Facade & Adapter

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 8 — 02/11/2019

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

Goals of the Lecture

- Some interactive class stuff
- Introduce two design patterns
  - Facade
  - Adapter
- Compare and contrast the two patterns
- Look at multiple inheritance
- Review some housekeeping (midterm, grad presentation outline)

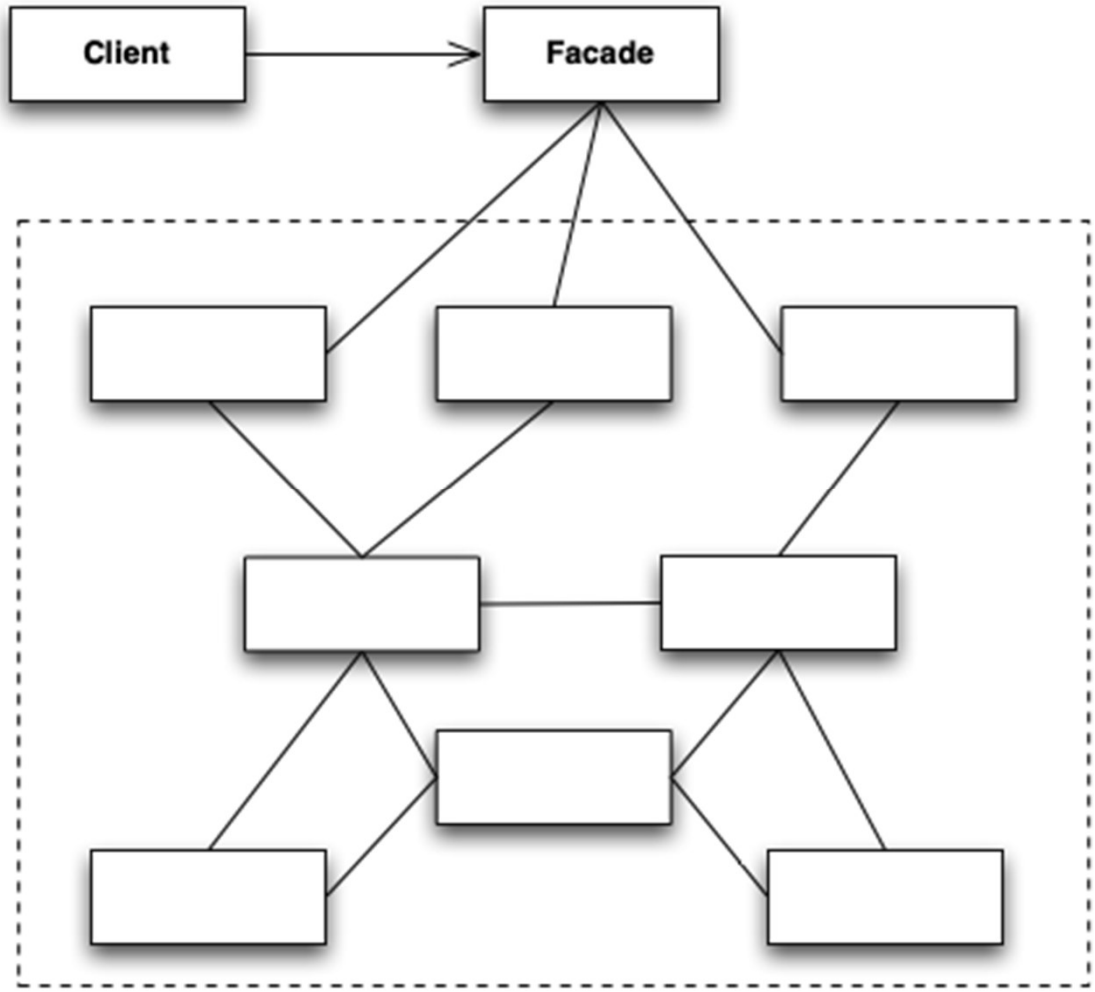
The dreaded name-card based activity!

- I will offer a question
- You will raise your hand if you'd like to answer it
- I will recognize you (via your handy name-card)
- I will provide a question for you
- You will answer it so that the class can hear you
- The class will vote, via a show of hands, on the correctness of your answer
- If you are voted correct (>50%), you will receive 3 Extra Credit Homework Points!
- If not, it will go to the next student who wants to answer it for 2 Points, and so on.
- There are 10 questions today
- Distance students, I will come up with a similar Extra Credit opportunity for you. There will also be other little extra credit opportunities if you don't like this one.
  Do not panic.

Facade (I)

- “Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.”
  - Design Patterns, Gang of Four, 1995
- There can be significant benefit in wrapping a complex subsystem with a simplified interface
  - If you don’t need the advanced functionality or fine-grained control of the former, the latter makes life easy

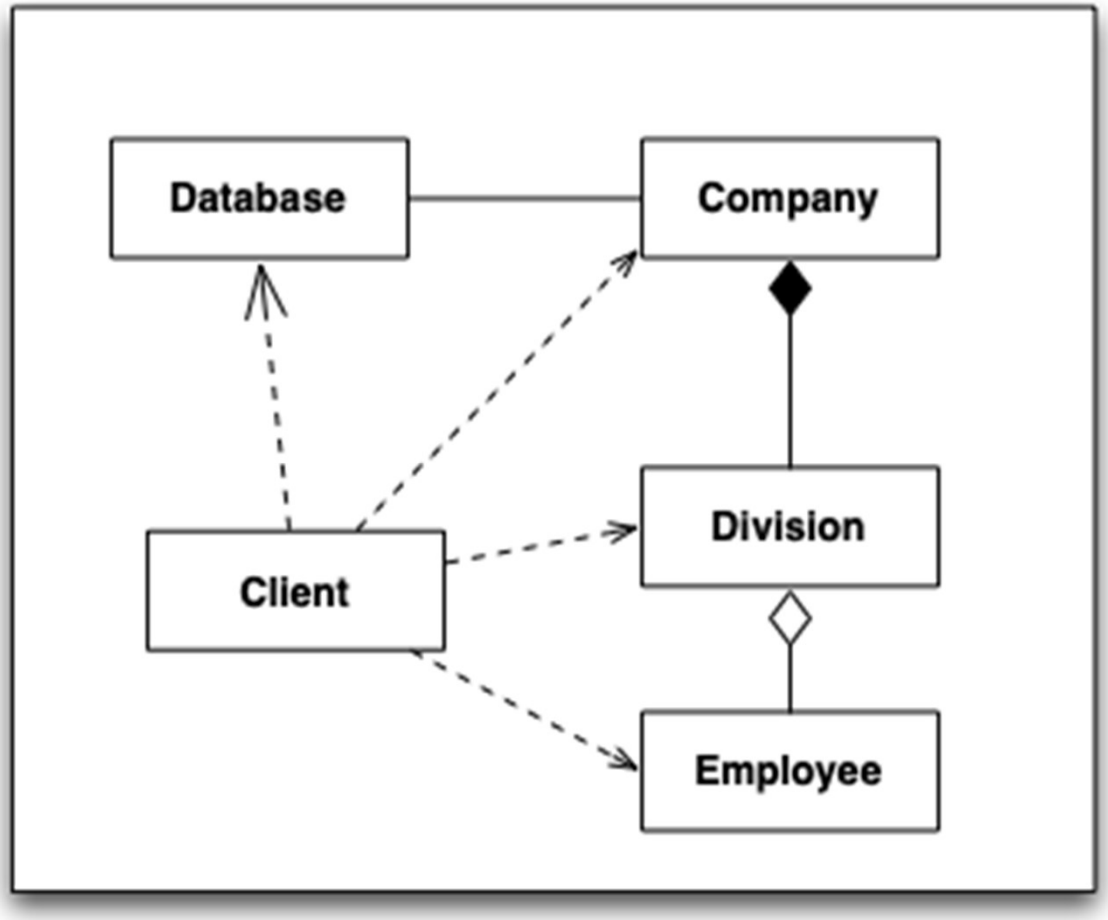
Facade Pattern: Structure



Facade (II)

- Facade works best when you are accessing a subset of the subsystem's functionality
  - You can also add new features by adding it to the Facade (not the subsystem); you still get a simpler interface
- Facade not only reduces the number of methods you are dealing with but also the number of classes
  - Imagine having to pull Employees out of Divisions that come from Companies that you pull from a Database
  - A Facade in this situation can fetch Employees directly

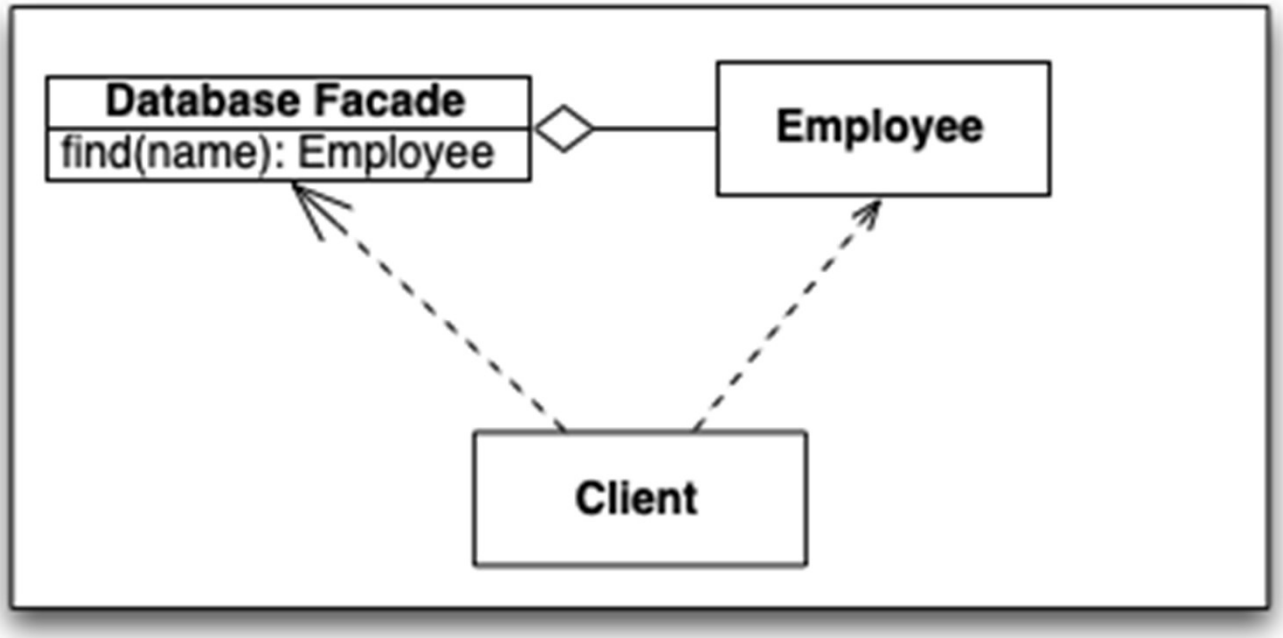
Example (Without a Facade)



Without a Facade, Client contacts the Database to retrieve Company objects. It then retrieves Division objects from them and finally gains access to Employee objects.

It uses four classes.

Example (With a Facade)



With a Facade, the Client is shielded from most of the classes. It uses the Database Facade to retrieve Employee objects directly.

Real World Example: Core Audio

- Consider Core Audio, included in iOS
  - If you want to access that subsystem directly, you have up to 8 frameworks that you need to deal with
    - AudioToolbox, AudioUnit, AVFoundation, CoreAudio, CoreAudioKit, CoreMIDI, CoreMIDIServer & OpenAL
  - However, if all you need to do is play a sound, you can use a single class, AVAudioPlayer, which acts as a Facade

Facade Example (I)

- Imagine a library of classes with a complex interface and/or complex interrelationships
  - Home Theater System
    - Amplifier, DvdPlayer, Projector, CdPlayer, Tuner, Screen, PopcornPopper (!), and TheatreLights
      - each with its own interface and interclass dependencies
- Imagine steps for “watch movie”
  - turn on popper, make popcorn, dim lights, screen down, projector on, set projector to DVD, amplifier on, set amplifier to DVD, DVD on, etc.
- Now imagine resetting everything after the movie is done, or configuring the system to play a CD, or play a video game, etc.

Facade Example (II)

- For this example, we can place high level methods...
  - like “watch movie”, “reset system”, “play cd”
- ... in a facade object and encode all of the steps for each high level service in the facade
- Client code is simplified and dependencies are reduced
  - A facade **not only simplifies an interface**, it **decouples a client from a subsystem of components**
- Indeed, Facade lets us **encapsulate subsystems**, hiding them from the rest of the system

Principle of Least Knowledge

- aka Talk only to your friends
- Be careful how many classes an object interacts with
- And also, how it comes to interact with those classes
- Reduce the chance of a change cascade when many classes interact
- Improve maintainability and reduce complexity

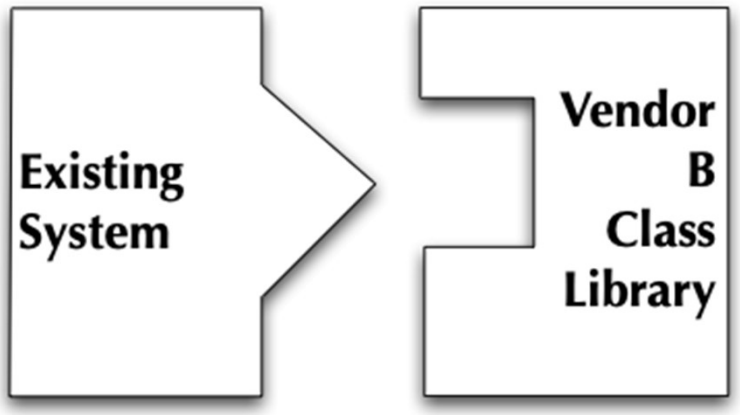
Adapters in the Real World

- Our next pattern provides steps for converting an incompatible interface with an existing system into a different interface that is compatible
  - Real World Example: AC Power Adapters
  - Electronic products made for the USA cannot be used directly with outlets found in most other parts of the world
    - To use these products outside the US, you need an AC power adapter
    - In some case, you also need a AC power transformer/converter
      - which is a separate, orthogonal issue
      - but these issues are sometimes mixed

OO Adapters (I)

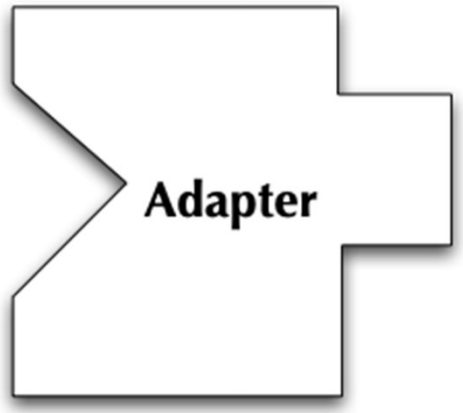
- Pre-Condition: You are maintaining an existing system that makes use of a third-party class library from vendor A
- Stimulus: Vendor A goes belly up and corporate policy does not allow you to make use of an unsupported class library.
- Response: Vendor B provides a similar class library but its interface is completely different from the interface provided by vendor A
- Assumptions: You don’t want to change your code, and you can’t change vendor B’s code.
- Solution?: Write new code that adapts vendor B’s interface to the interface expected by your original code

OO Adapters (II)



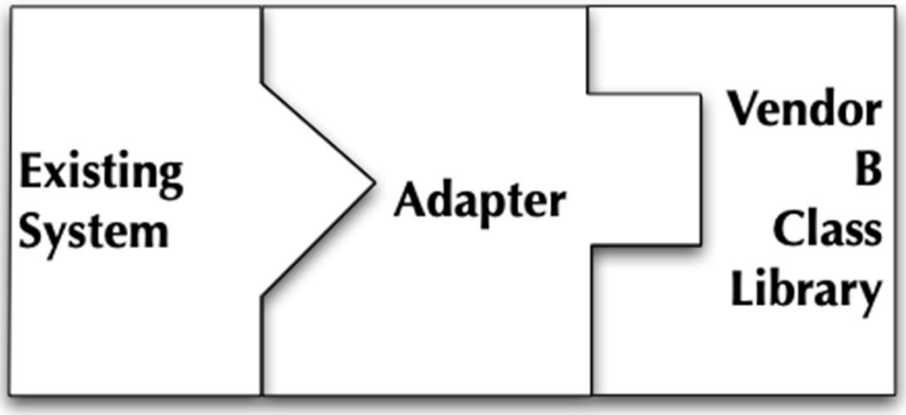
Interface Mismatch
Need Adapter

Create Adapter



And then...

OO Adapters (III)



...plug it in

Benefit: Existing system and new vendor library do not change, new code is isolated within the adapter.

Example: A turkey amongst ducks! (I)

- If it walks like a duck and quacks like a duck, then it must be a duck!

Or...

- If it walks like a duck and quacks like a duck, then it might be a turkey wrapped with a duck adapter… (!)

Example: A turkey amongst ducks! (II)

- Recall the Duck simulator from last lecture?

```
1  public interface Duck {
2      public void quack();
3      public void fly();
4  }
5
6  public class MallardDuck implements Duck {
7
8      public void quack() {
9          System.out.println("Quack");
10     }
11
12     public void fly() {
13         System.out.println("I'm flying");
14     }
15 }
16
```

Example: A turkey amongst ducks! (III)

- An interloper wants to invade the simulator

```
1  public interface Turkey {
2      public void gobble();
3      public void fly();
4  }
5
6  public class WildTurkey implements Turkey {
7
8      public void gobble() {
9          System.out.println("Gobble Gobble");
10     }
11
12     public void fly() {
13         System.out.println("I'm flying a short distance");
14     }
15
16 }
17
```

Example: A turkey amongst ducks! (IV)

- Write an adapter, that makes a turkey look like a duck

```
1  public class TurkeyAdapter implements Duck {
2
3      private Turkey turkey;
4
5      public TurkeyAdapter(Turkey turkey) {
6          this.turkey = turkey;
7      }
8
9      public void quack() {
10         turkey.gobble();
11     }
12
13     public void fly() {
14         for (int i = 0; i < 5; i++) {
15             turkey.fly();
16         }
17     }
18
19 }
20
```

Demonstration

1. Adapter implements target interface (Duck).

2. Adaptee (turkey) is passed via constructor and stored internally

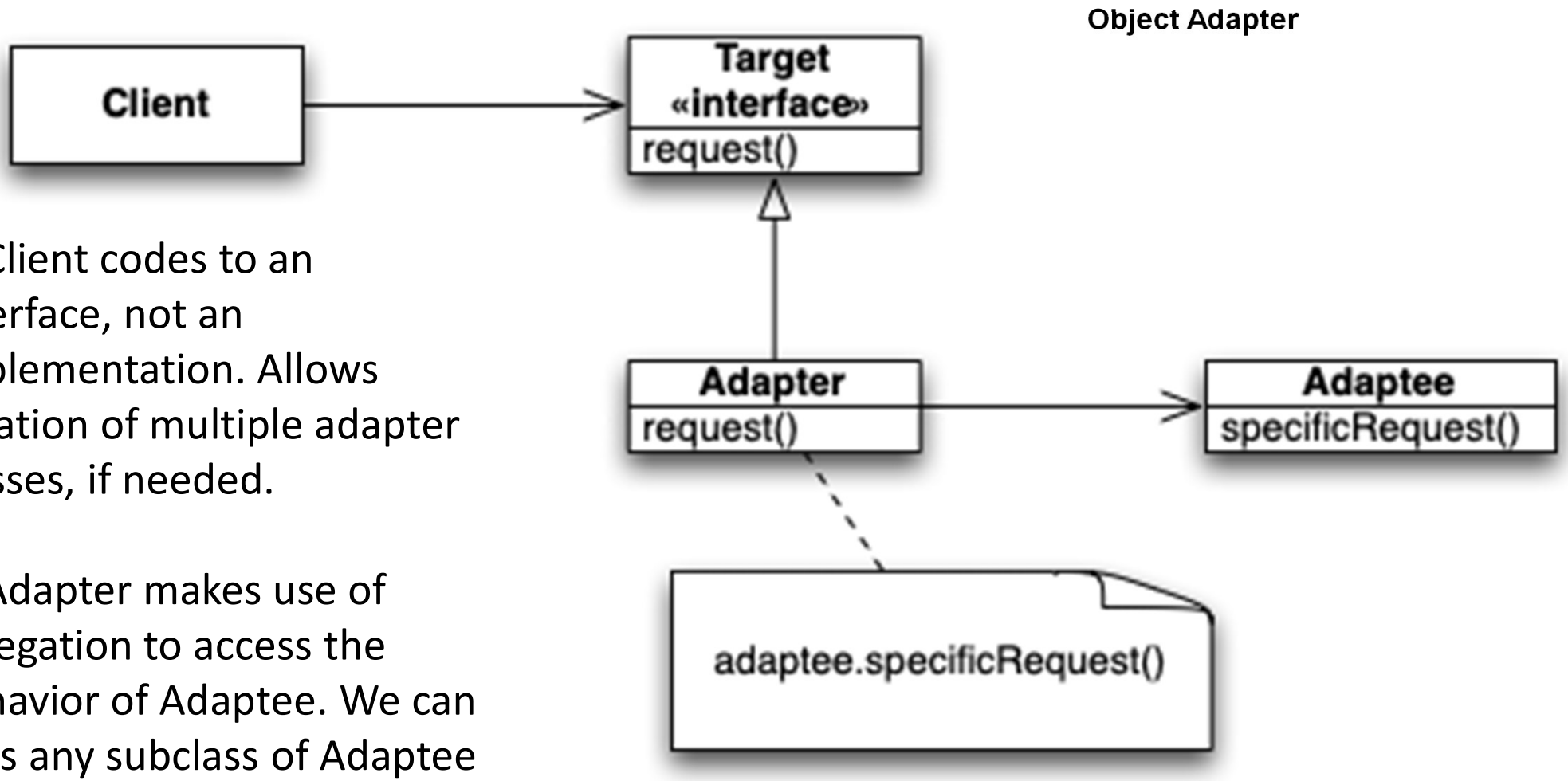
3. Calls by client code are delegated to the appropriate methods in the adaptee

4. Adapter is full-fledged class, could contain additional vars and methods to get its job done; can be used polymorphically as a Duck

Adapter Pattern: Definition

- The Adapter pattern converts the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces
  - The client makes a request on the adapter by invoking a method from the target interface on it
  - The adapter translates that request into one or more calls on the adaptee using the adaptee interface
  - The client receives the results of the call and never knows there is an adapter doing the translation

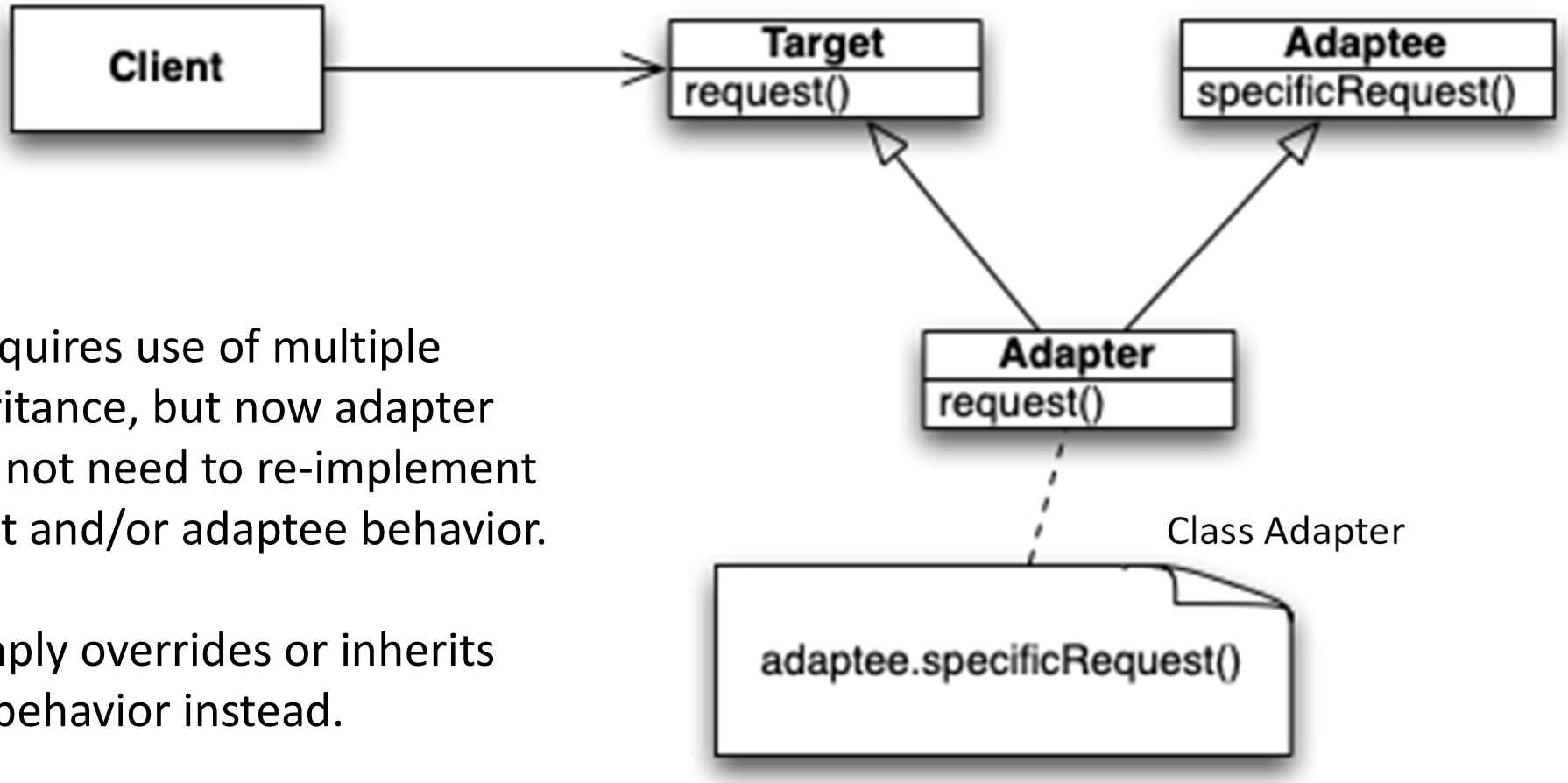
Adapter Pattern: Structure (I)



1. Client codes to an interface, not an implementation. Allows creation of multiple adapter classes, if needed.

2. Adapter makes use of delegation to access the behavior of Adaptee. We can pass any subclass of Adaptee to the Adapter, if needed.

Adapter Pattern: Structure (II)



1. Requires use of multiple inheritance, but now adapter does not need to re-implement target and/or adaptee behavior.

It simply overrides or inherits that behavior instead.

Comparison (I)

- To many people, these two patterns (Adaptor/Facade) appear to be similar
  - They both act as wrappers of a preexisting class
  - They both take an interface that we don't want and convert it to an interface that we can use
- With Facade, the intent is to simplify the existing interface
- With Adapter, we have a target interface that we are converting to
  - In addition, we often want the adapter to plug into an existing framework and behave polymorphically

Comparison (II)

- Superficial difference
  - Facade hides many classes; Adapter hides only one
- But
  - a Facade can simplify a single, very complex object
  - an Adapter can wrap multiple objects at once in order to access all the functionality it needs
- The key is simplify (facade) vs convert (adapter)

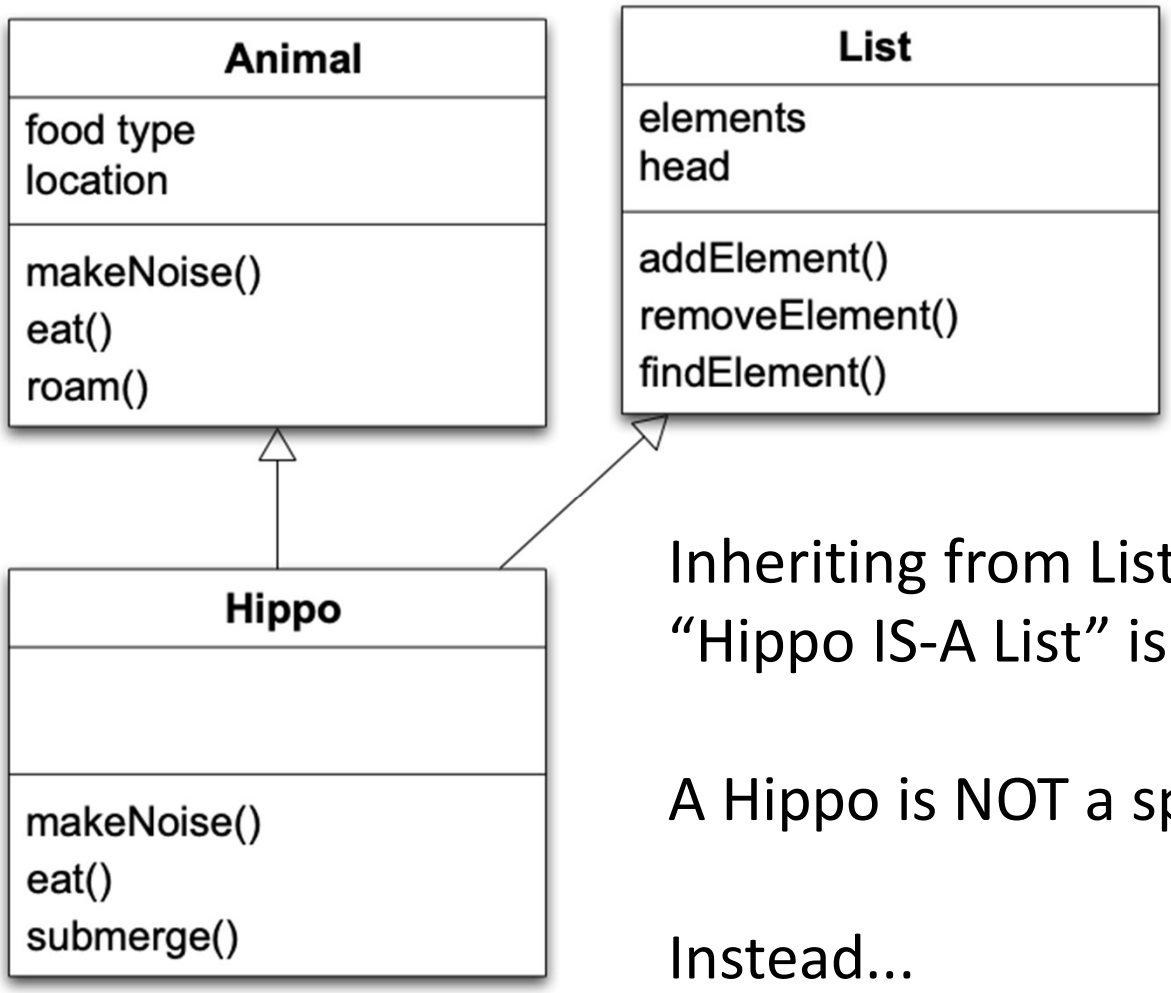
Multiple Inheritance

- Let’s talk a little bit more about multiple inheritance
  - Some material for this section taken from
    - Object-Oriented Design Heuristics by Arthur J. Riel
      - Copyright © 1999 by Addison Wesley
      - ISBN: 0-201-63385-X

Multiple Inheritance

- Riel does not advocate the use of multiple inheritance (its too easy to misuse it). As such, his first heuristic is
  - **If you have an example of multiple inheritance in your design, assume you have made a mistake and prove otherwise!**
- Most common mistake
  - Using multiple inheritance in place of containment
    - That is, you need the services of a List to complete a task
      - Rather than creating an instance of a List internally, you instead use multiple inheritance to inherit from your semantic superclass as well as from List to gain direct access to List's methods
        - You can then invoke List's methods directly and complete the task

Graphically

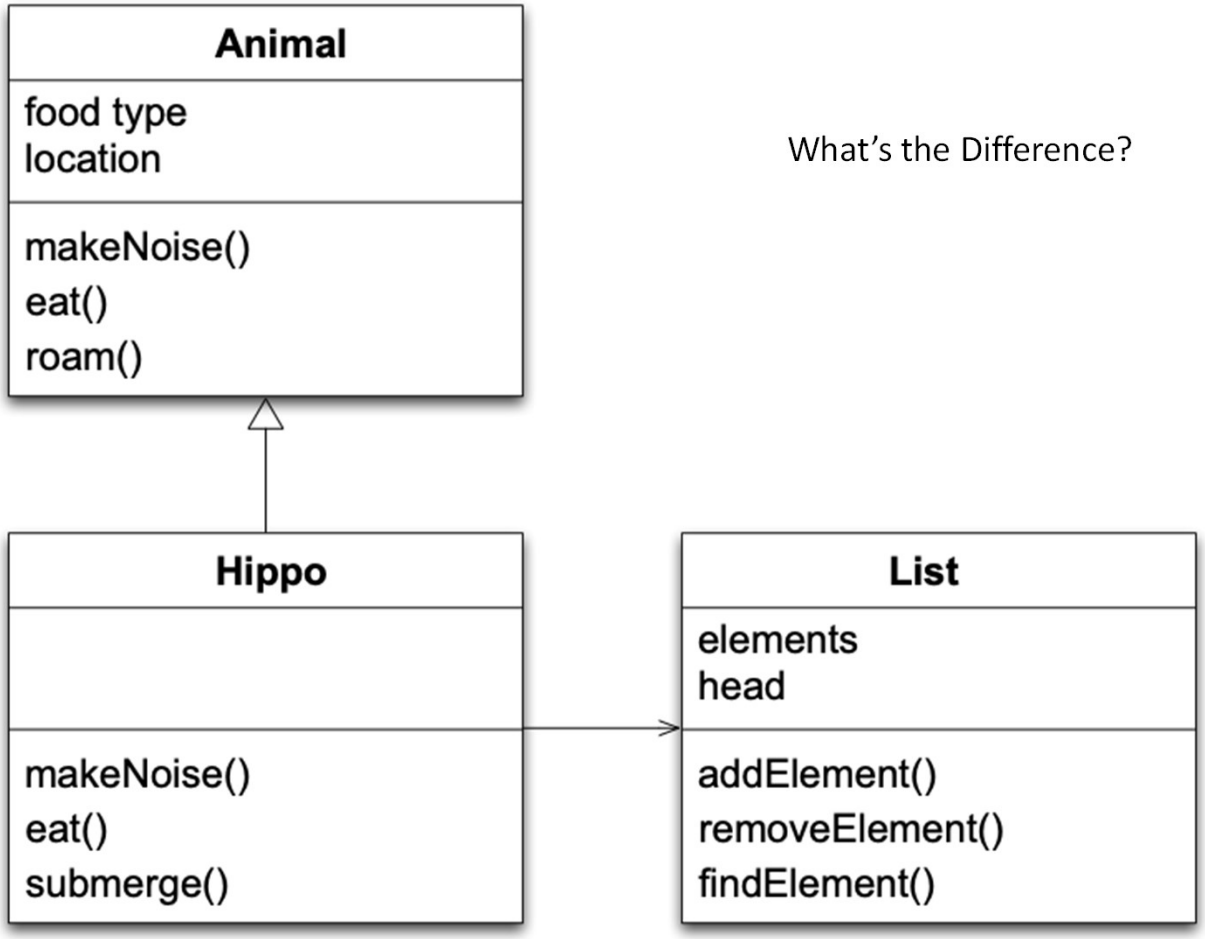


Inheriting from List in this way is bad, because "Hippo IS-A List" is FALSE

A Hippo is NOT a special type of List

Instead...

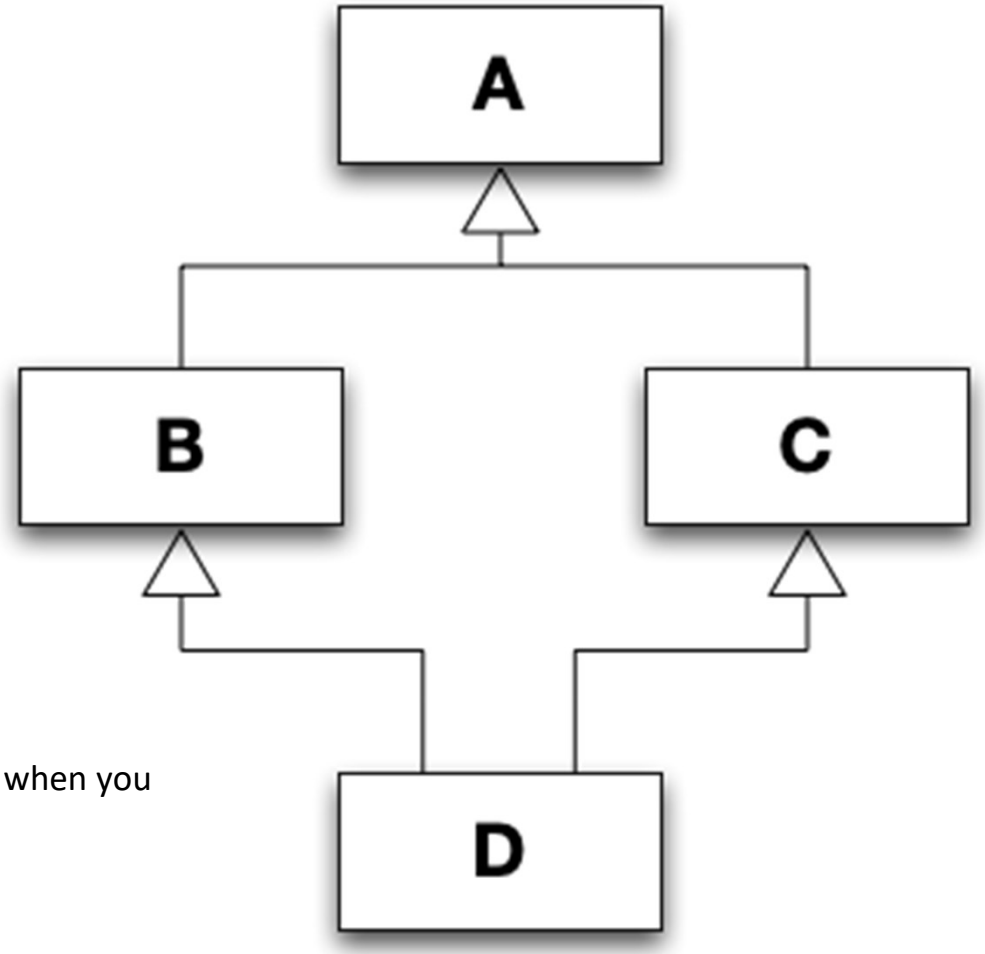
Do This



What's the Difference?

Another Problem

What's wrong with this?



Hint: think about what might happen when you create an instance of D

Multiple Inheritance

- A Second Heuristic
  - Whenever there is inheritance in an OO design, ask two questions:

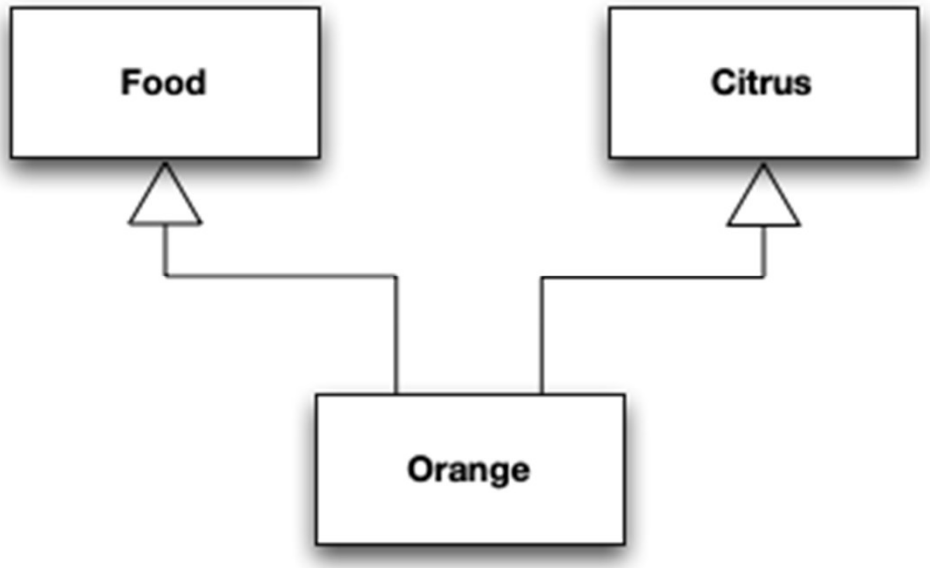
    1) Am I a special type of the thing from which I'm inheriting?

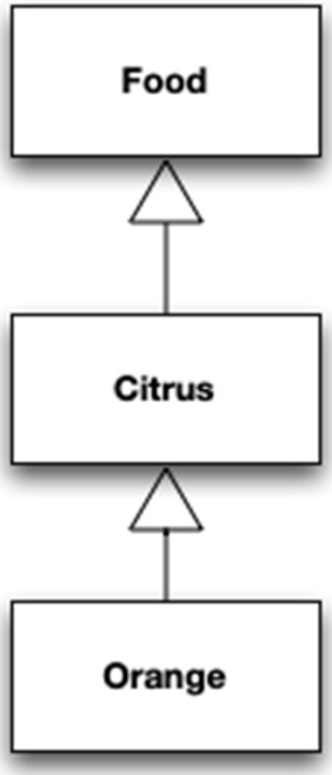
    2) Is the thing from which I'm inheriting part of me?
- A "yes" to 1) and "no" to 2) implies the need for inheritance
- A "no" to 1) and a "yes" to 2) implies the need for delegation
  - Recall Hippo/List example
- Example
  - Is an airplane a special type of fuselage? No
  - Is a fuselage part of an airplane? Yes

Multiple Inheritance

- A third heuristic
  - Whenever you have found a multiple inheritance relationship in an object-oriented design, be sure that no base class is actually a derived class of another base class
- Otherwise you have what Riel calls **accidental multiple inheritance**
  - Consider the classes “Citrus”, “Food”, and “Orange”; you can have Orange multiply inherit from both Citrus and Food…but Citrus IS-A Food, and so the proper hierarchy can be achieved with single inheritance

Example

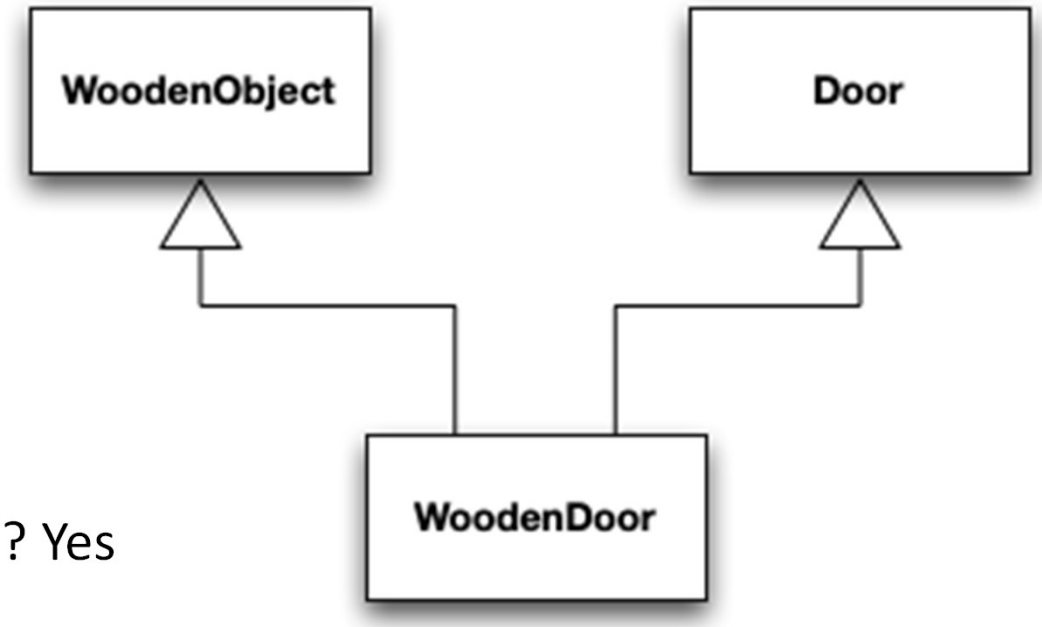




Multiple Inheritance

- So, **is** there a valid use of multiple inheritance?
  - **Yes**, sub-typing for combination
    - It is used to define a new class that is
      - a special type of two other classes
      - and where those two base classes are from different domains
  - In such cases, the derived class can then legally combine data and behavior from the two different base classes in a way that makes semantic sense

Multiple Inheritance Example



Is a wooden door a special type of door? Yes
Is a door part of a wooden door? No
Is a wooden door a special type of wooden object? Yes
Is a wooden object part of a door? No
Is a wooden object a special type of door? No
Is a door a special type of wooden object? No
All Heuristics Pass!

The Midterm Exam

- The first lecture mentions some October dates for the midterm, this is a cut-paste error. The exam is **March 4.**
- All work on this exam must be your own and only your own
  - Remember, discussing or providing exam content to others that have not taken the exam is an honor code violation for both parties
- **In-class on March 4**, 100 points, 20% of final grade
  - Distance students can come in to take the exam, or arrange for a proctor **prior to the exam**
  - Distance students have from March 4 at 11 AM to March 11 at 11 AM to take and turn in their proctored exam as a PDF on Canvas
- If you are a campus student, and cannot attend on **March 4**, see me PRIOR to the exam for an alternate arrangement. There will be few excuses I would accept to re-take the exam if you miss it on **March 4**.
- **Open-note, open-text** – but because of tight time constraints, you'll probably want to prepare a summary sheet of important points from slides and the textbook readings
- **Time-limited 11:00 AM to 11:50 AM** – this will be challenging – you may want to consider where to spend time on the exam based on point values of the questions if time is tight for you
- **On paper**, I recommend using a pencil you can erase
- Expect definitions, problems, and UML diagrams – a mix of theory and design exercises, a mix of question types. I will only cover design patterns and other content we have covered in class up to the exam.
  - I have not finalized content yet
- I reserve the right to curve or not curve results based on my assessment of the results
- I will not have a review of exam content until all exams (including distance student's) are turned in

Graduate Presentation - Outline

- This will be a 10 point submission in addition to the 100 points for the presentation project
  - I will award a full score for addressing each topic listed below to a level that lets me understand how you're approaching the topic you selected
  - I may come back with questions or clarification requests after the outline review, that will not affect your grade if all sections below are addressed
- One outline submitted per team or individual; both members of a two-person team will get the same score
- PDF containing the following sections:
  - Topic name
  - Team or individuals
  - Outline of full presentation content
    - As you've envisioned it to date, high level – topics only
    - I know this may change as you research the topic – just want to see what you think now
  - Intended code example(s) content and language(s)/tool(s)/libraries to be used
  - Any target literature or web citations identified to date
  - Try to limit response to two written pages maximum
- Due Monday 2/25 11 AM (2 weeks)

Next Steps

- Optional additional material
  - Dr. Anderson's lecture on façades and adapters
    - You can find it on the class Canvas site under Media Gallery starting in lecture 7
    - Again, very similar material as I'm using versions of his slides…
- This week
  - Wednesday 2/13 – Recitation session with Manjunath (optional)
  - Friday 2/15 – Lecture: Expanding Horizons, Commonality & Variability Analysis; Design Patterns & Agile (Chapter 8 in Textbook)
    - Homework 3 will be assigned also
- Things that are due
  - Quiz 3 due before Wed 2/13 recitation
  - Homework 2 is due Fri 2/15 at 11 AM
  - Grad Presentation Outline is due Mon 2/25 at 11 AM
  - Class Semester Project topic e-mail is due Friday 3/1/19 11 AM
  - Distance students, consider how you'll take the mid-term on 3/4 – in-class or by proctor – and make arrangements accordingly