Facade & Adapter, Part 2

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 8A — 02/15/2019

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

- Complete two design patterns
  - Facade
  - Adapter
- Compare and contrast the two patterns
- Look at multiple inheritance
- Review some housekeeping (HW3, semester topic)

- Note: This is lecture 8A, a continuing version of lecture 8

The dreaded name-card based activity!

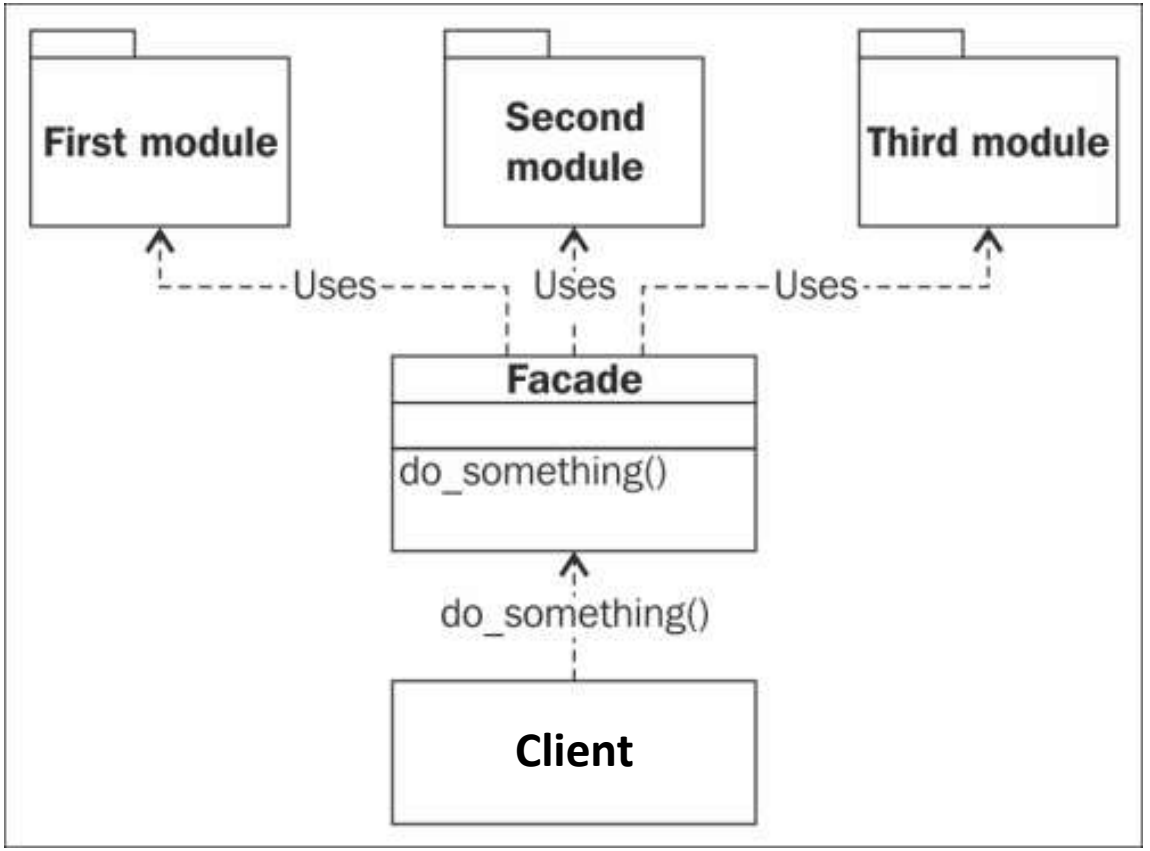
- I will offer a question
- You will raise your hand if you'd like to answer it
- I will recognize you (via your handy name-card)
- I will provide a question for you
- You will answer it so that the class can hear you
- The class will vote, via a show of hands, on the correctness of your answer
- If 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.

Included for your reference (if you wanted to see the questions)…

1. Describe cohesion and is it better strong or weak?
2. Describe coupling and is it better loose or tight?
3. What is the difference between aggregation and composition?
4. Which one is Is-A and which is Has-A; which should I favor in designs?
5. Give me a short (1-2 sentence description) of polymorphism
6. What does Design by Contract mean?
7. Public vs. Protected vs. Private accessibility?
8. Interface vs. Abstract Class?
9. You can do this – what's the WAVE rule for Use Cases?
10. On my Laptop, in the center, is a sticker that says "Don't Panic" – what is that saying from?

Façade Pattern = “Simpler”

- Provides a better and clearer API for the client code
- It maintains loose coupling between client and subsystems
- It provides an interface to a set of interfaces in a subsystem (without changing them)
- It wraps a complicated subsystem with a simpler interface
- Subsystem implementation gains flexibility and clients gain simplicity



From Learning Python Design Patterns
https://learning.oreilly.com/library/view/learning-python-design/9781783283378/ch04.html

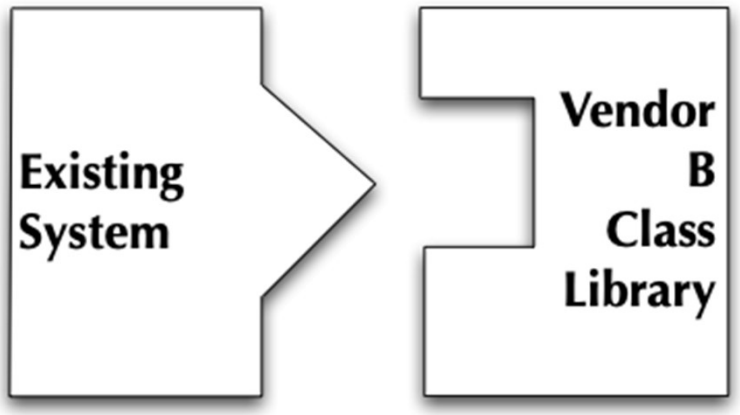
Principle of Least Knowledge

- 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

OO Adapters (I)

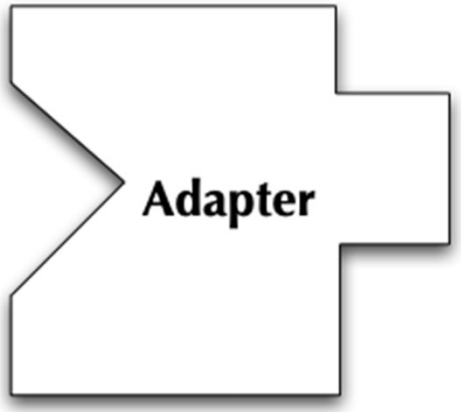
- Pre-Condition: You are maintaining an existing system that makes use of a third-party class library from vendor A
- Stimulus: Vendor A goes belly up and corporate policy does not allow you to make use of an unsupported class library.
- Response: Vendor B provides a similar class library but its interface 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)



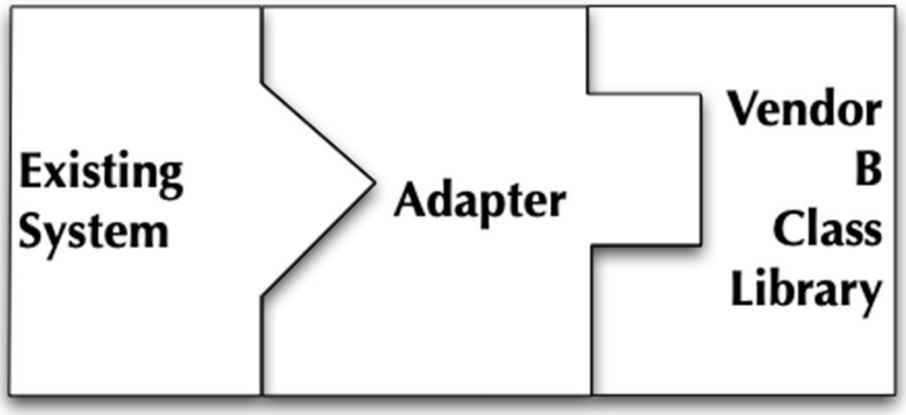
Interface Mismatch
Need Adapter

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

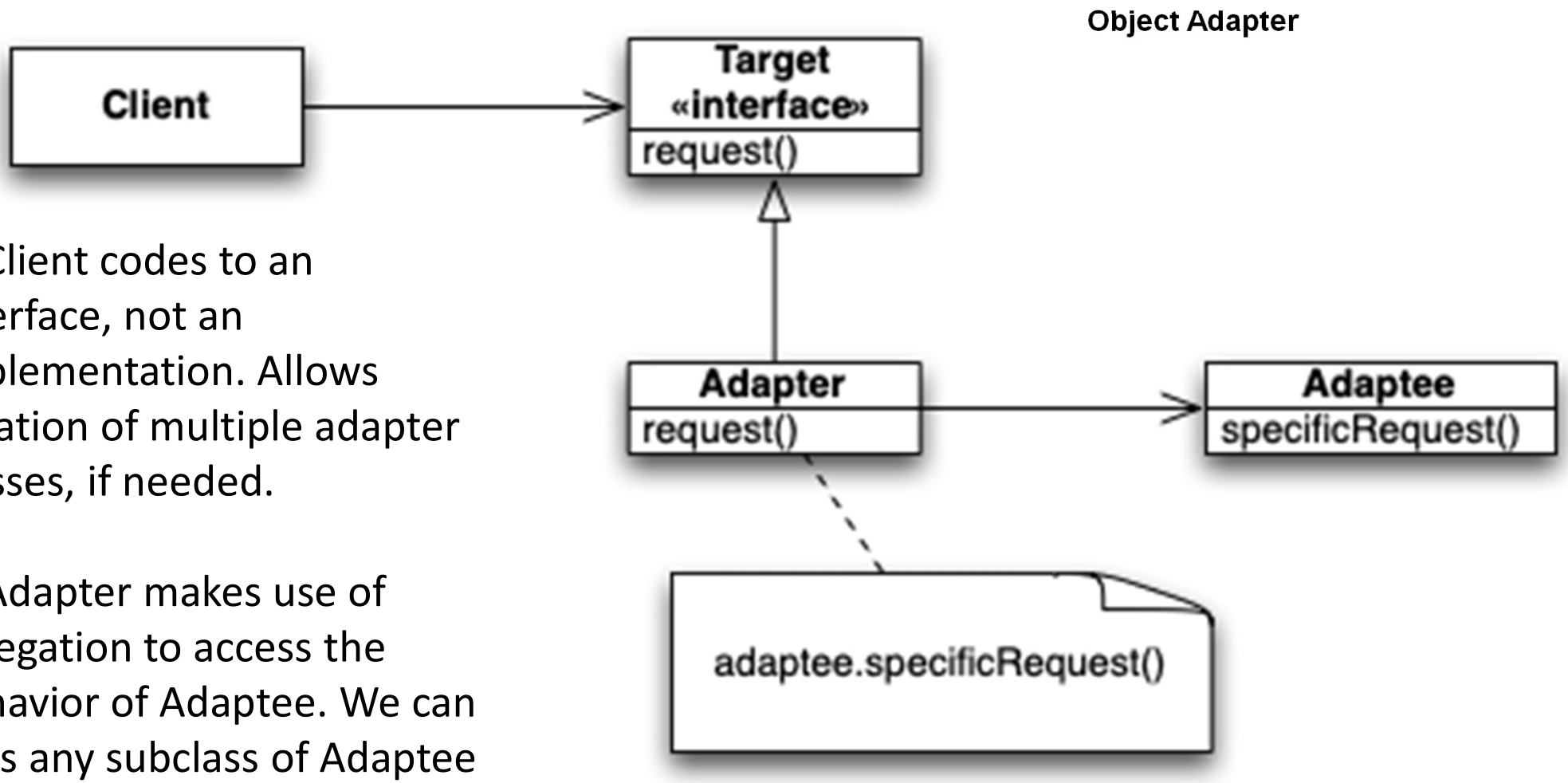
3. Calls by client code are delegated to the appropriate methods in the adaptee

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

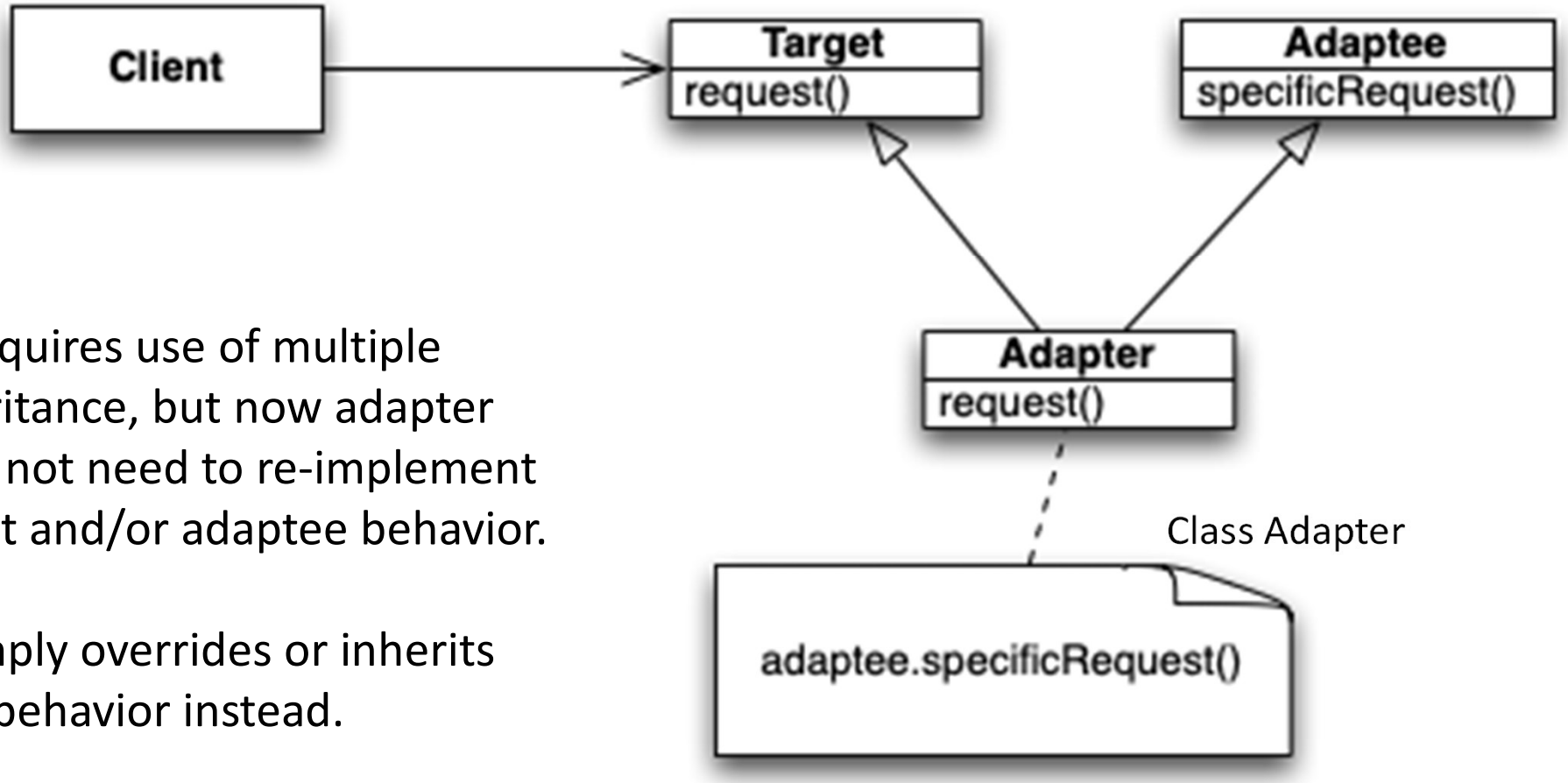
Adapter Pattern: Structure (I) via Interface



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) via multiple inheritance



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

- 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
- 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 (façade) vs convert (adapter)**

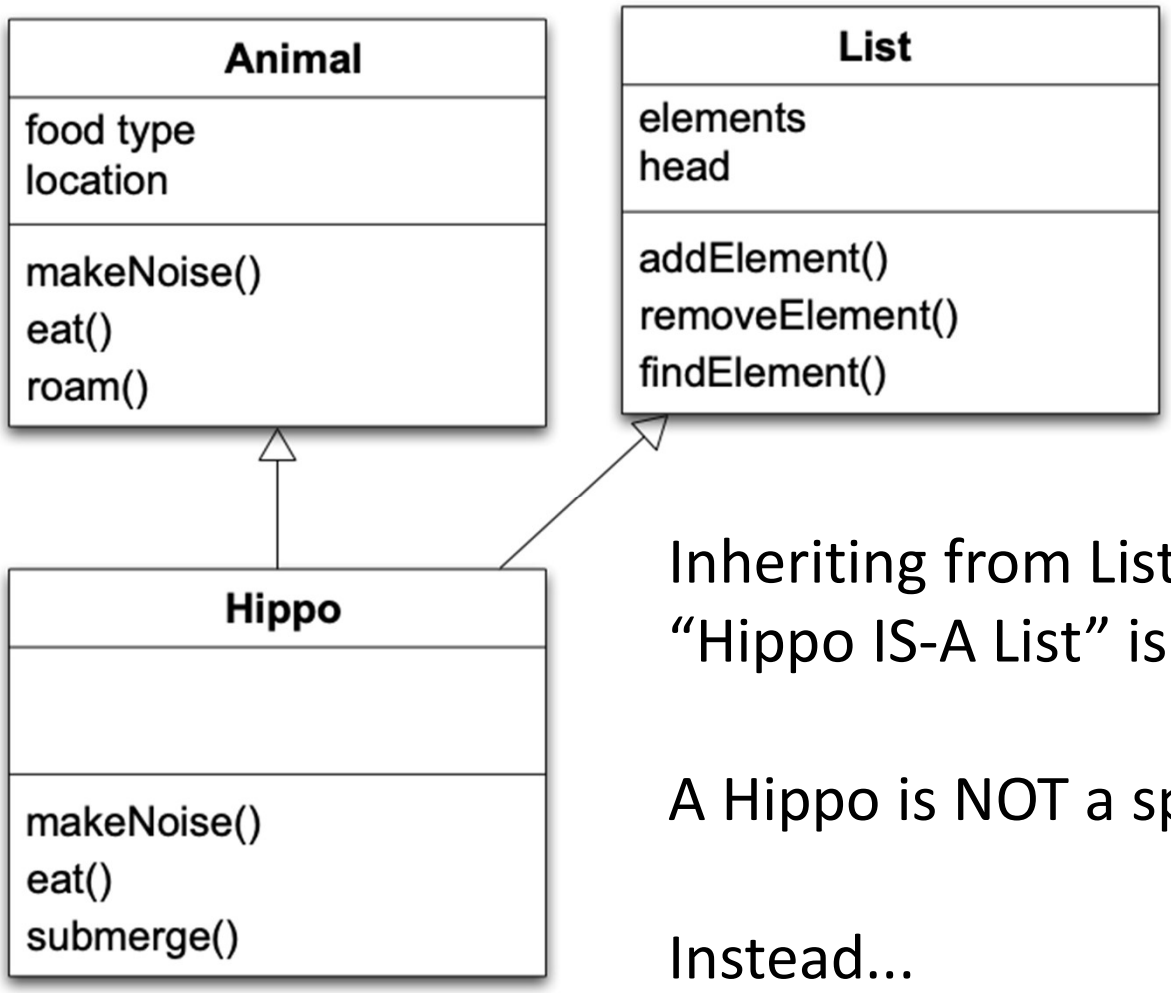
Multiple Inheritance (I)

- 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 (II)

- 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

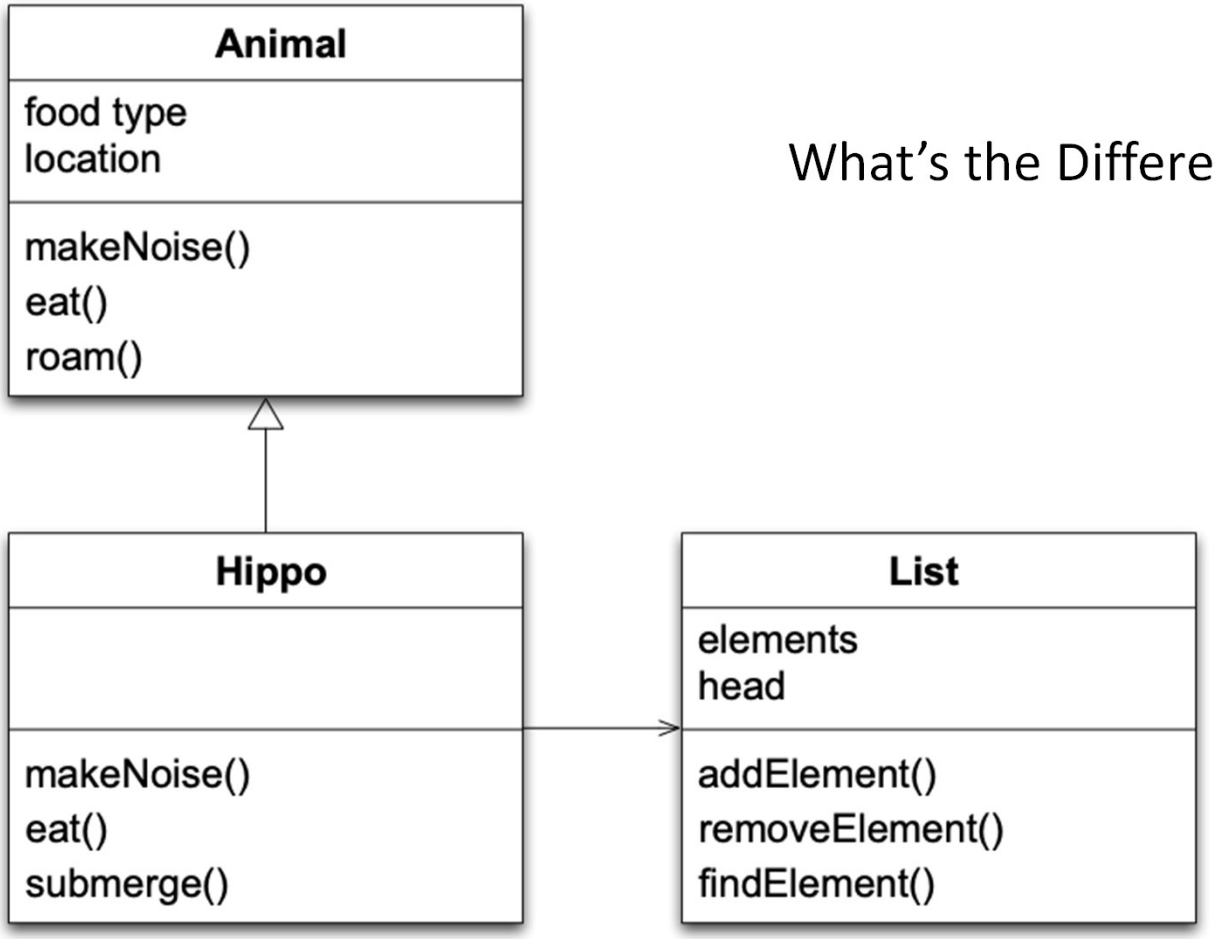


Inheriting from List in this way is bad, because "Hippo IS-A List" is FALSE

A Hippo is NOT a special type of List

Instead...

Do This



What's the Difference?

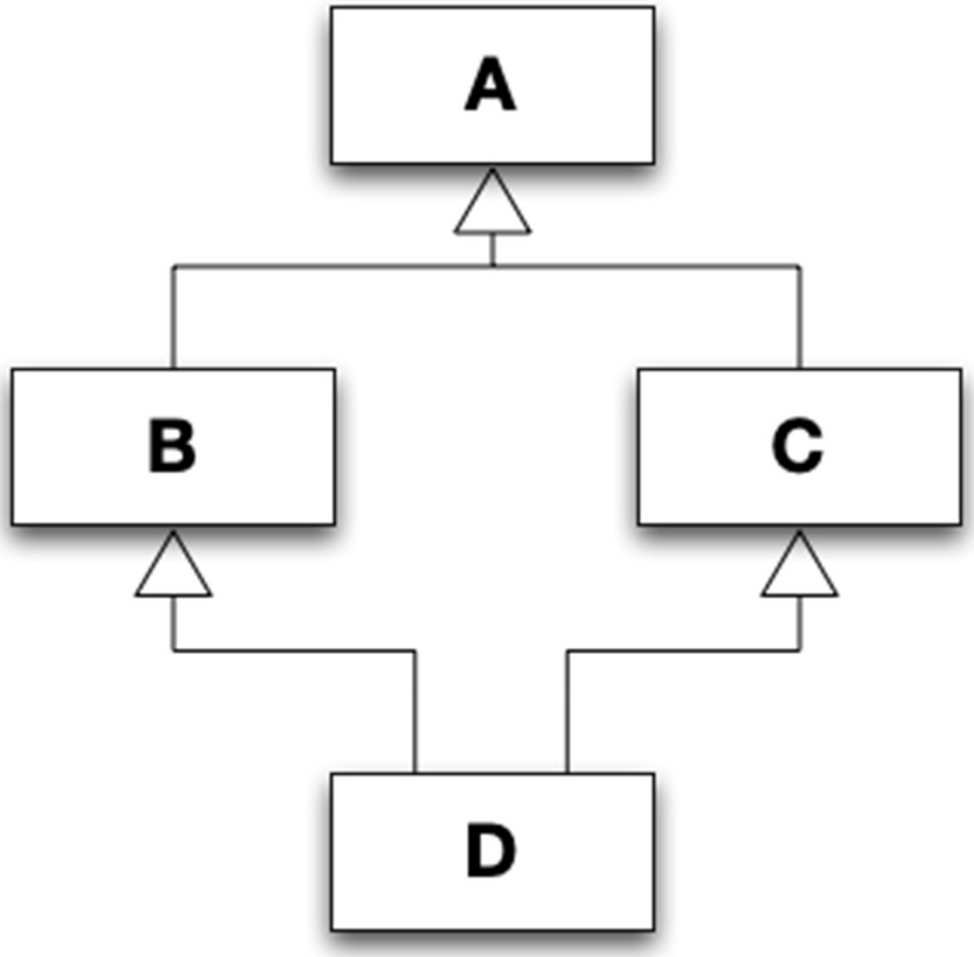
Another Problem

What's wrong with this?

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

This OO issue is known as the "Diamond Problem"

OO Languages handle this differently, some by limiting to single inheritance (a class can only derive from one base class).



Diamond Problem in Python 3

Program Output:
I am parent B1
I am parent B2

The ordering of base classes defines the order of search, for an implementation of method “call()”.

This ordering (along with what classes are inherited) can be changed at runtime using a method called “restructure”.

```
#    Python 3 adapted from http://www.aizac.info/
#    a-solution-to-the-diamond-problem-in-python/

class A:
  def call(self):
    pass

class B1(A):
  def call(self):
    print "I am parent B1"

class B2(A):
  def call(self):
    print "I am parent B2"

class ME1(B1, B2):
  def whichCall(self):
    print(self.call())

class ME2(B2, B1):
  def whichCall(self):
    print(self.call())

def main():
    m1 = ME1()
    m1.whichCall()
    m2 = ME2()
    m2.whichCall()

main()
```

Multiple Inheritance (III)

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

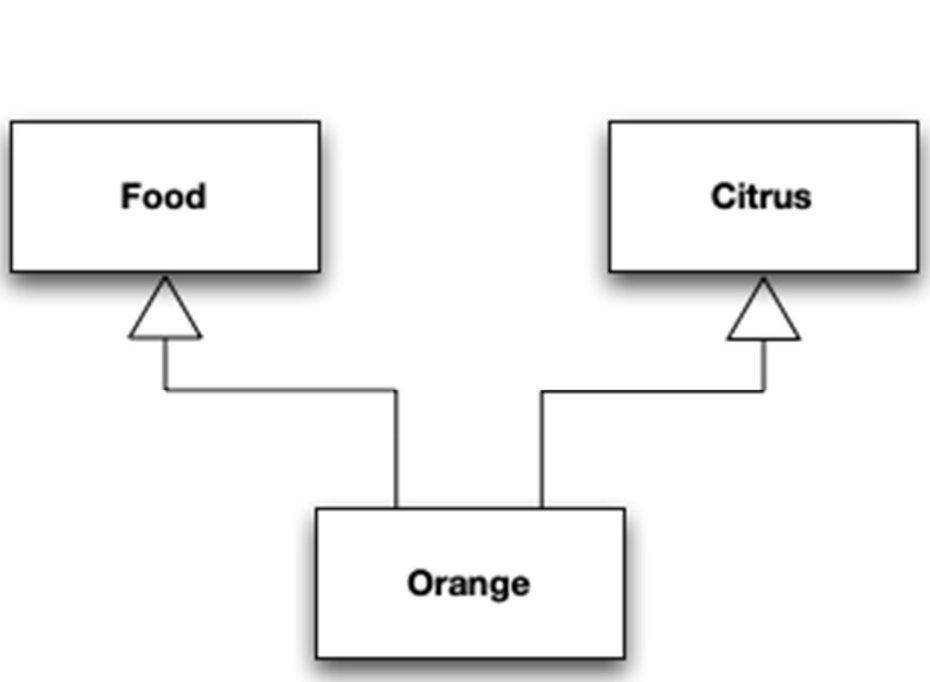
    1) Am I a special type of the thing from which I'm inheriting?

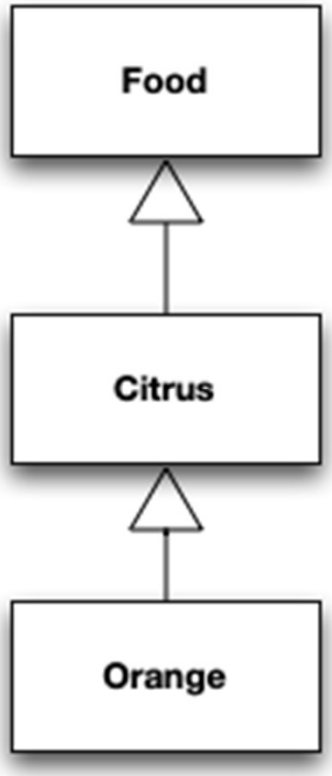
    2) Is the thing from which I'm inheriting part of me?
- A "yes" to 1) and "no" to 2) implies the need for inheritance
- A "no" to 1) and a "yes" to 2) implies the need for delegation
  - Recall Hippo/List example
- Example
  - Is an airplane a special type of fuselage? No
  - Is a fuselage part of an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

Multiple Inheritance (IV)

- 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

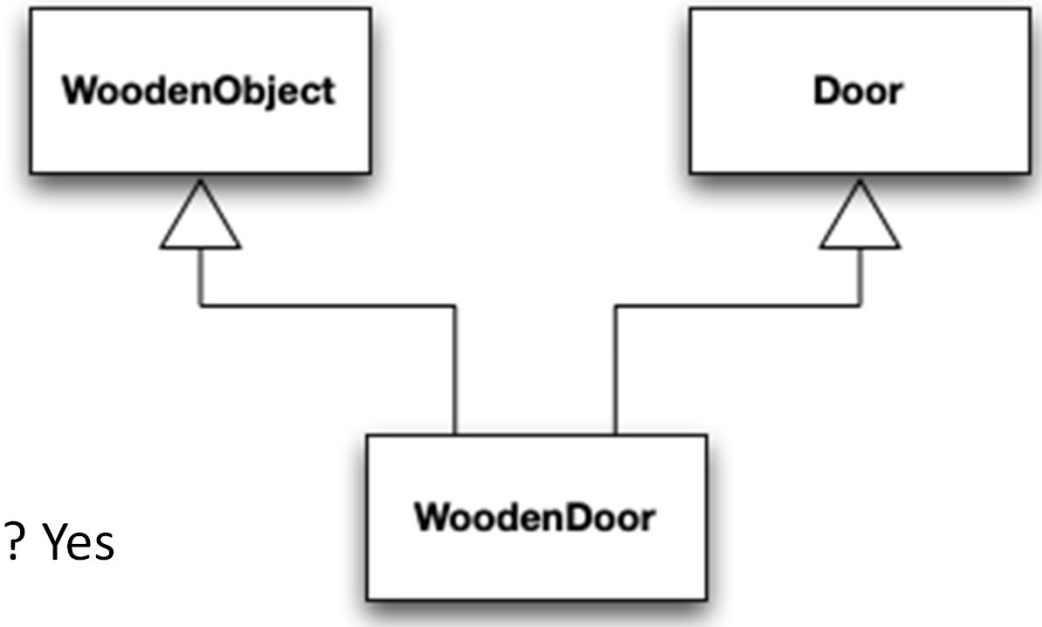




Multiple Inheritance (V)

- 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!

Summary

- Façade pattern = “Simpler”
- Adapter pattern = “Convert”
  - By interface or
  - By multiple inheritance
- Multiple inheritance heuristics
  1. It’s a mistake unless you prove otherwise
  2. The 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**
  3. Be sure that no base class is actually a derived class of another base class
- The Diamond Problem

Homework 3 – OO Programming Exercise (I)

Problem Domain: Hardware Rental Store

- For this homework, you will design an object-oriented program to model the following problem domain.
- A hardware rental store has a catalog of 20 different tools to rent, spread across 5 different categories (Painting, Concrete, Plumbing, Woodwork, Yardwork). Each tool has a unique name (e.g. “Paint Tool 1”) and belongs to a specific category; the price per day to rent a tool varies by category. You may decide on the pricing of the rental categories.
- Customers are allowed to rent a tool for up to 7 nights. (Thus a tool rented for 7 nights on a Monday must be returned by the following Monday. A tool rented for "one day" would, for example, be rented on a Tuesday and returned the next morning before the rental store opens for business on Wednesday.) Customers are allowed to have at most three tools rented at any one time.
- This store has 10 customers; each customer has a unique name and is associated with one of three types. **Casual** customers rent one or two tools for one or two nights. **Business** customers always rent three tools for seven nights. **Regular** customers will rent one to three tools each time they visit for 3 to 5 nights.
- Each time a customer comes into the store, a Rental is created that will keep track of what tools they rented and how many nights they will keep the tools. A customer rents the "group" of tools and returns them all at the same time. They will NOT, for example, rent three tools and then return one after 2 days, the second after 5 days, and the third after seven days. They will instead return all of the tools they rented at the specified time. (That means, for instance, that a customer will never be late in returning their tools.)
- The store keeps track of the existing rentals along with the current inventory of the store. As such, when it has zero rentals, there will be 20 tools in its inventory. When it has zero tools in its inventory, it will have multiple rentals that between them account for all 20 tools.
- Finally, a customer pays up front for their rental. If, for example, a customer rents three Paint tools for three nights at a price of $3 per night, they will pay the store $27 dollars before they leave the store with their three tools.

Homework 3 – OO Programming Exercise (II)

Assignment

Write an object-oriented program that implements the problem domain and does the following:

- Simulates the activity of the rental store for 35 days (34 nights). On each day, a random number of customers will visit the store as long as there are tools to rent. Each customer will create one Rental that follows the rules of their associated type before they leave the store. That is, no customer will show up and then leave without making a rental. Note: if the store has less than 3 tools, then a Business customer will NOT arrive (as they wouldn't be able to create a Rental that follows their rules). As soon as the store has zero tools, customers will magically stop arriving until tools are once again available.
- At the end of the simulation, the program will produce a report that includes the following information:
  - the number of tools currently in the store along with a list of their names
  - the amount of money the store made during the 35 days (including any rentals that occurred on the 35th day)
  - a list of all the completed rentals including which tools were rented by which customer for how many days along with the total amount of that rental
  - a list of all the active rentals that includes all of the information listed in the previous bullet
- A customer can have more than one active rental. That is, they can show up on day 1 and rent 1 tool for 5 nights. They can then show up on day 2 and rent another tool for 4 nights. As long as they do not have more than 3 tools rented, they are allowed to have multiple rentals.
- Returns occur at the beginning of the day before the store opens for business. A tool rented for one night is available to customers the very next day; that's because the customer rented the tools for one night, used it, and got it back to the store early the next morning.
- Your program should be single-threaded; you do not need to handle the case of multiple customers trying to rent tools concurrently.

Homework 3 – OO Programming Exercise (III)

- The purpose of this assignment is NOT to meet the requirements by any means necessary. A program that does the simulation above and produces the requested report but makes use of structured programming techniques (i.e. no objects, just data structures and a main program) will receive zero points (for the whole assignment).
- An object-oriented program that meets the requirements but doesn't make use of polymorphism, has poor abstractions, and poor encapsulation will lose many of the 30 points allocated to the program.
- Only object-oriented programs that show good use of abstraction, encapsulation and polymorphism and meet the above requirements will be able to get full credit for the program.
- Note in your documentation if any OO patterns are applied.
- 50 Point Submission is in two parts:
  - 30 Points - GitHub URL with
    - running, commented OO code (language of your choice) and
    - a README Markdown file with the names of team members and any special instructions to run the code (graders may request demonstrations)
  - 20 Points - PDF containing
    - Names of team members
    - Text description of program design
    - UML Class diagram that shows classes and relationships from your design (include data attributes and methods in the class diagram)
- Due Friday 3/1 11 AM (2 weeks)

Semester Project Topic

- The Semester Project – A Demonstrated OOAD Effort
  - Create a non-trivial set of classes and services that make use of design patterns and provide a nice set of functionality to the end-user
  - A Native Mobile Application
  - A Responsive Web Application for Mobile Use
  - A Web Service Paired with a Mobile Client
  - A Cloud-based Application (using Student accounts for AWS, Google, Azure) with a Web or Mobile Interface
    - Think ahead on this – getting the cloud account approved can take a little time
  - Other Client/Server Applications
  - Another thought - An IoT style device or devices – I have SBCs (Raspberry Pi) if you need them!
- ~~Project idea and language/toolset will be submitted to me by e-mail, due for approval before Friday 3/1/19!~~
- Submit a PDF on Canvas – Ungraded, but required
  - Title of Project
  - Members of Team
  - Language/Toolsets for Code
  - A Paragraph (just a few sentences) on What This Project Idea Is
  - No more than a page!
  - I will review the projects, and may ask questions or request revisions
- Due Friday 3/1 11 AM (2 weeks) – One submission per team!

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…
- Next week
  - Monday 2/18 – Lecture: Expanding Horizons, Commonality & Variability Analysis; Design Patterns & Agile (Chapter 8 in Textbook)
  - Wednesday 2/20 – Recitation w/Manjunath (optional)
  - Friday 2/22 – Lecture: Strategy, Bridge, Abstract Factory Patterns (Ch 9, 10, 11 in Textbook) – last patterns before midterm
- Coming soon – Java, Python OO Reviews
- Things that are due
  - Quiz 4 due before Wed 2/20 recitation, will be up Fri/Sat
  - Grad Presentation Outline is due Mon 2/25 at 11 AM
  - Class Semester Project topic ~~e-mail~~ Canvas submission is due Friday 3/1 11 AM
  - Homework 3 is due Friday 3/1 11 AM
  - Distance students, consider how you'll take the mid-term on 3/4 – in-class or by proctor – and make arrangements accordingly