

# Ready for the Real World

---

Kenneth M. Anderson  
University of Colorado, Boulder  
CSCI 4448/6448 — Lecture 8 — 09/18/2007

# Lecture Goals

---

- Review material from Chapter 4 of the OO A&D textbook
  - Taking Your Software into the Real World
  - Alternative Designs / Design Trade-Offs
  - Use Case Analysis / More about Class Diagrams
  - Discuss the Chapter 4 Example: Todd & Gina's Dog Door, Take 3
  - Emphasize the OO concepts and techniques encountered in Chapter 4

# Real World Context

---

- A key problem in software development is gaining an understanding of the context in which your software must operate
  - Chapter 4 starts out by identifying a problem with our bark recognizer software from the end of Chapter 3: It opens for ANY bark... even if the bark belongs to some other dog!
- In the perfect world, everyone uses your system just like you expect
  - As the book says “Everyone is relaxed and there are no multi-dog neighborhoods here!”
- In the real world, (unexpected) stuff happens and things go wrong
- Analysis is the tool that can help you understand your software’s real-world context, identify potential problems, and help you avoid them

# The Role of Use Cases

---

- A well written use case can aid us in our goal of identifying real-world problems during the analysis phase
  - They are your means of communicating with your customers, your managers, and other developers about how your system will work in the real world
    - A customer may look at your scenarios and say “these are not very realistic”
    - Be open to comments like this, because you can then learn how to change your use cases to take into account the problems that will be encountered in the real world
- Once your use cases are updated, you can use them to glean the new requirements your system has to meet

# Initial Changes

---

- Make use case more generic
  - We've been a bit "folksy" up to now, referring to "Todd and Gina" and "Fido" in our use case
  - We'll switch to using phrases like "owner" and "dog"
    - Except that in Boulder, we have to say "guardian" not "owner" 😊
- We'll update the use case to make sure that we specify that the bark recognizer opens the door ONLY for the "owner's dog"
  - We were playing fast and loose with requirements last time
    - looking at what we needed to do to introduce BarkRecognizer to our design, without thinking long and hard about what it really needed to do

# New Use Case

## What the Door Does

### Main Path

1. The owner's dog barks to be let out.
- 2. The bark recognizer "hears" a bark.**
- 3. The bark recognizer detects the owner's dog and opens the door.**
4. The dog door opens.
5. The owner's dog goes outside.
6. The owner's dog does his business.
  - 6.1 The door shuts automatically
  - 6.2 Fido barks to be let back inside.
  - 6.3 The bark recognizer "hears" a bark (again).**
  - 6.4 The bark recognizer detects the owner's dog and opens the door**
  - 6.5 The dog door opens (again).
7. Fido goes back inside.

### Alternate Paths

- 2.1 The owner hears her dog barking.**
- 3.1 The owner presses the button on the remote control.**
  
- 6.3.1 The owner hears her dog barking (again).**
- 6.4.1 The owner presses the button on the remote control.**

# Discussion

---

- Note: I did things slightly differently from the book
  - I changed step 3 to say “The bark recognizer detects the owner’s dog and opens the door”
  - The book said “If it’s the owner’s dog barking, the bark recognizer sends a request to the door to open”
    - I didn’t like the use of “if” in this action step, instead I just decided that the bark we hear is always the owner’s dog.
    - I can add an additional path to this use case in which I can say something like: “The bark recognizer detects an unknown dog. Use case terminates.” Or I can create a separate use case that documents this behavior
- Note: my version of step 3 can be further improved by splitting it into two steps: one that does the detection and one that asks for the door to open

# New Use Case

---

- If the bark recognizer is going to determine if a bark belongs to the owner's dog, we need to store a representation of that dog's bark

## Storing a dog bark

1. The owner's dog barks "into" the door.
2. The door stores the owner's dog's bark.

- This may seem like its "not enough":
  - Pros: simple, primary actor should be dog in this use case
  - Cons: It feels a bit weird not to have a step that says something like "The owner issues a command to the door to prepare it to store the dog's bark"
    - But since that step sounds awkward, make it a precondition

# The Competition

---

- The book now holds a design competition between two programmers
  - Randy: simple is best right?
    - Bark sounds are just strings... I'll store the owner's dog's bark in the dog door and then just do a string comparison in bark recognizer
  - Sam: object lover extraordinaire
    - A “bark” is an important concept in our application domain. Lets make it a class and have it take care of “bark comparison”

# Discussion

---

- **Randy's approach**

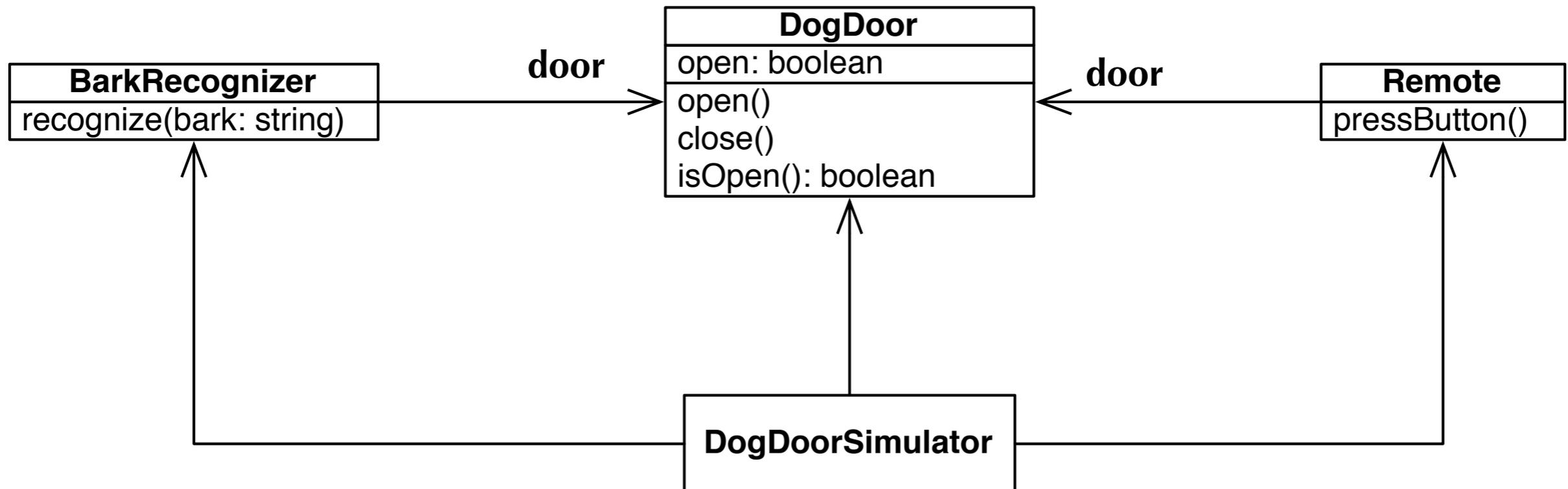
- Agile approach to software development
  - What is the simplest thing I can do today to meet my requirements?
  - Avoids “speculative complexity”
- Fast: doesn't take long to modify the DogDoor class and update the BarkRecognizer to do the appropriate string comparison

- **Sam's approach**

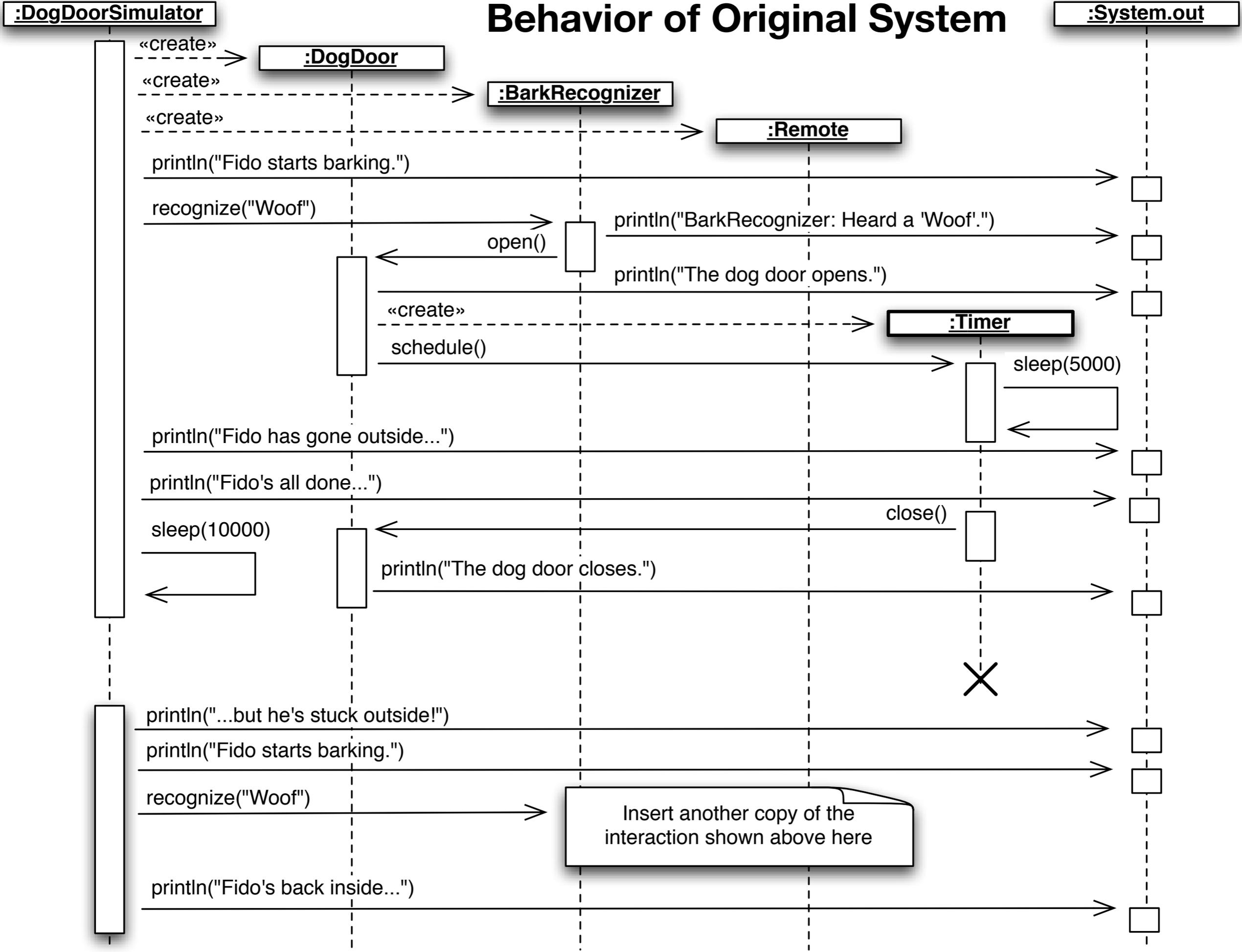
- Makes use of good OO design principles
  - Encapsulation and Delegation
    - A “bark” is something we need to track  $\Rightarrow$  it should be a class
    - Barks are strings now; But what if they turn into .wav files?
    - Use delegation to hide those details from the rest of the system

# Original Class Diagram

---



# Behavior of Original System



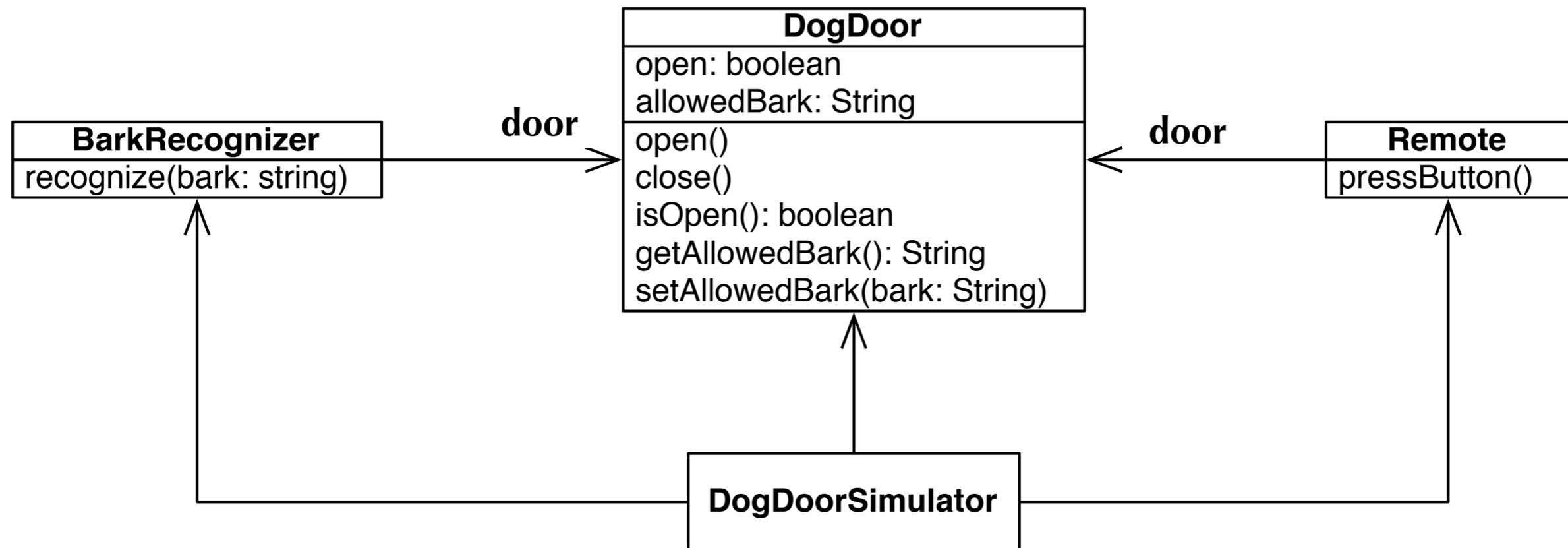
# Introduction to Sequence Diagrams

---

- 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”
- 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 (!)

# Randy's Class Diagram

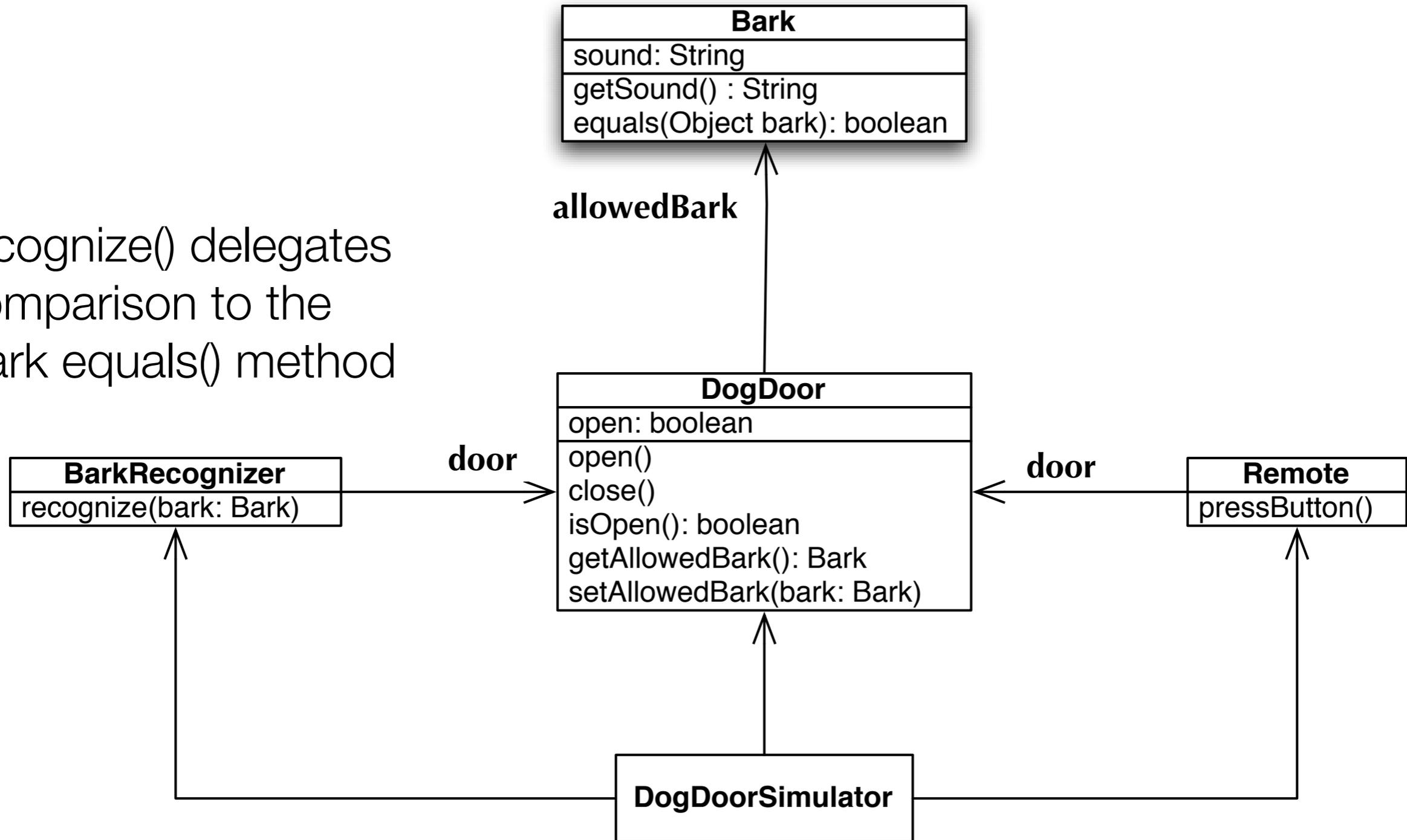
---



BarkRecognizer's `recognize()` method has been updated to call `getAllowedBark()` and check to see if it matches the bark passed to it

# Sam's Class Diagram

recognize() delegates comparison to the Bark equals() method



# The Power of Delegation

---

- Sam's application is shielded from the details of how a "bark" is implemented
  - By using delegation to do the comparison of bark objects, his BarkRecognizer doesn't have to know that internally a bark is represented as a String
    - If we change the way a bark is represented, BarkRecognizer will be unaffected
  - Contrast with an alternative approach of BarkRecognizer calling the getSound() method of its two Bark objects and then doing a comparison itself; BarkRecognizer would then be tied to the implementation of the Bark class
- Delegation shields your objects from implementation changes to other objects in your software
  - The coupling between Bark and BarkRecognizer is looser having used delegation; there is still some coupling between them, but its not tight

# The Results

---

- Sam's and Randy's solutions both work but both of them lost the competition!
  - They lost to a summer intern (and "junior" programmer), Maria
- Why?
  - She did a deeper analysis of the problem domain and identified a problem that both Sam and Randy ignored
    - The same dog can have different types of barks!
      - when its excited, sleepy, hungry, angry, scared, etc.
- Sam's and Randy's solutions would both fail in the real world
  - Maria succeeded because she applied **textual analysis** to the use case
    - She realized that it was the **dog** that was the focus, not the **bark**

# Textual Analysis

---

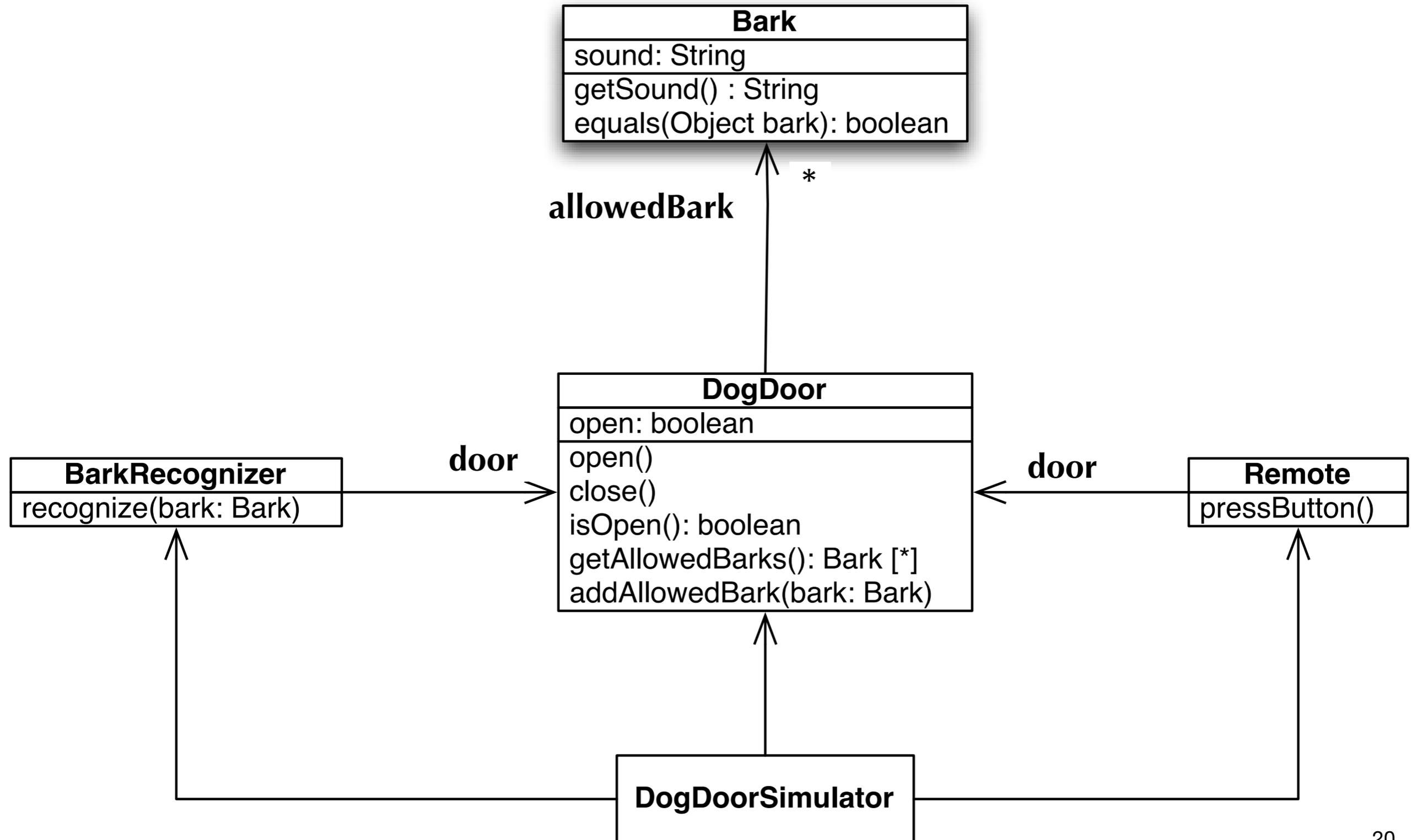
- Pay attention to the **nouns** in your use case
  - They may indicate a **potential candidate for a class** in your system
    - Some things don't need to be tracked
    - For example, we don't need a class for "Dog" in this system
  - They also provide hints on what your design should focus on
- Pay attention to the **verbs** in your use case as well
  - They may indicate **potential candidates for methods** in your system
  - They will also provide hints as to where a method should "live"
    - i.e. what class should be assigned the responsibility of handling the service provided by the method

# Soft Science?

---

- The book discusses the potential problems with textual analysis
  - Wouldn't a slightly different wording of the use case lead to different results?
    - Yep
  - But, as they point out, only one or two wordings will accurately capture the real-world context that your system will find itself in
    - If you get your analysis wrong, you'll end up focused on the wrong thing, and even if your design is good, your system will fail
  - Also, "everyone does it": this technique shows up in all OO A&D methods
- A good use case clearly and accurately explains what a system does, in language that's easily understood with real world context captured
- With a good use case, complete textual analysis is a quick and easy way to identify the **potential** classes and methods of your system

# Maria's Class Diagram



# New Notation

---

- In Maria's class diagrams in the book, you encountered a new notation
- For Attributes
  - allowedBarks: Bark [\*]
- For Methods
  - getAllowedBarks(): Bark [\*]
- (actually, its just a new type notation)
- It means that the type of allowedBarks and the return type of getAllowedBarks() is a collection of zero or more Bark objects
- You can indicate a specific multiplicity like this
  - allowedBarks: Bark [2..6] or allowedBarks: Bark [20]

# Demonstration

---

- Lets take a look at the final version of the software

# Class Diagrams are Incomplete

---

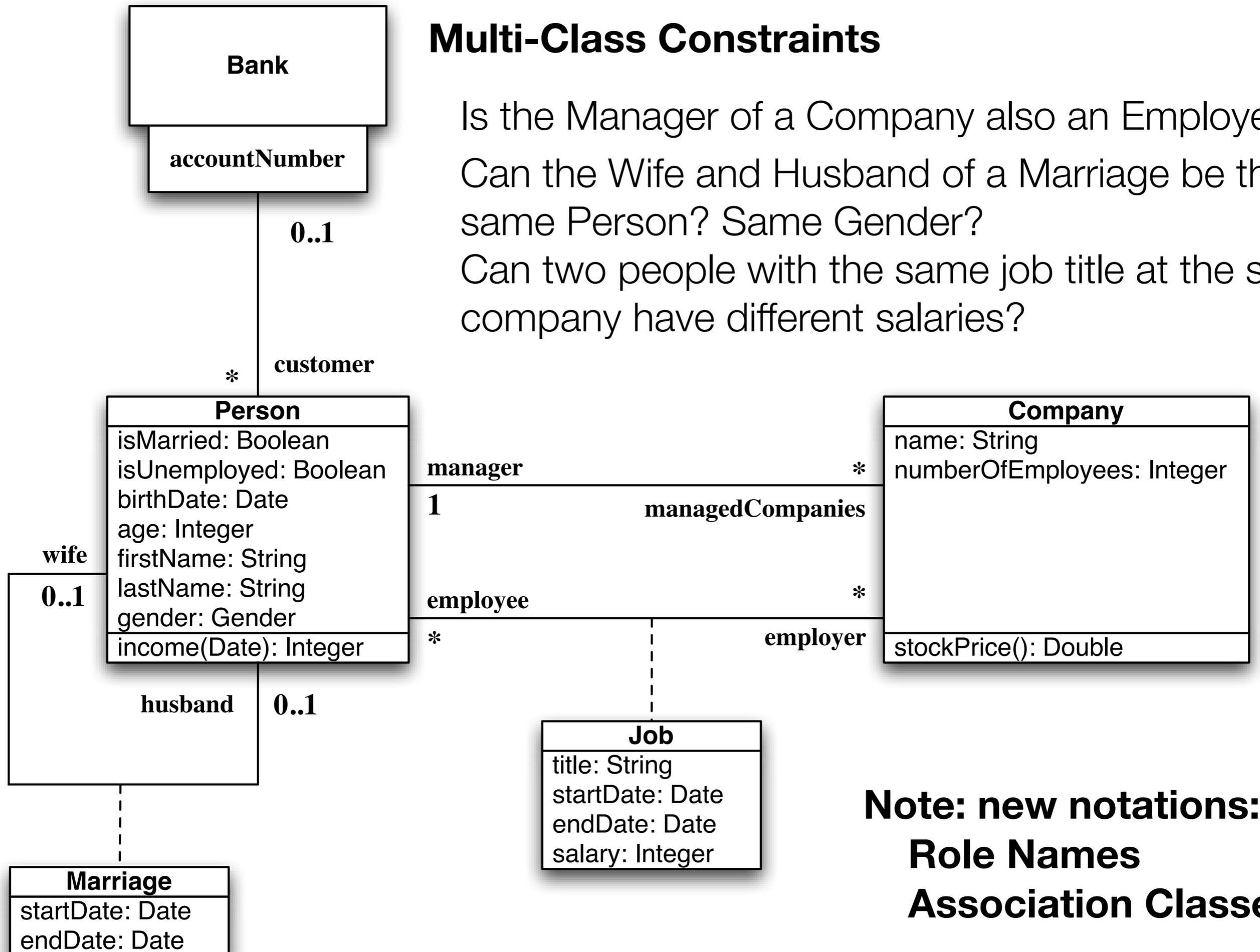
- Class diagrams are useful but they do not provide a **complete** picture
  - They provide limited type information
    - Types are optional, and when a type specifies a multiplicity it does not indicate what collection class should be used
  - They don't tell you how to code a method
    - You'll need a use case or sequence diagram for that
  - They almost never talk about constructors
  - They do not provide information on how associations are instantiated
  - They don't provide explicit information on the purpose of a class
    - That info comes from its associated requirements and use cases
  - They are horrible at specifying constraints that span multiple classes

# Multi-Class Constraints

Is the Manager of a Company also an Employee?

Can the Wife and Husband of a Marriage be the same Person? Same Gender?

Can two people with the same job title at the same company have different salaries?



**Note: new notations:**  
**Role Names**  
**Association Classes**

# OCL to the Rescue!

---

- To handle the problem of constraints that span multiple classes, a designer can make use of the object constraint language
  - OCL version 2.0 specification located here:
    - <http://www.omg.org/docs/ptc/05-06-06.pdf>
  - Essentially defines a first-order predicate logic that has its domain in UML models; you can construct statements like
    - All Persons employed by Company X with Job title Y have salary Z
    - For all Persons X, manager(X, Company) and employee(X, Company)
    - etc.
  - You can then annotate UML models with OCL constraints using notes
    - Example of “note” shown on slide 12
  - Knowledge of OCL is NOT required for this class

# Wrapping Up

---

- Systems fail if their developers failed to take into account the problems that they will encounter in the real world
  - Its tough to model the real world accurately but it can be done
    - if you are willing to expend the effort to create good use cases
    - A good use case precisely lays out what a system does, but does not indicate how the system accomplishes that task
- Textual analysis can provide you with information on the candidate classes and methods of your system
  - they also indicate where to focus when creating the design of your system
    - get the use case wrong, and you'll focus on the wrong thing

# Ken's Corner

---

- Lets Practice Modeling
  - Jewel Quest II
    - Lets take a look at the game and then identify classes and methods
      - Be writing down potential classes during the demo
  - iTunes (if we have time)
    - Huge number of responsibilities
      - It used to be just a music player; Now it:
        - plays music, podcasts, tv shows, movies, stores apps/games
        - synchronizes devices (ipods, Apple TV, iPhones)
        - iTunes store (purchases and rentals)
        - App store for iPhone

# Coming Up Next

---

- Lecture 9: Nothing Stays the Same
  - Read Chapter 5 (part 1 and interlude) of the OO A&D book
- Lecture 10: Flexible Software
  - Read Chapter 5 (part 2) of the OO A&D book