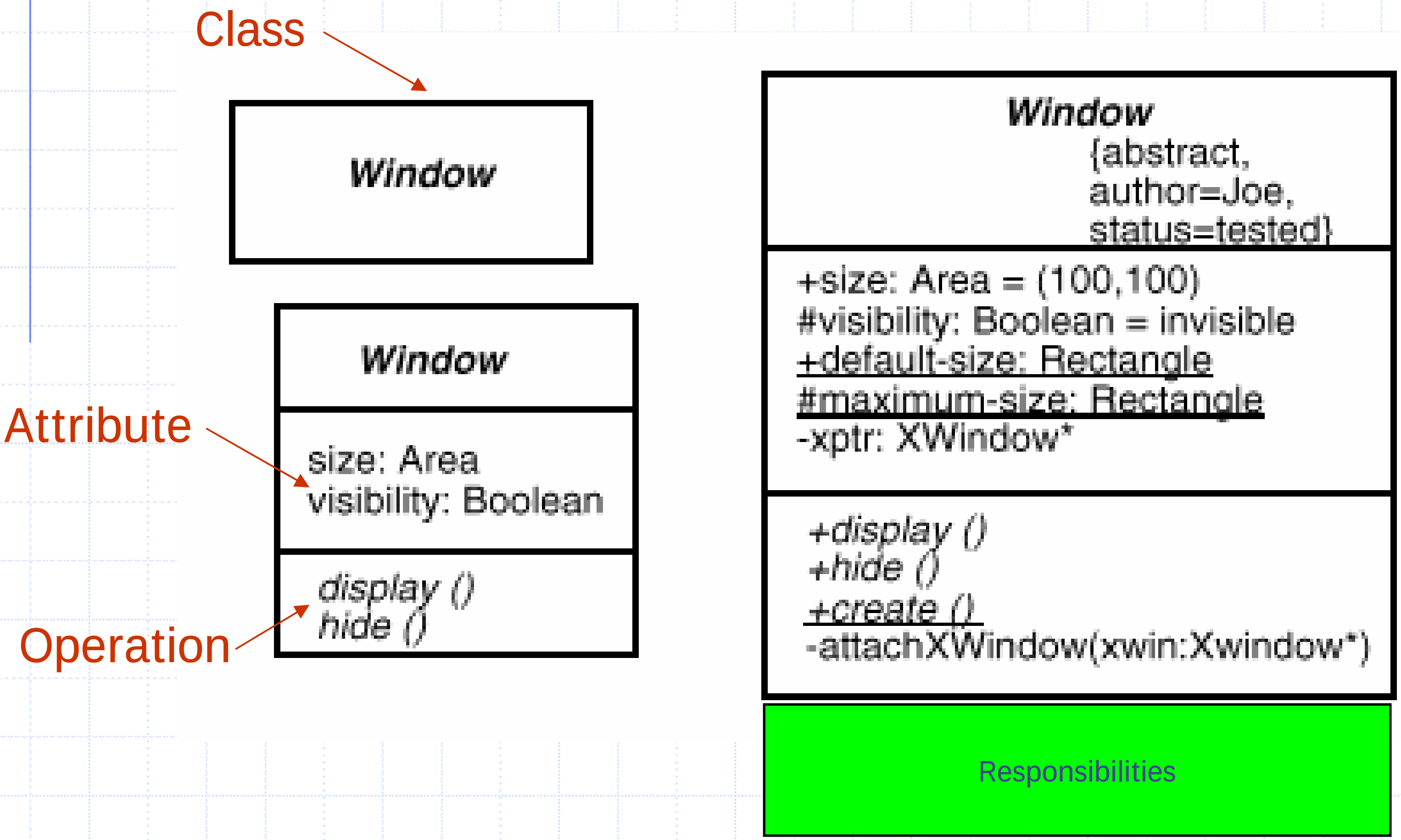
Class Diagram

- Describes the types of objects in the system and various kinds of static relationships that exist among them
- Shows attributes and operations of a class and constraints that apply to the way objects are connected
- Can be drawn from different perspectives:
  - Conceptual
  - Specification
  - Implementation

Class Diagramming Perspectives

- Conceptual Model
  - Represents the concepts in the domain under study
  - Describes business architecture rather than software architecture in a programming language-independent way
- Specification Model
  - Interfaces of software
    - “Type”
      - Can have many classes that implement it
- Implementation Model
  - Defining classes in OO languages
  - Most often used perspective to date

Class Notation



Class

Window

Attribute

Operation

Window

size: Area
visibility: Boolean

display ()
hide ()

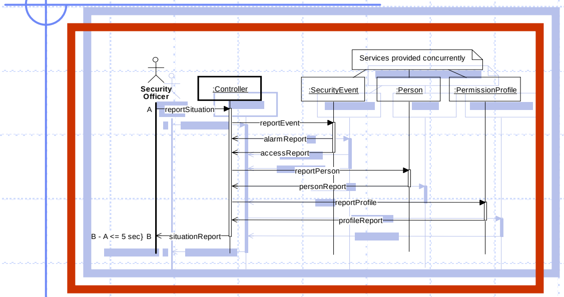
Window
{abstract,
author=Joe,
status=tested}

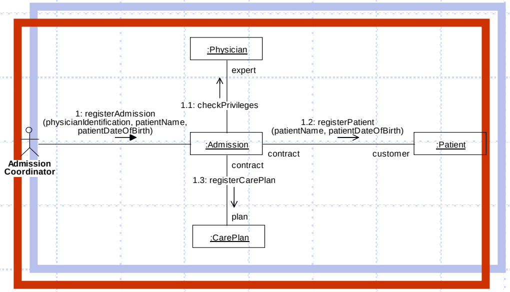
+size: Area = (100,100)
#visibility: Boolean = invisible
+default-size: Rectangle
#maximum-size: Rectangle
-xptr: XWindow*

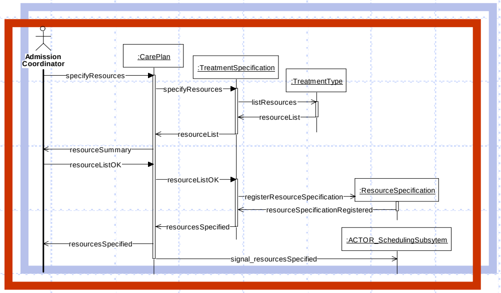
+display ()
+hide ()
+create ()
-attachXWindow(xwin:Xwindow*)

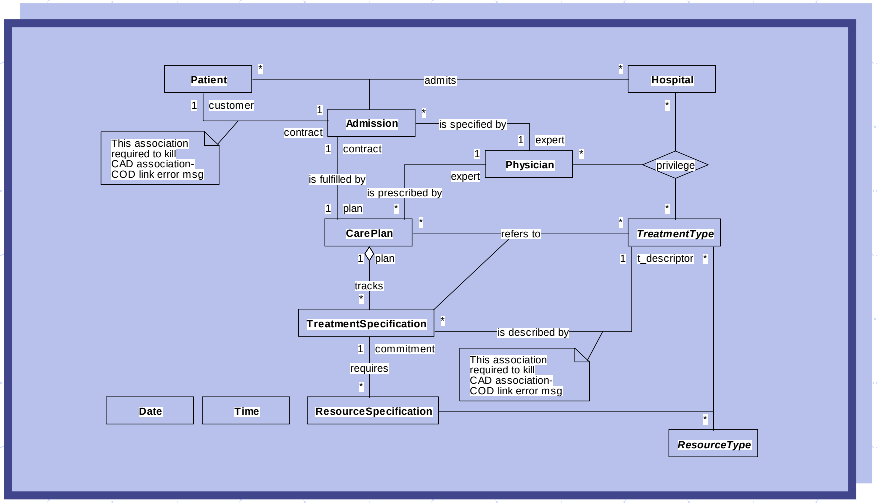
Responsibilities

Relationship between Classes and Interaction Diagrams



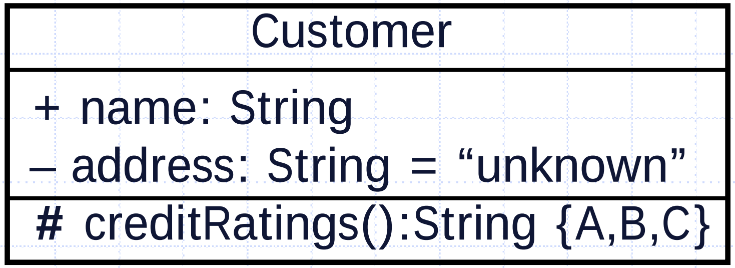






Interaction diagrams coalesce into class diagrams:
objects into classes, links into associations

Attributes


Customer

+ name: String

– address: String = “unknown”

# creditRatings():String {A,B,C}

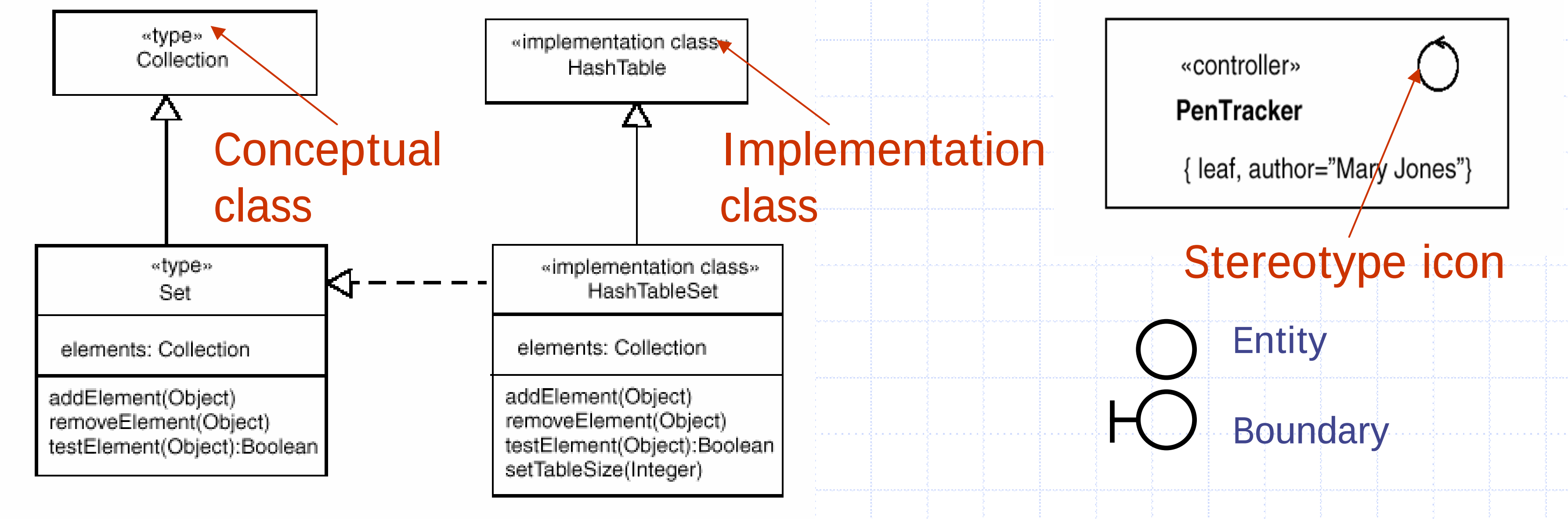
- Conceptual level: semantics as in ERD
  - E.g., Customers have names
  - Responsibilities e.g., a Customer object can tell its name (knows) and has some way of setting a name
- Implementation level: physical implementation
  - E.g., a Customer has a field (i.e., instance variable) for its name
- Syntax: *visibility name : type-expression = initial value {property-string}*
  - Visibility: + (public), # (protected), - (private)
  - Name may not have a pefix or suffix
- Class-scoped attributes (static attributes) are underlined.
  - E.g., - number of invoices: integer for class Invoice

Operations

- Conceptual level: a few words summarizing the principal **responsibility** of the class
- Implementation level: public, protected and private methods
- Operations that simply manipulate attributes are normally not shown
- Syntax: *visibility name (parameter-list) : return-type-expression { property-string }*
- Name, parameters and return type are together called *signature* of the operation
- Class-scoped operations (which can access only class-scoped attributes) are underlined
  - e.g., <u>getCounter (): Integer</u> that retrieves the value of <u>car counter</u> attribute of class Car

Stereotypes and Properties

- A persistent class is described by putting {persistent} property in the name compartment
- Conceptual, and implementation classes are described using «type» and «implementation class» stereotypes
- Classes can be stereotyped to «entity», «boundary» (or «interface») and «controller», based on the model-view-controller pattern (next classes)



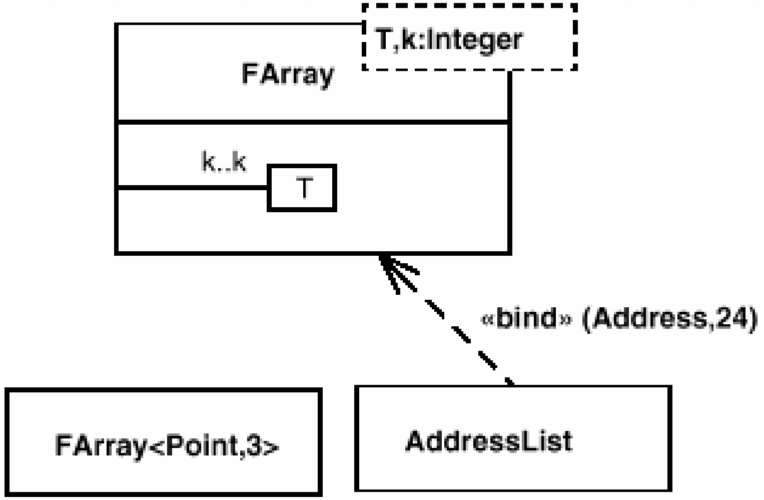
Conceptual class

Implementation class

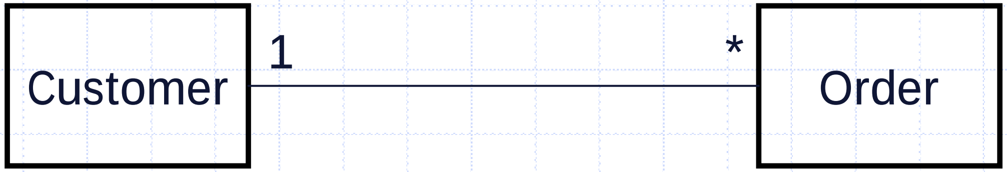
Stereotype icon

Parameterized Classes

- Defines a family of classes, each class specified by binding the parameters to actual values
- Attributes and operations within the template are defined in terms of the formal parameters



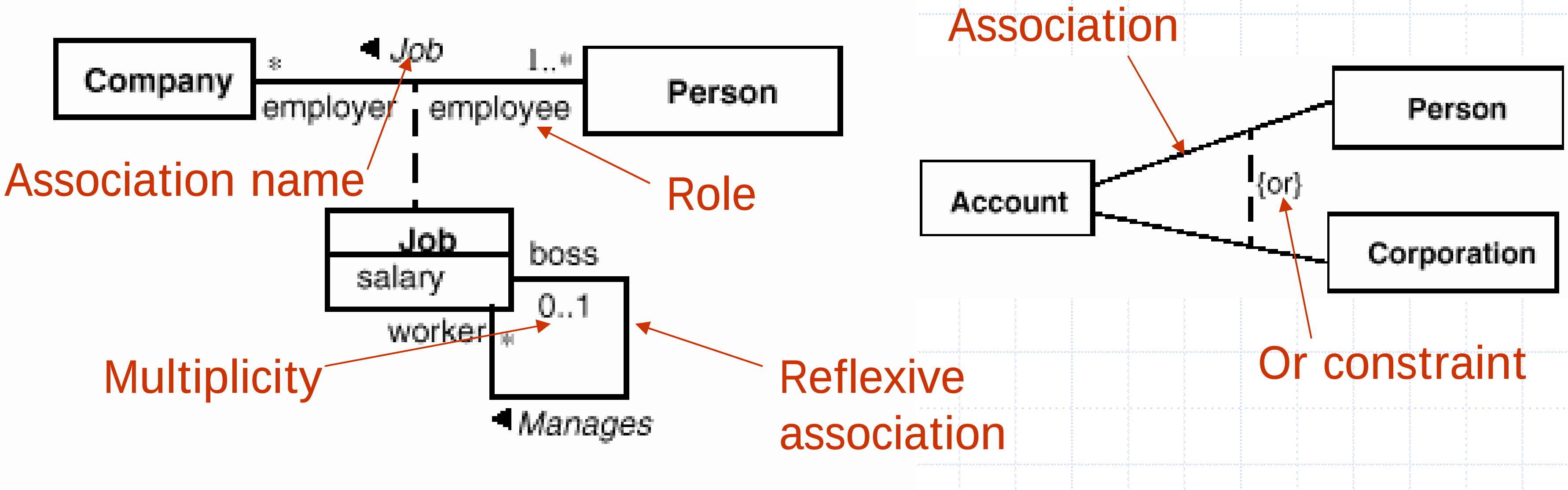
Associations



- Conceptual level: semantics as in ERD
    - E.g., an Order must come from a single Customer, and a Customer may make several Orders over time
- Specification level: responsibilities (i.e., interface)
    - E.g., there are one or more methods assoc. w/ Customer that tell what orders a given Customer has made, and those assoc. w/ Order telling which Customer placed a given Order; further, Customer could be specified in the constructor for Order
- Implementation level: physical implementation
    - E.g., Customer has a field that is a collection of pointers to Orders

Association

- Each association has two roles, one for each direction
  - Role name: verb phrase or noun (responsibility or operation)
- Multiplicity
  - 0..1; 1 (default); * (0..∞); 1..*; 2,4; 5
- Navigability: An arrow at the end of the association line indicates that the assoc. can be used in only that direction
  - Meaningful only in spec. and implem. diagrams



Association

Association name

Role

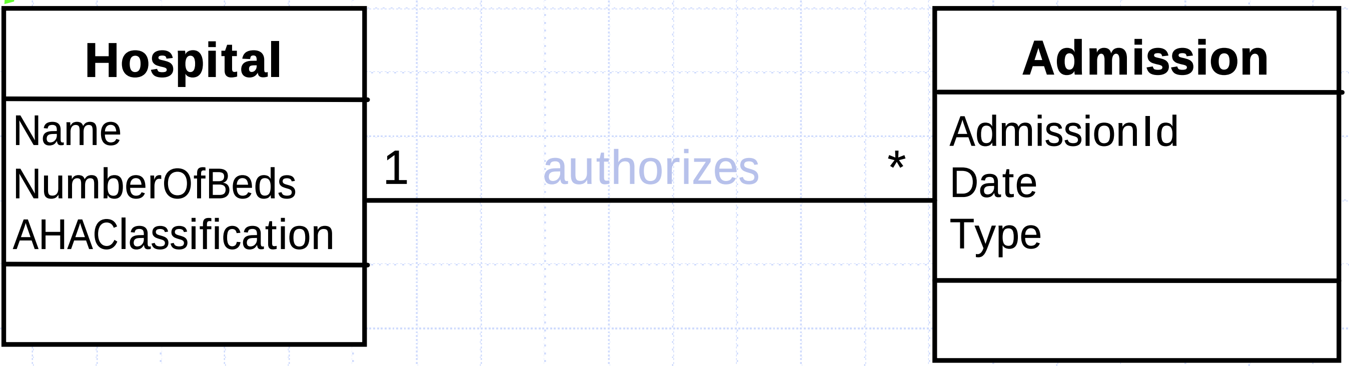
Multiplicity

Reflexive association

Or constraint

Qualified Associations

Unqualified



Hospital

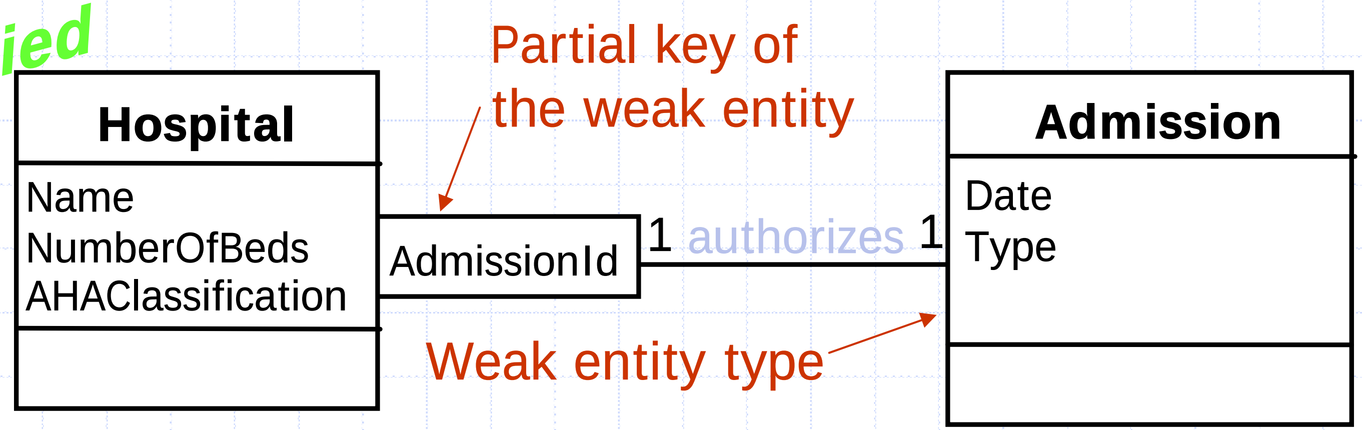
Name
NumberOfBeds
AHAClassification

1 authorizes *

Admission

AdmissionId
Date
Type

Qualified



Hospital

Name
NumberOfBeds
AHAClassification

AdmissionId

1 authorizes 1

Admission

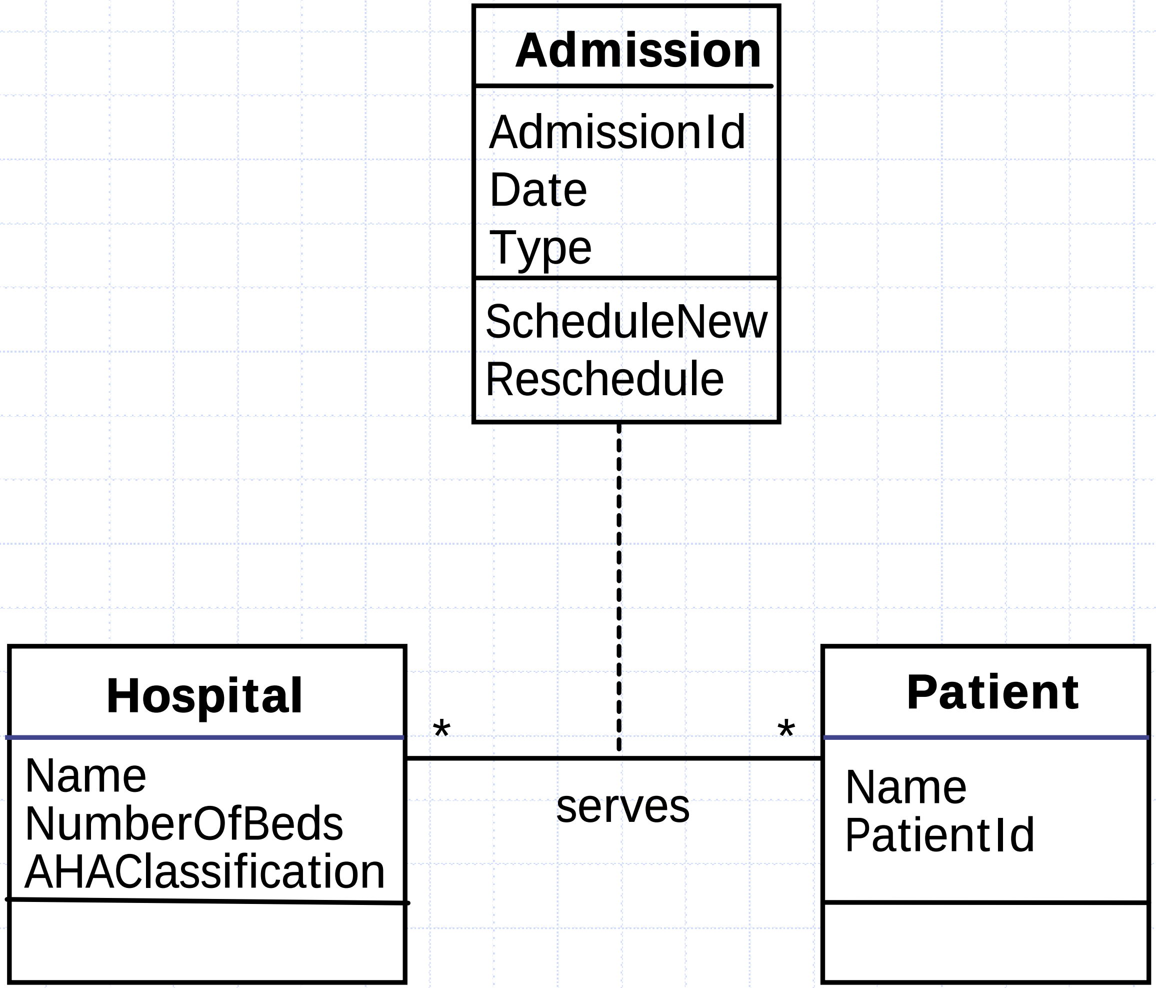
Date
Type

Partial key of
the weak entity

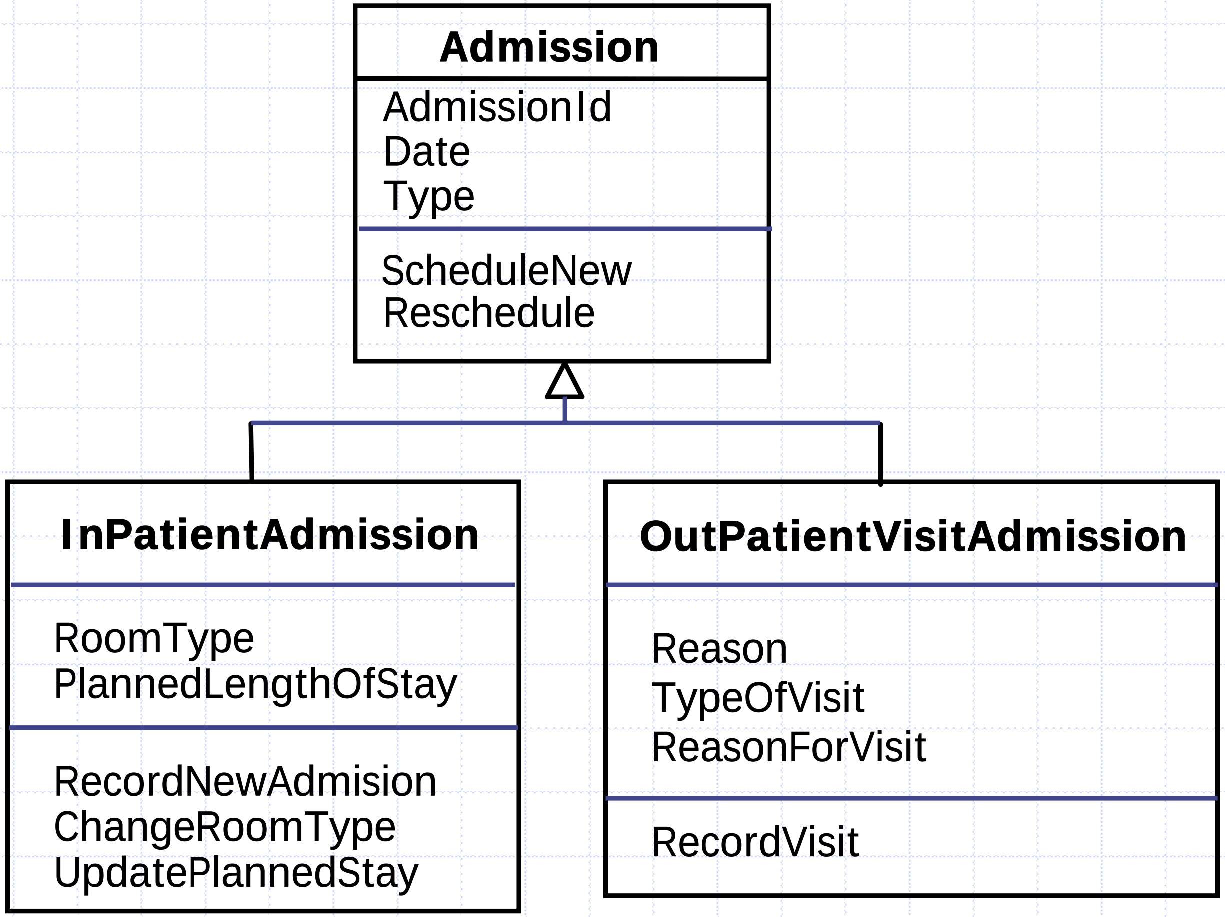
Weak entity type

Given a Hospital and an AdmissionId,
there can be only one Admission

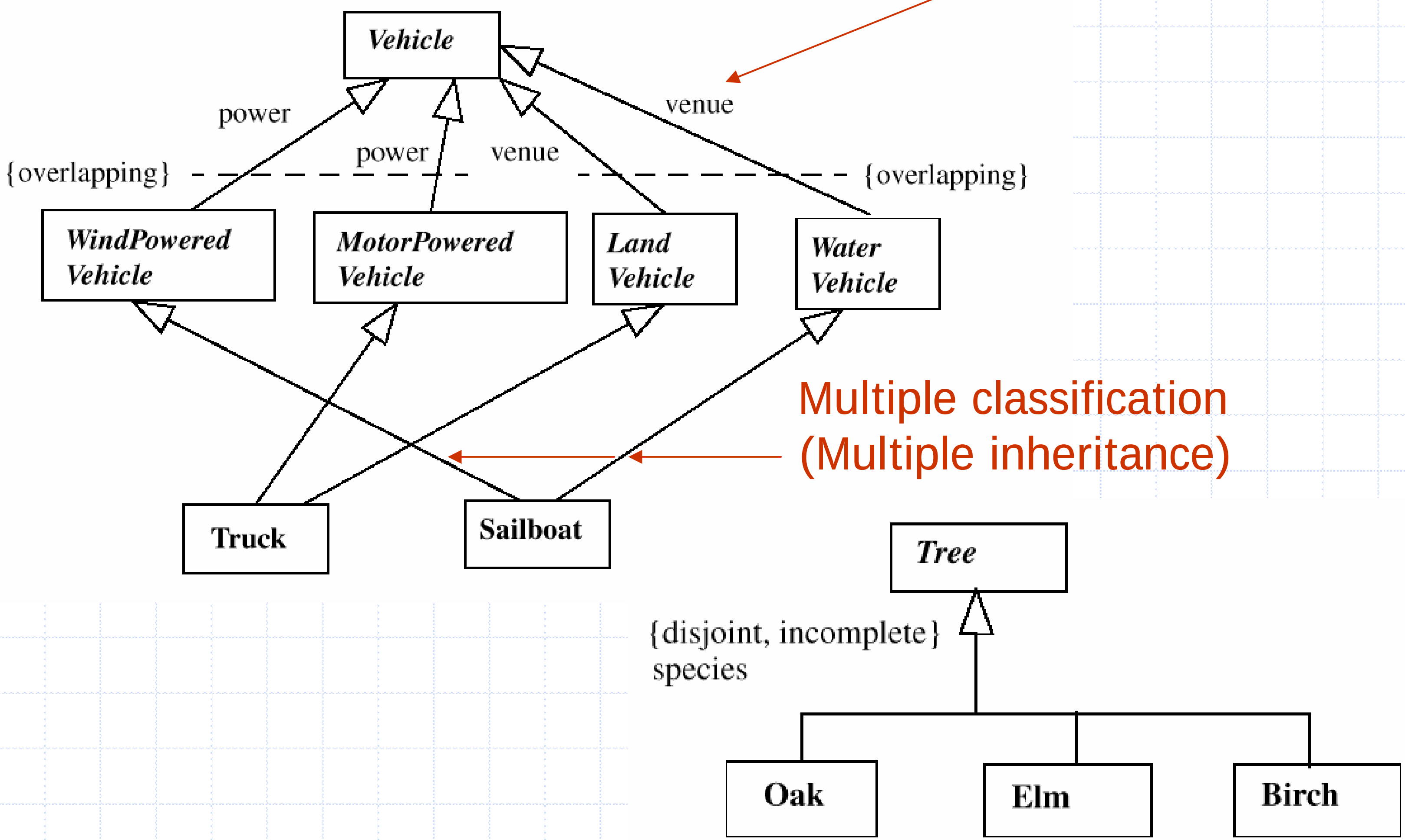
Example: Association Class



Example: Generalization



Classification

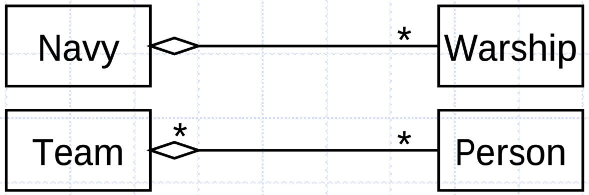


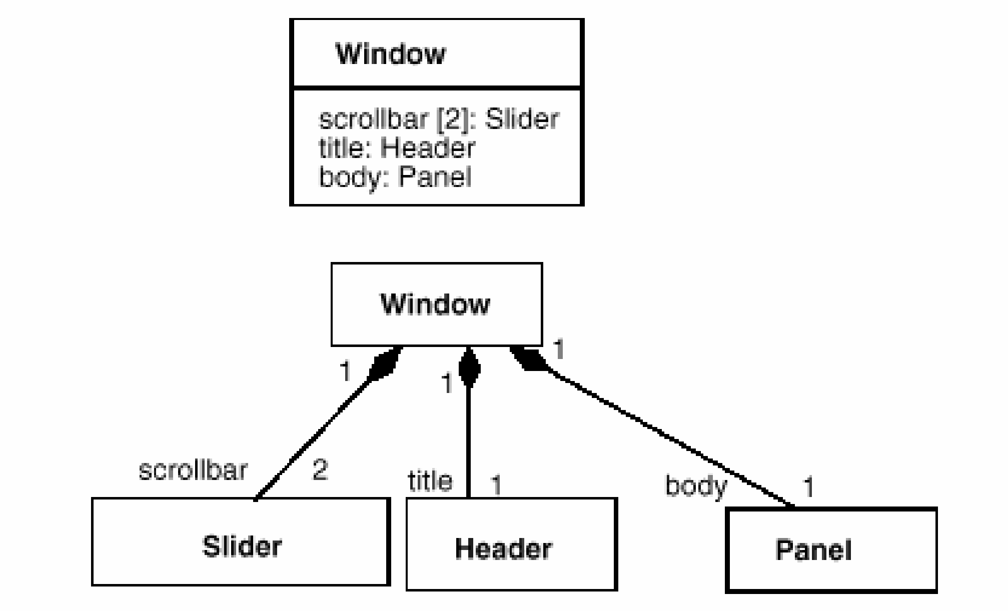
Discriminator

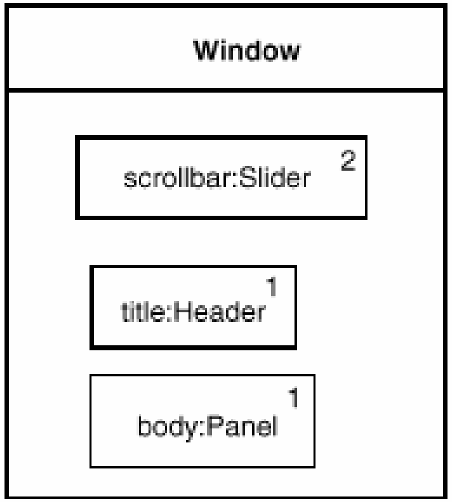
Multiple classification (Multiple inheritance)

Aggregation

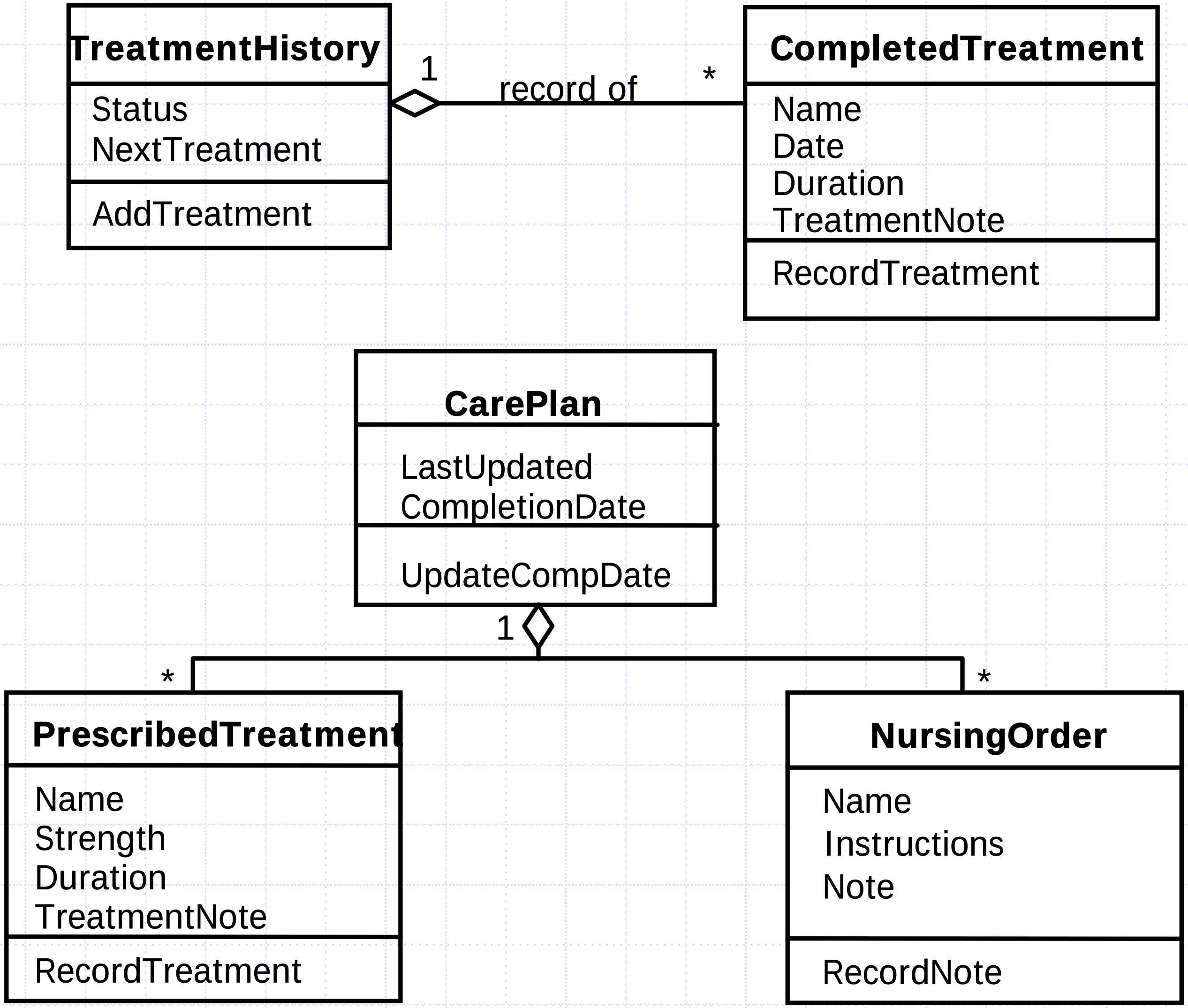
- Whole-part relationship
  - normal aggregation - 1:N
  - shared aggregation - M:N
  - **composition** aggregation - 1:N owner relationship





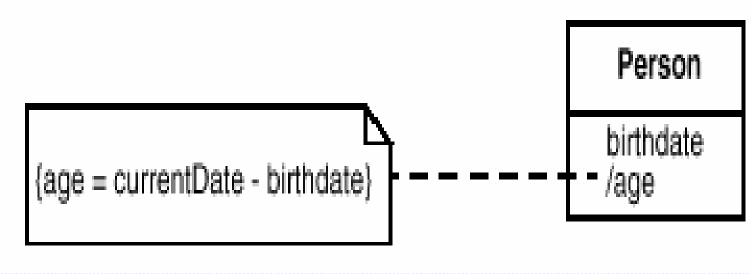


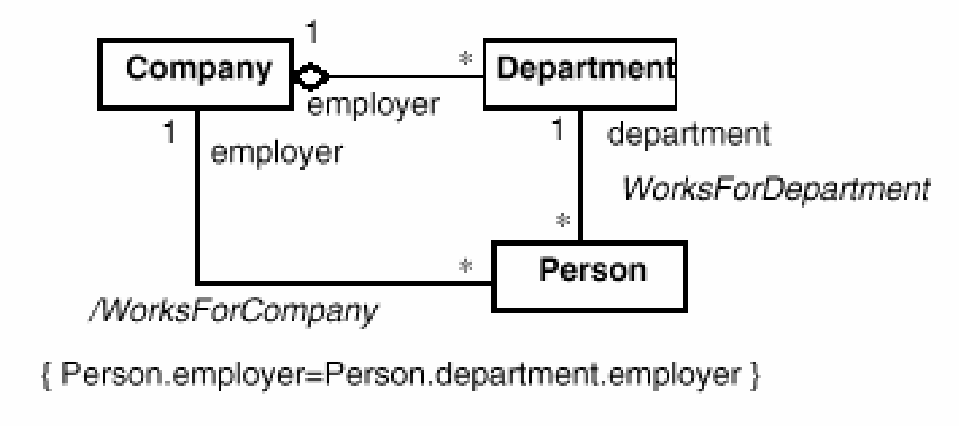
Example: Aggregation



Derived Association / Attributes

- Derived assoc. and attributes can be calculated from other assoc. or attributes, respectively
- {Frozen} constraint can be used to indicate for an attribute or role, that its value may not change during the lifetime of the source object
- {Read-only} indicates that a value cannot be changed directly, but may change due to a change in some other values


Person
birthdate
/age
{age = currentDate - birthdate}


Company
1
employer
*
Department
1
employer
1
department
WorksForDepartment
*
*
Person
/WorksForCompany
{ Person.employer=Person.department.employer }