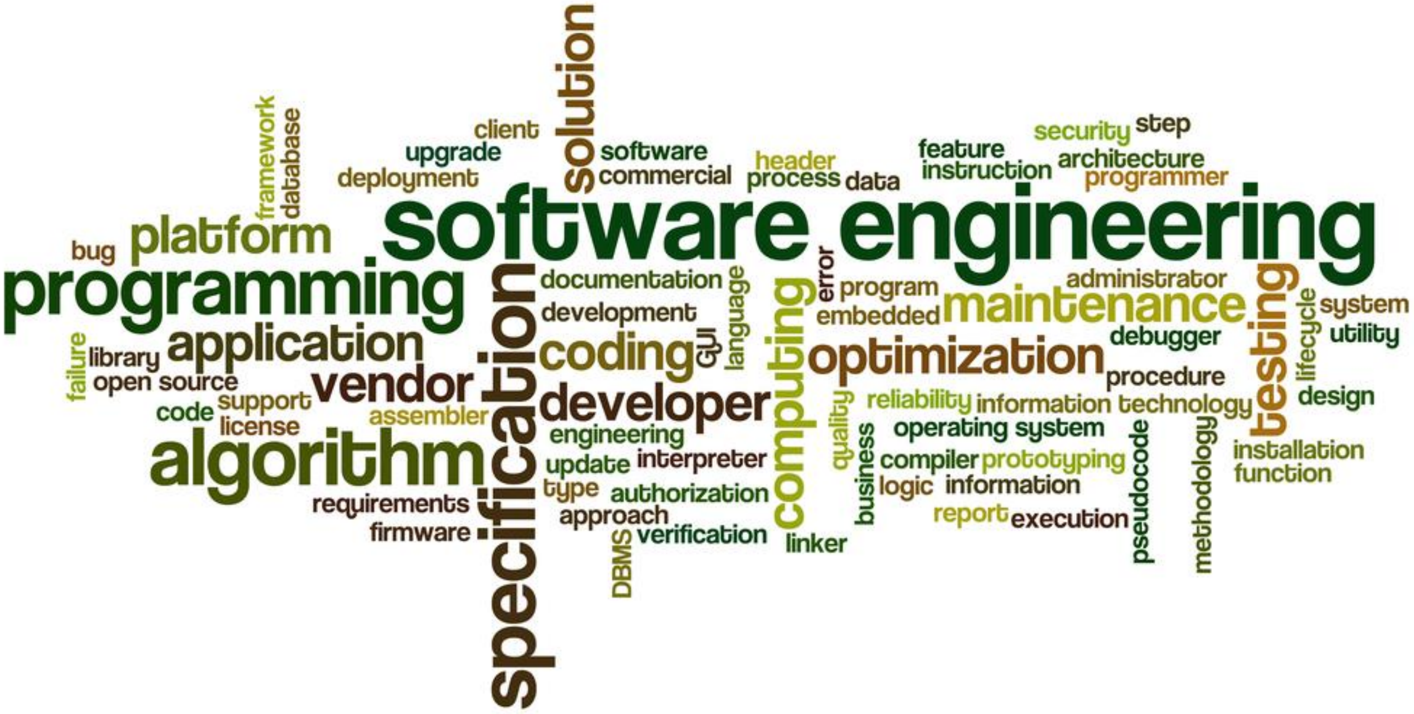


SWE 205: Introduction to Software Engineering

Lecture 12

Class Diagram

Course Topics

- ~~Introduction~~
- ~~Software Process Models~~
- ~~Requirements Engineering~~
- Modeling
- Programming Languages
- Software Construction Techniques
- Testing
- Project Management
- Refactoring
- Ethical Issues

Lecture Objectives

- ✓Modeling Classes
- ✓Associations
- ✓Generalizations
- ✓Interfaces

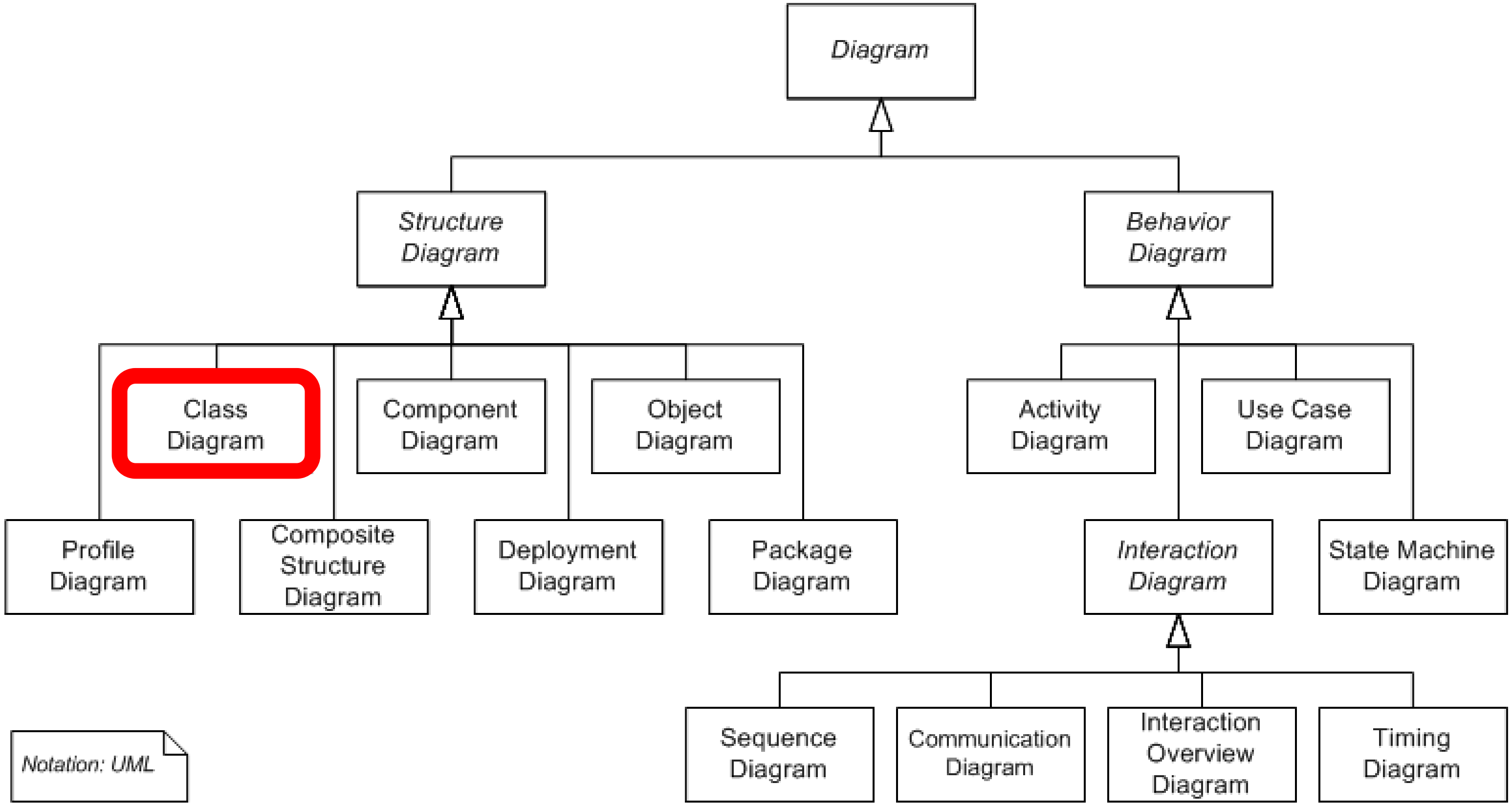
What is UML?

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

  - Developed by Rumbaugh, Booch and Jacobson
  - Based on earlier languages they had each developed
  - They worked together at the Rational Software Corporation, later bought by IBM
    - Much development of UML has been done at IBM Rational Ottawa

  - In 1997 the Object Management Group (OMG) started the process of UML standardization

UML diagrams



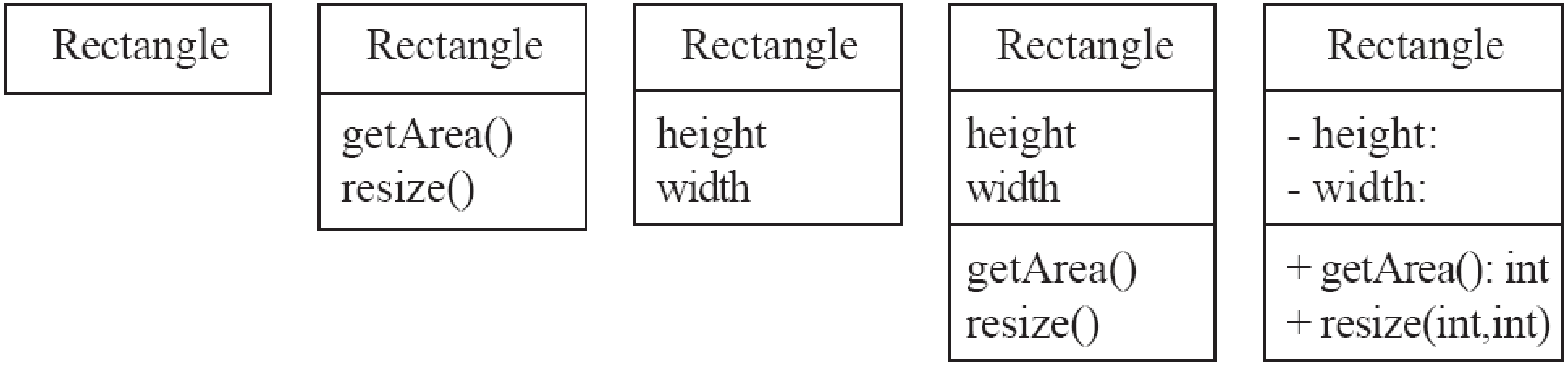
Essentials of UML Class Diagrams

- 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 abstract functions performed by the classes and their instances, as well as specific methods implementing these
  - Generalizations
    - group classes into inheritance hierarchies

Classes

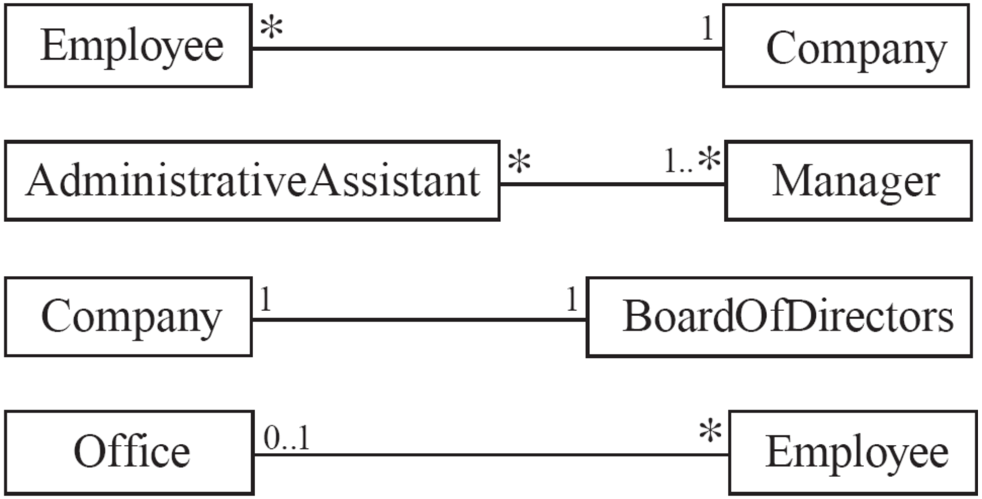
- A class is simply represented as a box with the name of the class inside
  - The diagram may also show the attributes and operations
  - The complete signature of an operation is:

    operationName(parameterName: parameterType …): returnType



Associations and Multiplicity

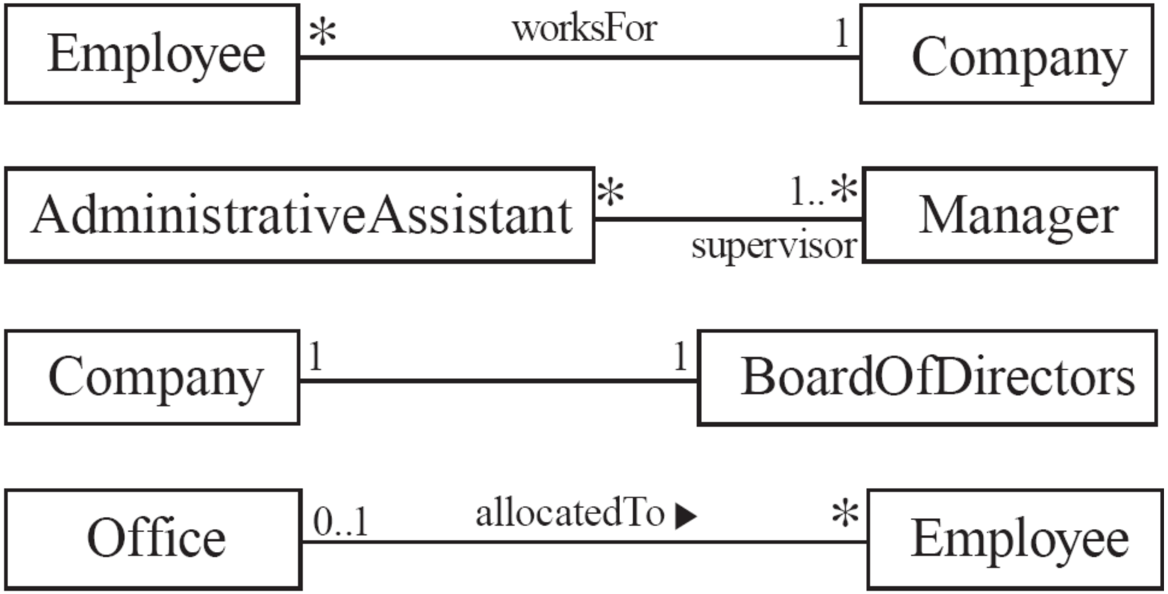
- An **association** is used to show how two classes are related to each other
  - Symbols indicating **multiplicity** are shown at each end of the association



Exactly one	1
Zero or more (unlimited)	* (0..*)
One or more	1..*
Zero or one (optional association)	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6, 8..10

Labelling associations

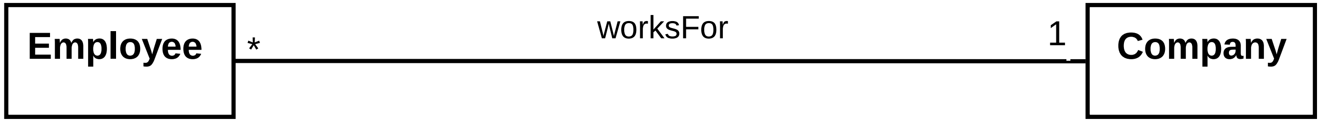
- Each association can be labelled, to make explicit the nature of the association



Analyzing and validating associations

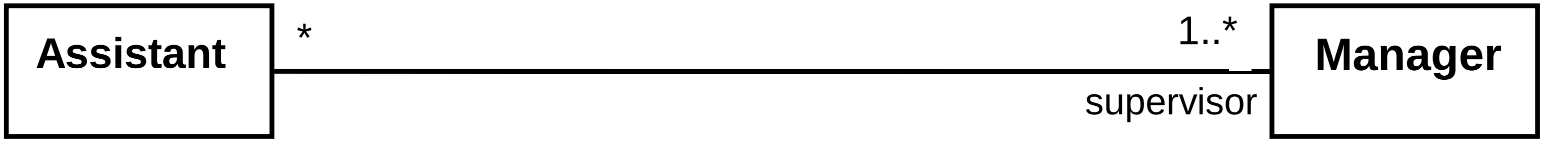
- Many-to-one

  - A company has many employees,
  - An employee can only work for one company.
  - A company can have zero employees
  - It is not possible to be an employee unless you work for a company



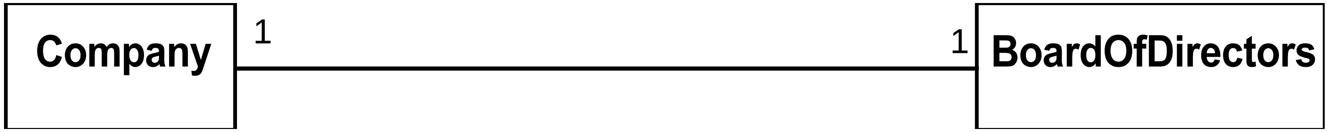
Analyzing and validating associations

- **Many-to-many**
  - An assistant can work for many managers
  - A manager can have many assistants
  - 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?



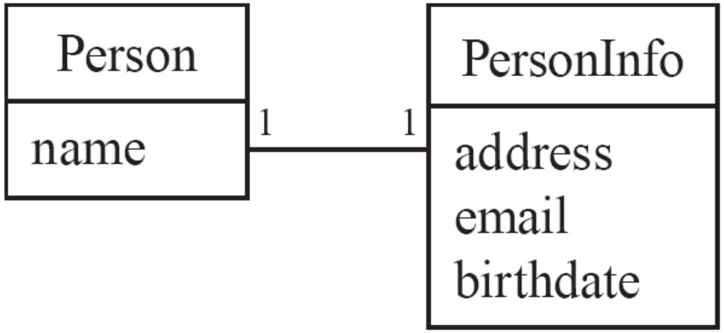
Analyzing and validating associations

- 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

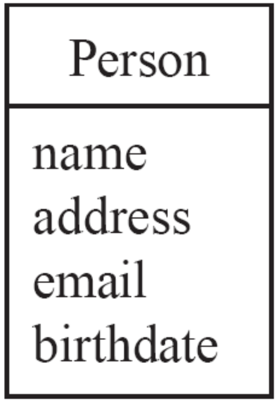


Analyzing and validating associations

- Avoid unnecessary one-to-one associations
- Avoid this

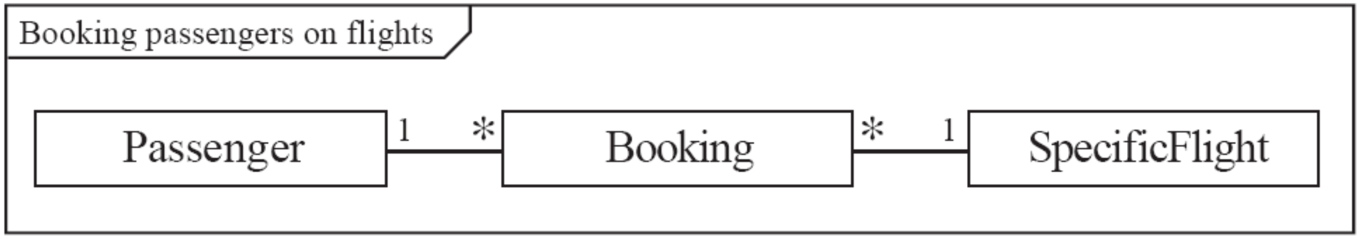


- do this



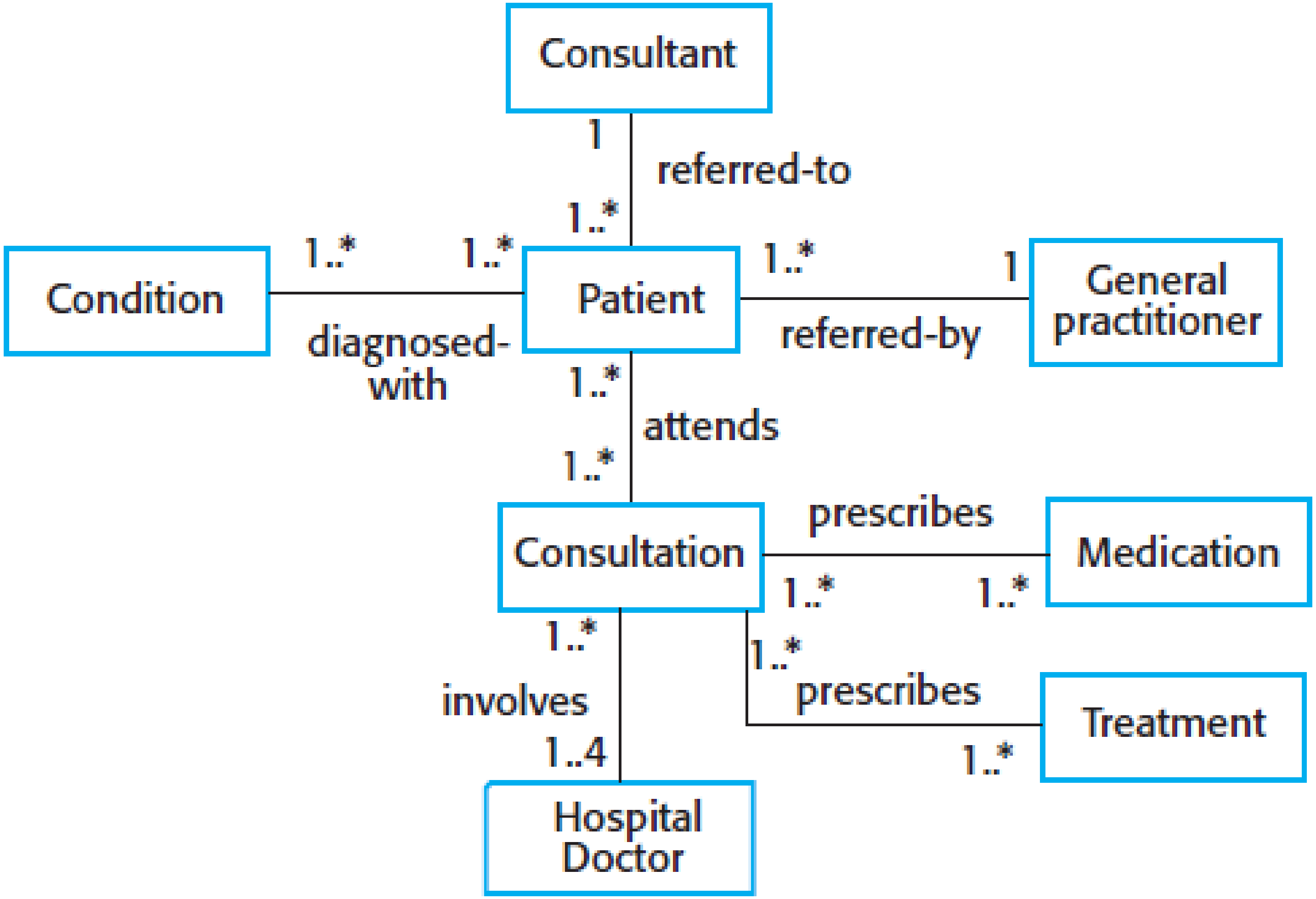
A more complex example (1)

- 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



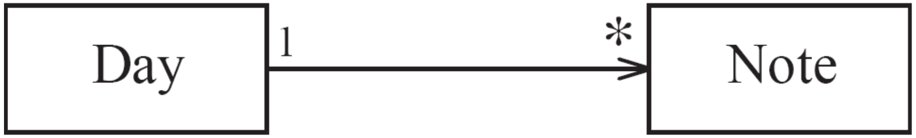
- The *frame* around this diagram is an optional feature that any UML 2.0 diagram may possess.

A more complex example (2)



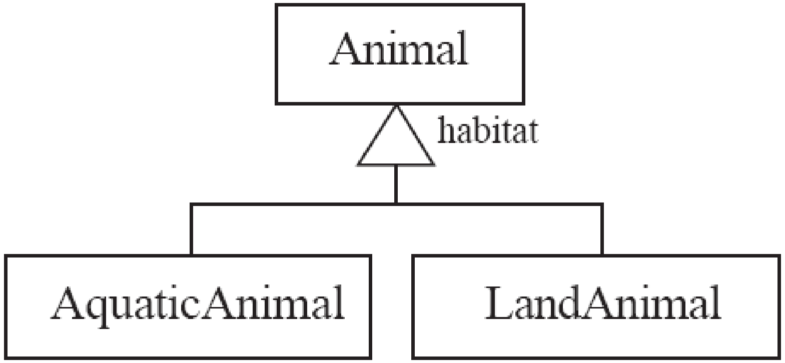
Directionality in associations

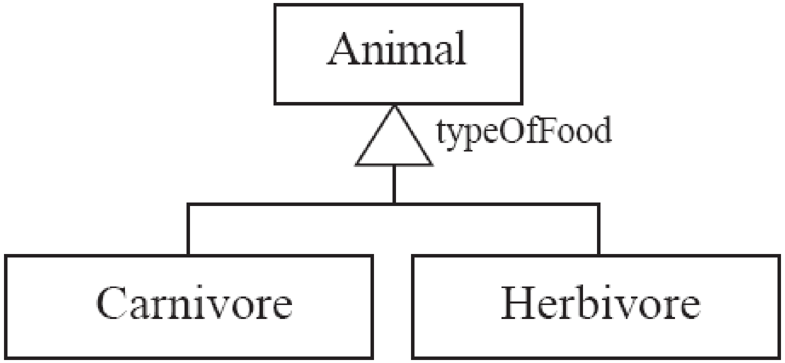
- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end



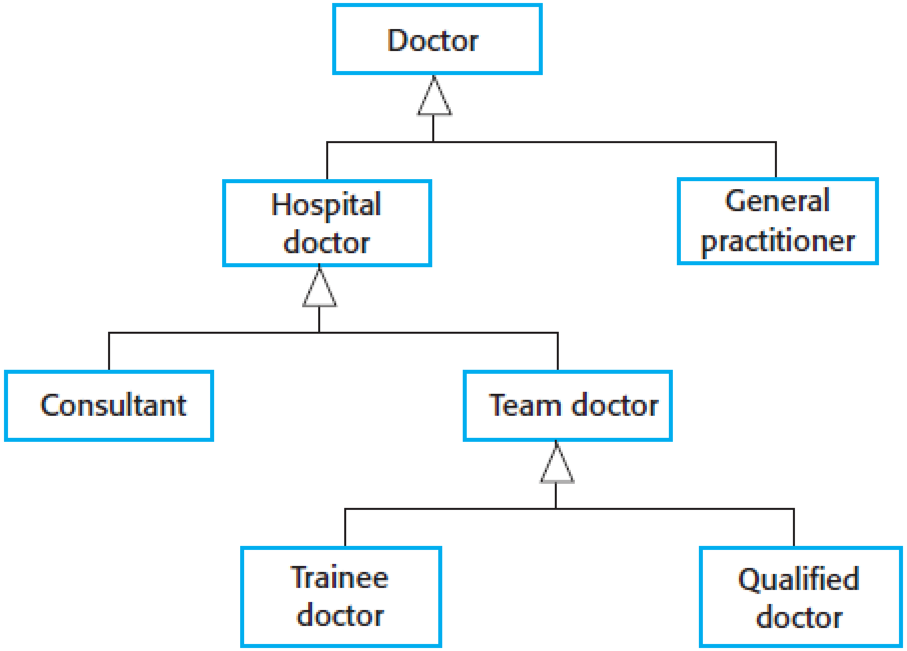
Generalization

- 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





Generalization Example


Doctor

Hospital doctor

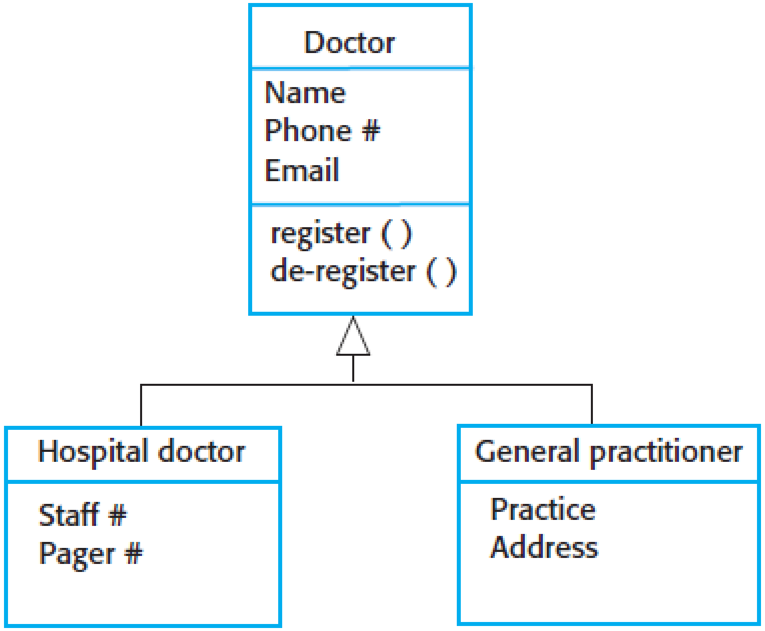
General practitioner

Consultant

Team doctor

Trainee doctor

Qualified doctor


Doctor

Name
Phone #
Email

register ( )
de-register ( )

Hospital doctor

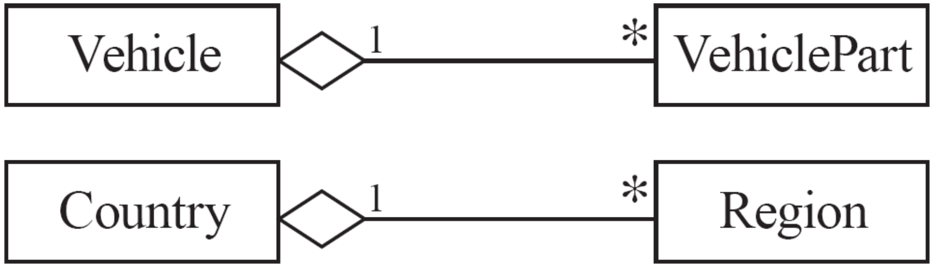
Staff #
Pager #

General practitioner

Practice
Address

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

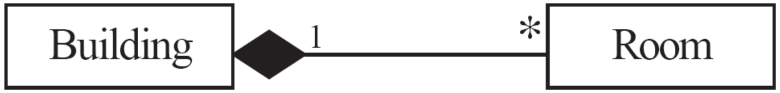


When to use an aggregation

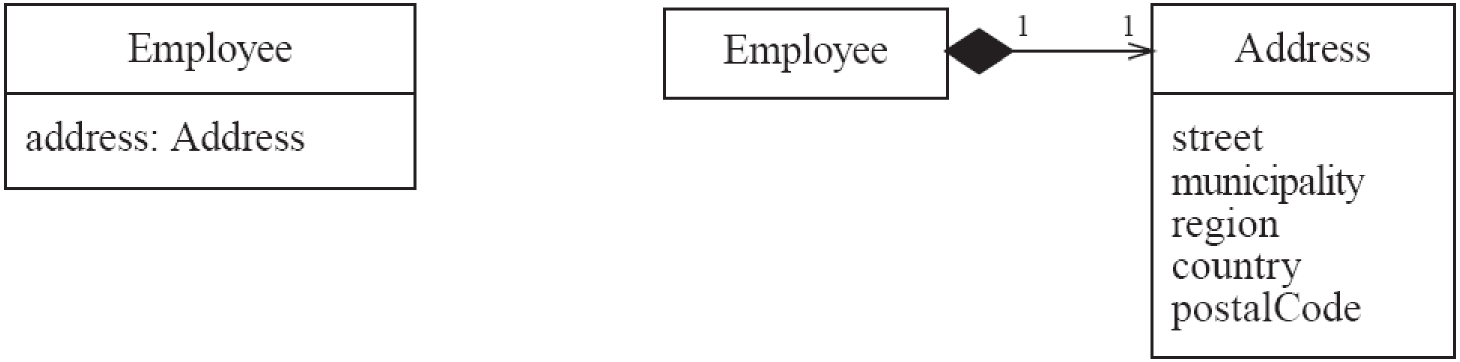
- As a general rule, you can mark an association as an aggregation if the following are true:
  - You can state that
    - the parts 'are part of' the aggregate
    - or the aggregate 'is composed of' the parts
  - When something owns or controls the aggregate, then they also own or control the parts

Composition

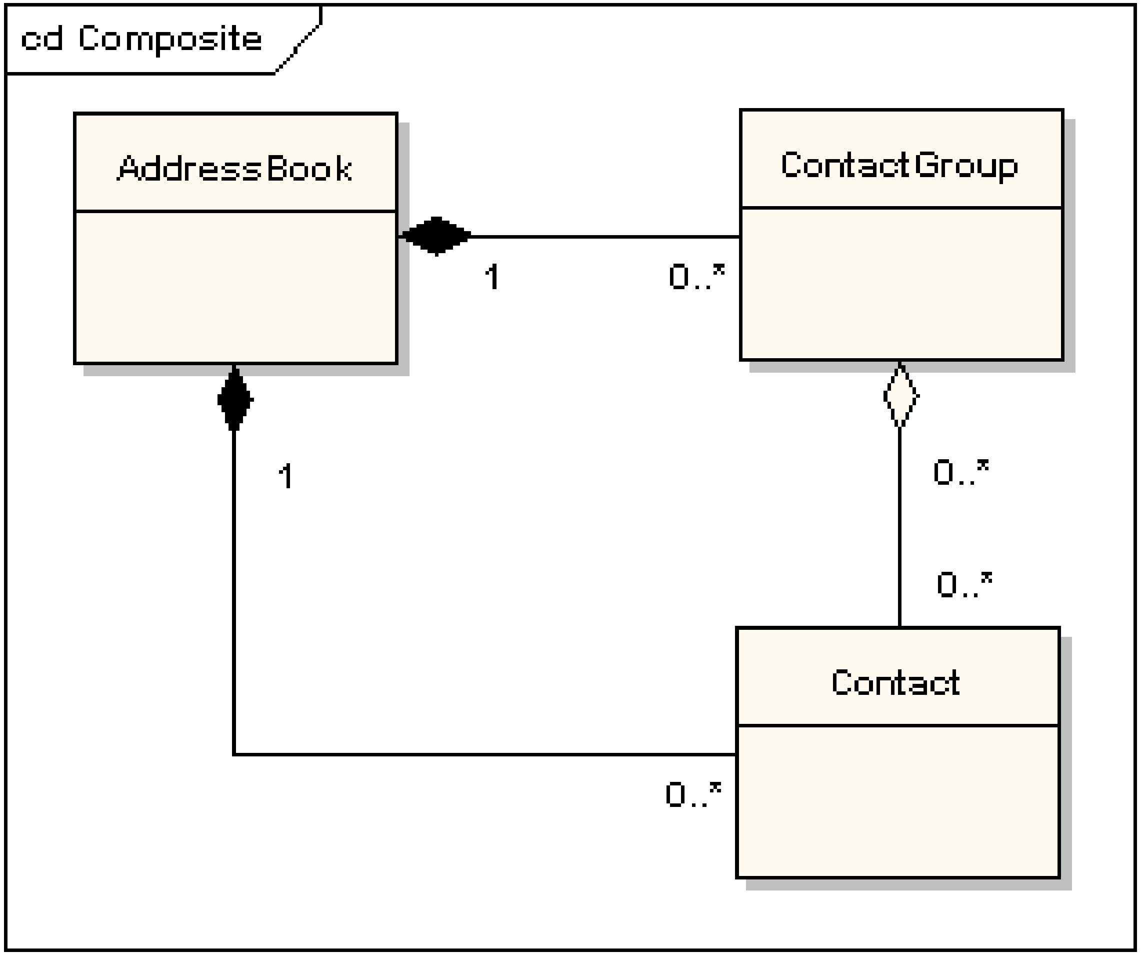
- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well



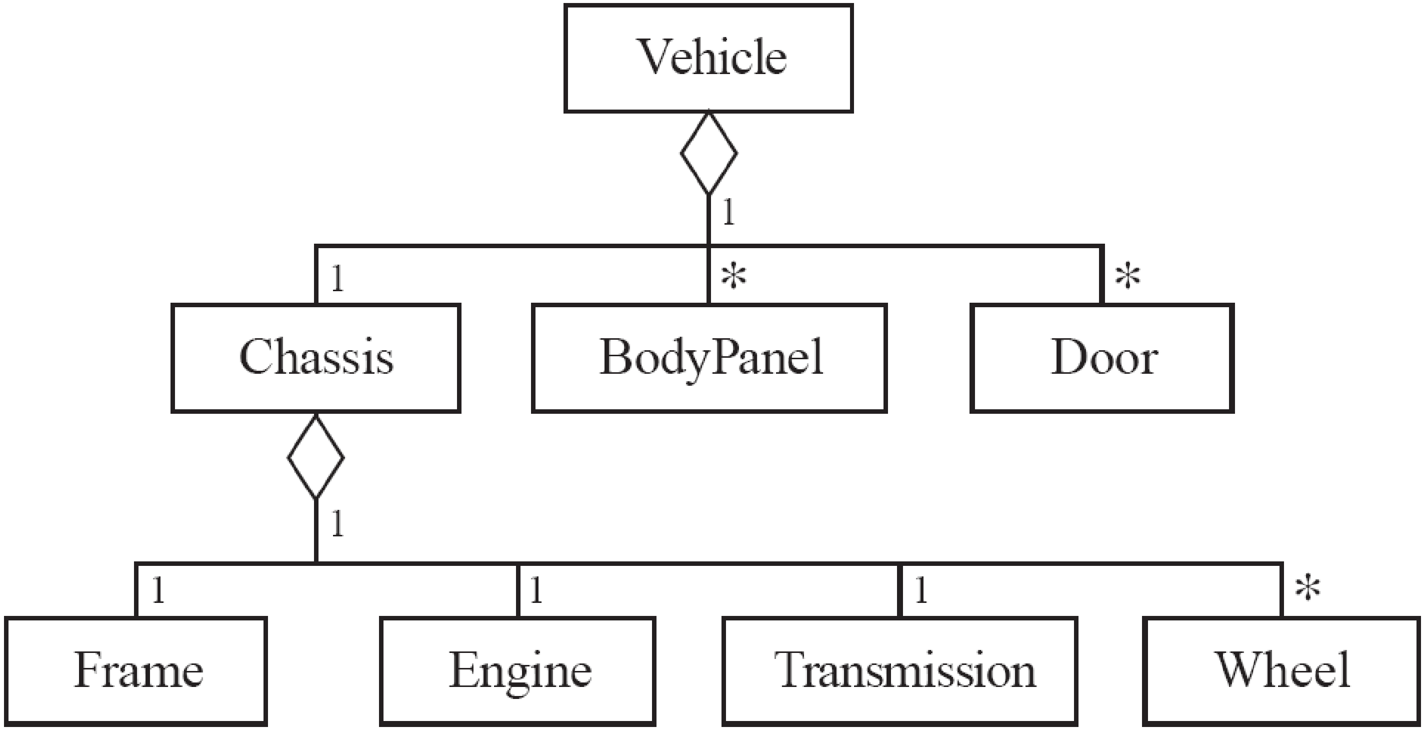
- Two alternatives for addresses



Composition vs. Aggregation

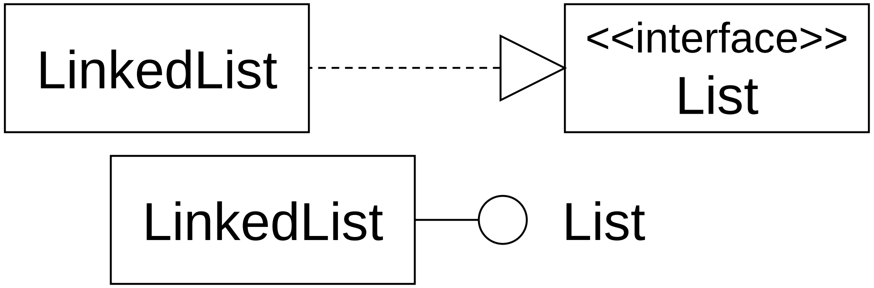


Aggregation hierarchy



Interfaces

- An interface is a bit like a class, except that an interface can only contain method signatures and fields.
- An interface cannot contain an implementation of the methods, only the signature (name, parameters and exceptions) of the method.
- A class can have an actual instance of its type, whereas an interface must have at least one class to implement
- An interface can be realized by many classes.
- A class may realize many interfaces.

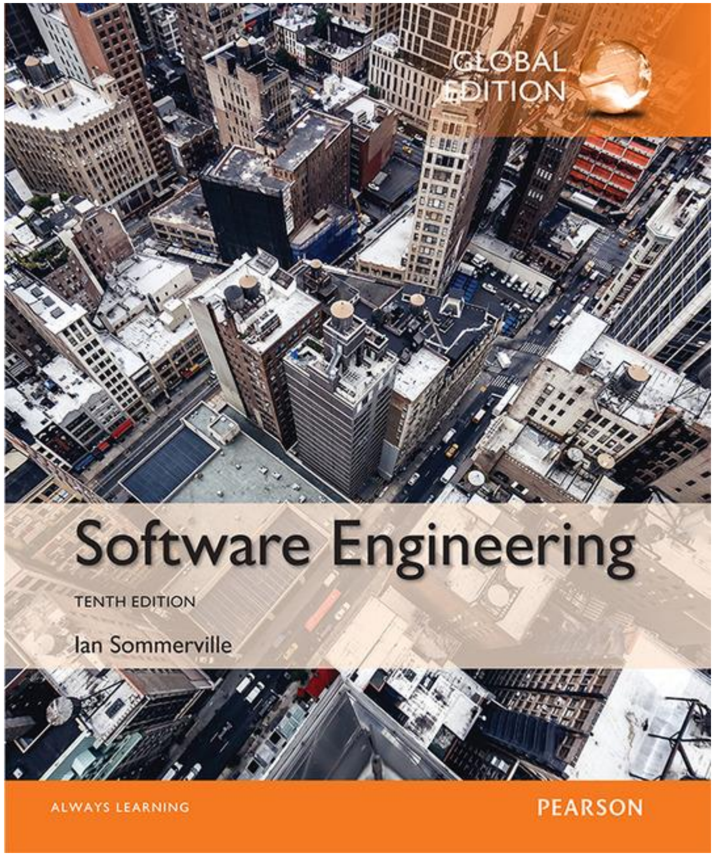


Suggested sequence of activities

- 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.*

Read



Chapter 5 and 7

References

- Ian Sommerville, “Software Engineering”, 10th Edition, Addison-Wesley, 2015.
- Timothy C. Lethbridge and Robert Laganière, “Object-Oriented Software Engineering: Practical Software Development using UML and Java”, 2nd Edition, McGraw Hill, 2001.
- R. S. Pressman, Software Engineering: A Practitioner’s Approach, 10th Edition, McGraw-Hill, 2005.