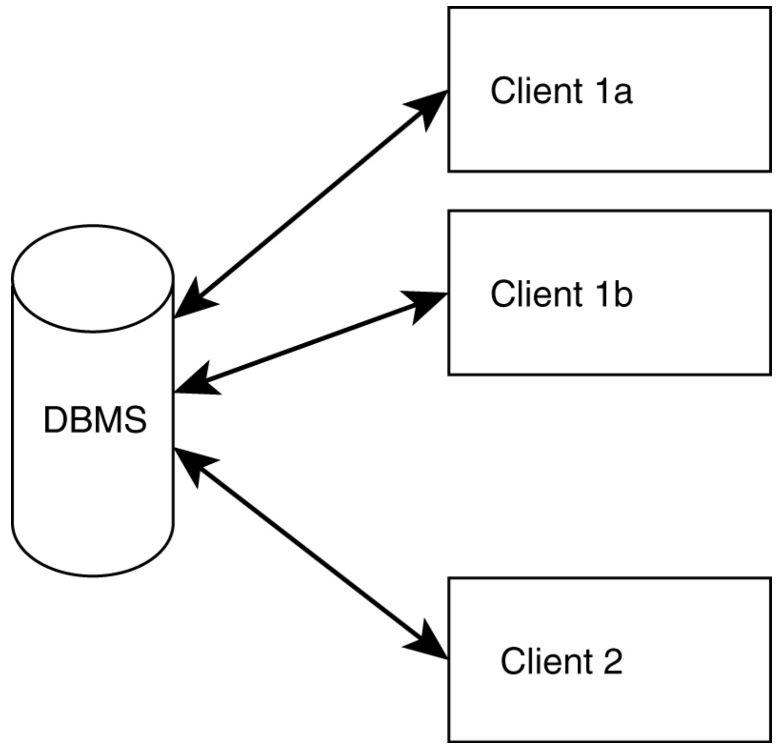
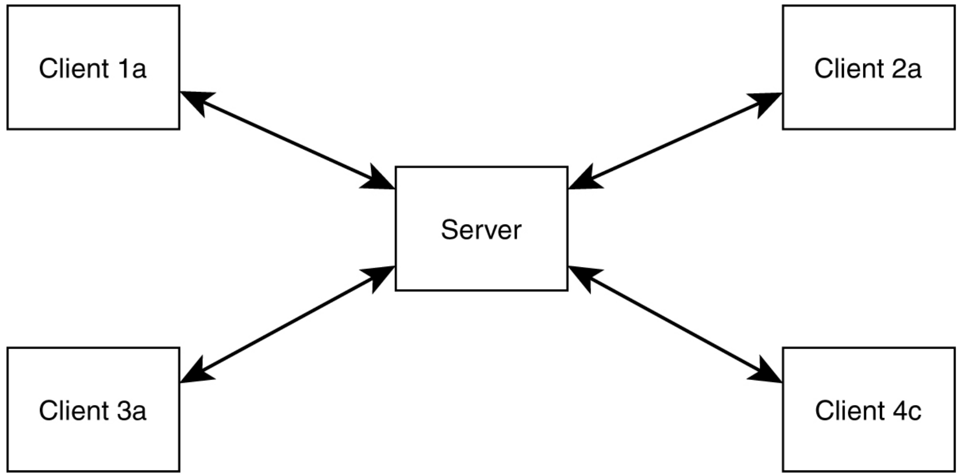
Data-Centric Architecture



- Components are data store and clients
- Connectors are access from clients to data store
- Constraint is the clients are independent

Client-Server Architecture



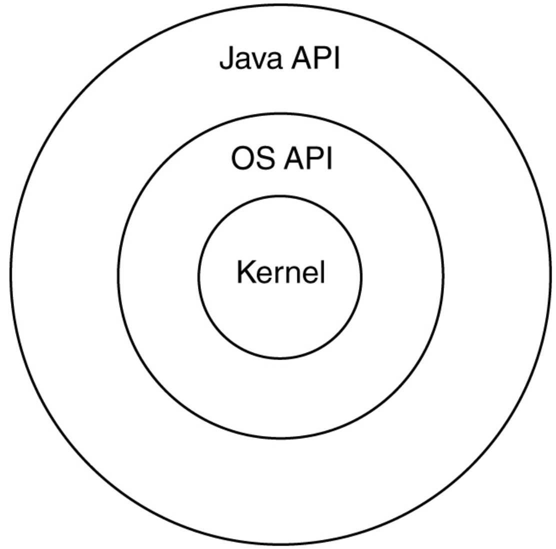
(a)

- A variant of data-centric
  - Processing shared between client and server
- Components are server and client
- Connectors are access from clients to server
- Constraint is the clients are independent

Data-Flow Architecture

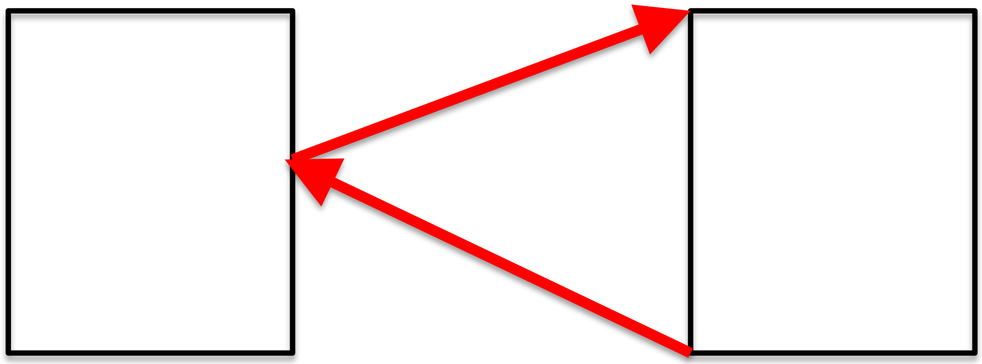
- Components are elements that do processing (typically called **filters**)
  - Transform or perform computations on data
- Connectors transport data between components (typically called **pipes**)
- Constraint is filters are independent

Layered Architecture



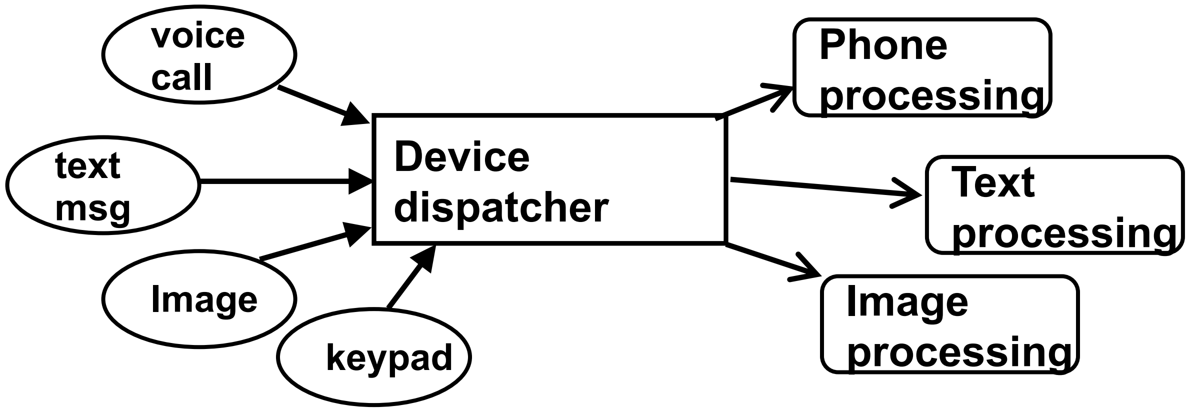
- Outer layers are more abstract, inner are more concrete
- Components are layers
- Connectors are calls between layers
- Constraint is layers can only communicate with adjacent layers

Call and Return Architecture



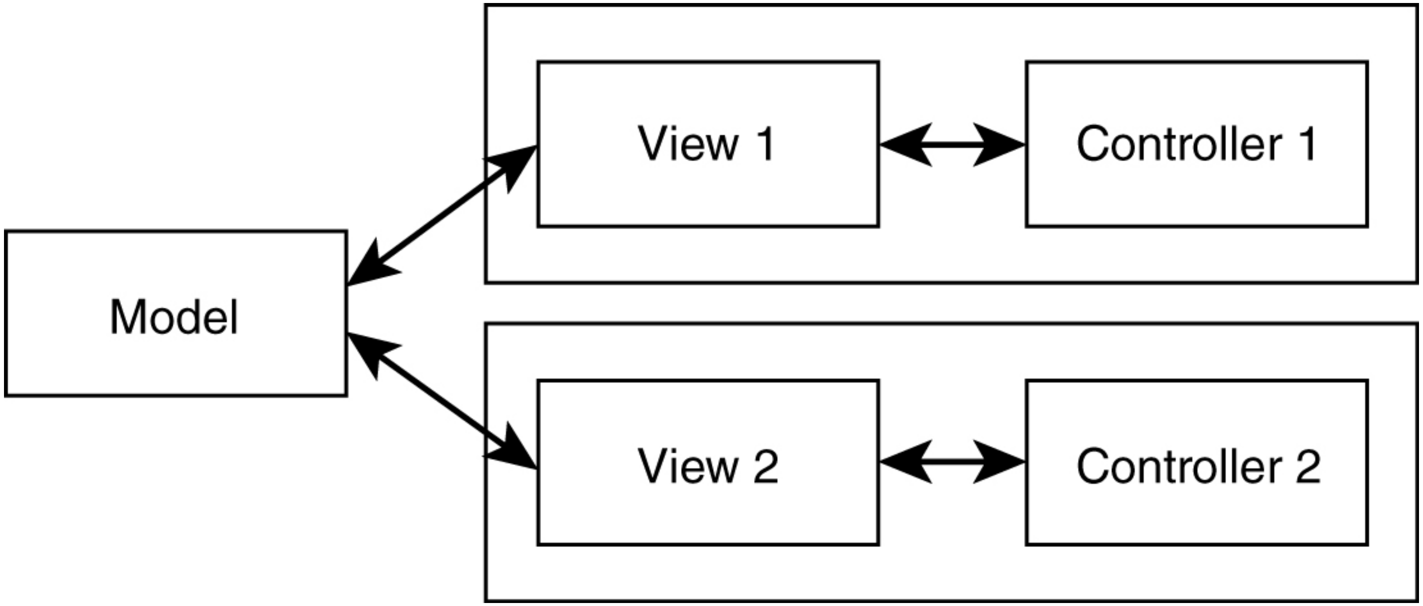
- Components are functions
- Connectors are function calls
- Constraint is LIFO
- Advantages
  – Simple, functional view of world
- Disadvantages
  – Control is distributed, state is hard to represent

Event-Based Architecture



- Components are modules, objects, functions, …
- Connectors are messages, function calls (explicit invocations), events (implicit invocation)
- Constraints
  – Announcer doesn’t know who’s listening
  – Event handlers are invoked in arbitrary order

Hybrid Architecture (MVC)



- Also known as mediated architecture
- Components are model, views, controllers
  - Controllers not needed in modern GUIs
- Connectors are data transfers between components
- Constraint is asynchronous operation

Object-Oriented Architecture

- Components are objects (not classes)
- Connectors are messages sent between objects
- Constraint is objects must know identity of another object to send a message
- Advantages
  - Data encapsulation
  - Reuse
- Disadvantage is need to know identity of other object

OO Concepts

- **Classes** are data types that hold variables (**attributes**) and operations (**methods**) to manipulate those
- **Objects** are instances of classes
- **Messages** are triples
  - Name of method
  - Parameters
  - Name of destination object

OO Concepts (cont.)

- Classes support
  - **Abstraction**
    - Hiding details to facilitate understanding
  - **Encapsulation**
    - Protecting state behind an interface
    - Makes reuse easier
  - **Information hiding**
    - Hiding design decisions behind an interface
    - Makes maintenance/re-implementation easier

OO Concepts (cont.)

- Classes naturally form a hierarchy
  - **Superclass/subclass**
  - Subclass is often a specialization of its superclass
    - Sub “IS_A” super
    - Subclass **inherits** attributes and methods of superclass
      - Can redefine inherited methods
      - Can define additional methods
      - CANNOT remove inherited methods
      - Example: bird class with method fly, subclass penguin
        - » Redefine fly to generate error message for penguin
        - » Have two subclasses of bird (flying and flightless), then penguin is a subclass of flightless

Inheritance Example

Kingdom: Animal
- Phylum: Vertibrate
  - Class: Mammal
    - Order: Primate
      - Family: Hominids
        - Genus: Homo
          - Species: Sapiens

Multiple Inheritance Example

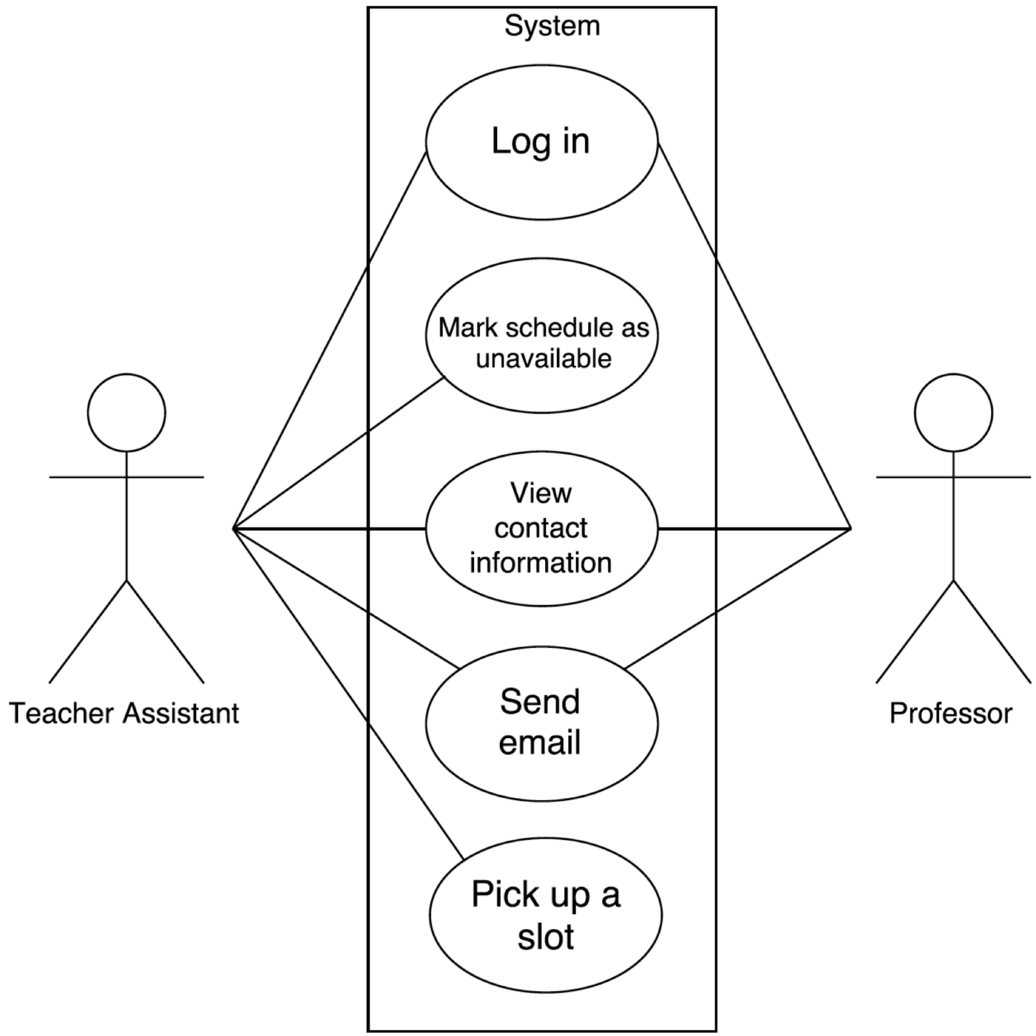
Shape
- Ellipse
- Polygon
  - Triangle
  - Quadrilateral
    - Rectangle
    - Rhombus

Square is both a Rectangle and a Rhombus, so inherits from both

Unified Modeling Language

- A graphical language to assist with detailed design of OO systems
  - Includes class diagrams, use cases
- OO Design process
  - Create and refine use cases
  - Decide
    - Which classes to create
    - How classes are related
  - Document using UML

Use Cases

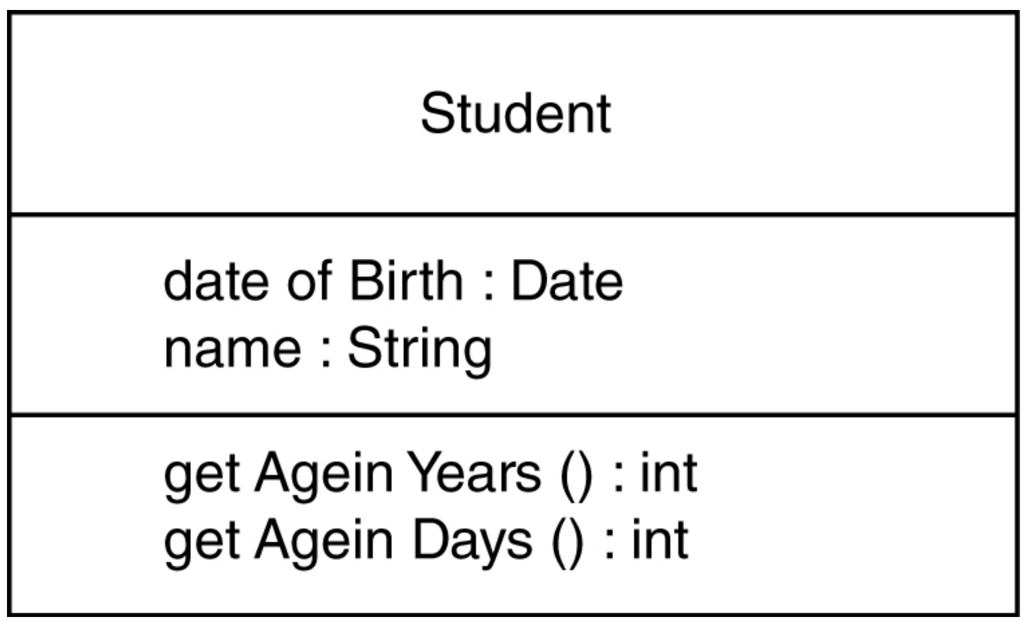

System
Log in
Mark schedule as unavailable
View contact information
Send email
Pick up a slot
Teacher Assistant
Professor

Use Cases (cont.)

- For each use case you need to list
  - Goal
  - Actor(s)
  - Preconditions
  - Steps
  - Postconditions
  - Exceptions

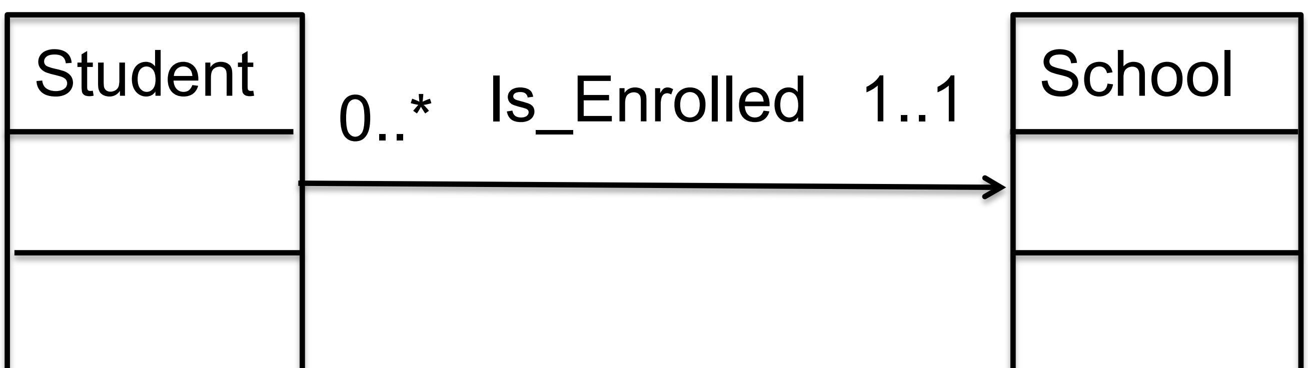
Class Diagrams

- Classes are rectangles with boxes for
  - Class name
  - Attributes
  - Methods
    - + means public, - means private, # means protected



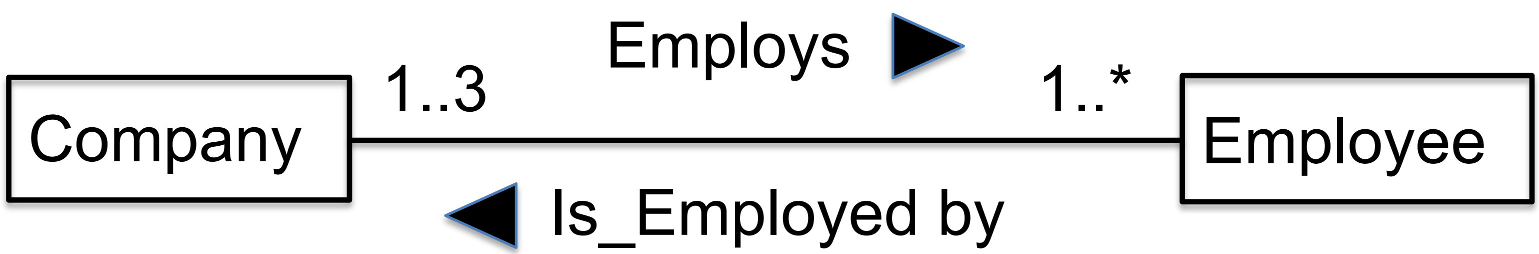
Class Diagrams (cont.)

- **Association** indicated with arrow from one class to another
  - Association is labeled
  - **Cardinality** (number of instances of one class associated with other class) indicated with numeric annotation
    - 1..4, 6
    - * indicates no limit
    - If omitted assume 1



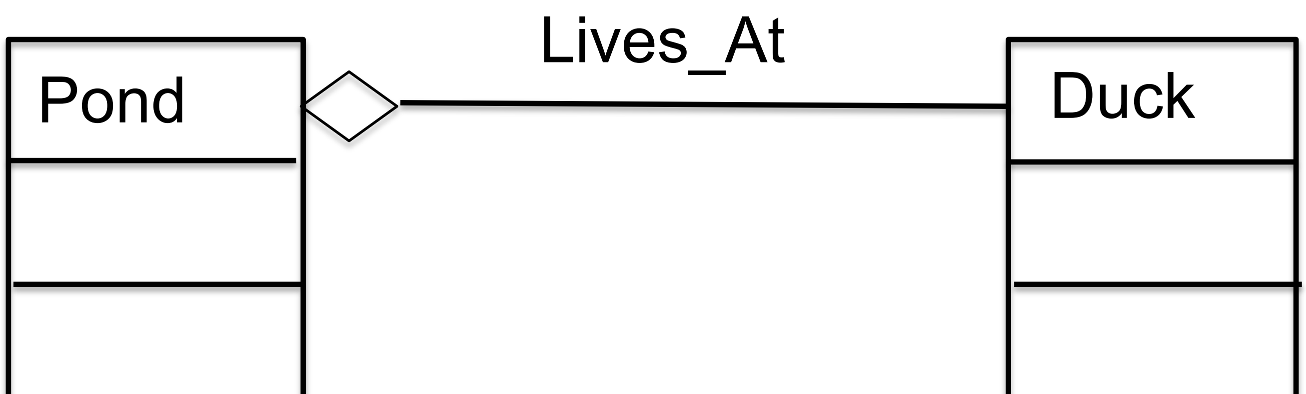
Class Diagrams (cont.)

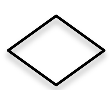
- Sometimes association can be bi-directional



Class Diagrams (cont.)

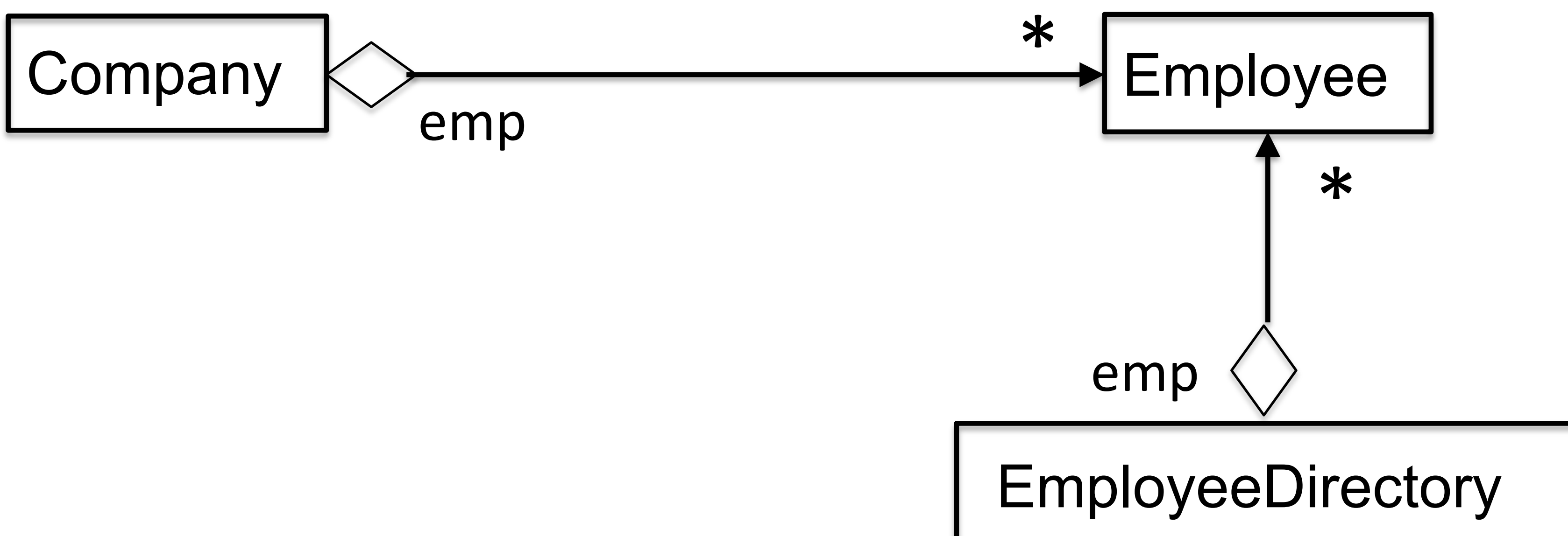
- Aggregation indicated with unfilled diamond from one class to another
  - “Has_a” relationship
  - If pointed to class disappears, pointing class still exists
  - Pond class includes an attribute of type Duck



 Goes on end of association that does the aggregating

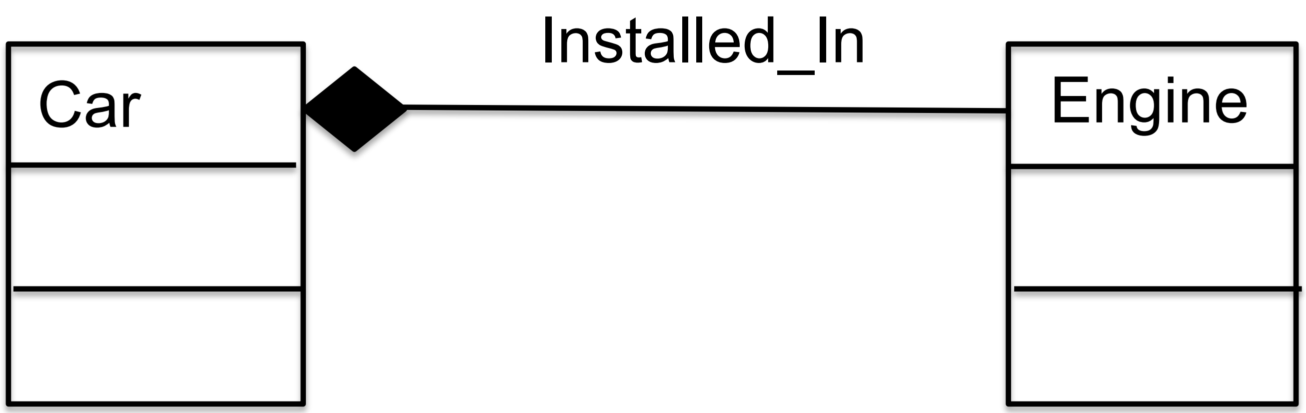
Class Diagrams (cont.)

- Multiple aggregation of classes is possible
- "emp" denotes reference in Company and EmployeeDirectory to aggregated Employee
- One instance of Employee is shared by both



Class Diagrams (cont.)

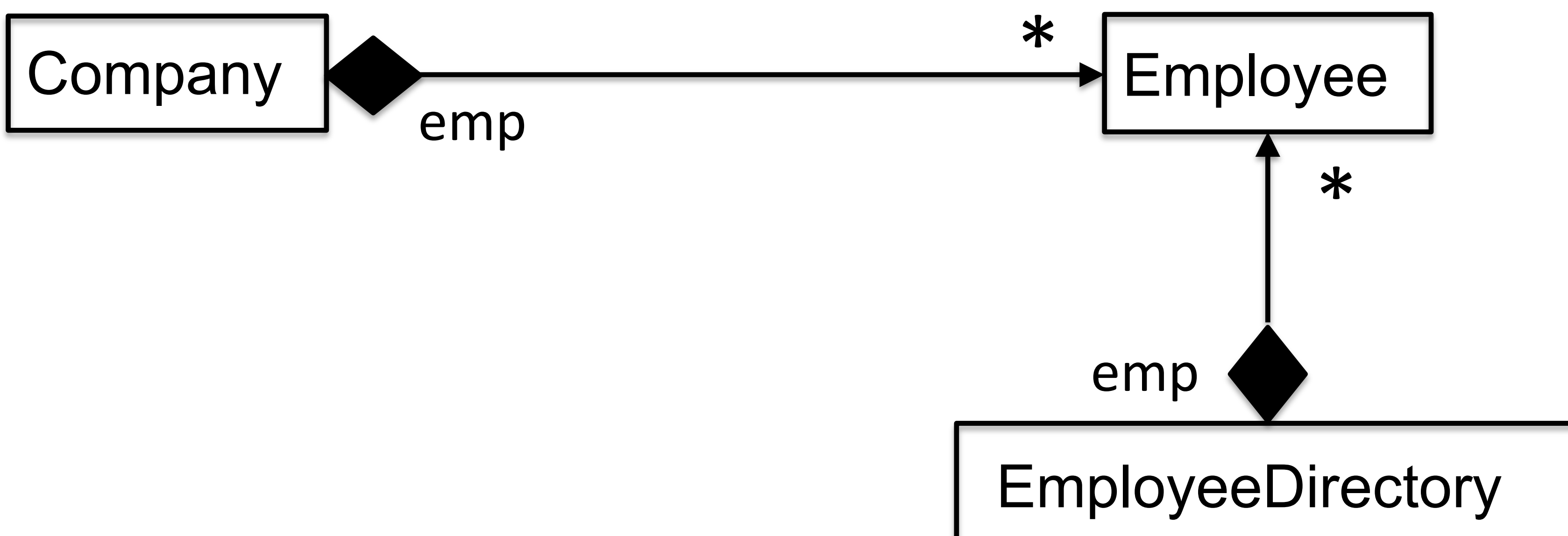
- Composition indicated with filled diamond from one class to another
  - "Has_a" relationship
  - If pointed to class disappears, pointing class disappears



Goes on end of association that does the composing

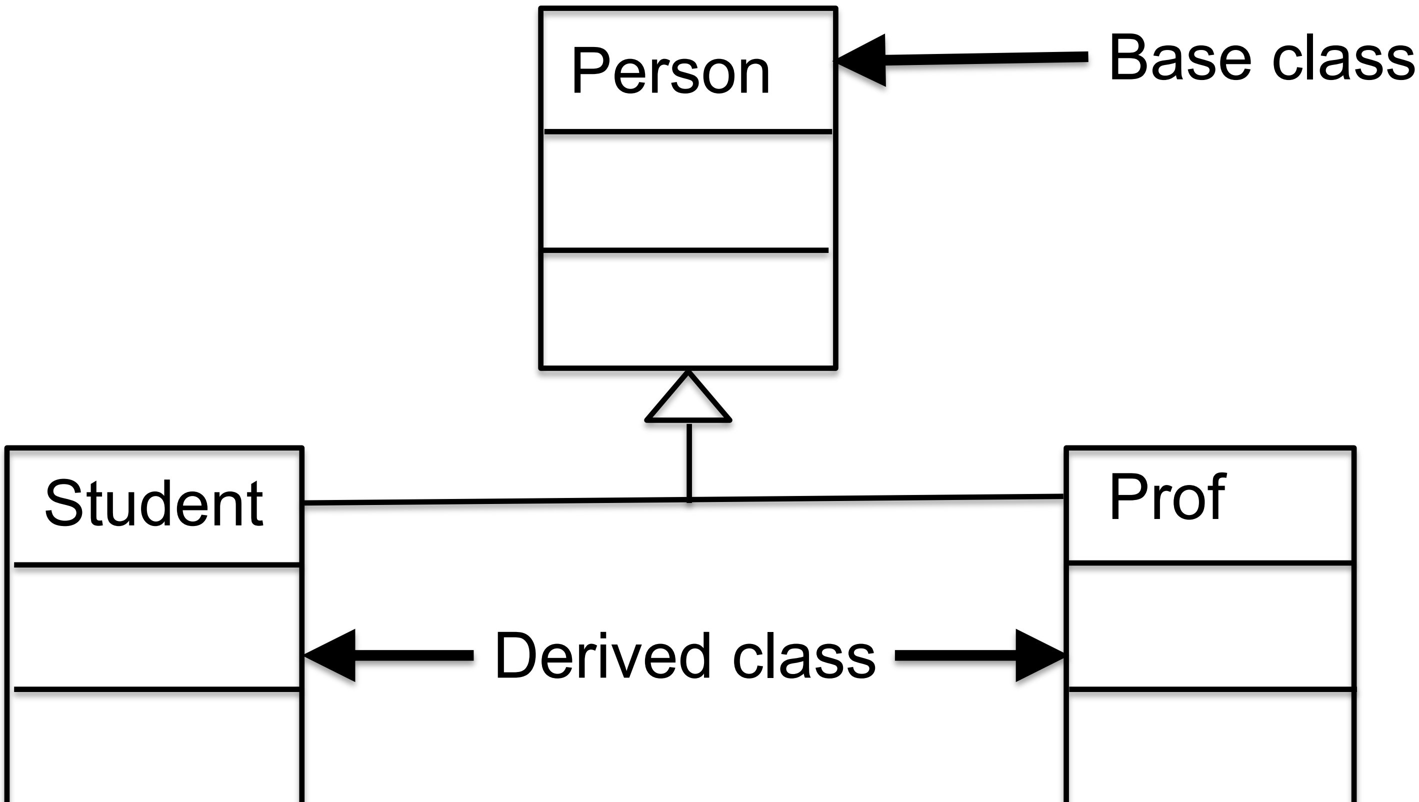
Class Diagrams (cont.)

- Multiple aggregation of classes is possible
- Separate instances of Employee created for Company and Employee Directory



Class Diagrams (cont.)

- inheritance indicated with unfilled arrow from subclass to superclass



Building Classes

- Start from scratch
- Subclass an existing class in the hierarchy
- Add an intermediary class, then subclass that (penguin example)

Architecture Choice Example

- Suppose there are two modules
  - V = a set of vertices
  - E = a set of edges
- Relation G
  - Inserting an edge in E requires verifying that the vertices of that edge are in V
  - Removing a vertex from V required removing all the incident edges from E
- Possible future changes
  - Modify G (to G') by changing the relationship
  - Add a module C which keeps track of the number of vertices in V