Patterns Review

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 10B — 03/01/19

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

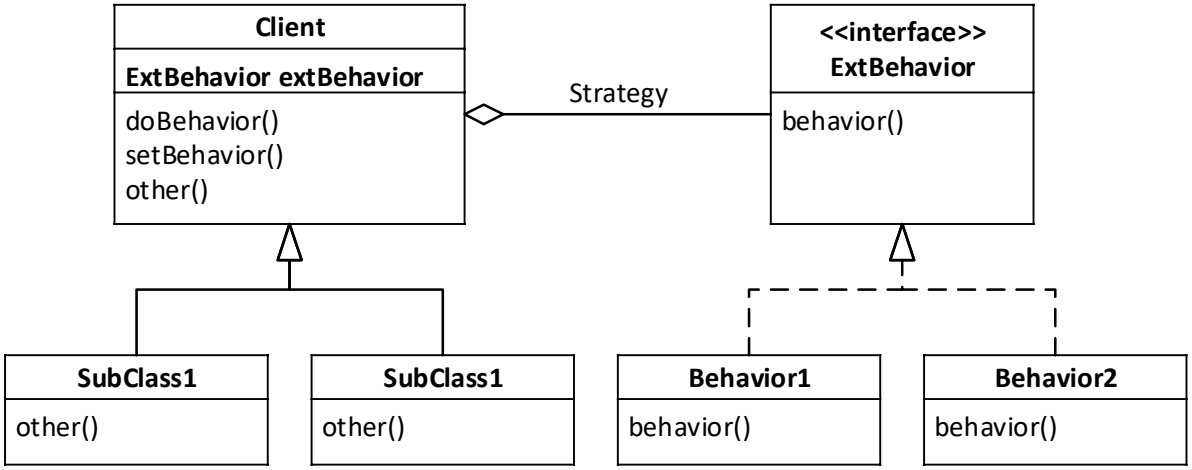
- Dr. Ken Anderson is a Professor of the Department of Computer Science and the Associate Dean for Education for the College of Engineering & Applied Science
- Ken taught this OOAD class on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- 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

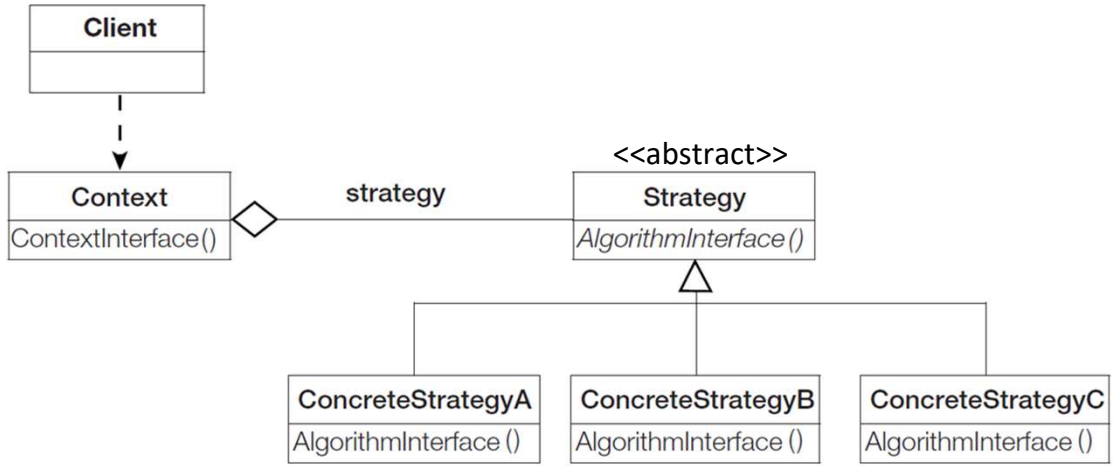
Goals of the Lecture

- Walk through the patterns one more time
  - Strategy
  - Façade
  - Adapter
  - Bridge
  - Simple Factory (not really one of the named patterns…)
  - Factory
  - Abstract Factory
- Present Homework 4

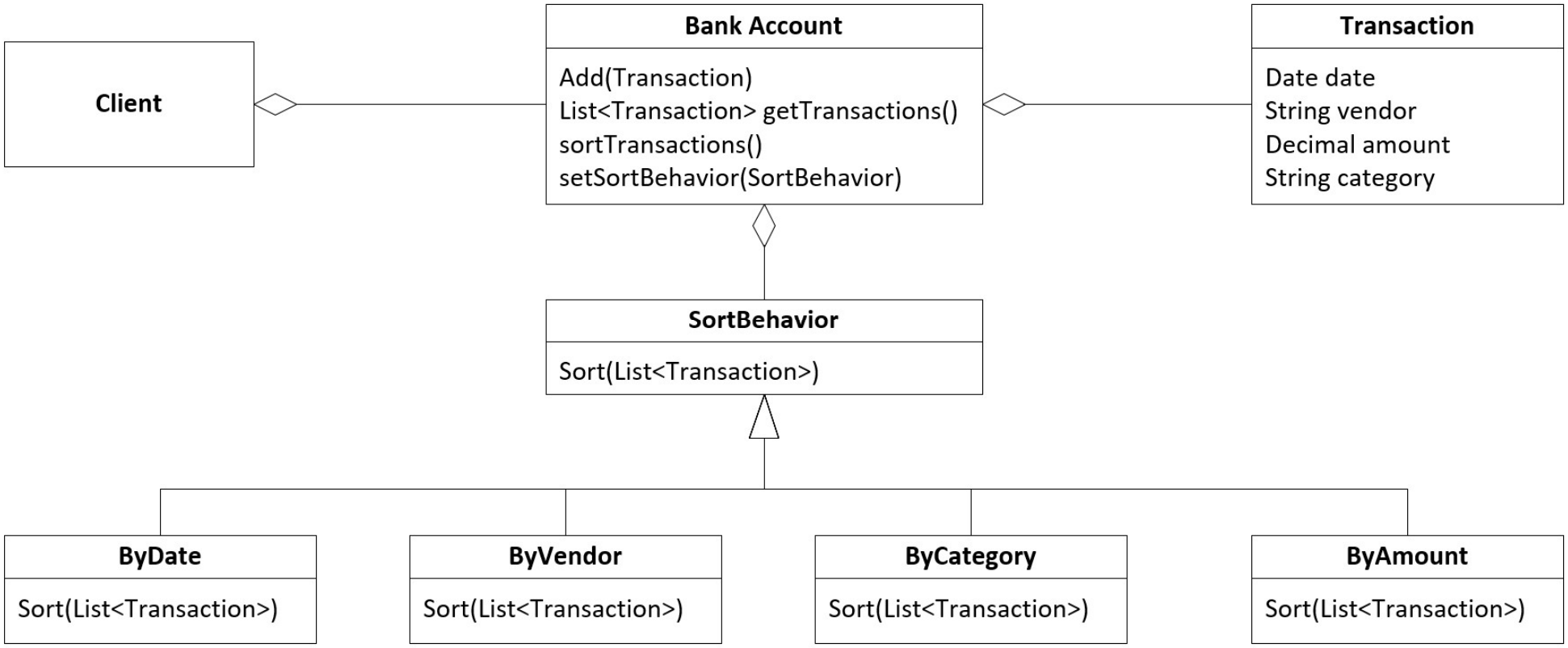
Strategy

- Intent: Allow use of varying rules or algorithms independent from clients
- Problem: The algorithm selected depends on the client making the request
- Solution: Separate the selection of the algorithm from the implementation of the algorithm
  - Can be implemented with an interface or an abstract class
- Result: The pattern defines a family of algorithms, all invoked the same way
- OO Principles:
  - **Favor composition over inheritance**
  - Program to interfaces not implementations
  - Encapsulate what varies



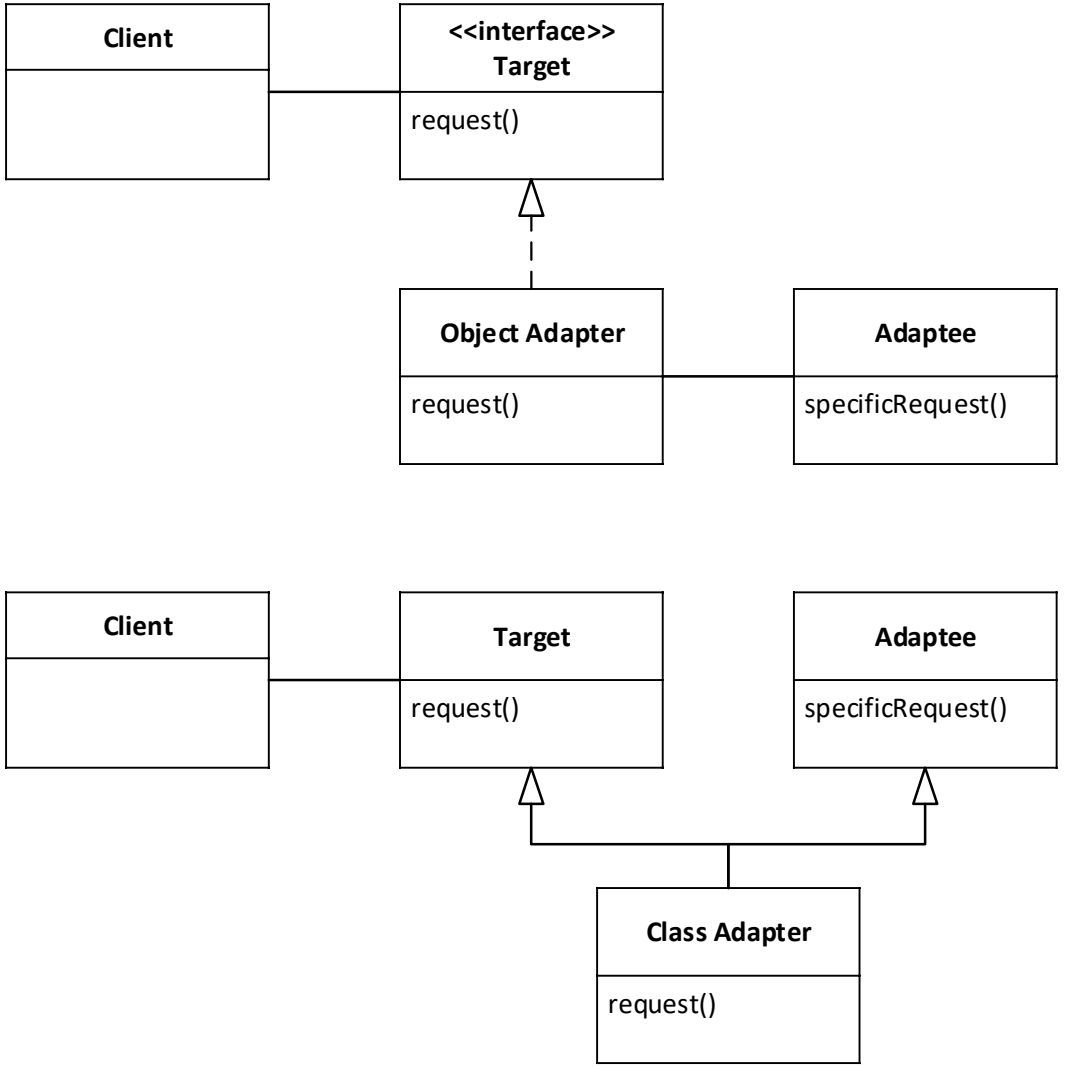


Strategy Example



Adapter

- Intent: Match an external object to an existing interface
- Problem: External object has right data and behavior but the wrong interface
- Solution: Wrap the external interface
- Two standard types of Adapter
  - Object Adapter – Interface based
  - Class Adapter – Multiple Inheritance based
- Result: Adapted external interfaces seem to act as an expected interface to the client
- OO Principles:
  - **Principle of Least Knowledge (talk only to your immediate friends)**
  - Program to interfaces not implementations
  - Favor composition over inheritance
  - Depend on abstractions, not concrete classes (Dependency Inversion Principle)



Example: A turkey amongst ducks!

- 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
```

1. Adapter implements target interface (Duck).

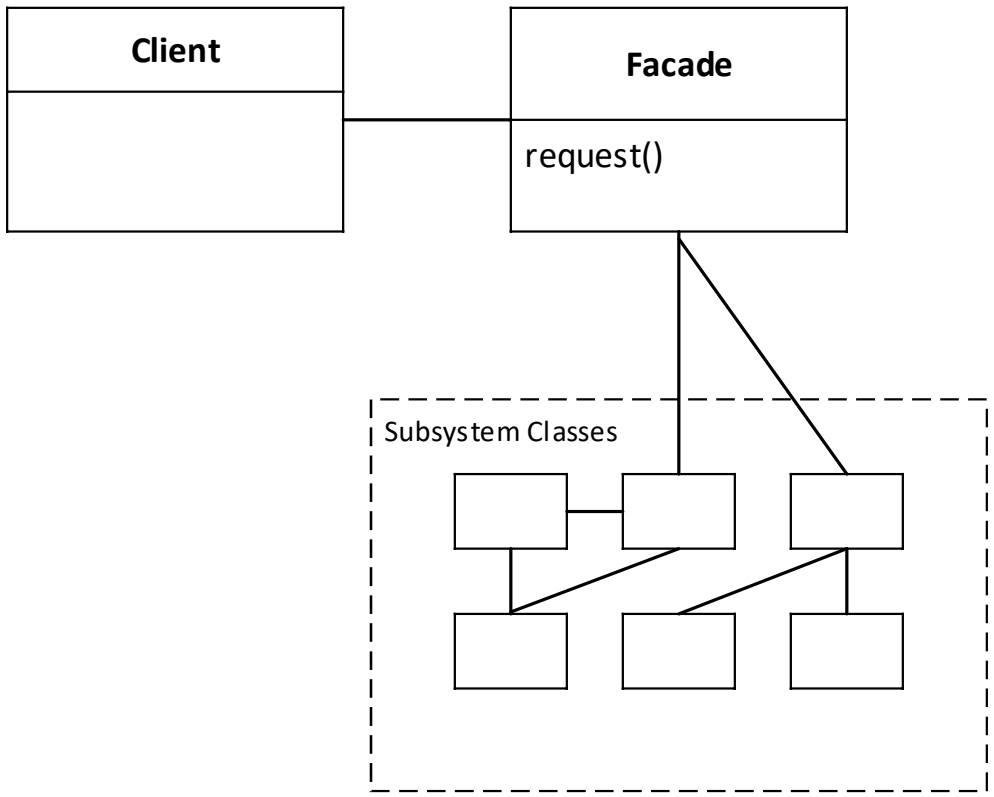
2. Adaptee (turkey) is passed via constructor and stored internally

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

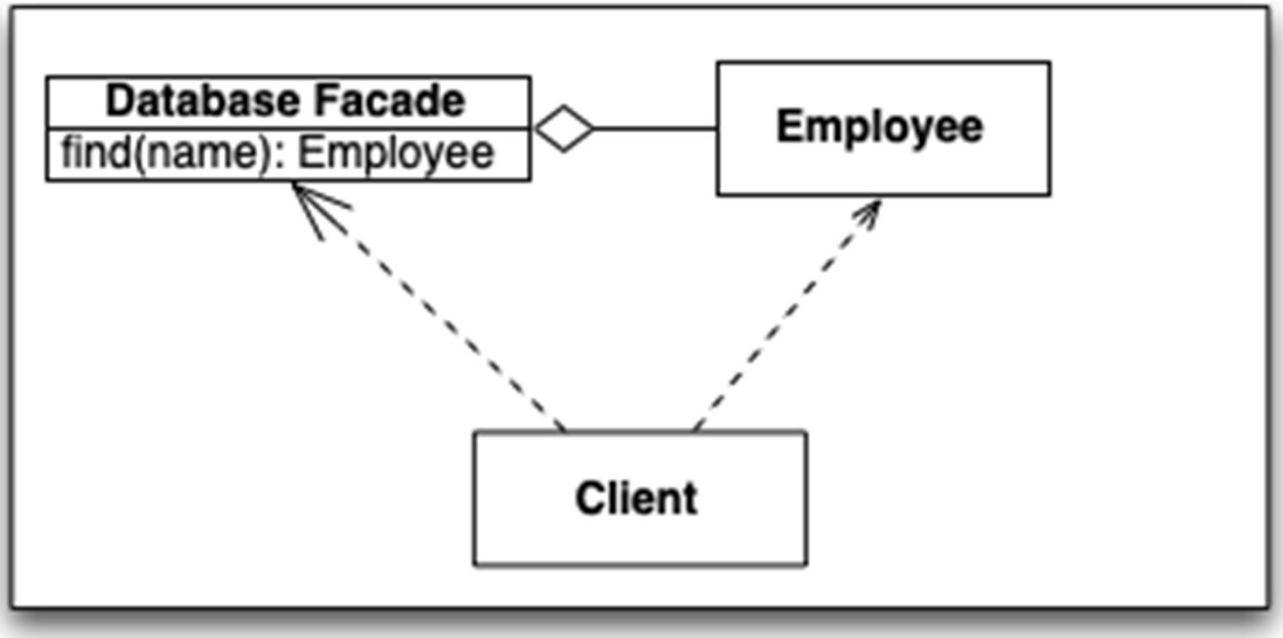
4. Adapter is full-fledged class, could contain additional vars and methods to get its job done; can be used polymorphically as a Duck

Facade

- Intent: Provide a simple interface to an existing complex subsystem of classes for specific functionality
- Problem: You only need a subset of that subsystem's functionality, or you want to perform a specific task
- Solution: Create a façade interface that provides a simplified method to access the desired functionality
- Result: The simplified interface presented to the client is significantly easier to use
- OO Principles:
  - **Principle of Least Knowledge (talk only to your immediate friends)**
  - Program to interfaces not implementations
  - Favor composition over inheritance
  - Depend on abstractions, not concrete classes (Dependency Inversion Principle)



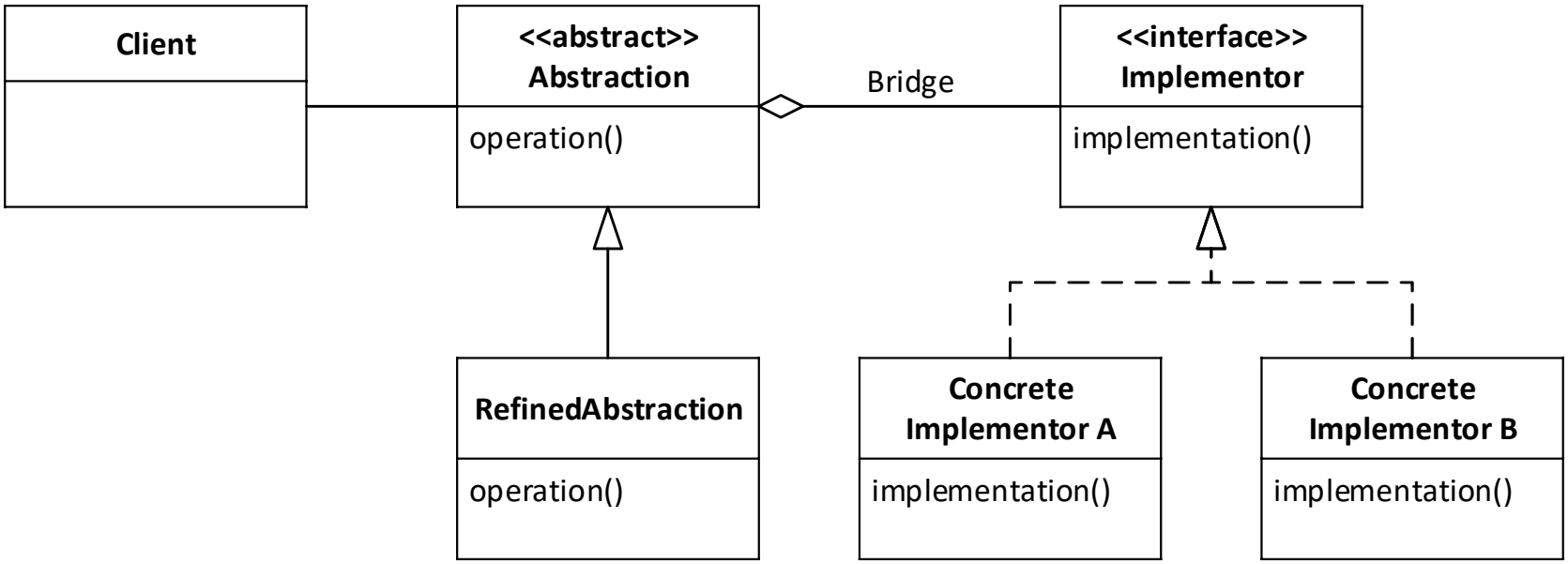
Example (With a Facade)

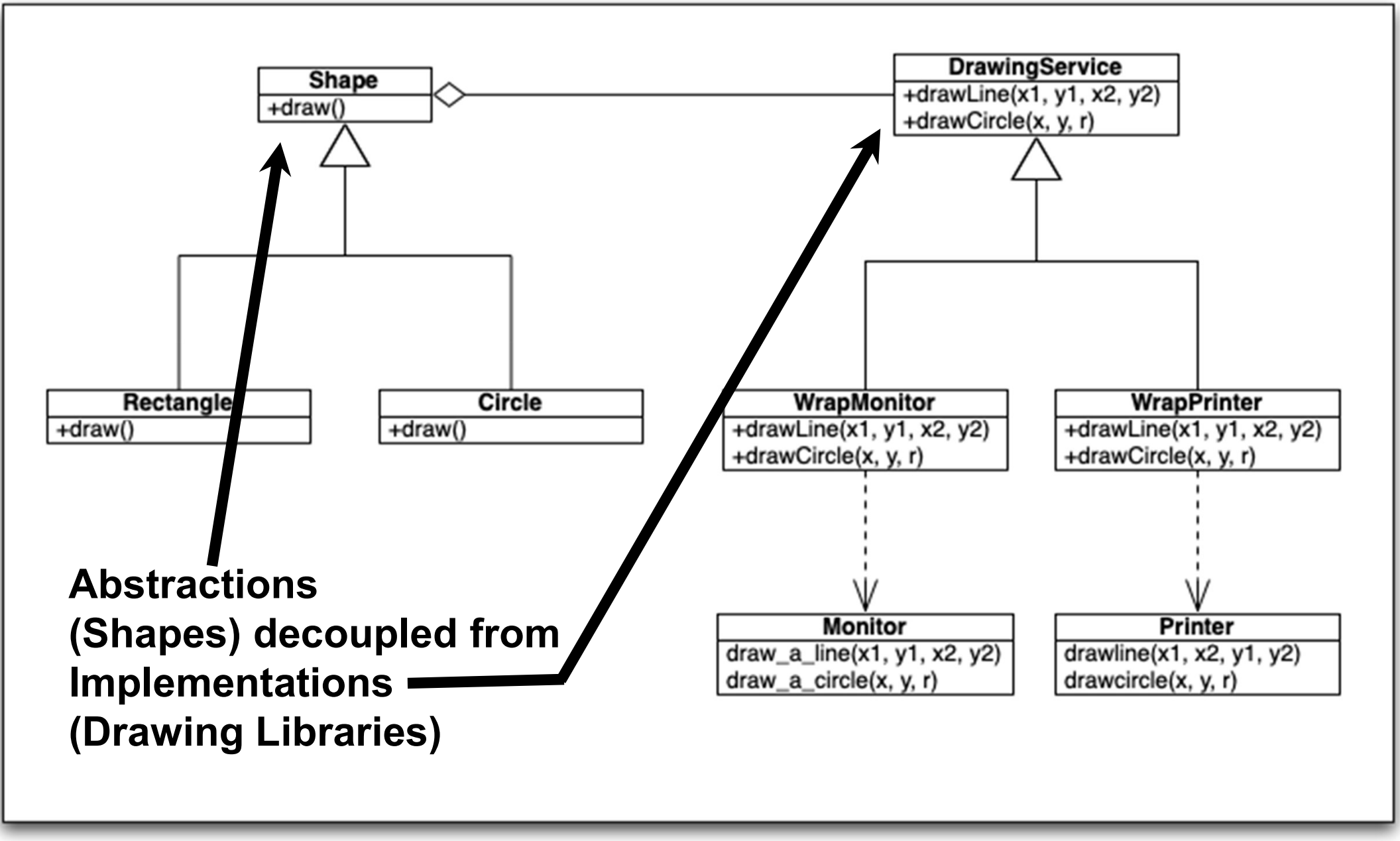


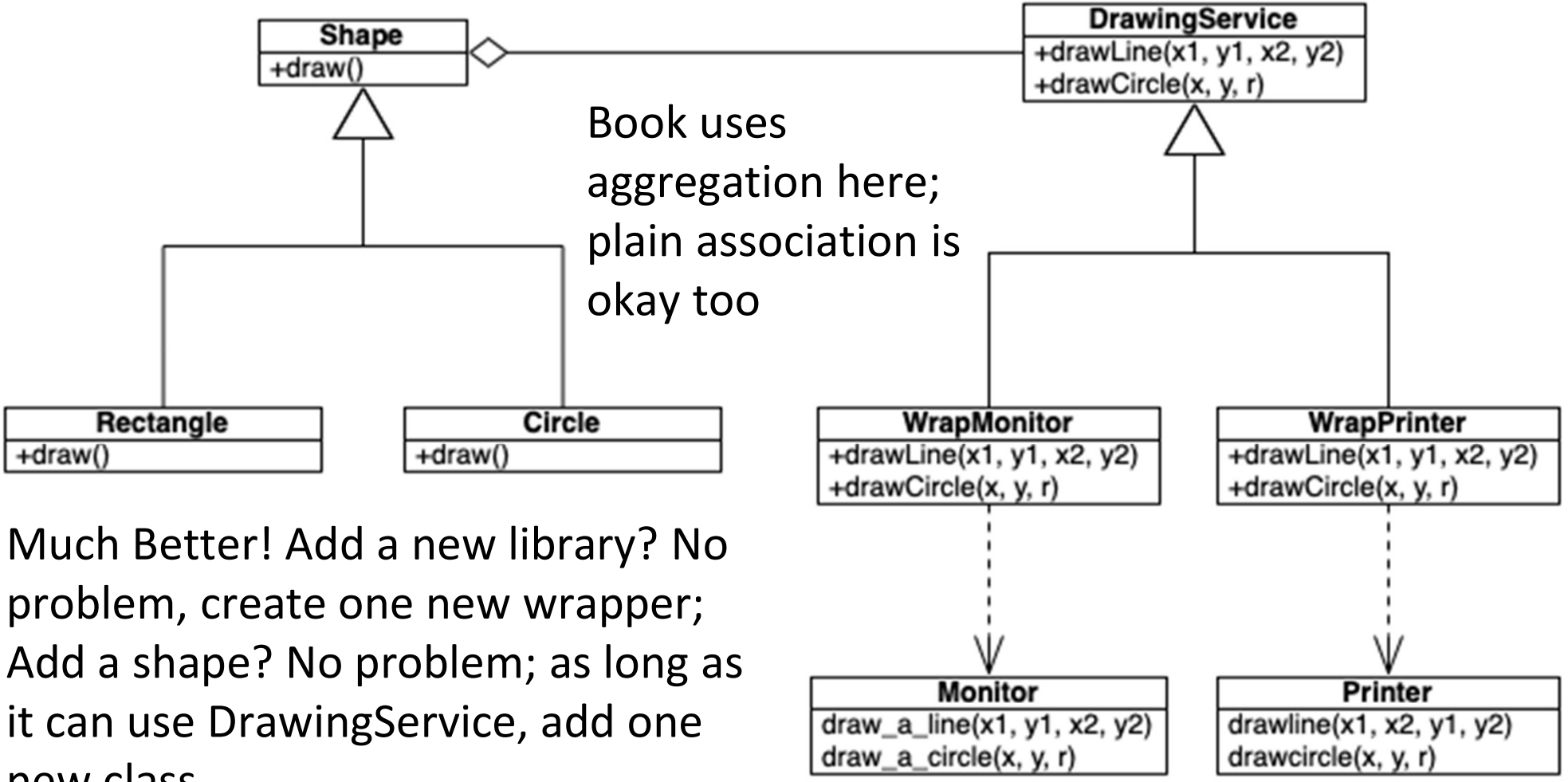
With a Facade, the Client is shielded from most of the classes. It uses the Database Facade to retrieve Employee objects directly.

Bridge

- Intent: Vary both abstractions and implementations
- Problem: Potential for varying both the abstracted operations and their implementations
- Solution: Decouple implementations from the interface
- Result: Both abstractions and implementations can be extended independently without impacting the client, allowing for a scalable system
- OO Principles:
  - **Favor composition over inheritance**
  - Find what varies and encapsulate it, program to interfaces not implementations




Shape
+draw()
DrawingService
+drawLine(x1, y1, x2, y2)
+drawCircle(x, y, r)
Rectangle
+draw()
Circle
+draw()
WrapMonitor
+drawLine(x1, y1, x2, y2)
+drawCircle(x, y, r)
WrapPrinter
+drawLine(x1, y1, x2, y2)
+drawCircle(x, y, r)
Monitor
draw_a_line(x1, y1, x2, y2)
draw_a_circle(x, y, r)
Printer
drawline(x1, x2, y1, y2)
drawcircle(x, y, r)
Abstractions
(Shapes) decoupled from
Implementations
(Drawing Libraries)



Much Better! Add a new library? No problem, create one new wrapper; Add a shape? No problem; as long as it can use DrawingService, add one new class

Java code examples for the three steps through the Bridge pattern are on Canvas under Files, Class Lectures, 10-Code Examples, Bridge

Factory Patterns: The Problem With "New"

- Each time we use the "new" command, **we break encapsulation of type**
  - Duck duck = new DecoyDuck();
- Even though our variable uses an "interface", this code depends on "DecoyDuck"
- In addition, if you have code that instantiates a particular subtype based on the current state of the program, then the code depends on each concrete class

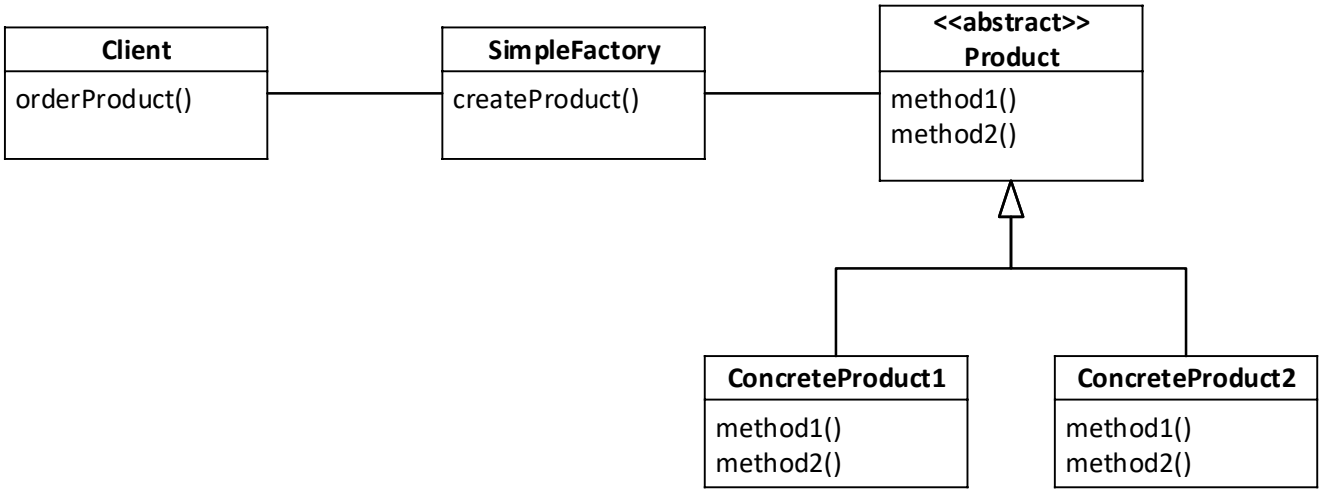
```
if (hunting) {
    return new DecoyDuck();
} else {
    return new RubberDuck();
}
```

Obvious Problems:

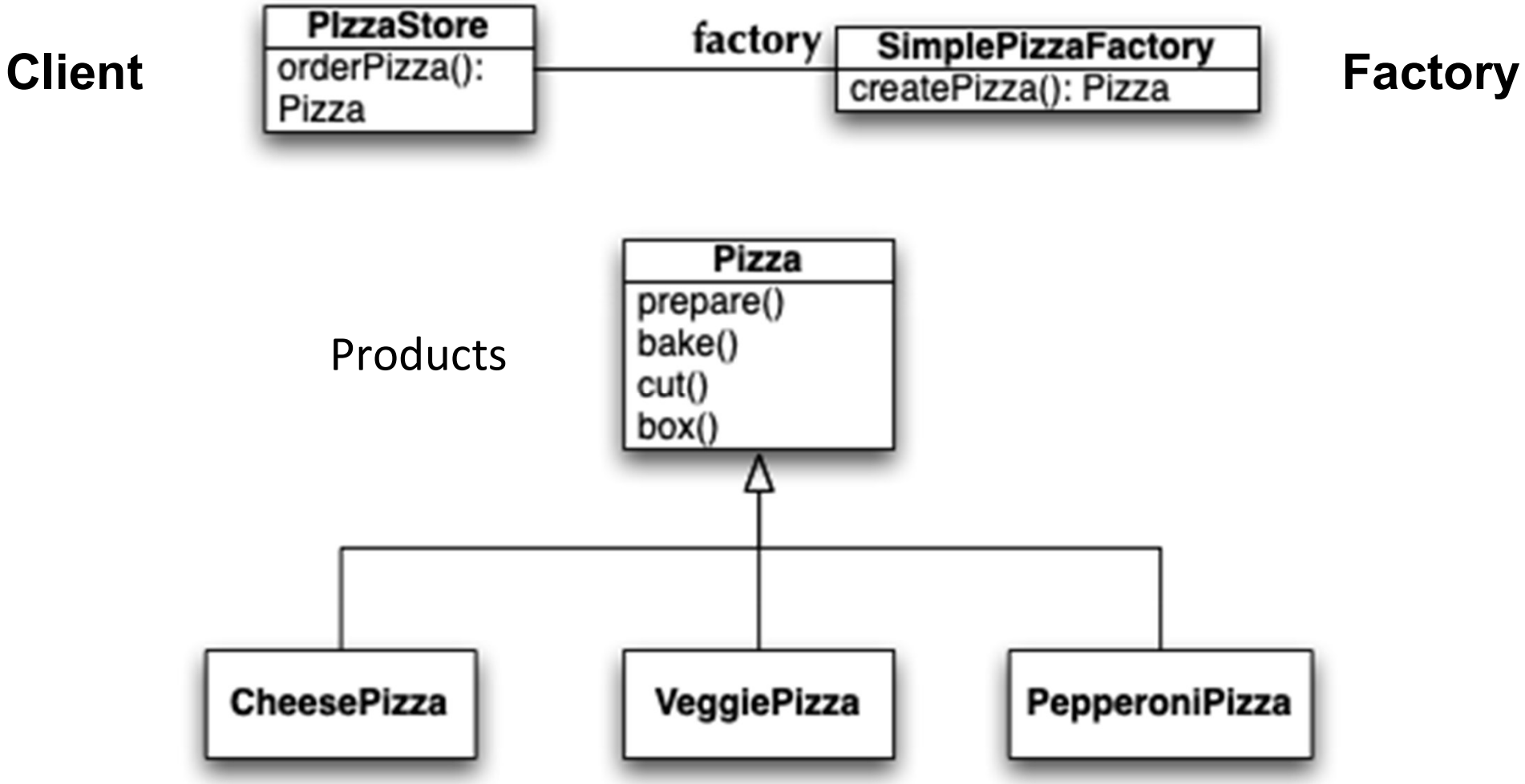
- **needs to be recompiled each time a dependency changes**
- **add new classes, change this code**
- **remove existing classes, change this code**

Simple Factory (not a high-level pattern)

- Intent: Avoid concrete instantiations that break encapsulation of type, avoids directly using new for concrete subclasses
- Problem: Dependency issues, adding or removing subclasses may cause rippling changes
- Solution: The SimpleFactory becomes the only part of the application that refers to concrete subclasses
- Result: The client is loosely coupled to the factory to get instances of the product subclasses
- OO Principles:
  - **Dependency Inversion Principle - Depend upon abstractions. Do not depend upon concrete classes.**
  - Encapsulate what varies
  - Favor composition over inheritance
  - Classes open for extension but closed for modification



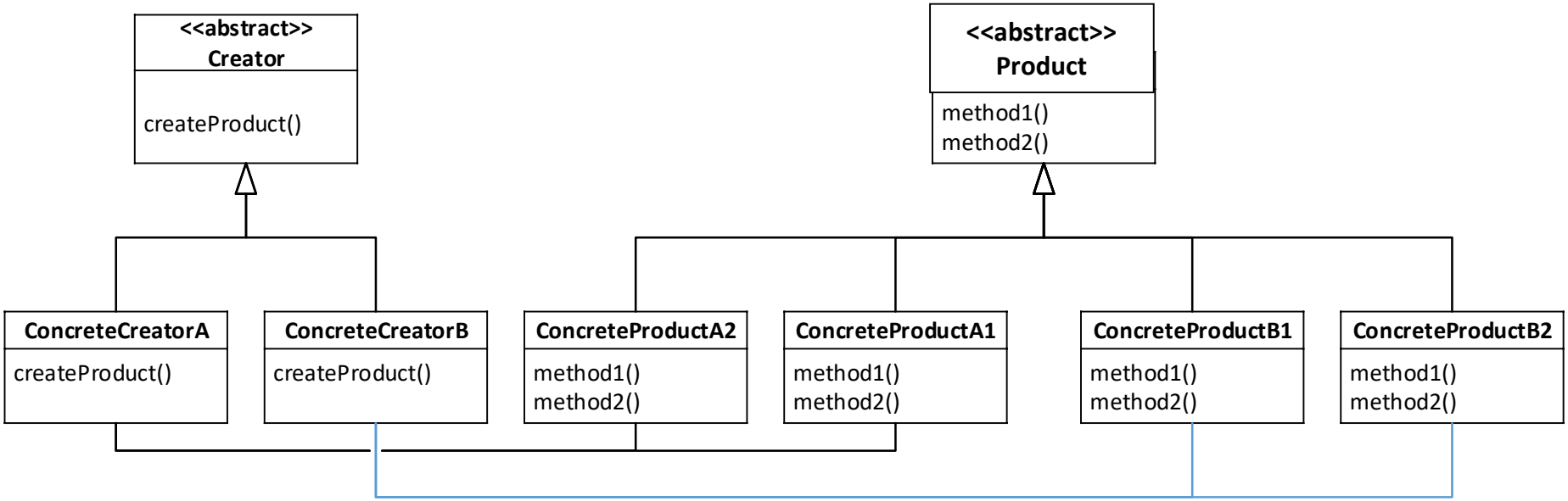
Class Diagram of New Solution (Simple Factory)



While this is nice, its not as flexible as it can be: to increase flexibility we need to look at two design patterns: Factory Method and Abstract Factory

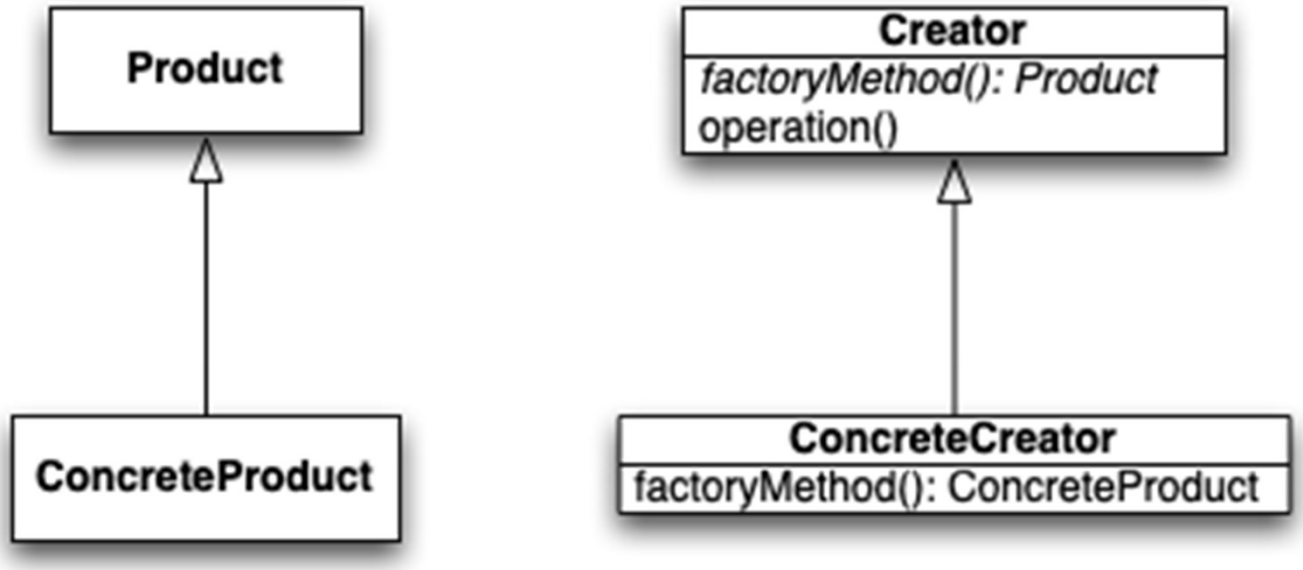
Factory Method

- Intent: Allow a class to defer instantiation to its subclasses
- Problem: Varying behaviors/implementations for both creating subclasses and created product subclasses
- Solution: Parallel class hierarchies that push implementations to subclasses; object creation is delegated to subclasses, which implement the factory method to create objects; relies on inheritance
- Result: Abstract “code to interface” is in superclasses, object creation is in subclasses
- OO Principles:
  - **Dependency Inversion Principle - Depend upon abstractions. Do not depend upon concrete classes.**
  - Encapsulate what varies
  - Favor composition over inheritance
  - Classes open for extension but closed for modification



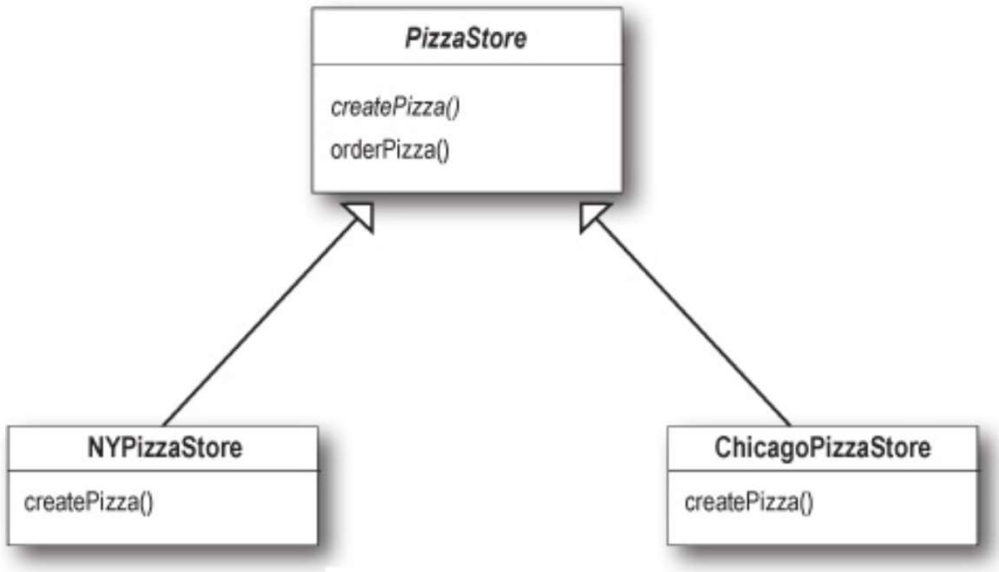
Factory Method: Definition and Structure

- The factory method design pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses

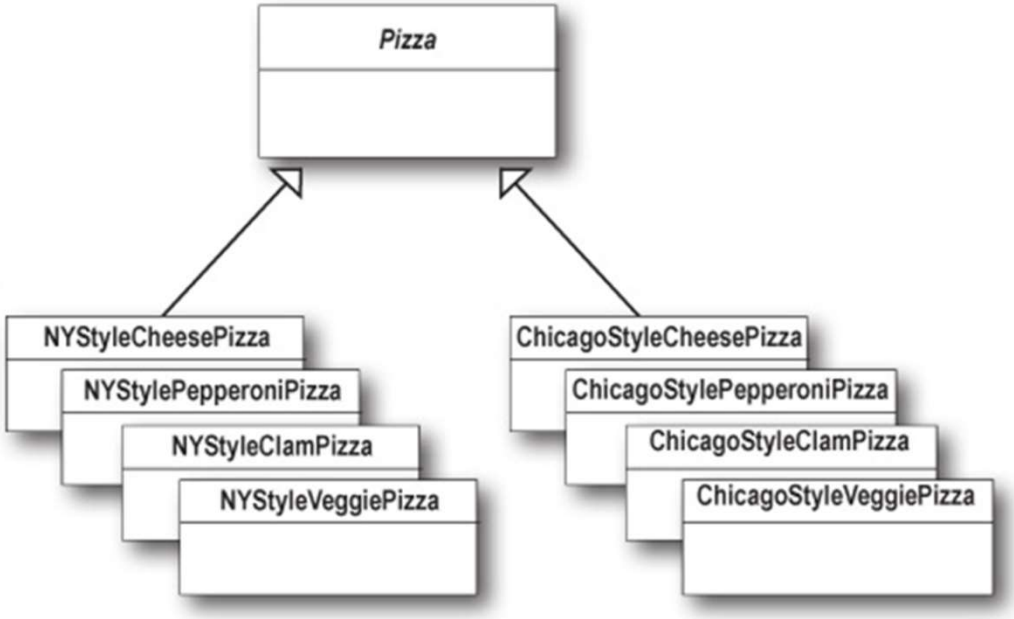


Factory Method leads to the creation of **parallel class hierarchies**; ConcreteCreators produce instances of ConcreteProducts that are operated on by Creators via the Product interface

Factory Method Example



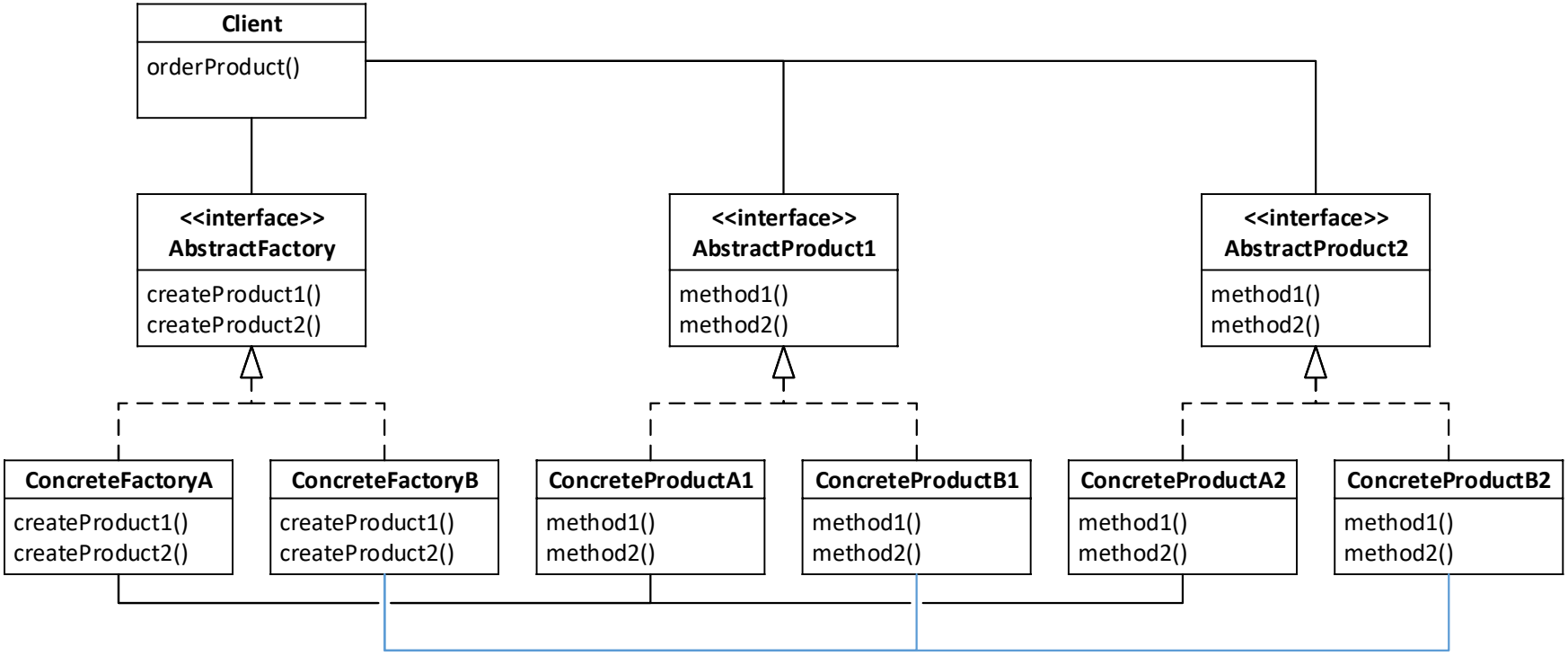
The subclassed PizzaStores are Concrete Creators with factory methods



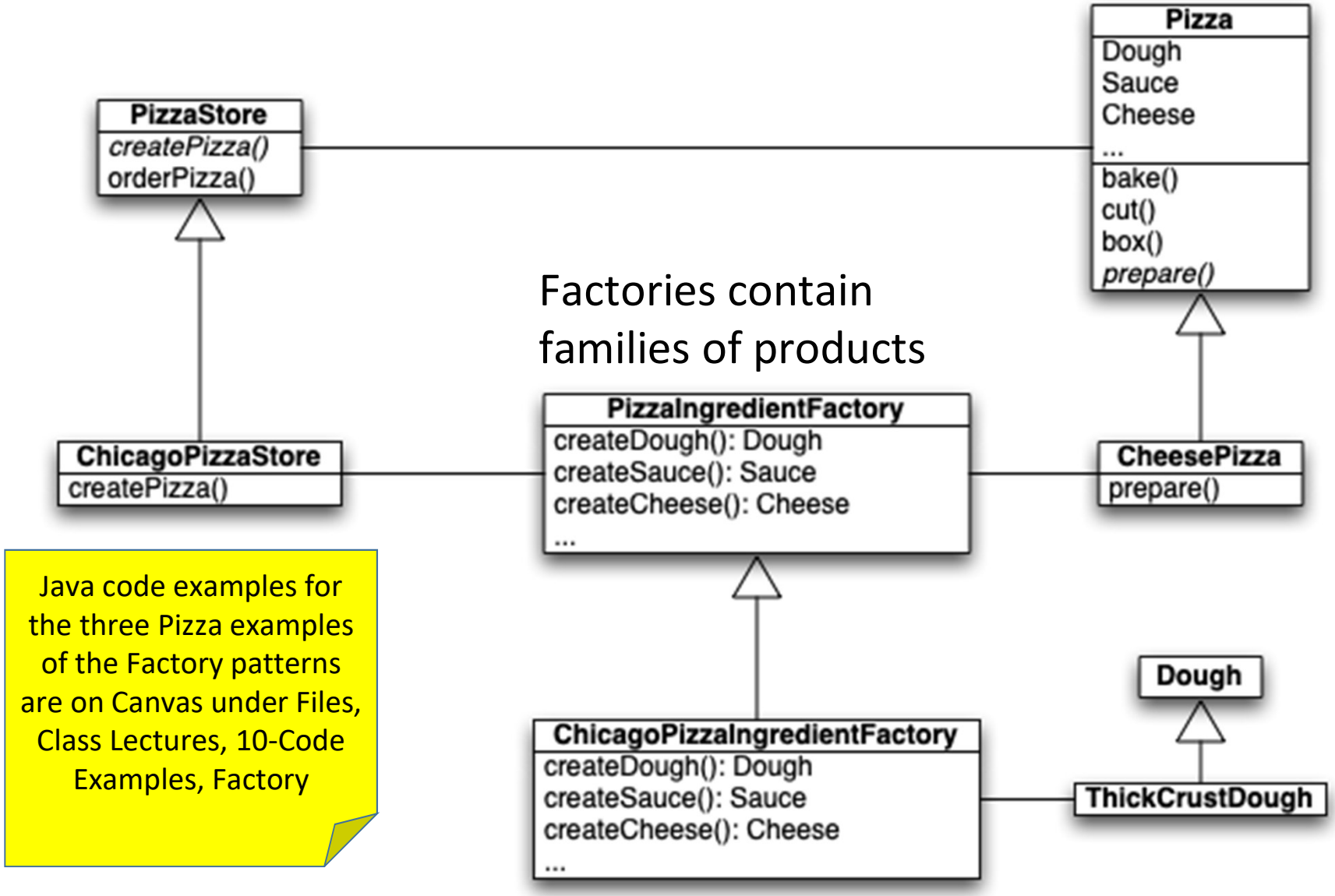
The subclassed Pizzas are Concrete Products

Abstract Factory

- Intent: Provide an interface for creating families of related or dependent objects without specifying concrete classes
- Problem: High levels of variability in both factory classes and product class families
- Solution: Provide the client with interfaces to abstract factories and product families, concrete factories can refer to their corresponding concrete products; relies on object composition
- Result: Consistency in factory and product properties and methods for clients, easily extended factories and concrete products
- OO Principles:
  - **Dependency Inversion Principle - Depend upon abstractions. Do not depend upon concrete classes.**
  - Encapsulate what varies
  - Favor composition over inheritance
  - Classes open for extension but closed for modification



Partial Class Diagram of Abstract Factory Solution



Factories Compared

- All factories encapsulate object creation
- **Simple Factory** is a simple way to decouple your clients from concrete classes
- **Factory Method** relies on inheritance: object creation is delegated to subclasses, which implement the factory method to create objects
- **Abstract Factory** relies on object composition: object creation is implemented in methods exposed in the factory interface
- All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes
- The intent of **Factory Method** is to allow a class to defer instantiation to its subclasses
- The intent of **Abstract Factory** is to create families of related objects without having to depend on their concrete classes
- The **Dependency Inversion Principle** guides us to avoid dependencies on concrete types and to strive for abstractions. Factories are a powerful technique for coding to abstractions, not concrete classes.

From Head First Design Patterns

OO Principles

- Encapsulate what varies
- Favor composition (delegation) over inheritance
- Program to interfaces not implementations
- Strive for loosely coupled designs between objects that interact
- Classes should be open for extension, but closed for modification
- Depend on abstractions, not concrete classes (Dependency Inversion Principle)
- Only talk to your (immediate) friends (Law of Demeter, Principle of Least Knowledge)
- Don’t call us, we’ll call you (the Hollywood Principle)
- A class should have only one reason to change

Don’t call us, we’ll call you

- Known as “The Hollywood Principle”
- It’s about reducing coupling
  - No unnecessary or improper references
- Often implemented through events or callbacks
- Seen in the observer pattern
  - An observer callback object is injected into a class to be observed, which raises an event to observers on a state change… (We’ll get there.)
- Also related to Dependency Injection and Inversion of Control (We’ll get there too.)
- Nice blog article at: http://matthewtmead.com/blog/hollywood-principle-dont-call-us-well-call-you-4/
- (Not on the midterm exam)

Homework 4: Semester Project OOAD

- Homework 4 asks you to engage in analysis and design activities for your semester project over the next three weeks.
  - You will generate a detailed set of tasks that can be accomplished with your proposed system and a comprehensive design of that system. The goal of these analysis and design activities is to generate information that will allow you to start implementing the system with confidence.
- Suggested Scope Approach
  - While I am asking for a non-trivial amount of information, you should only generate the information you need to achieve your desired functionality given the size of your team.
  - You will have four weeks (not counting the break) to develop your system prototype (2 iterations consisting of 2 weeks each).
  - During each iteration, each member of your group should expect to work on one to two use cases (possibly with other team mates). As a result, if you are a member of a two-person team, you'll want to target a 4-8 use cases for your system. A 4-person team is looking at a 8-16 use cases (16 is a lot!).
  - Use these guidelines to scope the work of this assignment and really focus on generating information that will guide your implementation efforts during the last four weeks of the semester.

Homework 4: Semester Project OOAD

- Your deliverable for Homework 4 is a PDF document that contains the information listed below:
- Project Summary
  - Who is on your team (include all names)? What is the high-level overview of your semester project. What are you trying to accomplish? What will your system do when you are done?
- Project Requirements
  - Based on the project summary, what are the requirements and responsibilities for your system? List the requirements and their associated responsibilities in this section. Identify the main goals of the system and its associated responsibilities. This list may be short but it should attempt to convey the "big picture" view of your system and what it is trying to accomplish. The requirements can include functional capabilities, constraints (such as platforms, number of users, etc.), and non-functional characteristics.
- Users and Tasks: Use Cases
  - How many different types of users will your system have? What tasks do they need to accomplish with your system. Document how the system will support each task via a use case (text or UML). Try to think of the various problems that can occur while trying to accomplish these tasks and document them in the use case.
- Activity Diagram
  - Pick your most complex use case and create an activity diagram that documents all of the possible paths through it. Typically, one activity diagram corresponds to a single use case, but if you can think of a way to combine multiple use cases into a single activity diagram that's fine too.
- Architecture Diagram
  - Draw a diagram that shows the architecture of your system. Is it a web app? A mobile app? Does your mobile app interact with a web service? Does your web app interact with both desktop and mobile clients? etc.
  - This diagram will be a boxes-and-arrows diagram that shows the various architectural components of your system (devices, databases, 3rd party services, etc.)
  - If you're building a web service or application, what frameworks will you be using, how will your service be structured, what request-response cycles will your service be supporting, etc.
  - The goal of this task is to get you to delve more deeply into the frameworks you'll be using to implement your system and to think up front about what classes and services in that framework are the most relevant to your application.
  - Examples of typical architecture diagrams are shown at https://www.edrawsoft.com/architecture-diagram.php; yours does not have to be in color or in a particular style – show what will talk to what at a component/framework level.
  - Note: not all of this information has to be conveyed in the diagram, you can augment the diagram with a textual description that goes into the details more deeply.

Homework 4: Semester Project OOAD

- Your deliverable for Homework 4 is a document that contains the information listed below (continued):
- Data Storage
  - Discuss how you will persist data in your application. What storage technology will you use? Text files? XML? CSV? MySQL? Where will the data be stored? Describe the classes that you will use to access this data at run-time.
- UI Mockups/Sketches
  - Create screen mockups for the user interface of various parts of your application. What will a user see as they work through the tasks identified in your use cases? What is the overall organization of your user interface? How will data be displayed? How will the user navigate from screen to screen?
  - Use this task as a means for focusing your thoughts about what you will actually be creating... iterate now on how your screens will be laid out and then include your final sketches in this section.
  - There is no fixed number of screen sketches. Include what you think is needed to convey an overall sense of your application. Note: it is okay to work on paper for this task and then scan in your work to include in your document. For an automated tool, I highly recommend Balsamiq at https://balsamiq.com/
- User Interactions
  - Use your use cases and UI mockups to identify at least three interactions that your user will have with your application.
  - For each interaction, describe how your system will support it. Then show a sequence diagram of the objects that will participate in the interaction.
  - Some of these objects will represent UI widgets, some will be model objects used to access/update persistent data, and some will be objects that sit in between the UI and the persistent data. This latter class of object is known as a controller and they contain application logic that decides how to respond to events, updating both model objects and UI widgets to represent the new state of the system.
  - Recall that sequence diagrams do not contain conditional constructs, so be sure to clearly describe the interaction that is being displayed in the sequence diagram. Do not try to show if-then-else scenarios in a single sequence diagram. If you find yourself needing to show such a situation, simply create two sequence diagrams, one that shows the true branch and one that shows the false branch.
- Class Diagram
  - Create a class diagram that documents everything you have gleaned from the other activities above with respect to what classes your system will contain: what relationships they have, what are their attributes and (public) methods, what design patterns are present in your design, etc.
  - If it is too difficult to fit everything into a single diagram, you can split your classes across more than one diagram in whatever way makes the most sense for your particular application.
- For the User Interaction Sequence Diagrams and Class Diagrams, you can use a scan of a paper or whiteboard diagram, or use your favorite UML tools.

Homework 4: Semester Project OOAD

- The point breakdown of this assignment is as follows:

Section	Points	Comments
Project Summary	10	Include your team's names!
Requirements	20	Main goals and other requirements
Use Cases	20	About 2 to 4 per team member
Activity Diagram	10	For most complex use case(s)
Architecture Diagram	20	Focus on components of system
Data Storage	10	Include tool and data description
UI Mockups/Sketches	20	Number of sketches varies
User Interactions	20	At least three sequence diagrams
Class Diagram	20	One or more describing system
Total:	**150**	

- Single PDF from your Team is Due Friday 3/22 11 AM on Canvas
- Please contact me EARLY in the cycle for questions, clarifications, or variations for your project

Next Steps

- Optional additional material
  - Dr. Anderson's lectures on these patterns
    - You can find it on the class Canvas site under Media Gallery
    - Again, very similar material as I'm using versions of his slides…
- This week
  - No Quiz prior to Exam
  - Monday 3/4 – Midterm Exam
  - Wednesday 3/6 – Recitation w/Manjunath
  - Friday 3/8 – OO Java
  - I'm travelling to DC Tue-Thur, please excuse delayed responses
- Things that are due
  - Grad Presentation Outline was due Mon 2/25 at 11 AM – get it in if you haven't
  - Homework 3 was due today Friday 3/1 11 AM
  - Class Semester Project topic Canvas submission is due Friday 3/1 11 AM
    - Not graded - this could come in as late as 3/8, but try to settle on your team's topic
  - Homework 4 – Design for Semester Project – is due Friday 3/22 11 AM
    - Your team can also start to develop preliminary code for your project
    - There will be two homeworks related to code delivery, an interim one on 4/12, and a final delivery on 4/26

*****NOTE, Homework 4 and 5 will NOT be accepted after the 1 week late period*****

*****NOTE, the final Homework 6 submission on 4/26 may NOT be turned in late*****

*****NOTE, the Graduate Presentation submission on 4/15 may NOT be turned in late*****