

# UML & OO Fundamentals

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 9

# 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

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD 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

# Goals of the Lecture

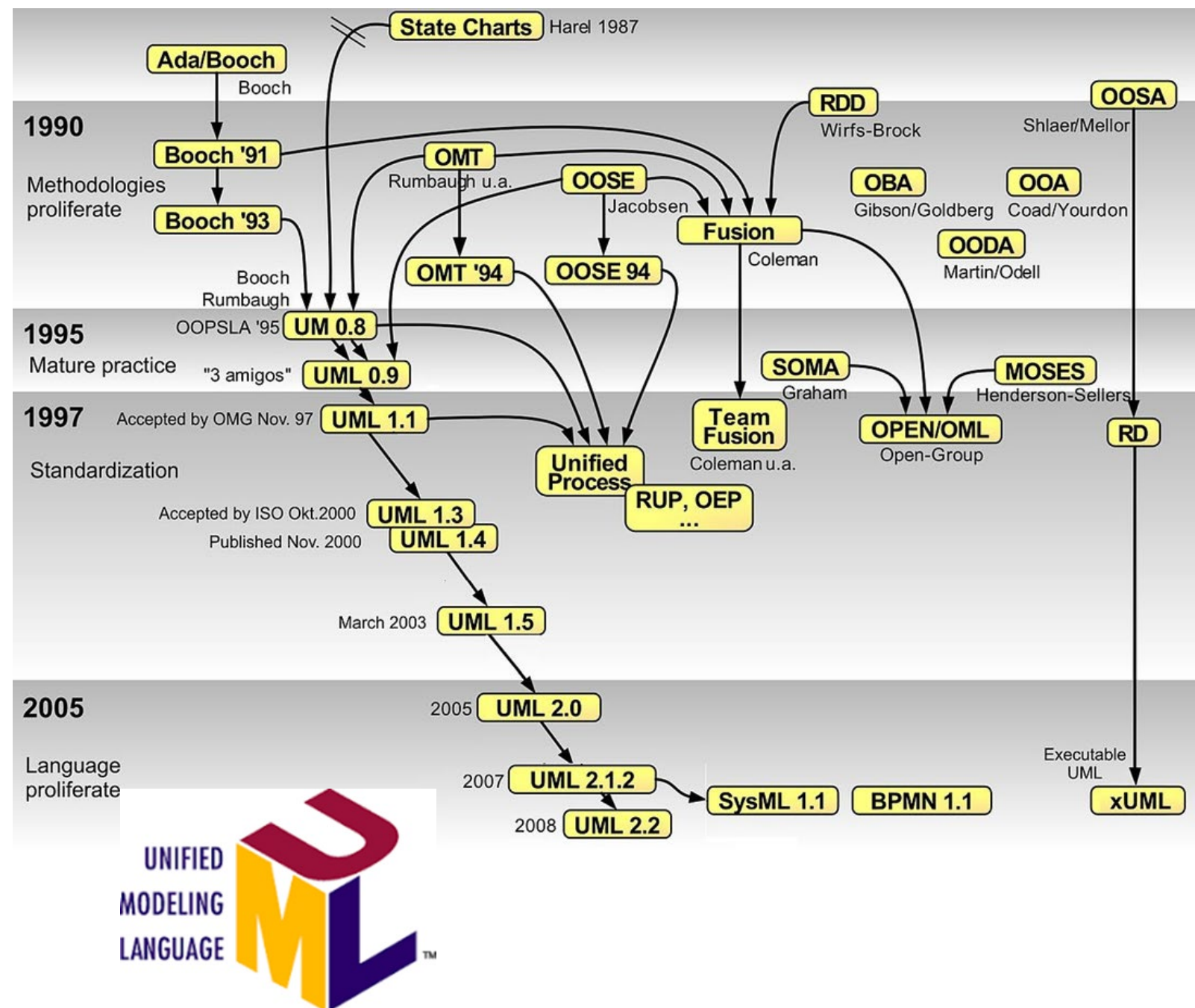
- Review using UML for OO Design
  - Cover key parts of the UML notation
  - Demonstrate some ways in which UML is useful
  - Give you a chance to apply the notation yourself to several examples
- Warning: important information is repeated several times in this lecture
  - this is a hint to the future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

# UML

- UML is short for **Unified Modeling Language**
  - The UML defines a standard set of notations for use in modeling object-oriented systems
- Throughout the semester we will encounter UML in the form of
  - class diagrams
  - sequence/collaboration diagrams
  - state diagrams
  - activity diagrams, use case diagrams, and more

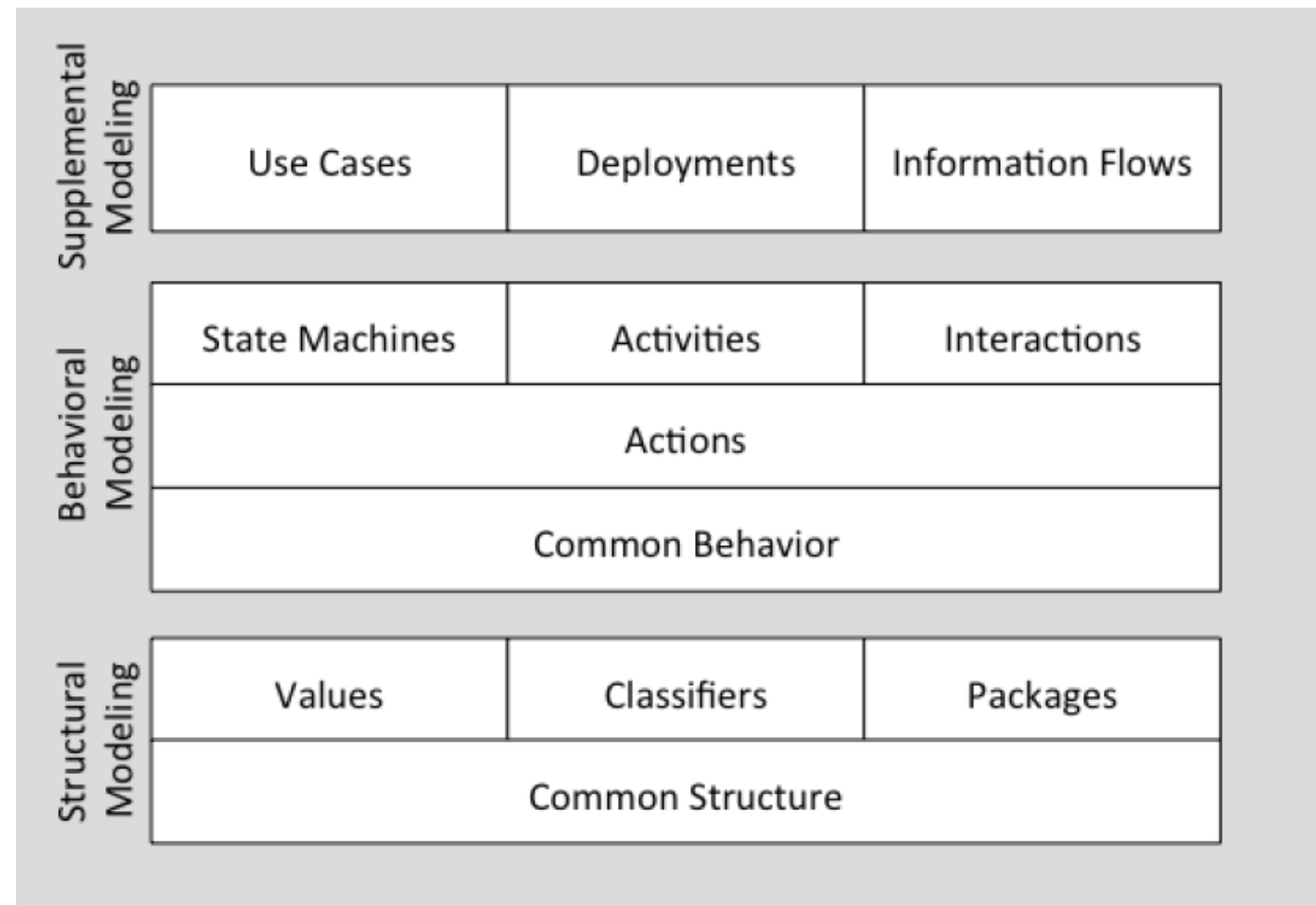
# Brief History of the UML

- In the 80s and early 90s, there were multiple OO A&D approaches (each with their own notation) available
- Three of the most popular approaches came from
  - James Rumbaugh: OMT (Object Modeling Technique)
  - Ivar Jacobson: Wrote “OO Software Engineering” book
  - Grady Booch: Booch method of OO A&D
- In the mid-90’s all three were hired by Rational and together developed the UML; known collectively as the “three amigos”
- Latest UML 2.5.1 Dec 2017  
<https://www.omg.org/spec/UML/>



# UML Diagrams

- 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

- References

- Tutorials

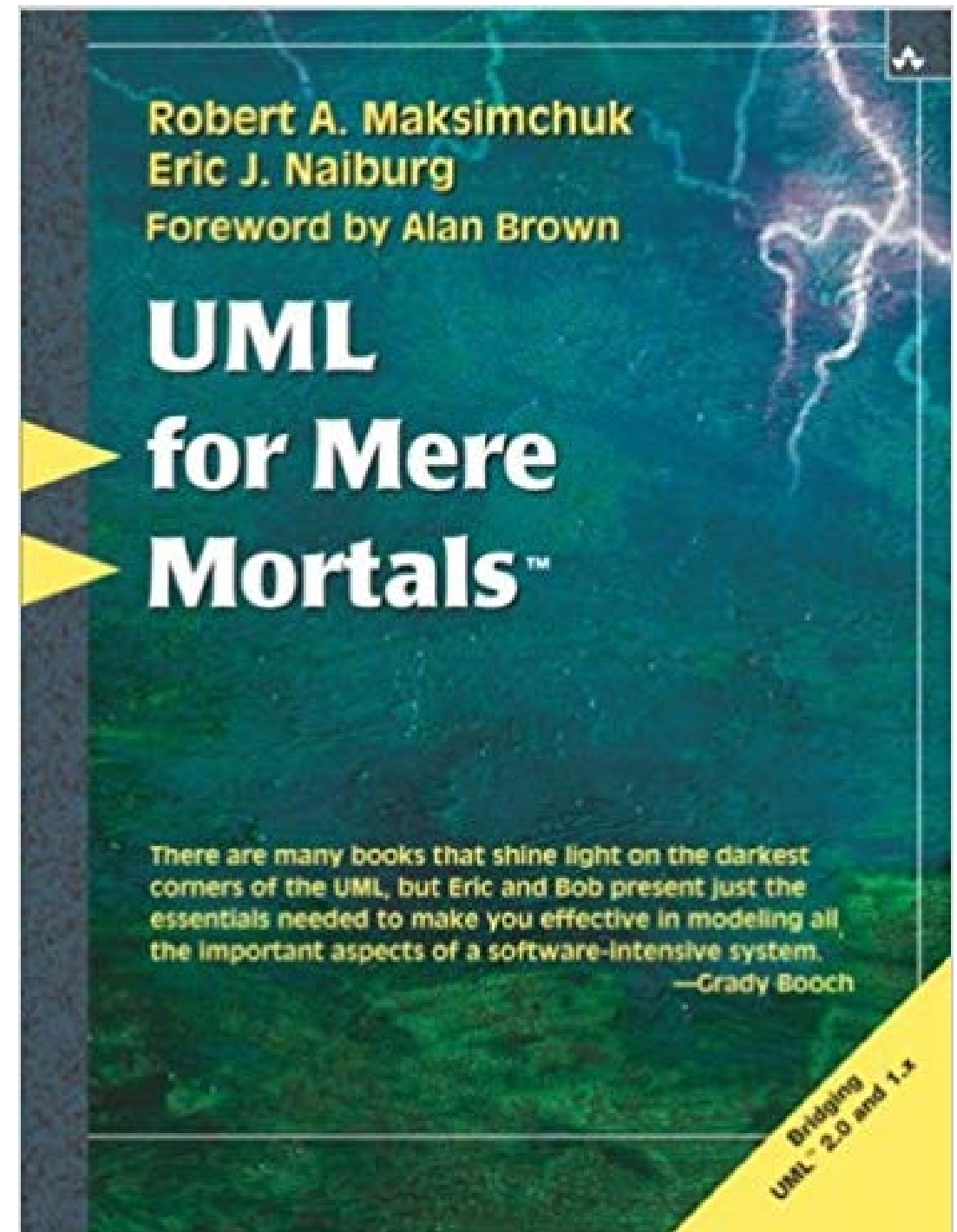
- <https://www.tutorialspoint.com/uml/index.htm>

- Book

- UML for Mere Mortals, Maksimchuk & Naiburg, 2005, Addison Wesley

- Tools

- **Draw.io** – has UML tools (Free!)
  - **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





# Big Picture View of OO Paradigm

- OO techniques view software systems as
  - **networks of communicating objects**
- Each **object** is **an instance of a class**
  - All objects of a class share similar **features**
    - **attributes**
    - **methods**
  - Classes can be **specialized** by **subclasses**
- Objects communicate by **sending messages**

# Objects (I)

- Objects are **instances of classes**
  - They have **state** (attributes) and **exhibit behavior** (methods)
- We would like objects to be
  - **highly cohesive**
    - have a single purpose; make use of all features
  - **loosely coupled**
    - be dependent on only a few other classes

# Objects (II)

- Objects interact by **sending messages**
  - Object A sends a message to Object B to ask it to perform a task
    - When done, B may pass a value back to A
    - Sometimes A == B
      - i.e., **an object can send a message to itself**

# Objects (III)

- Sometimes **messages can be rerouted**
  - invoking a method defined in class A may in fact invoke an **overridden** version of that method in subclass B
  - a method of class B may in turn invoke messages on its superclass that are then handled by overridden methods from **lower in the hierarchy**
- The fact that messages (**dynamic**) can be rerouted distinguishes them from procedure calls (**static**) in non-OO languages

# Objects (IV)

- In response to a message, an object may
  - update its internal state
  - return a value from its internal state
  - perform a calculation based on its state and return the calculated value
  - create a new object (or set of objects)
  - delegate part or all of the task to some other object
- i.e. they can do pretty much anything in response to a message

# Objects (V)

- As a result, objects can be viewed as members of multiple object networks
  - Object networks are also called **collaborations**
- Objects in an collaboration work together to perform a task for their host application

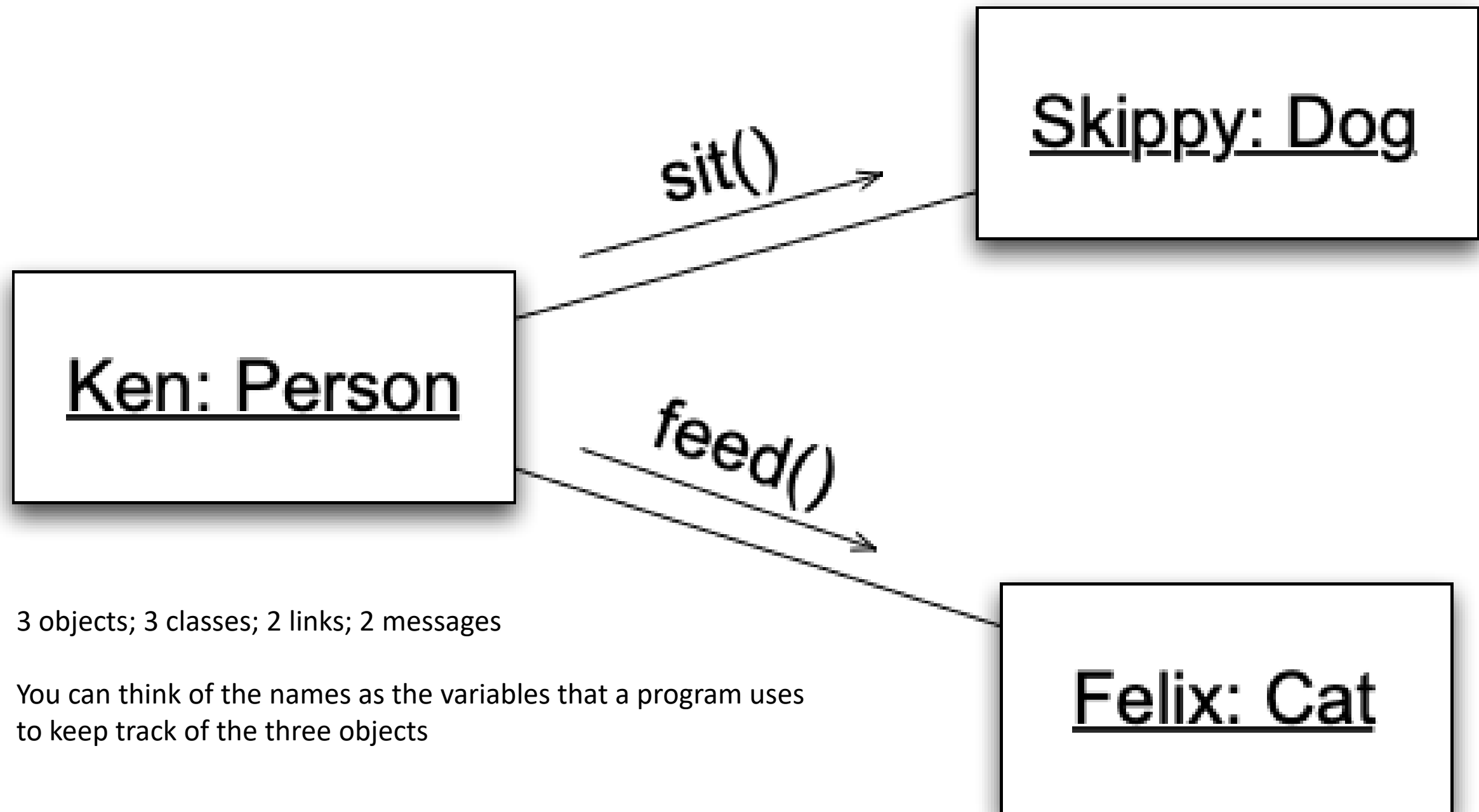
# Objects (VI)

- UML notation for Object Diagrams
  - Objects are drawn as rectangles with their names and types (class names) underlined
    - Ken : Person
  - The name of an object is optional. The type is required
    - : Person
  - Note: The colon is not optional.

# Objects (VII)

- Objects that *work together* **have lines drawn between them**
  - This connection has many names
    - object reference
    - reference
    - **link**
  - Messages are sent across links
    - Links are instances of associations (see [slide 31](#))





3 objects; 3 classes; 2 links; 2 messages

You can think of the names as the variables that a program uses to keep track of the three objects

# Classes (I)

- A **class** is a **blueprint for an object**
  - The blueprint specifies a class's **attributes** and **methods**
    - attributes are **things an object of that class knows**
    - methods are **things an object of that class does**
  - An object is **instantiated** (created) from the description provided by its class
    - Thus, objects are often called **instances**

# Classes (II)

- An object of a class **has its own values for the attributes of its class**
  - For instance, two objects of the Person class can have different values for the name attribute
- Objects **share the implementation of a class's methods**
  - and thus behave similarly
    - i.e. Objects A and B of type Person each share the same implementation of the sleep() method

# Classes (III)

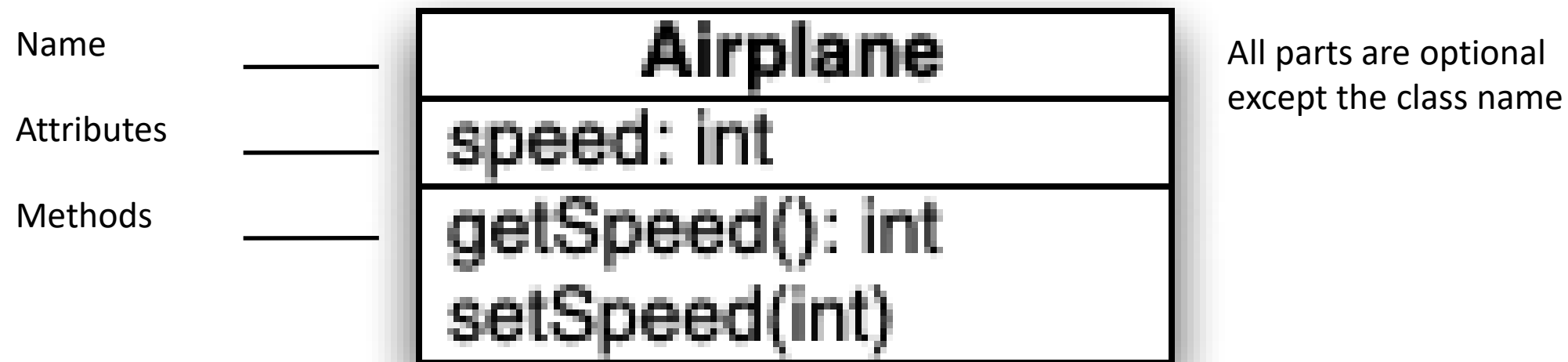
- Classes can define “class-based” (a.k.a. **static**) attributes and methods
  - A **static attribute** is shared among **all** of a class’s objects
    - That is, all objects of that class can read/write the static attribute
  - A static method is a **method defined on the Class itself**; as such, it does not have to be accessed via an object; you can invoke static methods directly on the class itself
    - In Lecture 2’s Java code: `String.format()` was an example of a static method

# Class Diagrams

- Classes in UML appear as rectangles with multiple sections
  - The first section contains its name (defines a type)
  - The second section contains the class's attributes
  - The third section contains the class's methods



# Class Diagrams, 2nd Example



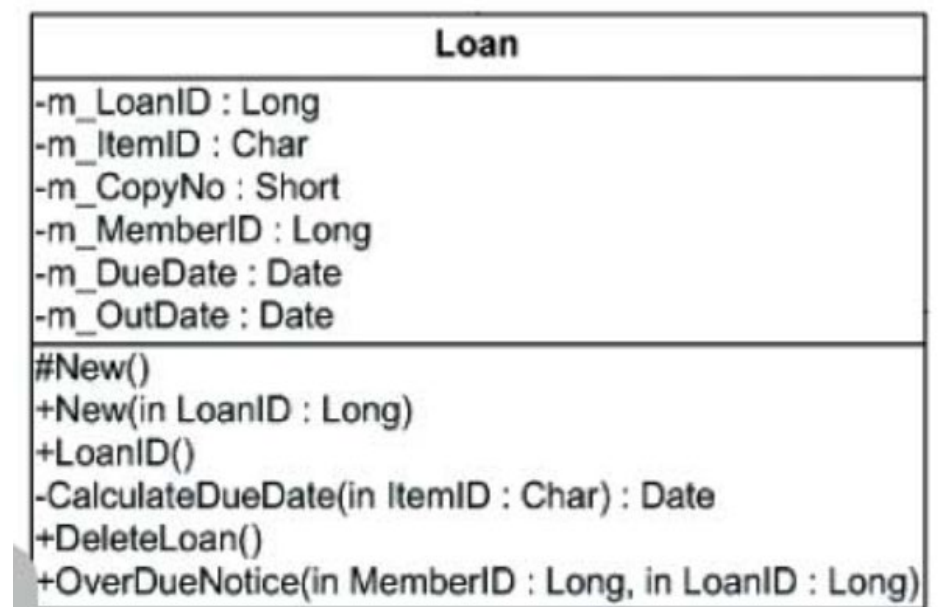
A class is represented as a rectangle

This rectangle says that there is a class called Airplane that could potentially have many instances, each with its own speed variable and methods to access it

# Clarification on Class Diagrams and Data/Method Accessibility

You can use UML to notate which accessibility you want each member to have. The three most common types of accessibility available in most object-oriented languages are as follows:

- **Public**—Notated with a plus sign (+). This means all objects can access this data or method.
- **Protected**—Notated with a pound sign (#). This means only this class and all of its subclasses (i.e. derivations) can access this data or method.
- **Private**—Notated with a minus sign (–). This means that only methods of this class can access this data or method.
- There are others – package, derived, static – expect to see variations in this by language!



<http://www2.sys-con.com/itsg/virtualcd/dotnet/archives/0105/clark/index.html>

# Translation to Code

- Class diagrams can be translated into code straightforwardly
  - Define the class with the specified name
  - Define specified attributes (assume private access)
  - Define specified method skeletons (assume public)
- May have to deal with unspecified information
  - Types are optional in class diagrams
  - Class diagrams typically do not specify constructors
    - just the class's public interface



# Airplane in Java

Using Airplane

```
Airplane a = new Airplane(5);
```

```
a.setSpeed(10);
```

```
System.out.println(  
    "" + a.getSpeed());
```

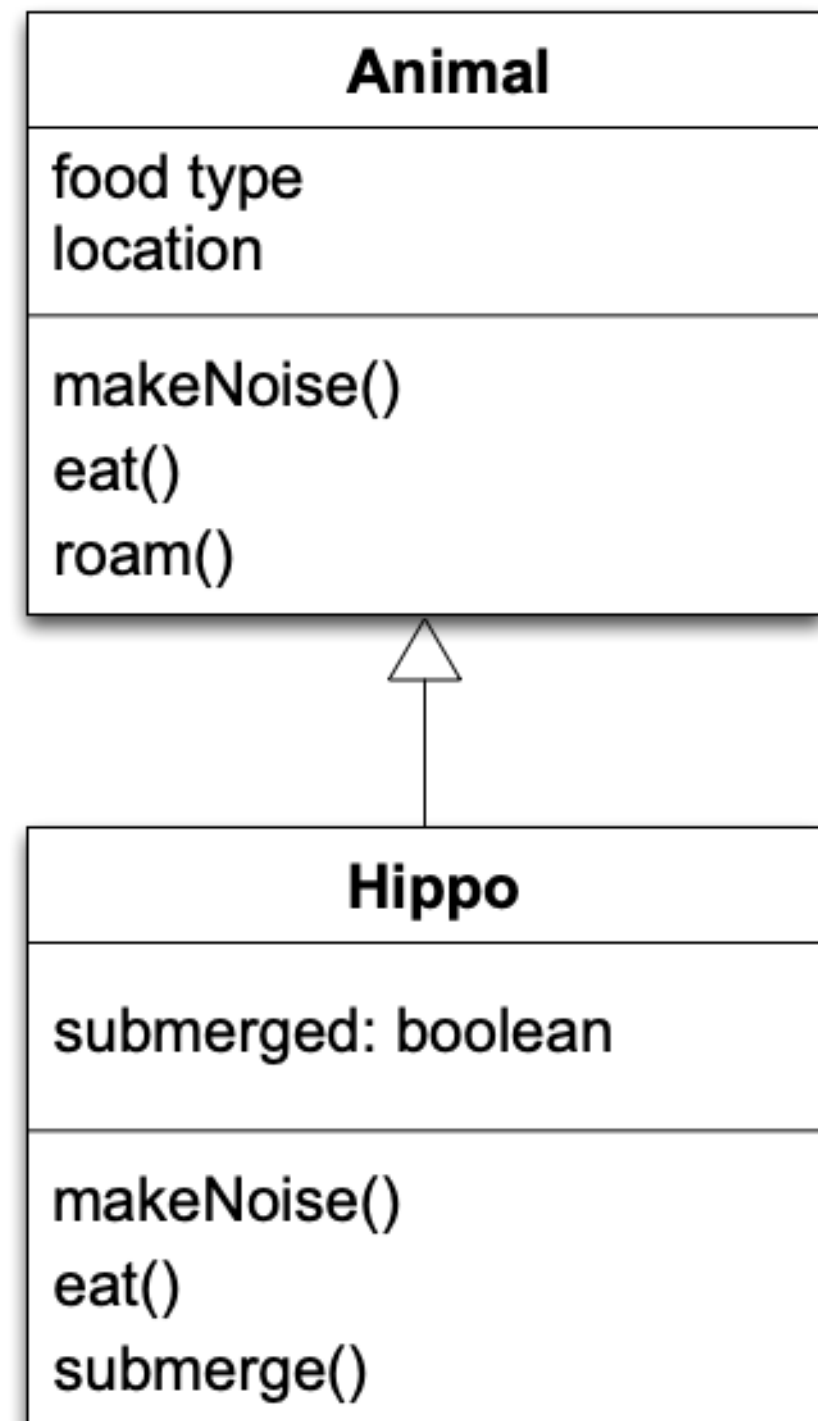
```
1 public class Airplane {  
2  
3     private int speed;  
4  
5     public Airplane(int speed) {  
6         this.speed = speed;  
7     }  
8  
9     public int getSpeed() {  
10        return speed;  
11    }  
12  
13    public void setSpeed(int speed) {  
14        this.speed = speed;  
15    }  
16  
17 }
```

# Relationships Between Classes

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

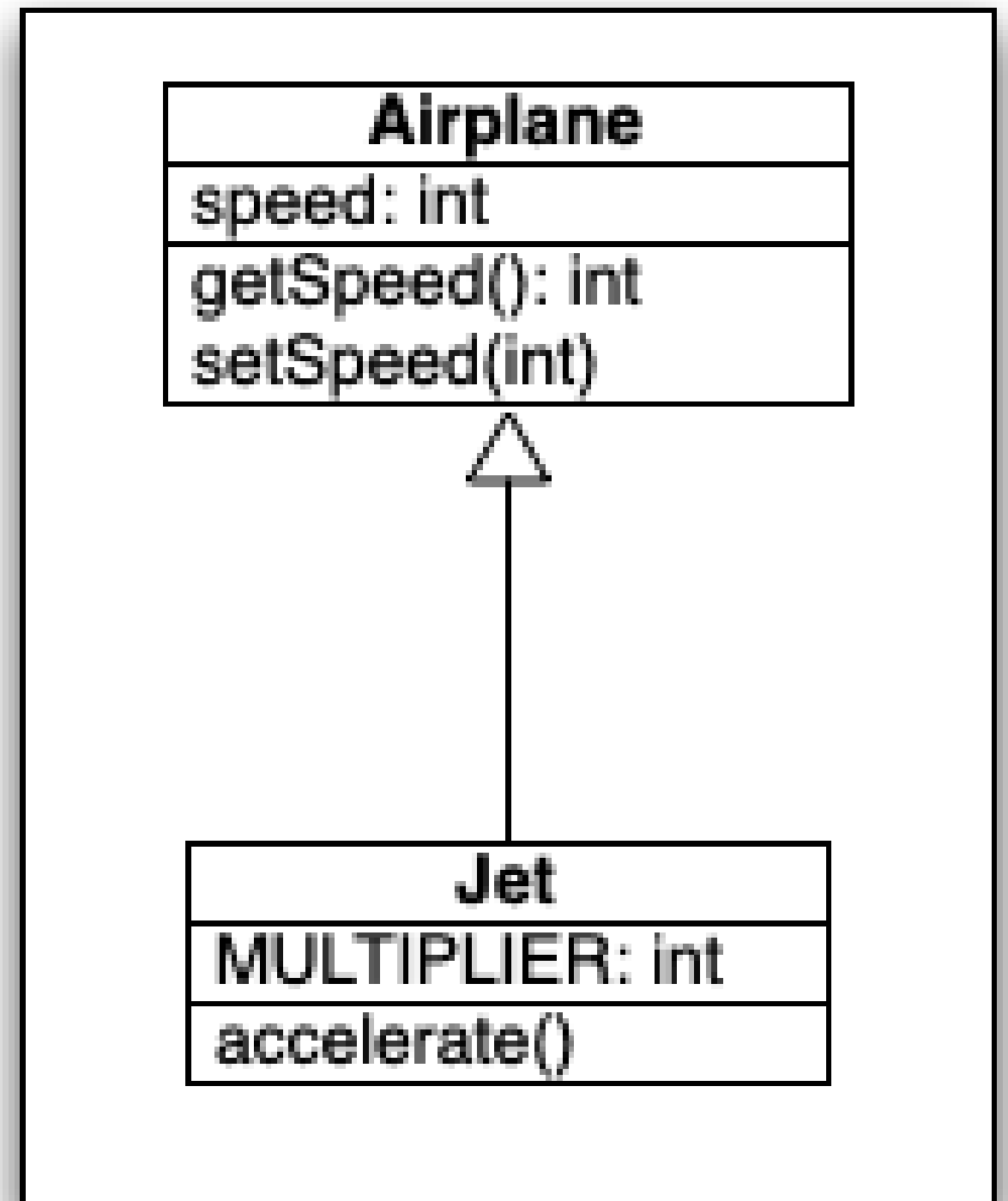
# Relationships: Inheritance

- 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



# Inheriting From Airplane (in Java)

```
1 public class Jet extends Airplane {  
2  
3     private static final int MULTIPLIER = 2;  
4  
5     public Jet(int id, int speed) {  
6         super(id, speed);  
7     }  
8  
9     public void setSpeed(int speed) {  
10        super.setSpeed(speed * MULTIPLIER);  
11    }  
12  
13    public void accelerate() {  
14        super.setSpeed(getSpeed() * 2);  
15    }  
16  
17 }  
18
```

Note:

**extends** keyword indicates inheritance

**super()** and **super** keyword is used to refer to superclass

No need to define `getSpeed()` method; its inherited!

`setSpeed()` method overrides behavior of `setSpeed()` in Airplane

subclass can define new behaviors, such as `accelerate()`

# Polymorphism: “Many Forms”

- “Being able to refer to different derivations of a class in the same way, ...”
  - Implication: both of these are legal statements
    - `Airplane plane = new Airplane();`
    - `Airplane plane = new Jet();`
- “...but getting the behavior appropriate to the derived class being referred to”
  - when I invoke `setSpeed()` on the second plane variable above, I will get Jet’s method, not Airplane’s method

# Encapsulation

- Encapsulation lets you
  - hide data and algorithms in one class from the rest of your application
  - limit the ability for other parts of your code to access that information
  - protect information in your objects from being used incorrectly

# Encapsulation Example

- The “speed” instance variable is private in Airplane. That means that Jet doesn’t have direct access to it.
  - Nor does any client of Airplane or Jet objects
- Imagine if we changed speed’s visibility to public
- The encapsulation of Jet’s `setSpeed()` method would be destroyed

```
1 Airplane
2
3 ...
4 public void setSpeed(int speed) {
5     this.speed = speed;
6 }
7 ...
8
9 Jet
10
11 ...
12 public void setSpeed(int speed) {
13     super.setSpeed(speed * MULTIPLIER);
14 }
15 ...
16
```



# Reminder: Abstraction

- Abstraction is distinct from encapsulation
- It answers the questions
  - What features does a class provide to its users?
  - What services can it perform?
- Abstraction is the **MOST IMPORTANT** concern in A&D!
  - The choices you make in defining the abstractions of your system will live with you for a **LONG** time

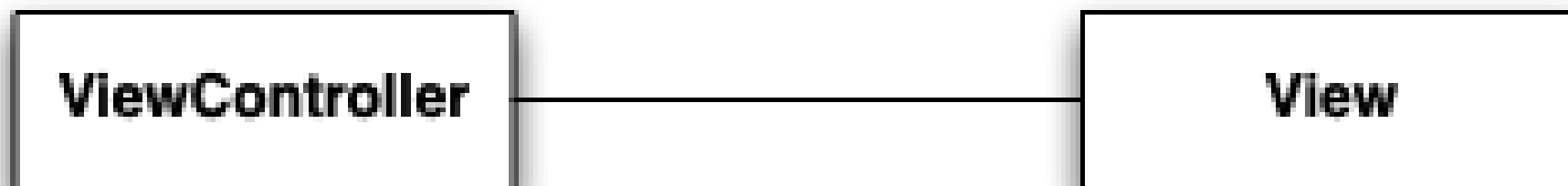
# The Difference Illustrated

- The `getSpeed()` and `setSpeed()` methods represent Airplane's abstraction
  - Of all the possible things that we can model about airplanes, we choose just to model speed
- Making the speed attribute private is an example of encapsulation; if we choose to use a linked list to keep track of the history of the airplane's speed, we are free to do so

```
1 public class Airplane {  
2  
3     private int speed;  
4  
5     public Airplane(int speed) {  
6         this.speed = speed;  
7     }  
8  
9     public int getSpeed() {  
10        return speed;  
11    }  
12  
13    public void setSpeed(int speed) {  
14        this.speed = speed;  
15    }  
16  
17 }
```

# Relationships: Association

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

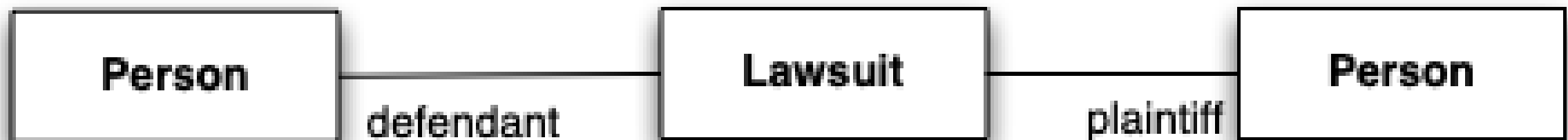


- This (particular) notation is a graphical shorthand that each class contains an attribute whose type is the other class



# 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

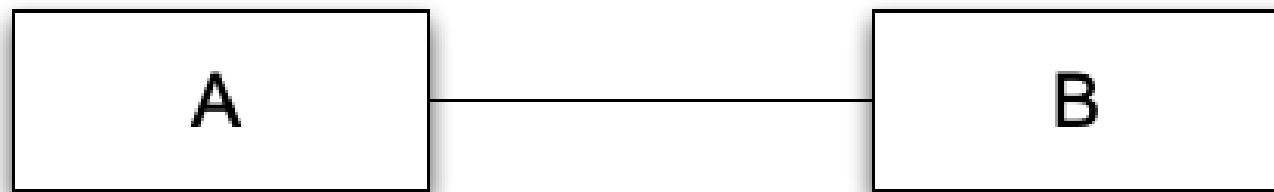


- In addition to roles and labels, associations can also have multiplicity annotations
  - Multiplicity indicates how many instances of a class participate in an association

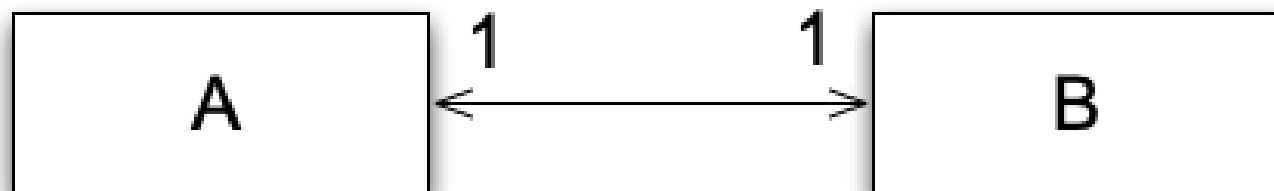
# Multiplicity

- 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

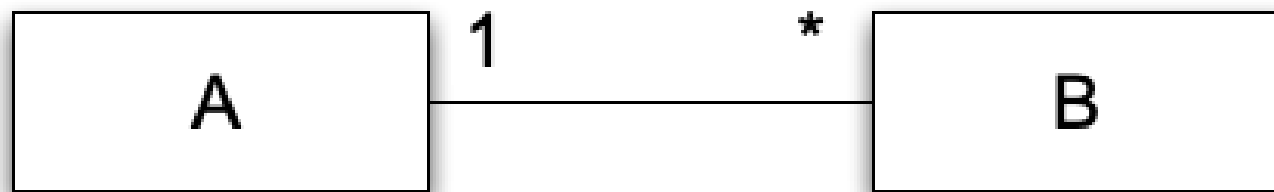
# Multiplicity Examples



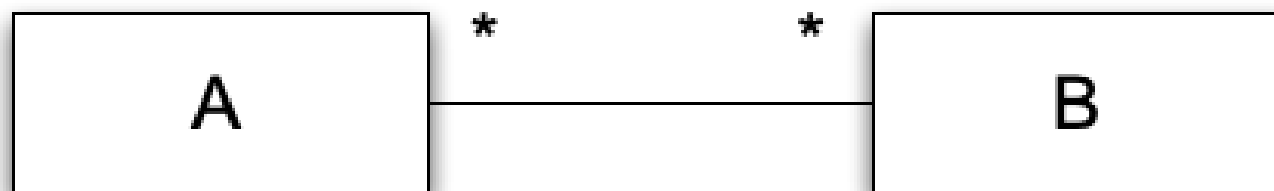
One B with each A; one A with each B



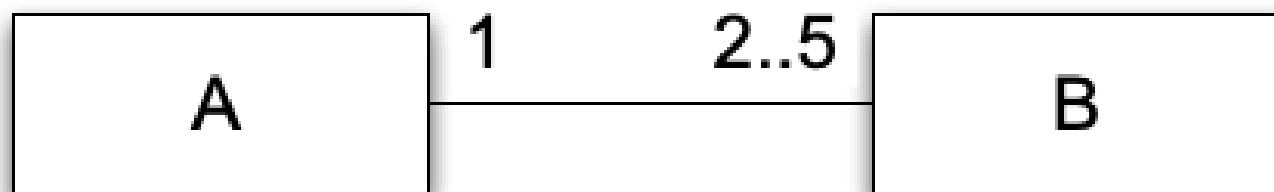
Same as above



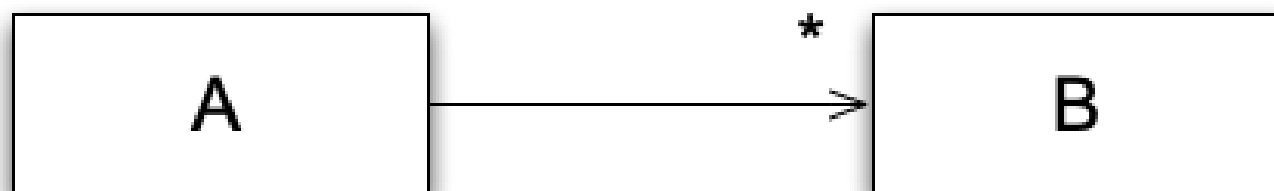
Zero or more Bs with each A; one A with each B



Zero or more Bs with each A; ditto As with each B

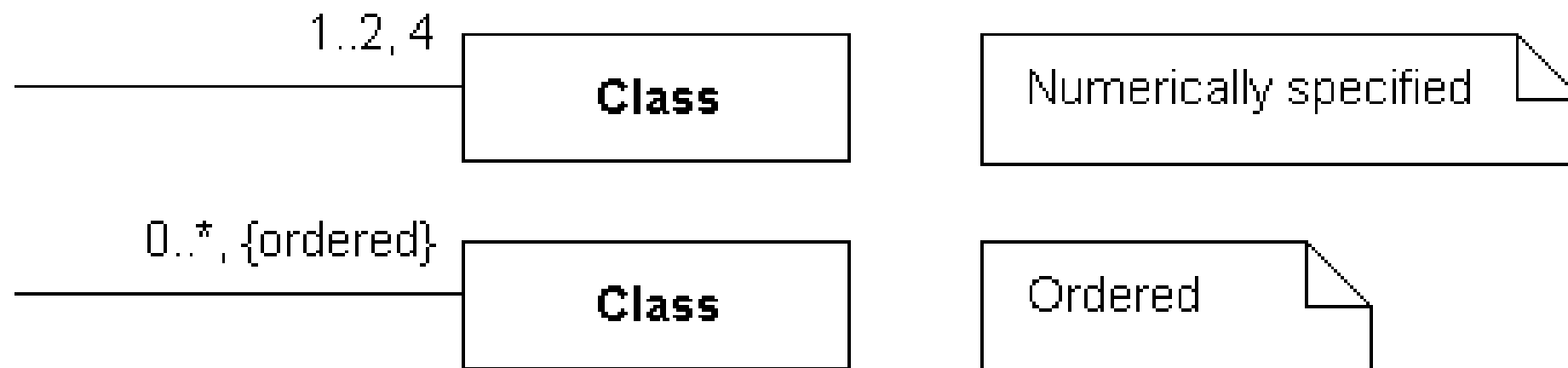


Two to Five Bs with each A; one A with each B



Zero or more Bs with each A; B knows nothing about A

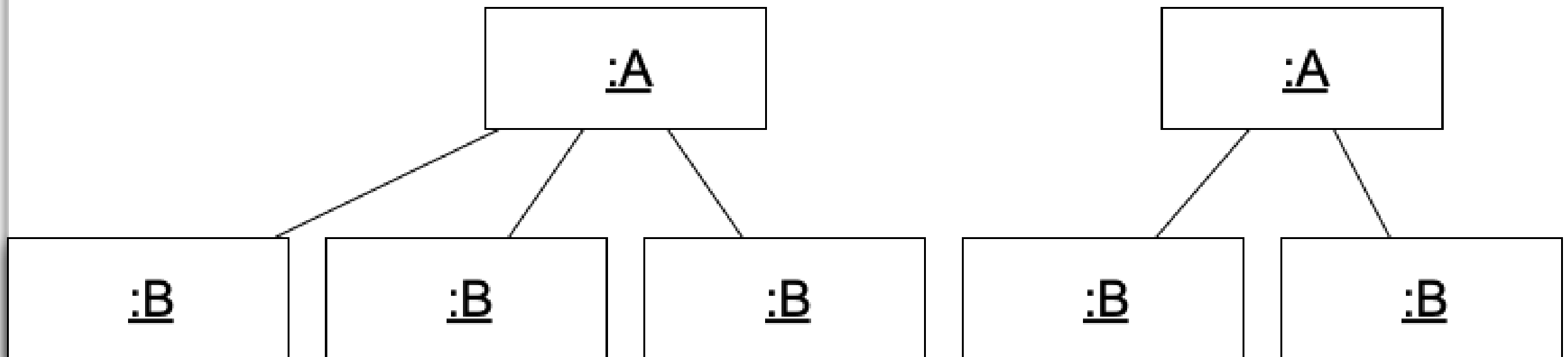
# Clarification on Multiplicity Notation



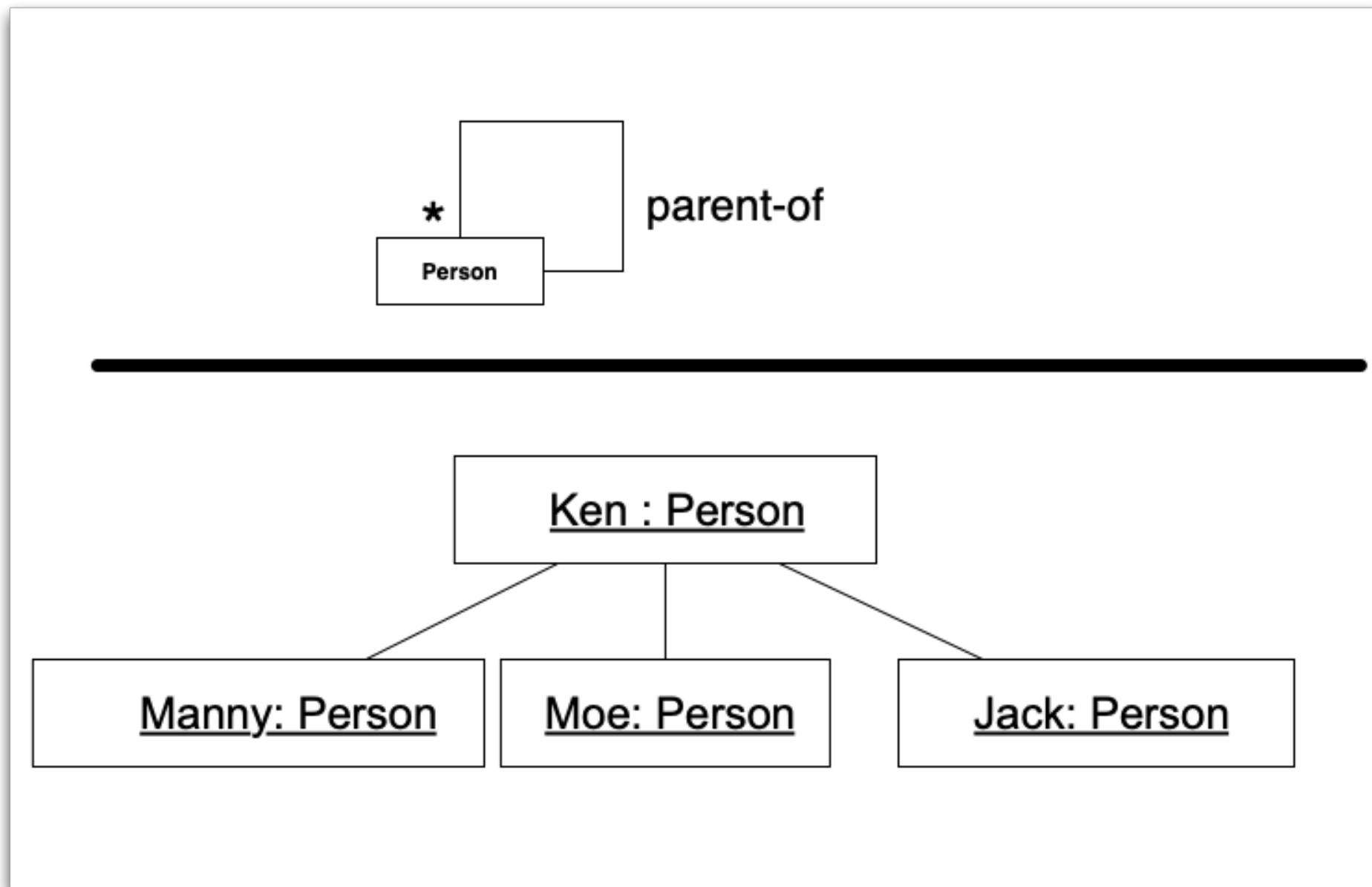
- The first example calls for 1, 2, or 4 instances of Class, not 0, 3, or more than 4
- The second one shows an added keyword indicating the instances have an order that is maintained



# Multiplicity Example



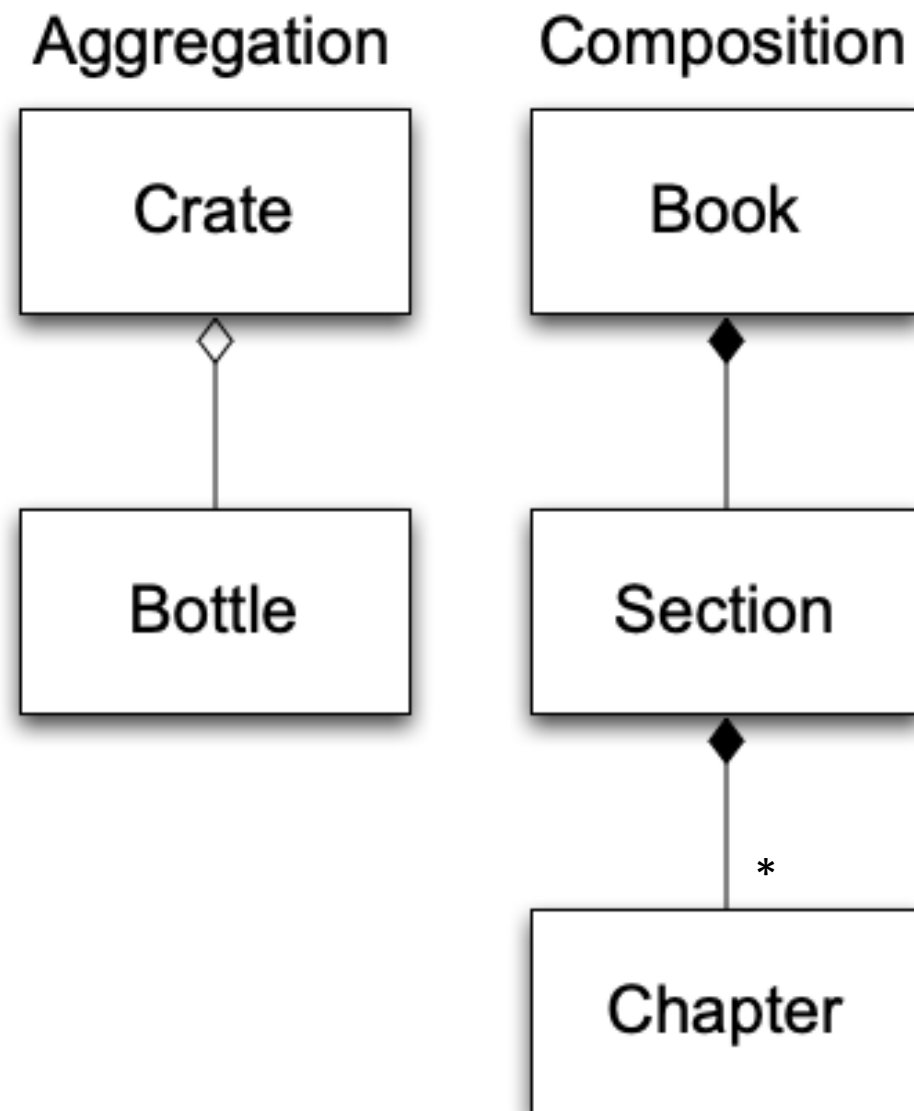
# Self Association



# 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

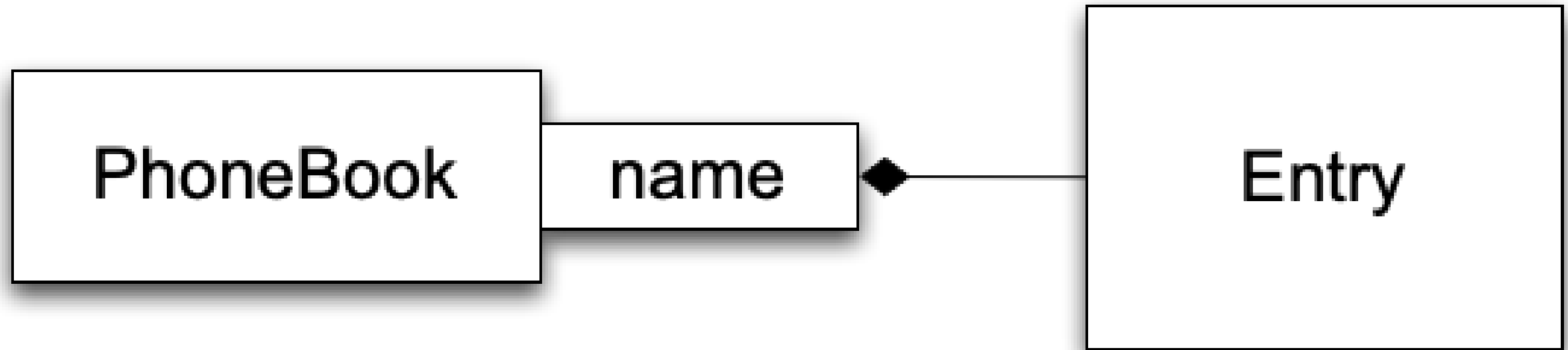
# Semantics of Aggregation

- 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.”

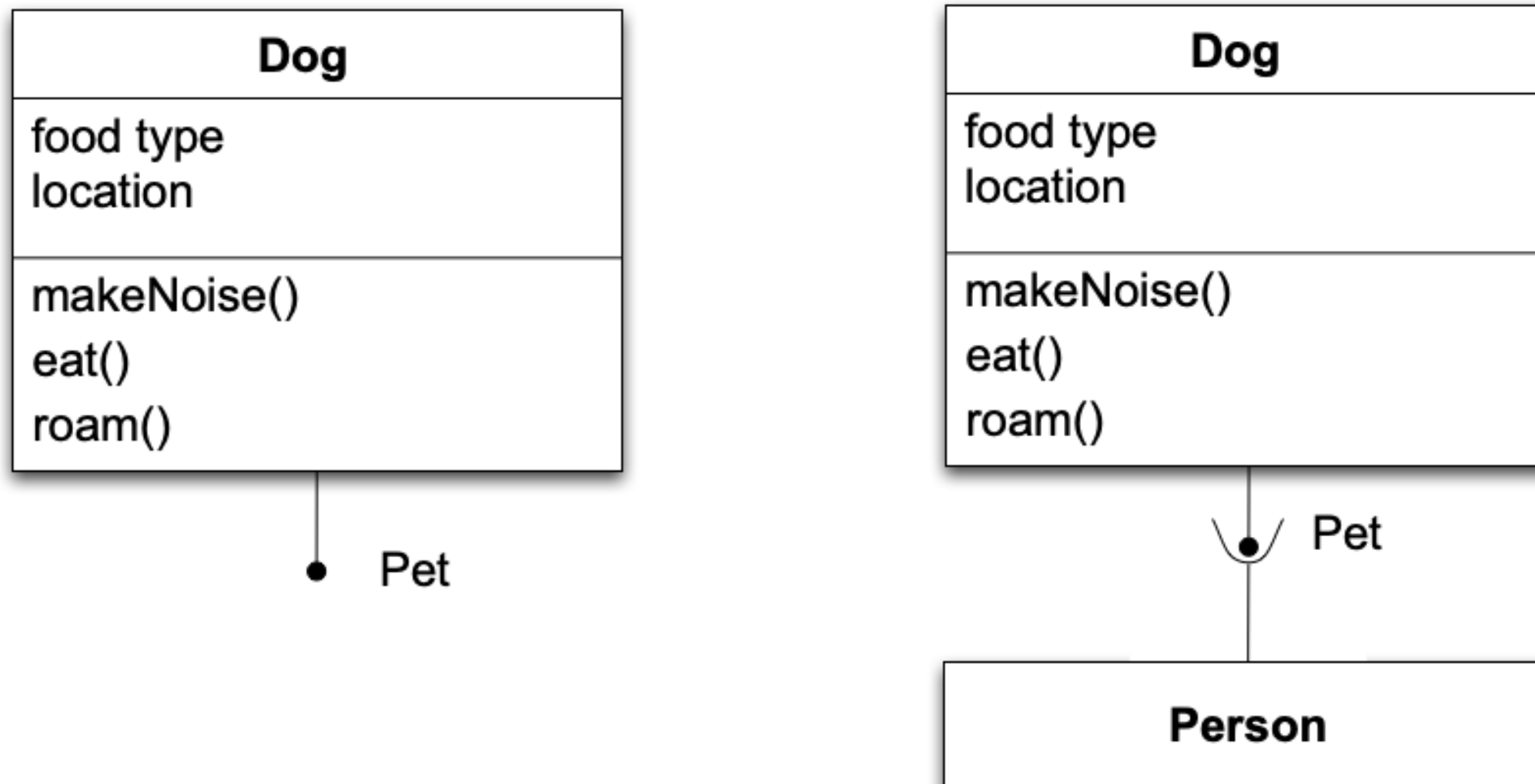
Qualification is **not used very often** – it’s a UML equivalent of programming constructs like associative arrays, maps, and dictionaries; the same information can be conveyed via a note or a use case that accompanies the class diagram

# 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

# OOAD Daily Double!



Sorry, Jeopardy...

# Let's draw a class diagram

- You'll need a partner (teams of 2 or 3 are ok) and a blank sheet of paper and a drawing thing
- The lecture notes on UML are on Canvas if you need to look at them while you're drawing
- Ready?

Remember, the lecture is out on Canvas under Class Files if you need it as a reference...

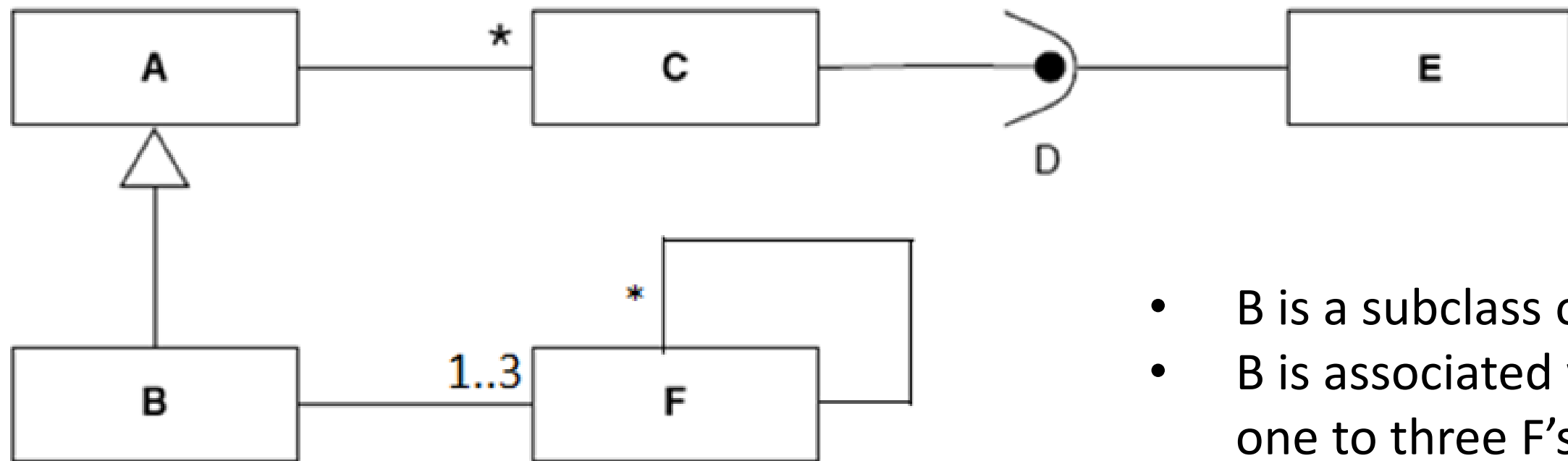
## Draw this class diagram...

- Draw a class diagram that models the following constraints.
  - B is a subclass of A.
  - B is associated with one to three F's.
  - Each F can also self-associate with zero or more other F's.
  - A is associated with zero or more C's.
  - C implements interface D.
  - E accesses C via the D interface.
- I'll give you a couple of minutes... Draw!

# Trade diagrams with another team

- Give your completed diagram to another team...
- Prepare to check the diagram...

The diagram should look like this...



- Check the diagram against the list to the right →
- It can be oriented differently.
- The D ball and socket could be a box labeled “Interface D”
- Perfect score is 6
- How did we do?

- B is a subclass of A.
- B is associated with one to three F’s.
- Each F can also self-associate with zero or more other F’s.
- A is associated with zero or more C’s.
- C implements interface D.
- E accesses C via the D interface.

# Menti Participation

- Please enter your CU identikey (mine is brmo3998, for example) on the menti.com screen for participation credit.

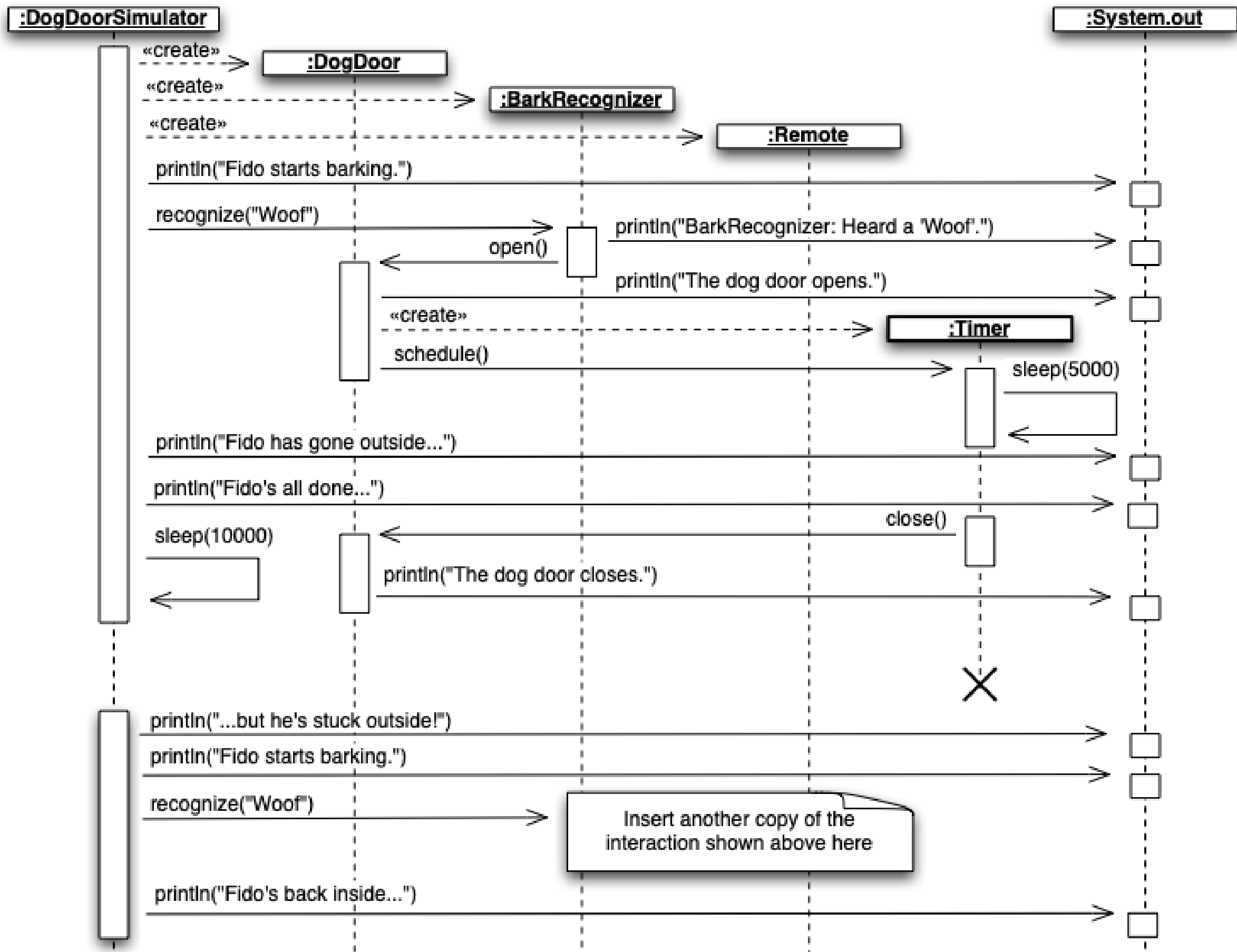


# Sequence Diagrams (I)

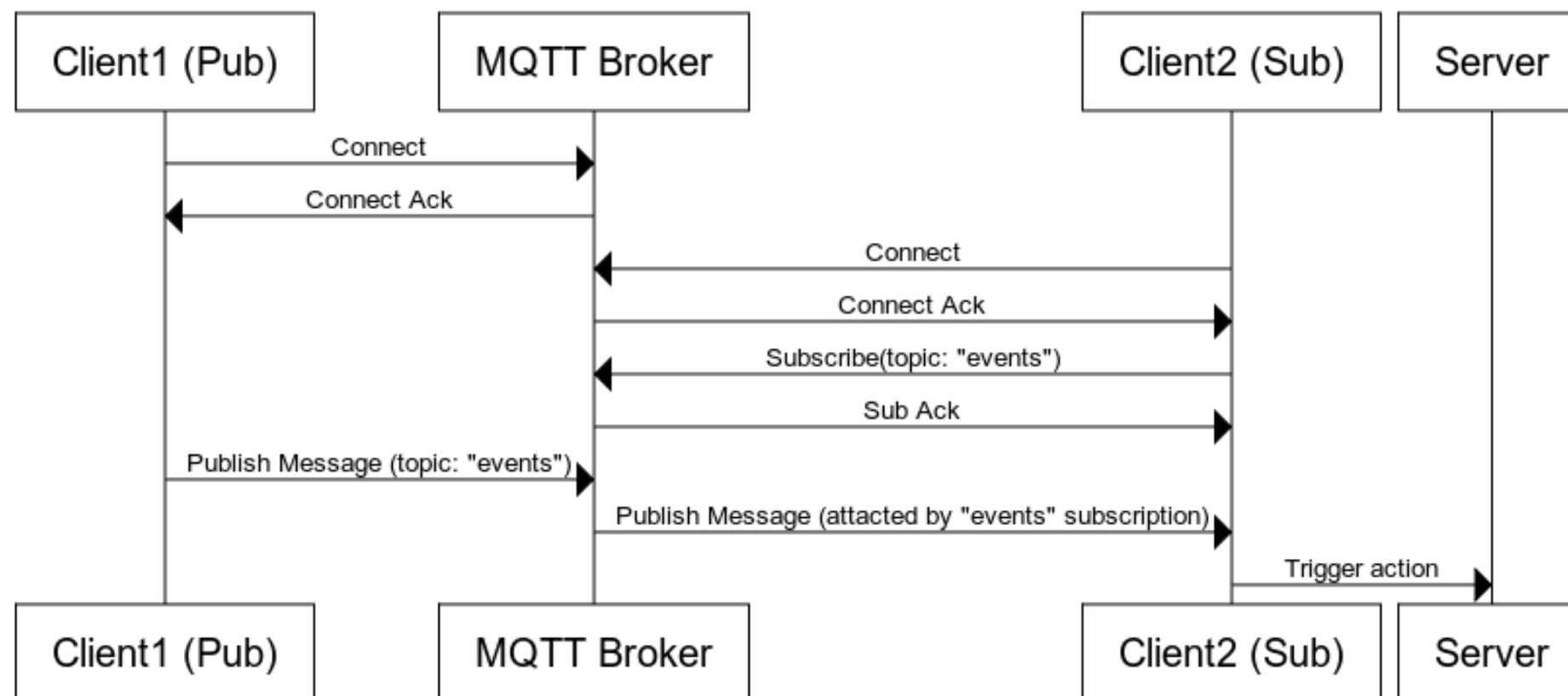
- Objects are shown across the top of the diagram
  - Objects at the top of the diagram existed when the scenario begins
    - All other objects are created during the execution of the scenario
- Each object has a vertical dashed line known as its lifeline
  - When an object is active, the lifeline has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

# 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
- Often needed to understand embedded system interactions that are timing dependant



# 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>

# User Perspective and Use Cases

- In analysis, as much as possible, we want to write our artifacts from the standpoint of a user
  - We will make frequent and consistent use of domain-related vocabulary and concepts
  - We will talk about the software system as a “black box”
  - We can describe its inputs and its expected outputs but we try to avoid discussing how the system will process or produce this information
- In UX oriented workflows, understanding the user and their tasks are key
  - A typical UX development process might include
    - Analysis and Planning
    - User and Task Research (<- Use cases)
    - Interface and Interaction Design
    - Verification and Validation
- Use cases help maintain the user perspective
  - We identify the different types of users for our system – “who”
  - We then develop tasks for each of the different types of user – “what”
- Use cases are used to capture functional requirements
  - They can be annotated to also describe non-functional requirements but typically the focus is on functional requirements only

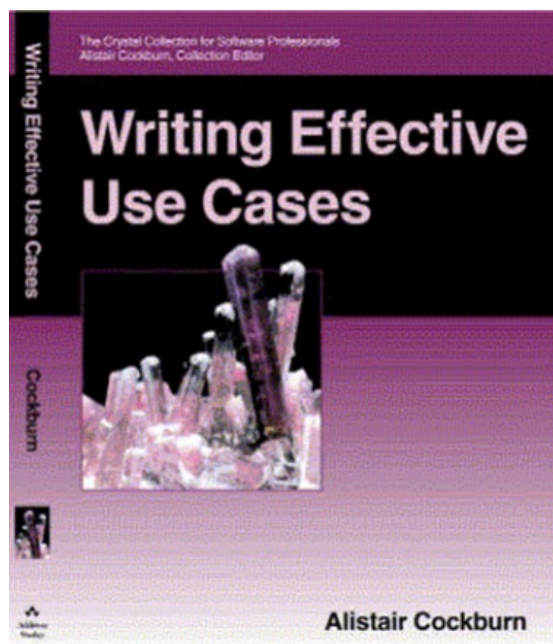
# Actors

- More formally, a user is represented by an actor
  - Each use case can have one or more actors involved
    - An actor can be either a human user or a software system
- Actors have two defining characteristics
  - They are external to the system under design
  - They take initiative and interact with our system
  - During a use case, they have a goal they are trying to achieve
- Each use case describes a task or tasks for a particular actor
  - The description typically includes one “success” case and a number of extensions that document “exceptional” conditions



# Text-based Use Cases

- From a presentation by Alistair Cockburn, author of Writing Effective Use Cases
  - Presentation is: Agile Use Cases
  - <http://alistaircockburn.us/get/2231>
  - What is and isn't a use case good for:



## **Good use cases** **are**                      **aren't**

**Text**  
**No GUI**  
**No data formats**  
**3 - 9 steps in main scenario**  
**Easy to read**  
**At user's goal level**  
**Record of decisions made**

**UML use case diagrams**  
**describing the GUI**  
**describing data formats**  
**multiple-page main scenario**  
**complicated to read**  
**at program-feature level**  
**tutorial on the domain**

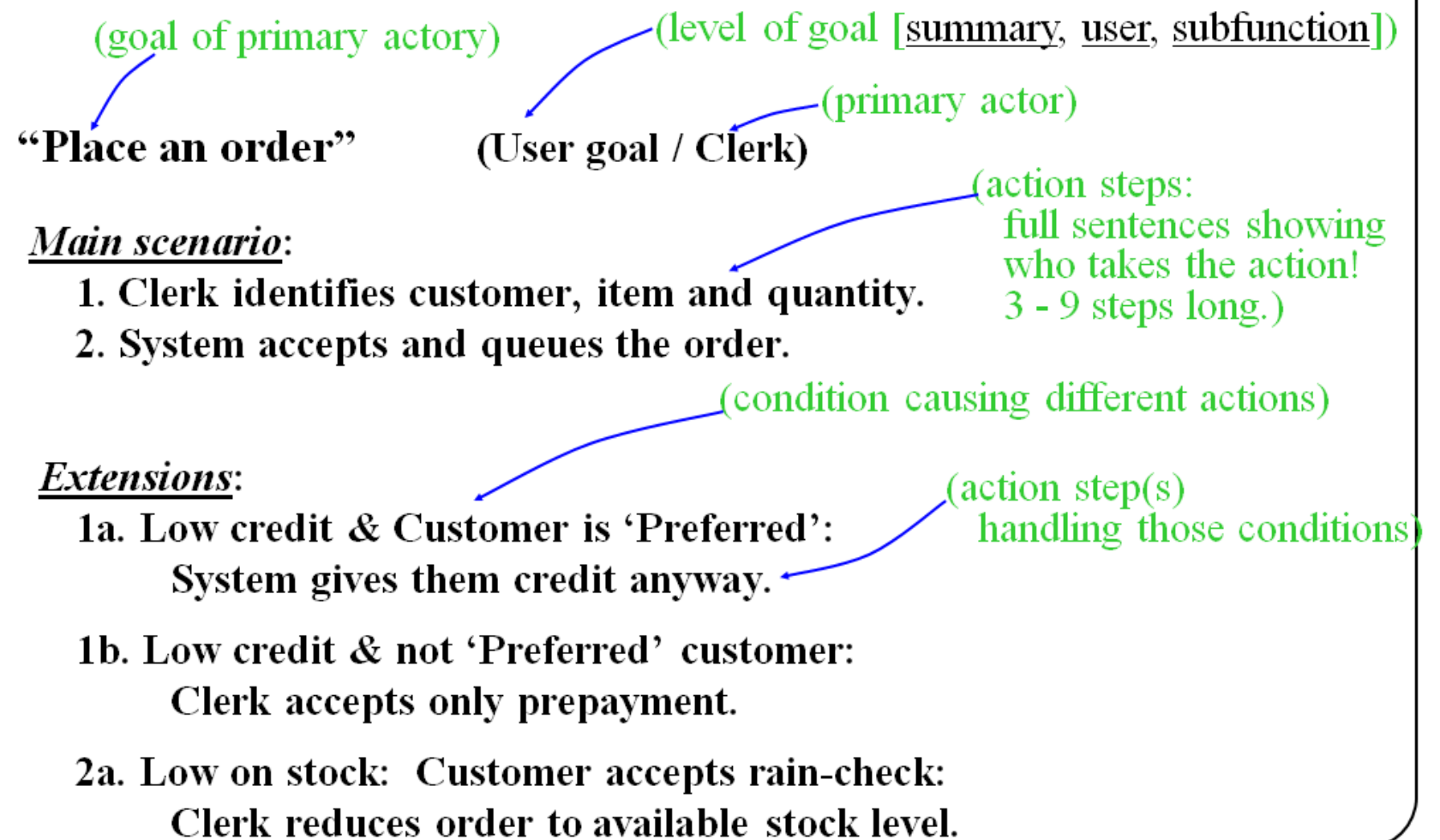
**Use cases \*can be\* written --**  
**all up front    --or--    just-in-time**  
**each to completion    --or--    in (usable) increments**

# Text-based Use Cases

- Four benefits:
  - Short summary of system goals
  - Main success scenario (system responsibility)
  - Extension conditions (things to watch for or consider)
  - Extension handling (decisions on policy)
- From a presentation by Alistair Cockburn
  - Agile Use Cases
  - <http://alistaircockburn.us/get/2231>

Robert Martin: "It shouldn't take longer than 15 minutes to teach someone how to write a use case!"

**Use case: Text describing scenarios of user succeeding or failing to achieve goal.**

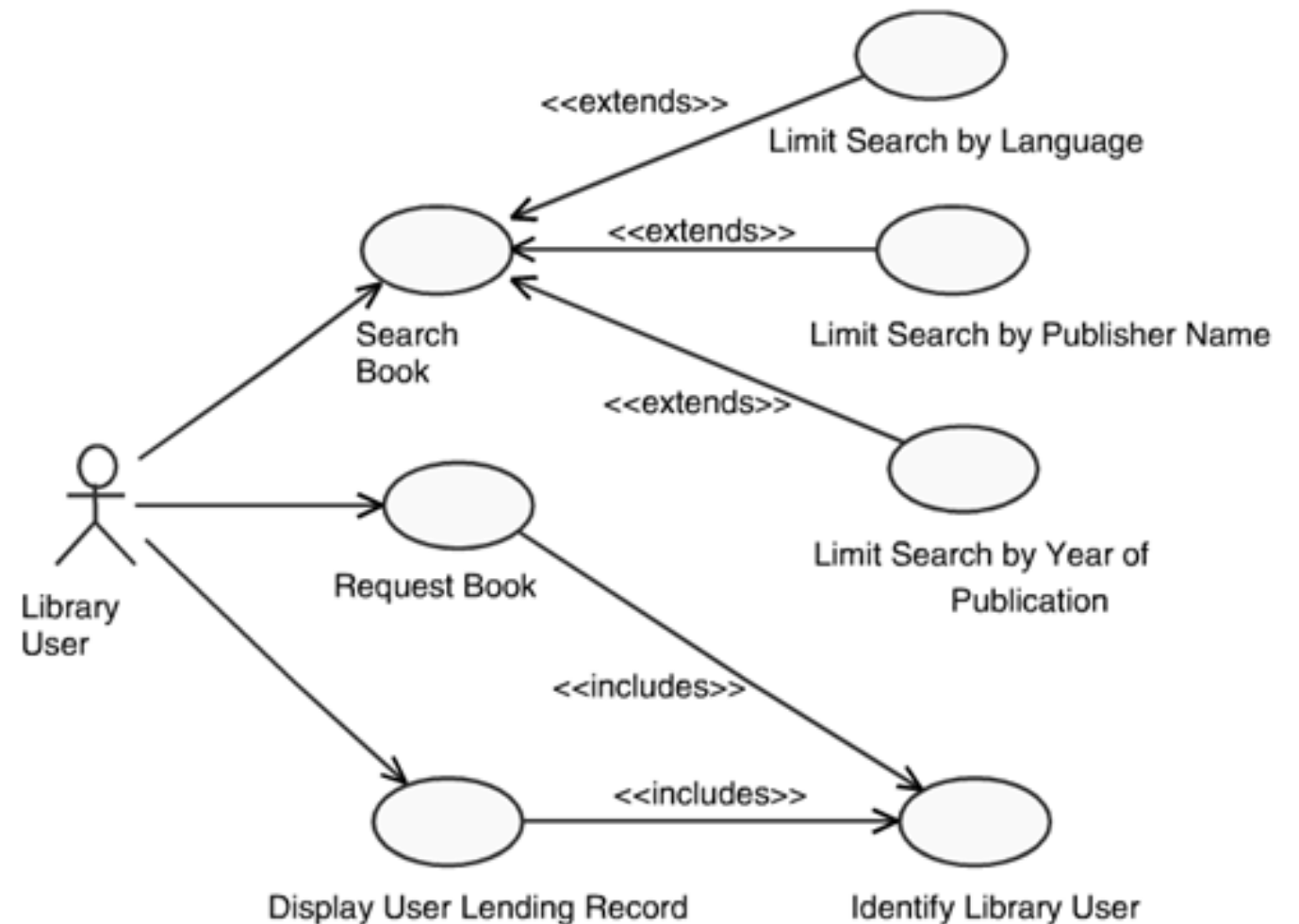


- Use Cases contain scenarios
  - A complete path through a use case from the first step to the last is called a **scenario**
  - Some use cases have multiple scenarios but a single user goal
    - All paths try to achieve victory



# UML Use Cases – Best Practices

- Always design use cases from the actor's point of view
- Model the entire flow of a given operation
- For most systems, use cases should number in the tens, not hundreds
- <include> cases: not optional, base use case not complete without it, not conditional, and doesn't change the base use case behavior
- <extend> cases: Can be optional, not part of base use case, can be conditional or change behavior



## **WAVE** Test for Use Cases (from Maksimchuk)

**W:** Use case describes WHAT to do, not how

**A:** ACTOR'S point of view

**V:** Has VALUE for actor

**E:** Use case models ENTIRE scenario

# What are Activity & State Diagrams?

- They represent alternate ways to record/capture design information about your system
- They can help you identify new classes and methods
- They are typically used after use case creation: for instance, create an activity diagram for a given use case scenario
- For each activity in the diagram, (you might) follow-on and draw a sequence diagram
  - Add a class for each object in the sequence diagrams to your class diagram, add methods in sequence diagrams to relevant classes
  - Remember – sequence diagrams may not needed for simple logic

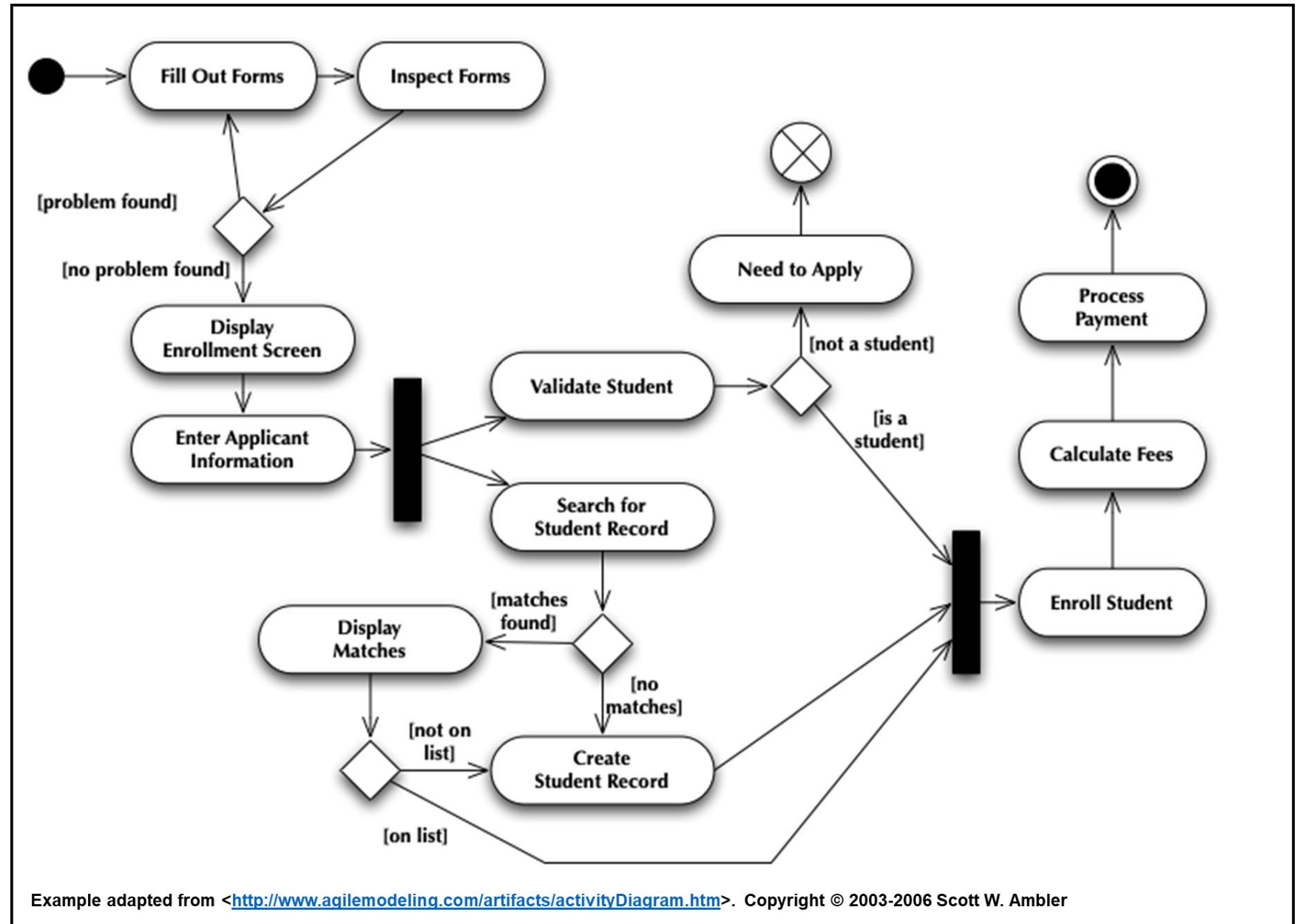
# What are Activity & State Diagrams?

- Activity Diagram
  - Think “Flow Chart on Steroids”
  - Able to model complex, parallel processes with multiple ending conditions
- State Diagram
  - Shows the major states of an object or system
  - partition an object’s behavior into various categories (initializing, acquiring info, performing calcs, ...)
  - documents these states and the transitions between them (transitions typically map to method calls)

# Activity Diagrams

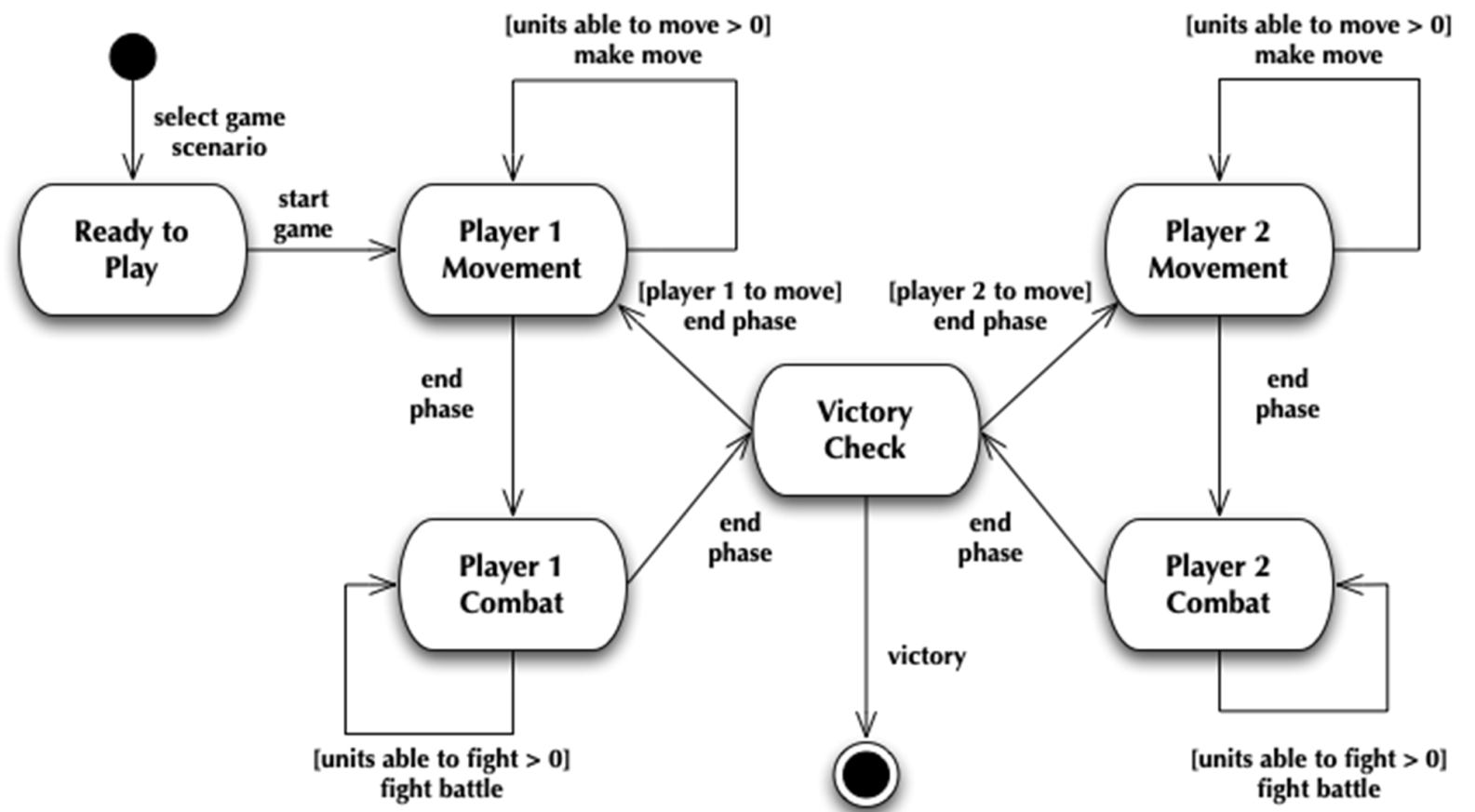
## Notation

- Initial Node (circle)/Final Node (circle in circle)/Early Termination Node (circle with x through it)
- Activity: Rounded Rectangle indicating an action of some sort either by a system or by a user
- Flow: directed lines between activities and/or other constructs. Flows can be annotated with guards "[student on list]" that restrict its use
- Fork/Join: Black bars that indicate activities that happen in parallel
- Decision/Merge: Diamonds used to indicate conditional logic.



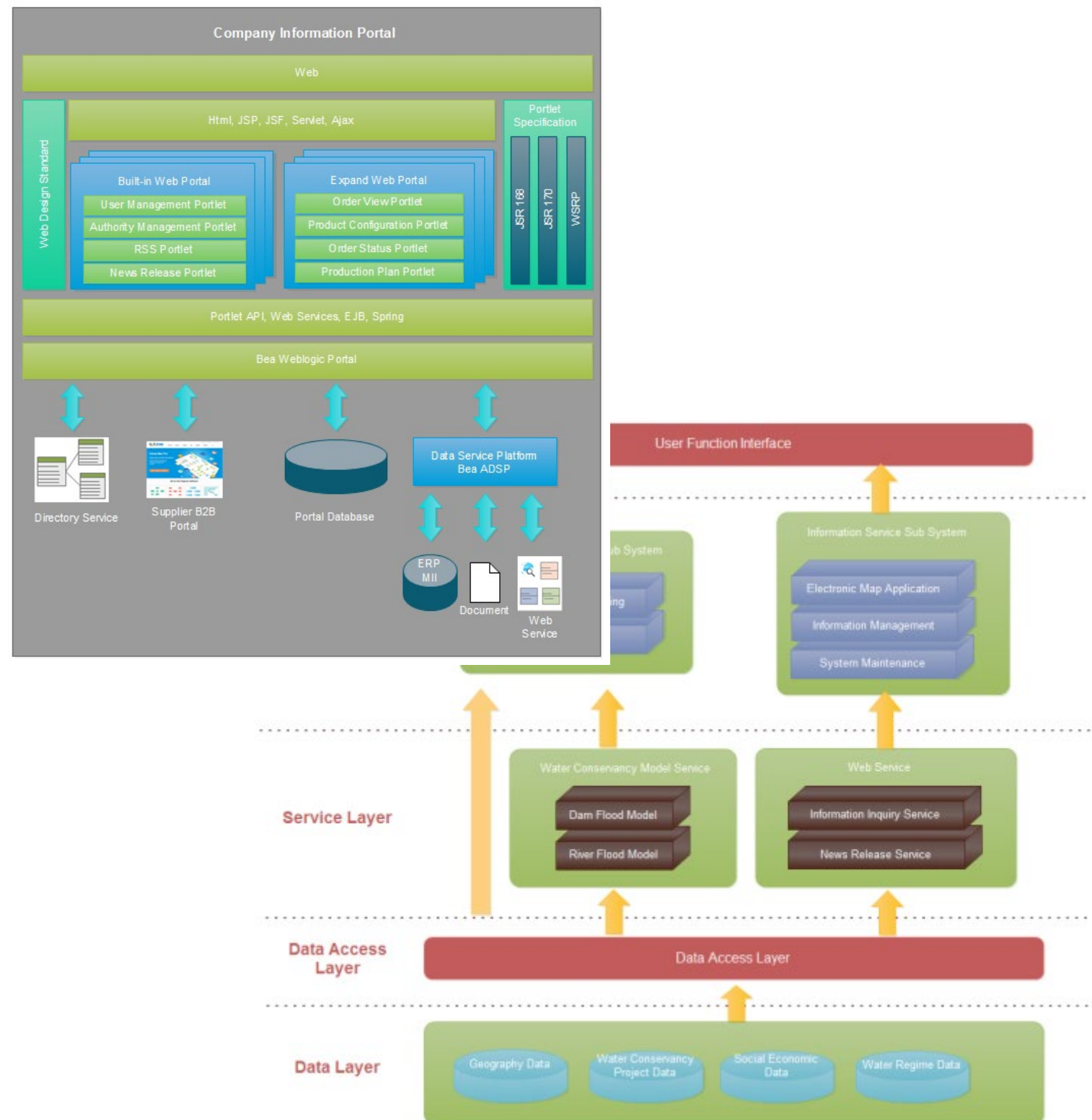
# State Diagrams

- Each state appears as a rounded rectangle
- Arrows indicate state transitions
  - Each transition has a name that indicates what triggers the transition (often times, this name corresponds to a method name)
  - Each transition may optionally have a guard that indicates a condition that must be true before the transition can be followed
- A state diagram also has a start state and an end state



# Architecture Diagram (not UML)

- A common diagramming style to present a high-level view of a system is an architecture diagram
- This is usually produced as a layered image of the internal major sub-systems of an application, external elements, and communication connections
- Typical examples at <https://www.edrawsoft.com/architecture-diagram.php>



# Iterative UML-based Development Process

- Once you have written requirements and use cases to fulfill them
  - and you've reviewed the use cases with clients to determine the various alternate paths
  - You're ready to start creating class diagrams, activity diagrams, state diagrams and sequence diagrams using information in the use cases as inspiration
  - Details are developed in iterative change and review
- Relationships between OO A&D Software Artifacts

