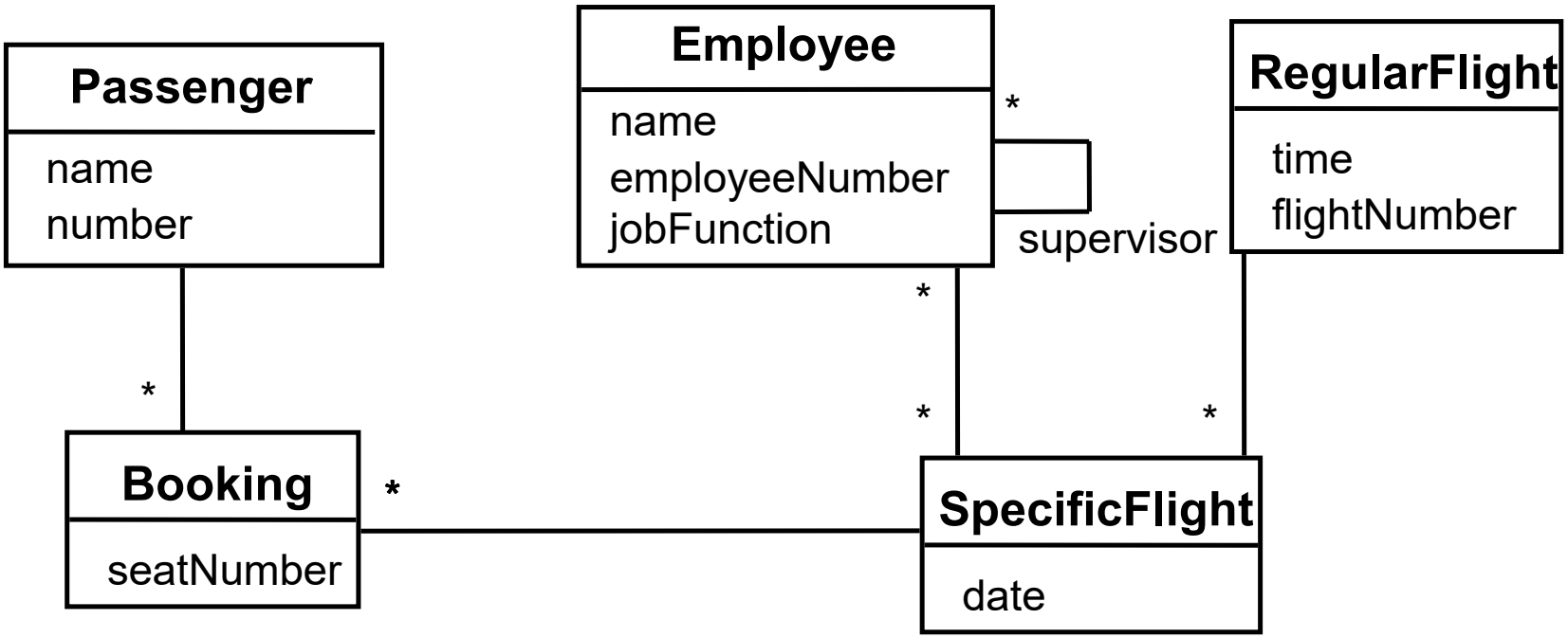


Bad due to too many attributes, and inability to add more addresses



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

An example (attributes and associations)



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

Categories of responsibilities

Functionality involves:

- 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, printing
- Computing numerical results
- Navigating and searching
- Other specialized work.

An example (responsibilities)

—Creating a new regular flight

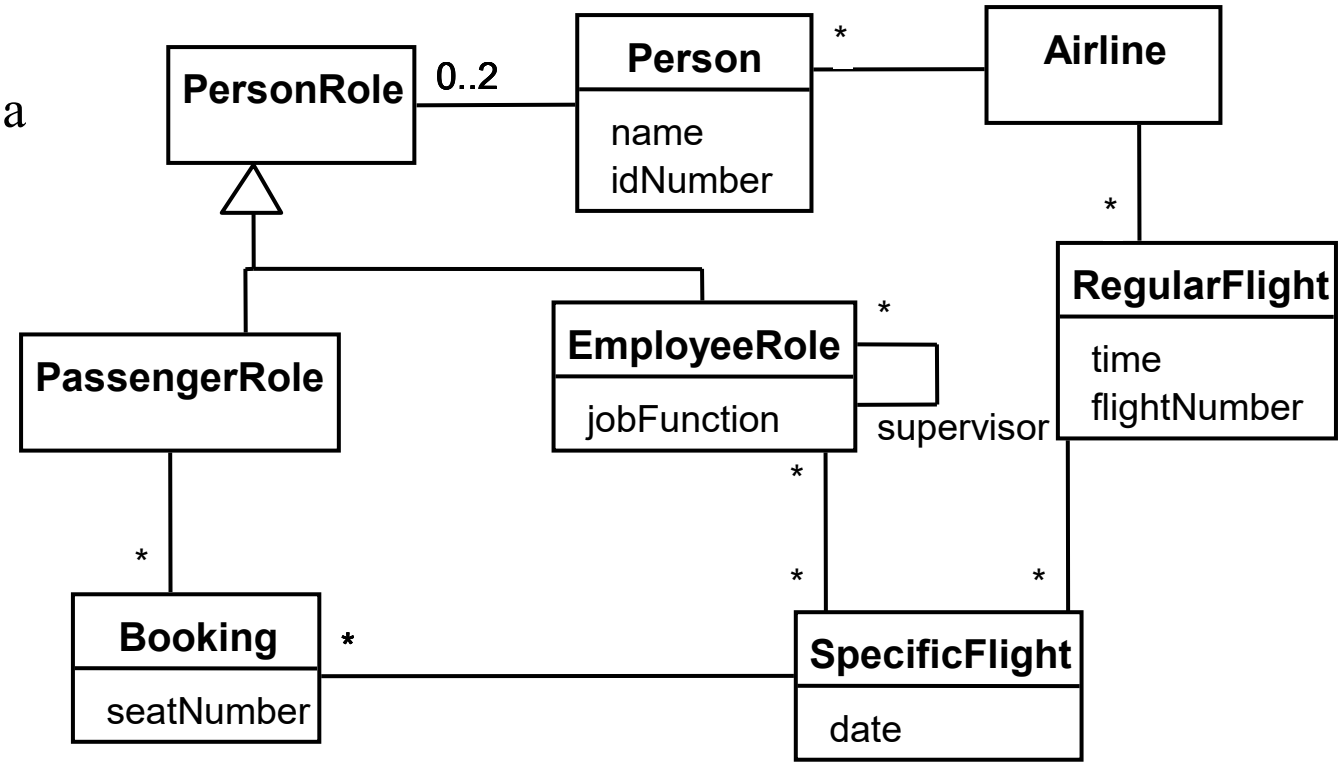
—Searching for a flight

—Modifying attributes of a flight

—Creating a specific flight

—Booking a passenger

—Canceling a booking



Prototyping a class diagram on paper

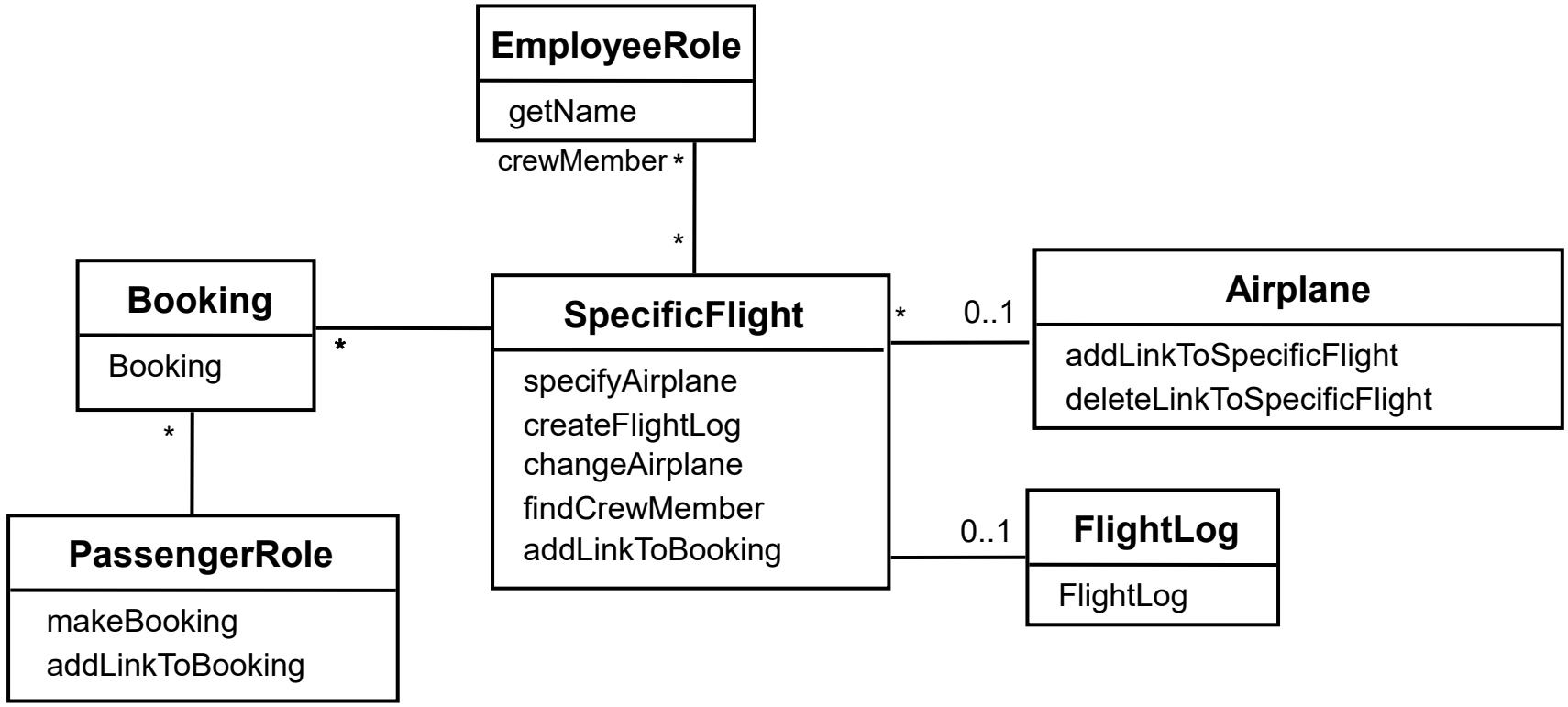
- As you identify classes, you write their names on small cards
- As you identify attributes and responsibilities, you list them on the cards
  - — If you cannot fit all the responsibilities on one card:
    - this suggests you should split the class into two related classes.
- Move the cards around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

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

An example (class collaboration)



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`

Example: `SpecificFlight`

```
class SpecificFlight
{
  private Calendar date;
  private RegularFlight regularFlight;
  private TerminalOfAirport destination;
  private Airplane airplane;
  private FlightLog flightLog;

  private ArrayList crewMembers;
   // of EmployeeRole
  private ArrayList bookings
  ...
}
```

Example: `SpecificFlight`

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

Example: **RegularFlight**

```
class RegularFlight
{
  private ArrayList specificFlights;
  ...
  // Method that has primary
  // responsibility

  public void addSpecificFlight(
    Calendar aDate)
  {
    SpecificFlight newSpecificFlight;
    newSpecificFlight =
      new SpecificFlight(aDate, this);
    specificFlights.add(newSpecificFlight);
  }
  ...
}
```

Creately Tool - video

1. Watch video, from minute 13

2. Down a tool that you like

1. Google “Free tools UML diagrams”
2. It could be Creately, free trial

3. Practice: YSC3232-06-UML-Homework-Solution