UML & OO Fundamentals

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 3a (Continued) — 01/28/2019

Task number one

- If you have your note card from the first class, please place it in front of you…
- If not…

- Get a marker and a piece of card stock
- Fold it in half to make a little tent
- Write the name you'd like to be called on that
- Face the name towards me
- Try to remember to bring that to class for the next several weeks

Acknowledgement & Materials Copyright

- Dr. Ken Anderson is a Professor of the Department of Computer Science and the Associate Dean for Education for the College of Engineering & Applied Science
- Ken taught this OOAD class on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Attendance

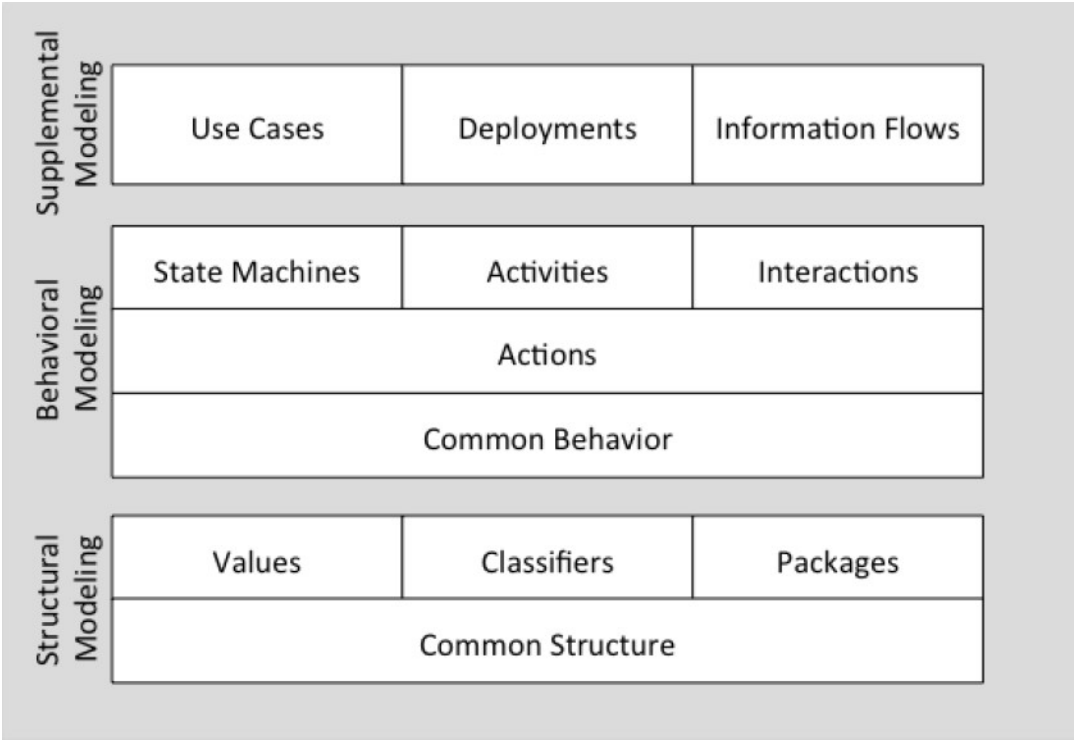
- Your attendance of lectures is part of your grade
- Please make sure you sign in for the class
- If you're absent for a good reason (interview, illness) please e-mail me, and I will excuse your absence
  - Please do this prior to class if at all possible
- Does not apply to distance students

Goals of the (Continuing) Lecture

- Review the material in Chapter 2 of the Textbook
  - Cover key parts of the UML notation
  - Demonstrate some ways in which UML is useful
  - Give you a chance to apply the notation yourself to several examples
- Warning: important information is repeated several times in this lecture
  - this is a hint to the future you when you are studying for the midterm

UML (Revisited)

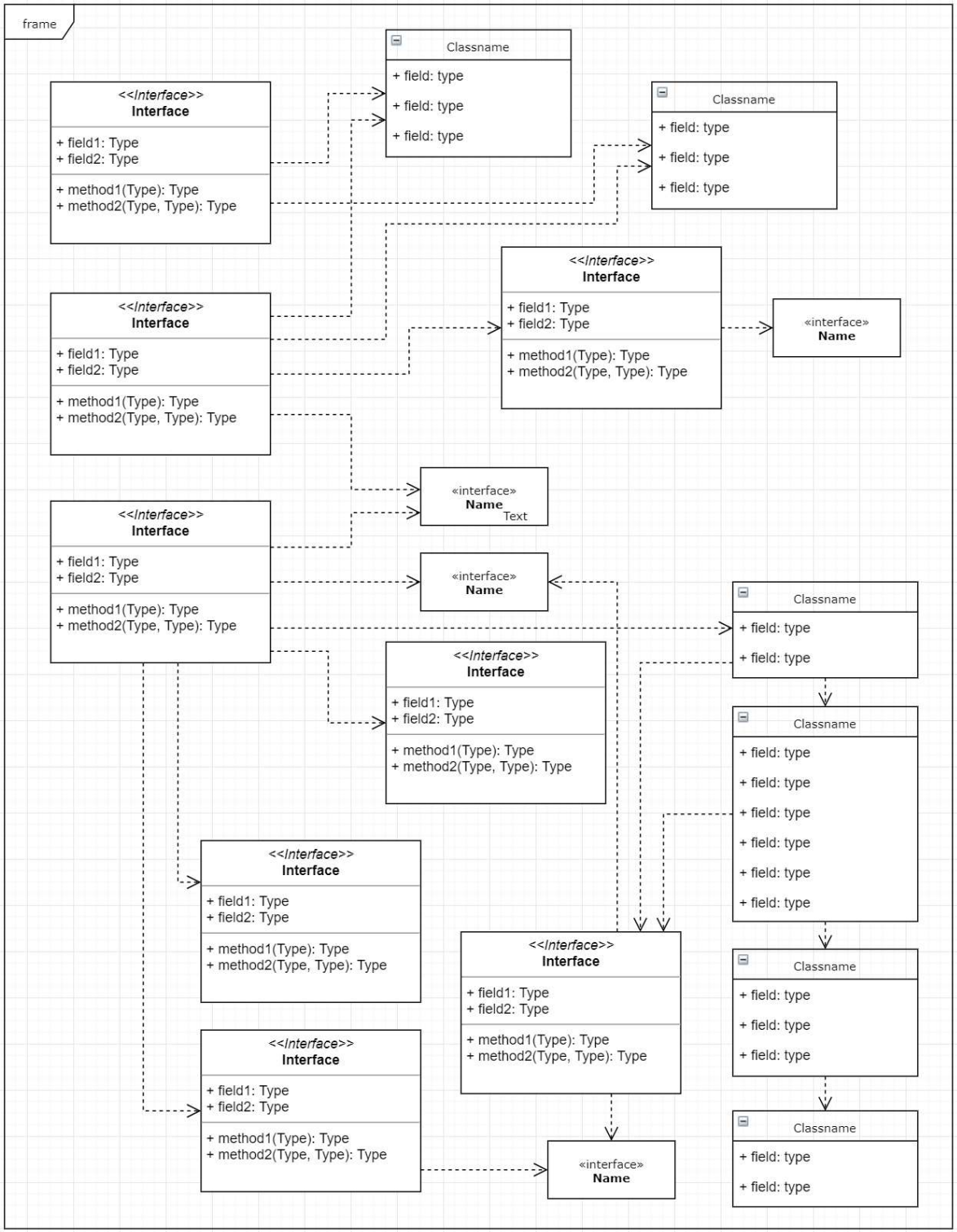
- Diagrams from the current UML release (https://www.omg.org/spec/UML/2.5.1/PDF)
- Structural
  - **Class**
  - Object
  - Package
  - Model
  - Composite Structure
  - Internal Structure
  - Collaboration Use
  - Component
  - Manifestation
  - Network Architecture
  - Profile
- Supplemental (both structural and behavioral elements)
  - **Use Case**
  - Information Flow
  - Deployment



- Behavior
  - **Activity**
  - **Sequence**
  - **State (Machine)**
  - Behavioral State Machine
  - Protocol State Machine
  - Interaction
  - Communication (was Collaboration)
  - Timing
  - Interaction Overview
- Diagrams we'll review are **BOLD**

UML Tools (Revisited)

- References
  - Tutorials
    - https://www.tutorialspoint.com/uml/index.htm
  - Book
    - UML for Mere Mortals, Maksimchuk & Naiburg, 2005, Addison Wesley
- Tools
  - **Draw.io – has UML tools (Free!)**
    - **Pictured here ->**
  - **Lucidchart.com – UML Templates**
    - **(Free access available)**
  - TopCoder UML Tool
    - sequence, class, use case, and activity diagrams
    - Free - Requires registration
    - https://www.topcoder.com/tc?module=Static&d1=dev&d2=umltool&d3=description
  - ArgoUML – open source
    - http://argouml.tigris.org/
  - Visio
  - Whiteboards and a phone/camera
  - Paper & pencil

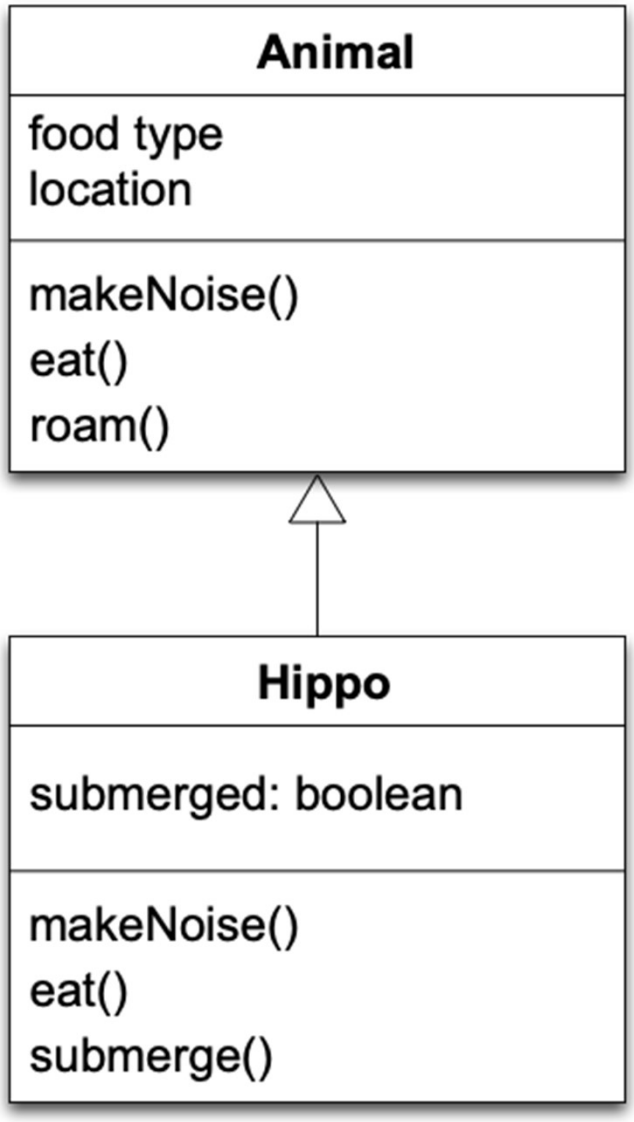


Relationships Between Classes

- Classes can be related in a variety of ways
  - Inheritance
  - Association
    - Multiplicity
  - Whole-Part (Aggregation and Composition)
  - Qualification
  - Interfaces

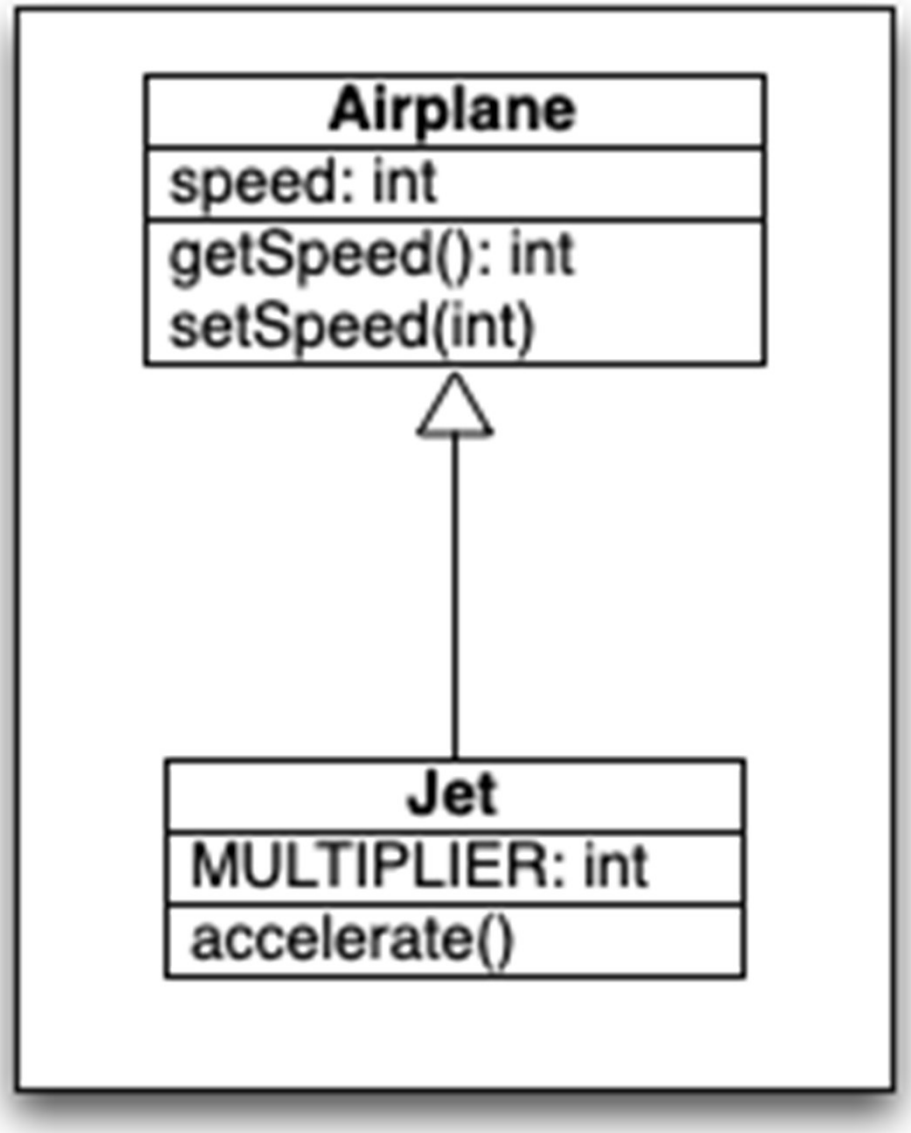
Relationships: Inheritance

- One class can extend another
- UML notation: a white triangle points to the superclass
  - the subclass can add attributes
    - Hippo adds submerged as new state
  - the subclass can add behaviors or override existing ones
    - Hippo is overriding makeNoise() and eat() and adding submerge()



Inheritance

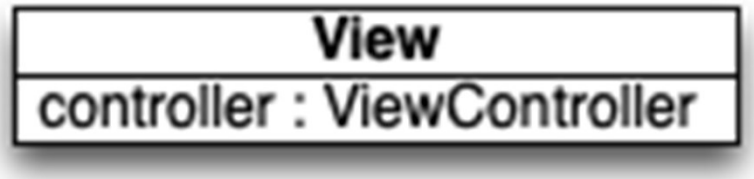
- Inheritance lets you build classes based on other classes and avoid duplicating code
  - Here, Jet builds off the basics that Airplane provides



Relationships: Association

- One class can reference another (a.k.a. association)
  - notation: straight line

- This (particular) notation is a graphical shorthand that each class contains an attribute whose type is the other class



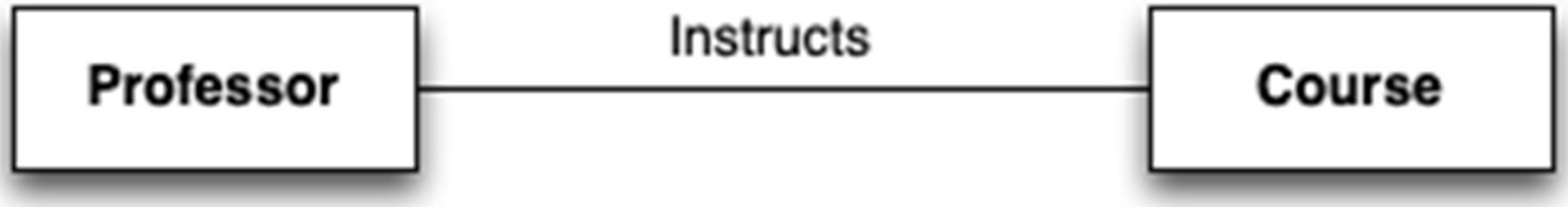
Roles

- Roles can be assigned to the classes that take part in an association

- Here, a simplified model of a lawsuit might have a lawsuit object that has relationships to two people, one person playing the role of the defendant and the other playing the role of the plaintiff
  - Typically, this is implemented via “plaintiff” and “defendant” instance variables inside of the Lawsuit class

Labels

- Associations can also be labelled in order to convey semantic meaning to the readers of the UML diagram

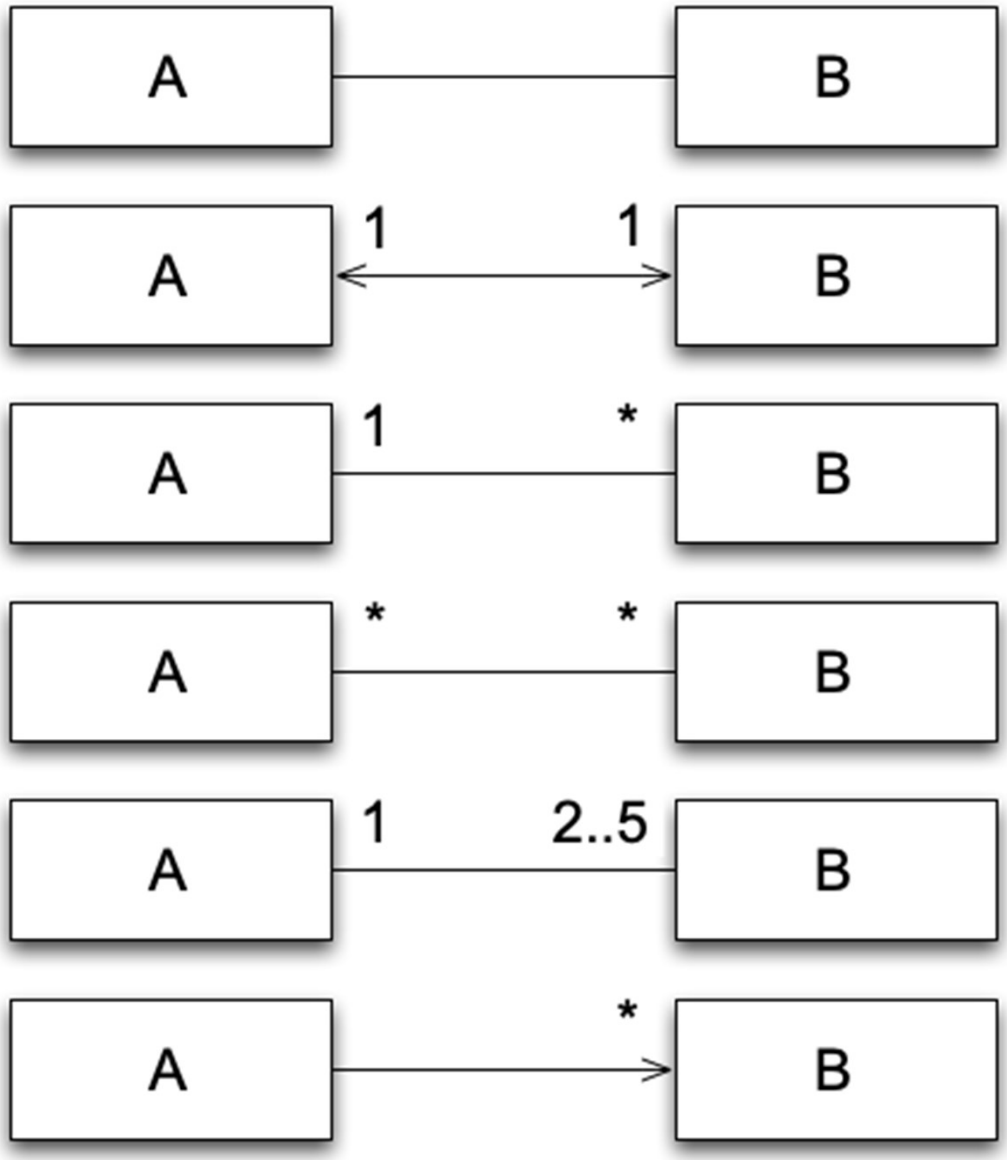


- In addition to roles and labels, associations can also have multiplicity annotations
  - Multiplicity indicates how many instances of a class participate in an association

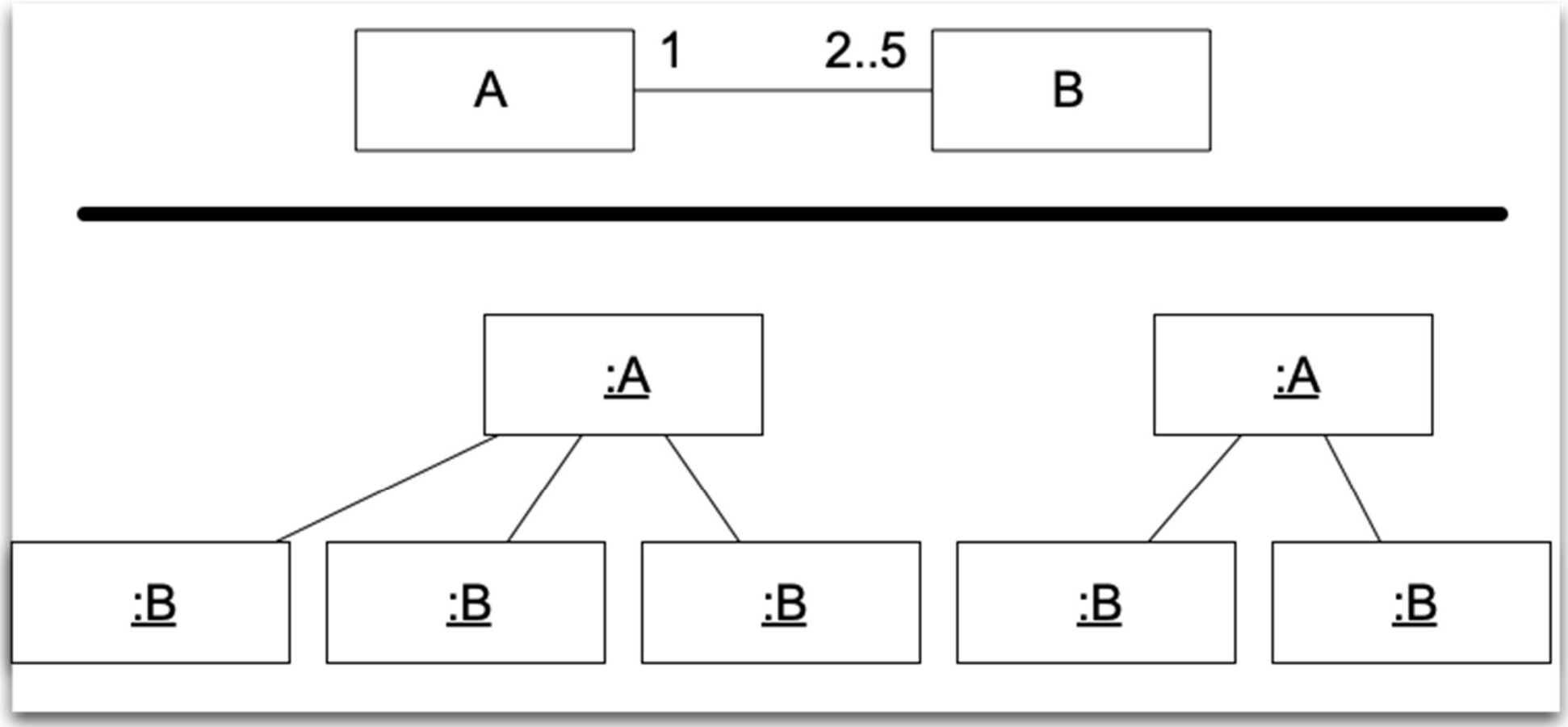
Multiplicity

- Associations can indicate the number of instances involved in the relationship
  - this is known as multiplicity
- An association with no markings is “one to one”
- An association can also indicate directionality
  - if so, it indicates that the “knowledge” of the relationship is not bidirectional
- Examples on next slide

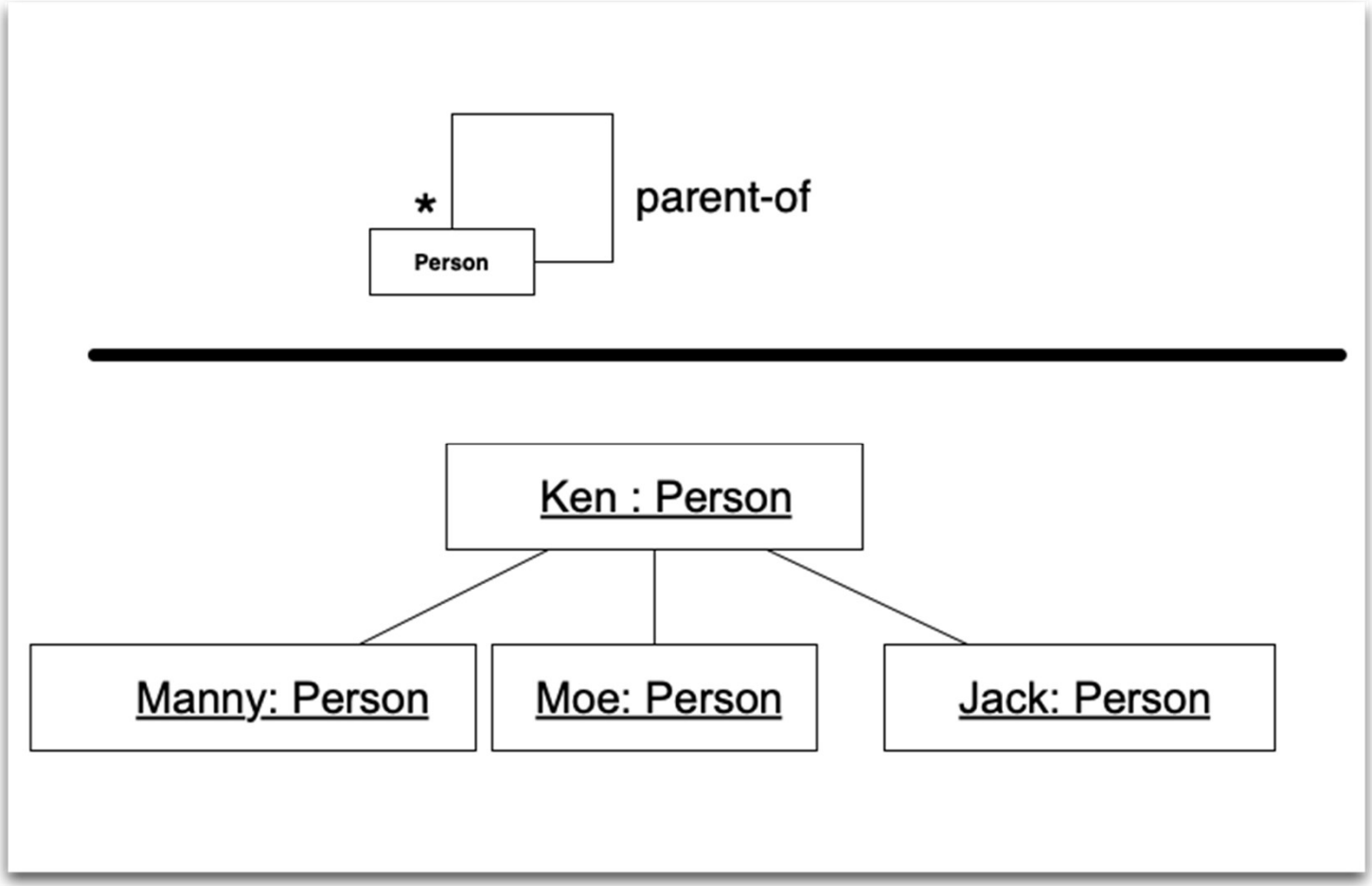
Multiplicity Examples



Multiplicity Example



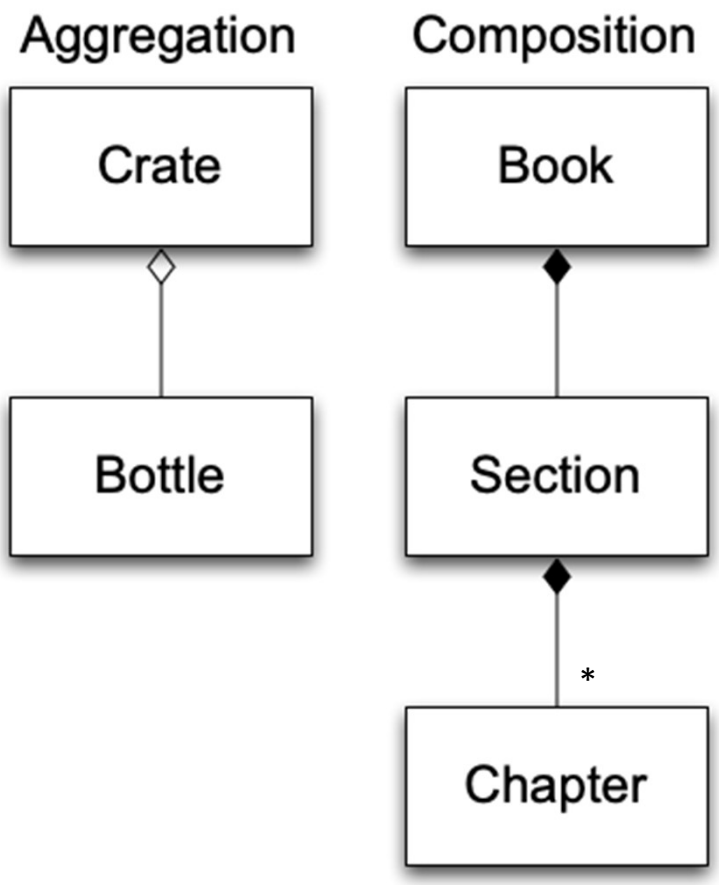
Self Association



Relationships: whole-part

- Associations can also convey semantic information about themselves
  - In particular, **aggregations** indicate that **one object contains a set of other objects**
    - think of it as a whole-part relationship between
      - a class representing a group of components
      - a class representing the components
  - Notation: aggregation is indicated with a **white diamond attached to the class playing the container role**

Example: Aggregation



Composition will be defined on the next slide

Note: multiplicity annotations for aggregation/composition is tricky

Some authors assume “one to many” when the diamond is present; others assume “one to one” and then add multiplicity indicators to the other end

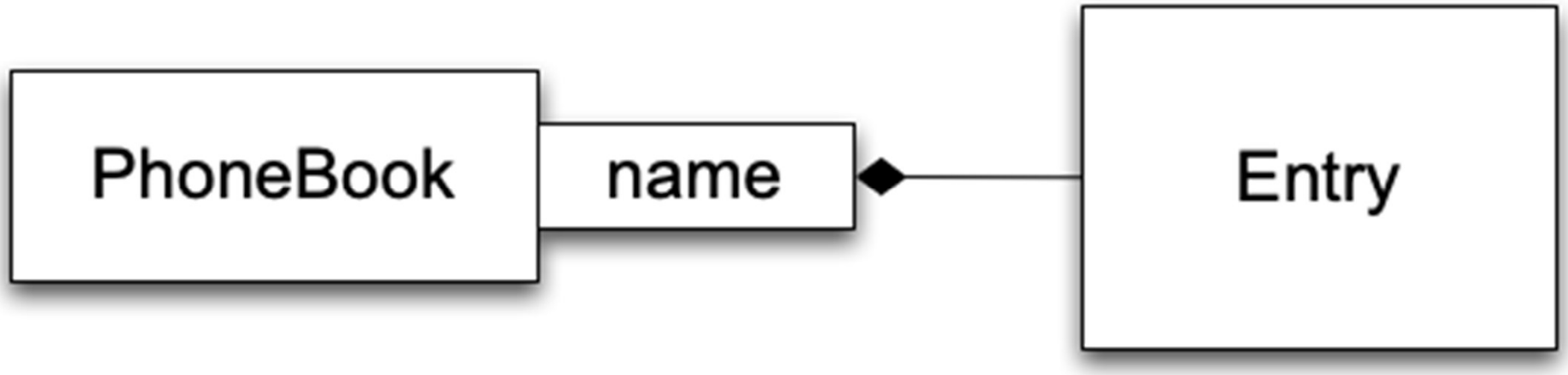
Semantics of Aggregation

- Aggregation relationships are **transitive**
  - if A contains B and B contains C, then A contains C
- Aggregation relationships are **asymmetric**
  - If A contains B, then B does not contain A
- A variant of aggregation is **composition** which adds the property of **existence dependency**
  - if A composes B, then if A is deleted, B is deleted
- Composition relationships are shown with a black diamond attached to the composing class

Relationships: Qualification

- An association can be **qualified** with information that indicates **how objects on the other end** of the association **are found**
  - This allows a designer to indicate that the association **requires a query mechanism of some sort**
    - e.g., an association between a phonebook and its entries might be qualified with a name
  - Notation: a qualification is indicated with a rectangle attached to the end of an association indicating the attributes used in the query

Qualification Example



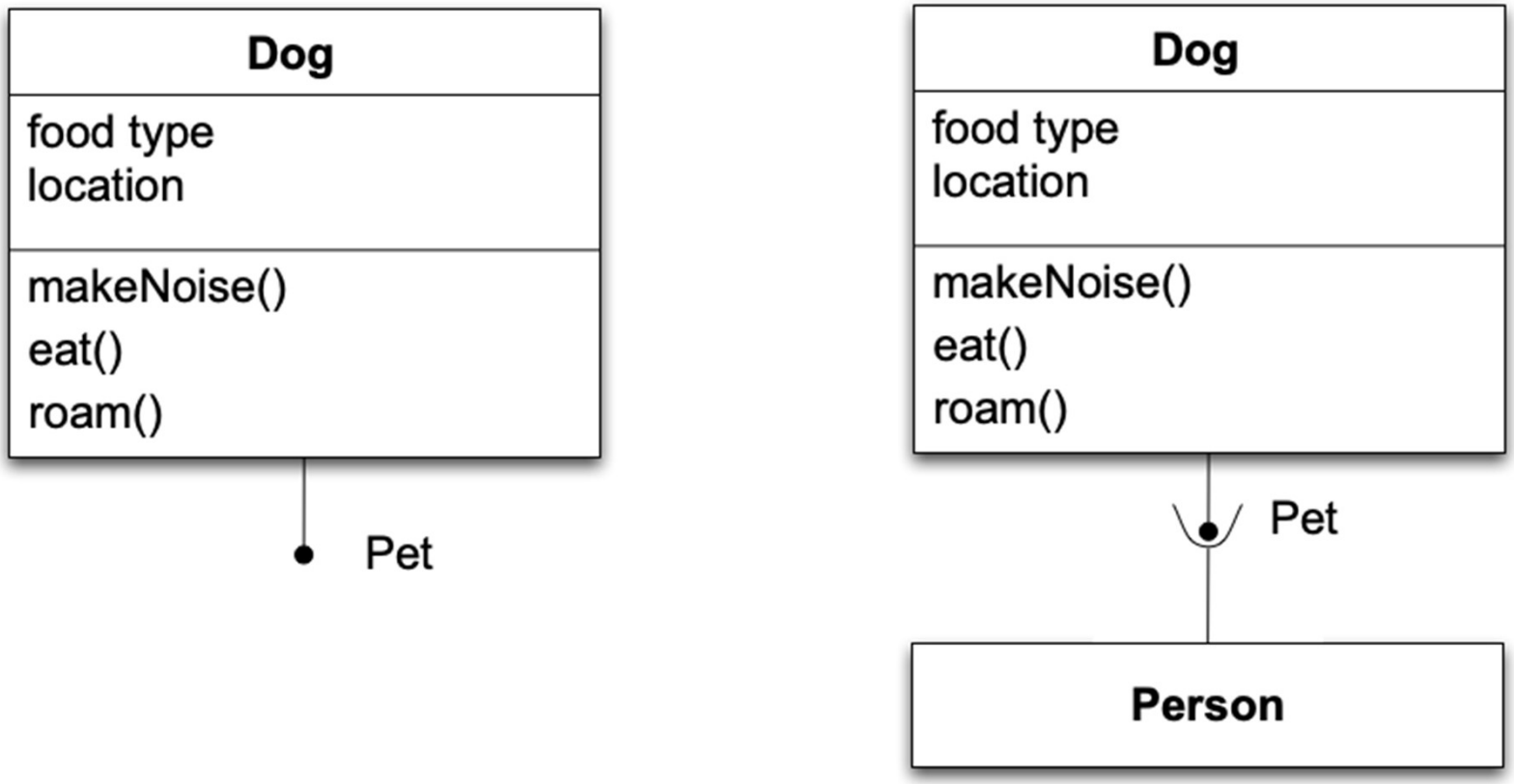
"With a Phonebook, there may be Entries for each instance of name."

Qualification is **not used very often** – it's a UML equivalent of programming constructs like associative arrays, maps, and dictionaries; the same information can be conveyed via a note or a use case that accompanies the class diagram

Relationships: Interfaces

- A class can indicate that it **implements an interface**
  - An interface is a type of class definition in which only method signatures are defined
- A class implementing an interface provides method bodies for each defined method signature in that interface
  - This allows a class to play different roles, with each role providing a different set of services
    - These roles are then independent of the class's inheritance relationships

Example



"Interface Pet is realized or implemented by Dog. Interface Pet is used or required by Person."

Other classes can then access a class via its interface

This is indicated via a "ball and socket" notation

Class Summary

- Classes are blue prints used to create objects
- Classes can participate in multiple types of relationships
  - inheritance, association (with multiplicity), aggregation/composition, qualification, interfaces

UML Sequence Diagrams (I)

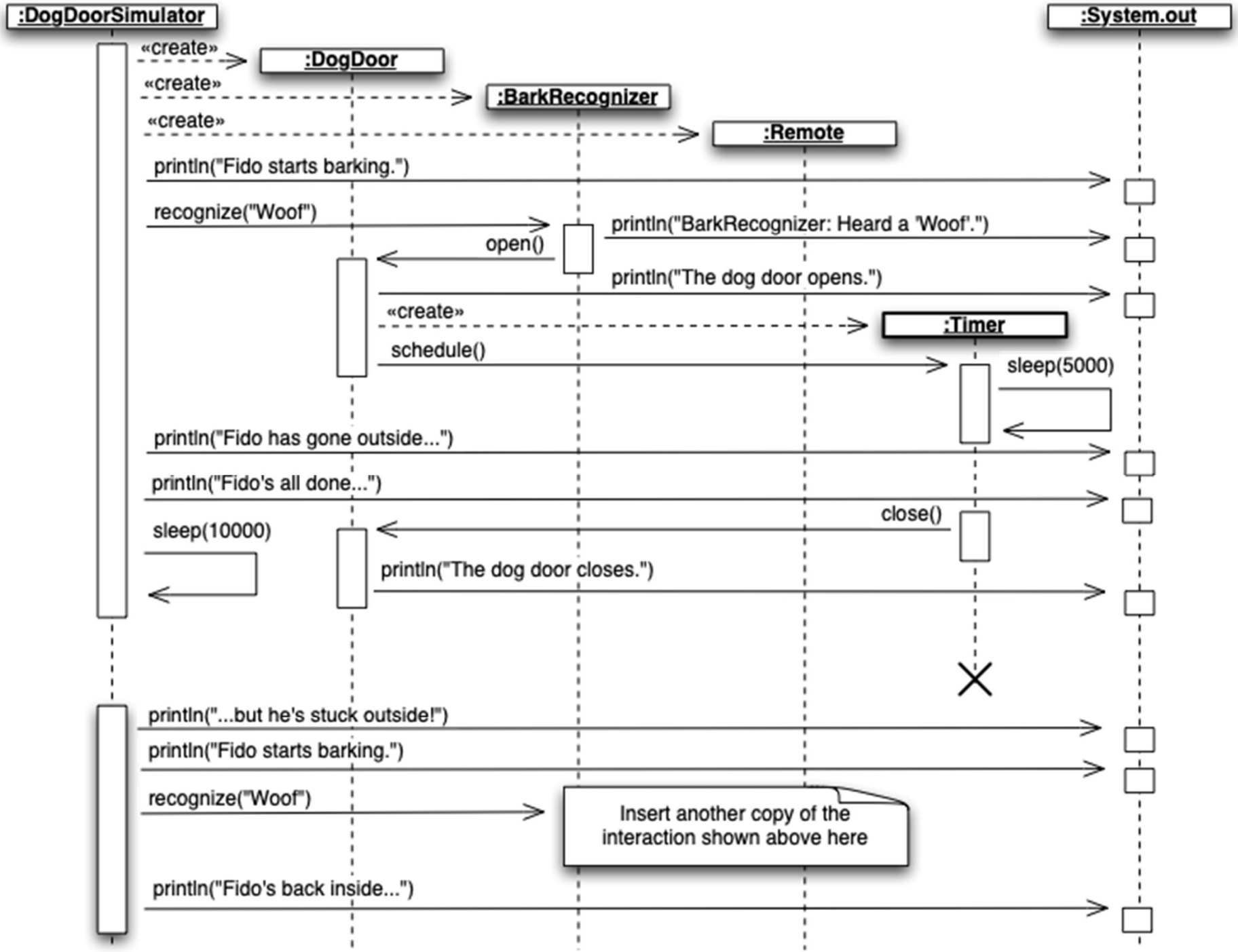
- Objects are shown across the top of the diagram
  - Objects at the top of the diagram existed when the scenario begins
    - All other objects are created during the execution of the scenario
- Each object has a vertical dashed line known as its lifeline
  - When an object is active, the lifeline has a rectangle placed above its lifeline
  - If an object dies during the scenario, its lifeline terminates with an “X”

UML Sequence Diagrams (II)

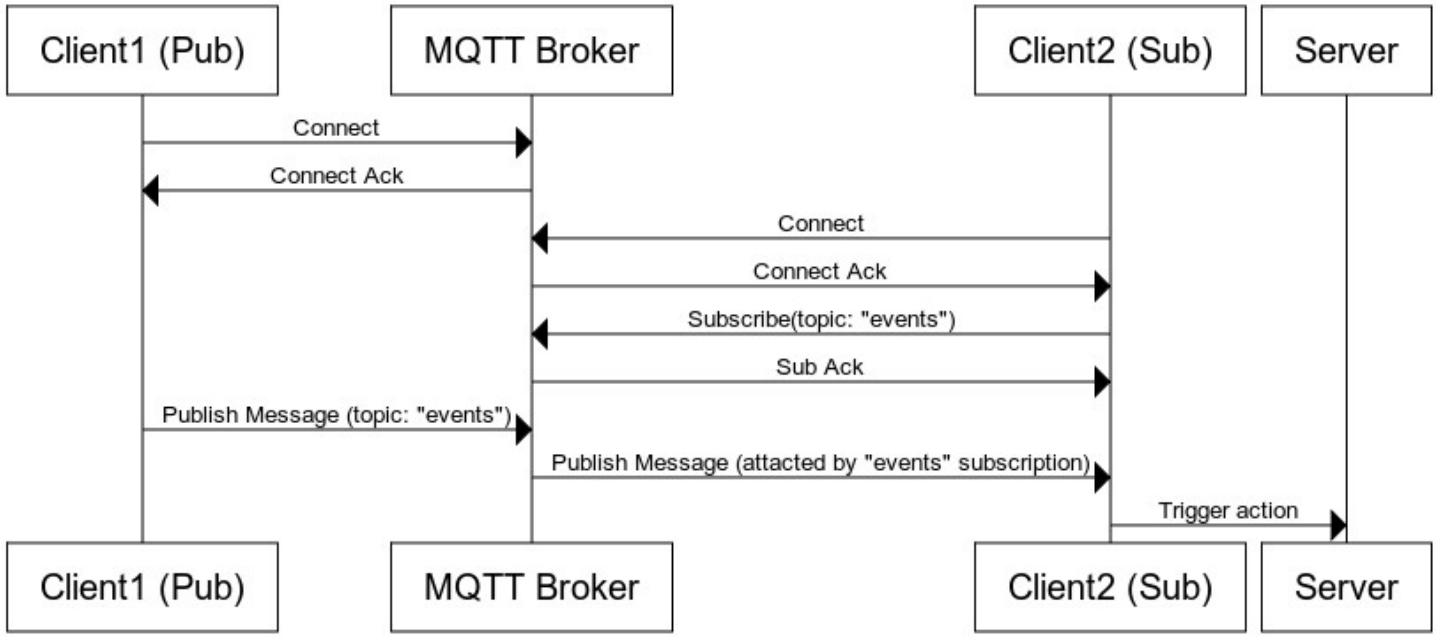
- Messages between objects are shown with lines pointing at the object receiving the message
  - The line is labeled with the method being called and (optionally) its parameters
- All UML diagrams can be annotated with “notes”
- Sequence diagrams can be useful, but they are also labor intensive

UML Sequence Diagrams (III)

- From Maksimchuk & Naiburg
  - Perhaps obvious but one of the major differences between sequence diagrams and class diagrams is their being dynamic vs. static
  - A class diagram does not show operations over time, a sequence diagram does
  - Generally the development of sequence diagrams is iterative, as developing the process and data flows uncover other behavior to model
- Best Practices
  - Avoid them for simple logic
  - Remove unneeded details
  - Messages show primarily move from left to right
  - Take care for changes that render the diagrams obsolete
  - Can be valuable at abstract levels without plumbing details
  - Useful for allocating what behavior classes will cover
  - More at: https://creately.com/blog/diagrams/10-common-mistakes-to-avoid-in-sequence-diagrams/


:DogDoorSimulator
:System.out
«create»
:DogDoor
«create»
:BarkRecognizer
«create»
:Remote
println("Fido starts barking.")
recognize("Woof")
println("BarkRecognizer: Heard a 'Woof'.")
open()
println("The dog door opens.")
«create»
:Timer
schedule()
sleep(5000)
println("Fido has gone outside...")
println("Fido's all done...")
close()
sleep(10000)
println("The dog door closes.")
println("...but he's stuck outside!")
println("Fido starts barking.")
recognize("Woof")
Insert another copy of the interaction shown above here
println("Fido's back inside...")

Another example: MQTT Broker



- My experience: often needed to understand embedded and/or connected system interactions that may have timing dependencies
- From an article on Node.JS publishing events to an MQTT broker
  - https://stackoverflow.com/questions/32538535/node-and-mqtt-do-something-on-message

Graduate Presentation Project

Graduate Students are required to create a presentation that covers a topic related to object-oriented analysis and design. Valid topics include:

- An in-depth look at an object-oriented language and the features it provides developers engaged in the task of implementing object-oriented designs
- An in-depth look at a “new” object-oriented feature or technique that is starting to appear in recent versions of OO programming languages
- An in-depth look at an object-oriented framework, including its history, its primary concepts (classes) and the services it provides to developers. Frameworks may be for any OO programing language
- An analysis and design technique that makes use of object-oriented principles and concepts (classes, objects, delegation, inheritance, etc.) in some way. You are essentially looking for a method that tells designers and developers how to use objects to create OO designs for software systems.

Graduate Presentation Project

Project Content (100 points):

1) A submitted set of slides, a recorded presentation, or a combination. Should be at least 30 minutes worth of material if the full presentation was given. The overall quality and thoroughness of the project will count for 40 points. Content for a two person team should evidence additional effort and detail. Appendices or supporting slides that include information not directly used in the presentation may be included.

   The submission must include citations supporting any claims made – any format is fine but citations should provide a path to find any referenced published literature: paper, book, web site, etc. Citations should be referenced in the above submission on the slide where used and summarized in slides at the end of the presentation. I expect at least ten solid supporting citations for these projects, if not more. At least five references must come from publications, not web sources. Citations will count for 10 points.

2) A runnable example program used to illustrate presentation points should be provided via GitHub, along with a README markdown file with team members, program description, and notes on execution and use. The program provided will count for 30 points, the README for another 10 points. [Note, if code is not applicable to your investigation, contact me regarding alternate graded submissions.]

3) A summary presentation will be provided to the class in a ten minute slot during the last few classes – those attending the class will present this in person. For distance graduate students, there is an option to provide a summarized recorded 10 minute presentation or to submit a similar length slide set. The summary presentation can use slides from the main project, and may include new ones as well. This summary presentation and participation is required, and will be worth 10 points.

Three deliverables will be turned in: The 30 minute full slide set or presentation with citations (a PDF and an optional media file), the GitHub URL for code and documentation, and the 10 minute summary slide set or presentation (a PDF with optional media file).

Graduate Presentation Project

Topic Selection:

- I will maintain a public selected topic list in Canvas Files
- Students should contact me directly by e-mail with a topic selection for themselves or their two person team. Once approved, the topic will be posted. I will allow topics to be selected by up to two teams.
- **Topics should be submitted and approved by Friday's class, 2/8/19.**
- Typical topics (not limited to this):
  - UML: Code generation, Using other diagrams in OO, UML Alternatives (ex. IDEF4)
  - OO Concepts: OO UI Design (MVC), GRASP, Interface Segregation, Concurrency, Dependency Injection/Inversion, Object Persistence
  - OO Techniques: Responsibility-based Design, Data-Driven Design, Domain-Driven Design
  - OO in Languages: C#, C++, Python, Ruby, Scala, Objective-C, Swift, Java, Delphi or Object Pascal, Eiffel, Curl, VB.NET, Smalltalk, Lisp
  - Two language OO element comparison or conversion
  - OO approaches for procedural languages like C
  - OO Design Patterns for specific development frameworks (iOS, Android, etc.)
  - OO Frameworks: Web (Django), Analysis (Pandas), GUI (QT), etc.

Next Steps

- As an option, take a look at Dr. Anderson's lecture on UML
  - You can find it on the class Canvas site under Media Gallery in Lecture 3 and 4
    - Again, very similar material as I'm using versions of his slides…
- First Quiz is up on Canvas, will be due before Wednesday recitation
- Wednesday 1/30 – Recitation session with Manjunath (optional)
- Reading for Friday – Chapter 3 & 4 from textbook
- Friday 2/1 – Lecture, Homework 1 due, and Homework 2 assigned (UML focused, find a tool you like for drawing UML diagrams!)
- Graduate Presentation topic research, submission, and selection: Due Friday 2/8/19 11 AM.