Strategy, Bridge & Factory

CSCI 4448/5448: Object-Oriented Analysis & Design

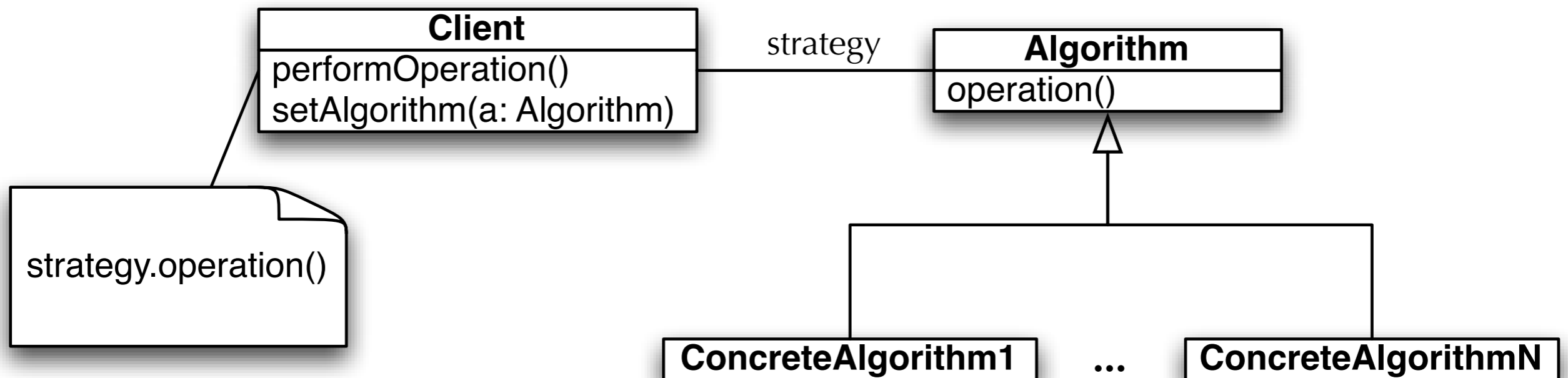### Lecture 9 — 09/20/2011

Goals of the Lecture

- Cover the material in Chapters 9, 10 & 11 of our textbook

    - The Strategy Pattern

    - The Bridge Pattern

    - The Abstract Factory Pattern
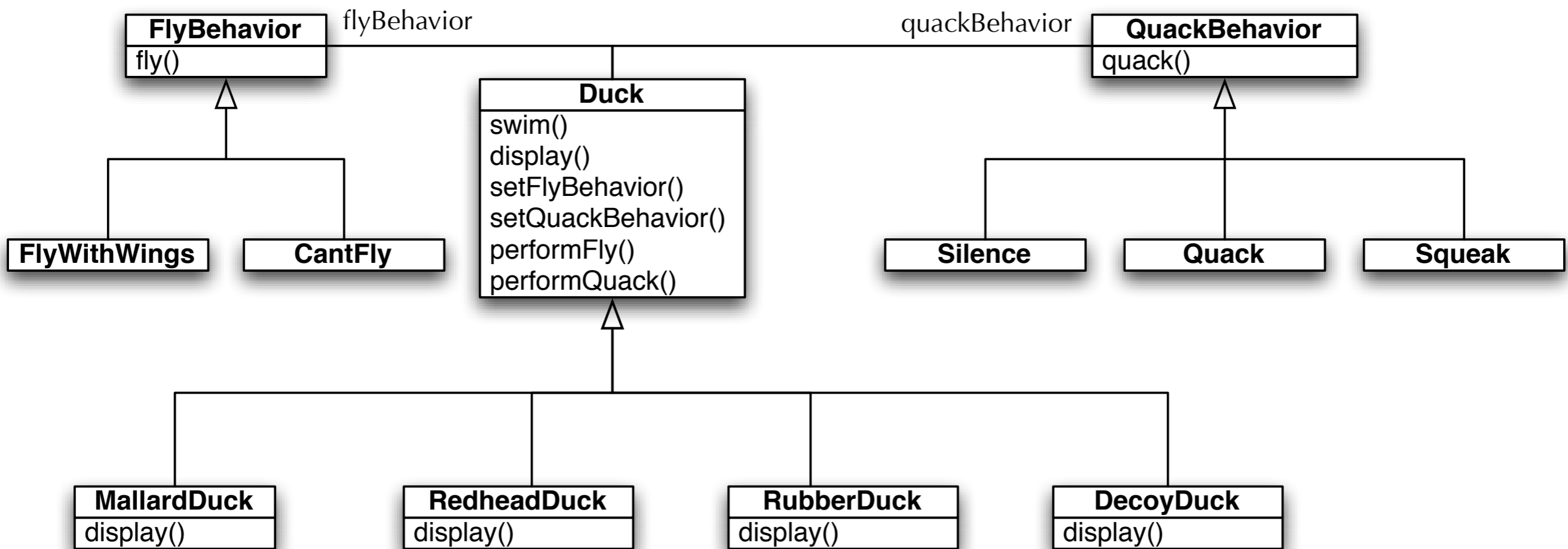
The Strategy Pattern

- We already covered this pattern in Lecture 6

- Intention

    - Lets you use different algorithms depending on context

        - Or lets you vary behavior across subclasses of some abstract class

The Structure of the Strategy Pattern

SimUDuck's use of Strategy

FlyBehavior
fly()

flyBehavior

Duck
swim()
display()
setFlyBehavior()
setQuackBehavior()
performFly()
performQuack()

quackBehavior

QuackBehavior
quack()

FlyWithWings

CantFly

Silence

Quack

Squeak

MallardDuck
display()

RedheadDuck
display()

RubberDuck
display()

DecoyDuck
display()

Duck's Fly and Quack behaviors are decoupled from Duck

Your Turn

- Work in a group and create a class diagram that shows a design that meets these requirements and uses Strategy

  - A bank account has a balance and a set of transactions

  - A transaction records the date, the vendor, the amount and a category

  - A client can add transactions, ask for a List of transactions and ask that the List be sorted in a certain way

    - The list can be sorted by date, vendor, amount or category

The Bridge Pattern

- The Gang of Four says the intent of the pattern is to "decouple an abstraction from its implementation so that the two can vary independently"

- What it doesn't mean

    - Allow a C++ header file to be implemented in multiple ways

- What it does mean

    - Allows a set of abstract objects to implement their operations in a number of ways in a scalable fashion

Bottom-Up Design

- The book presents an example that derives the bridge pattern

- Let a set of shapes draw themselves using different drawing libraries

  - Think of the libraries as items such as Monitor, Printer, OffScreenBuffer, etc.

   - Imagine a world where each of these might have slightly different methods and method signatures

Examples of Drawing Library

- The drawing library for Monitor has these methods

  - draw_a_line(x1, y1, x2, y2)

  - draw_a_circle(x, y, r)

- The drawing library for Printer has these methods

  - drawline(x1, x2, y1, y2)

  - drawcircle(x, y, r)

Monitor	Printer
draw_a_line(x1, y1, x2, y2) draw_a_circle(x, y, r)	drawline(x1, x2, y1, y2) drawcircle(x, y, r)

Examples of Shape

Shape
+draw()

Rectangle
+draw()
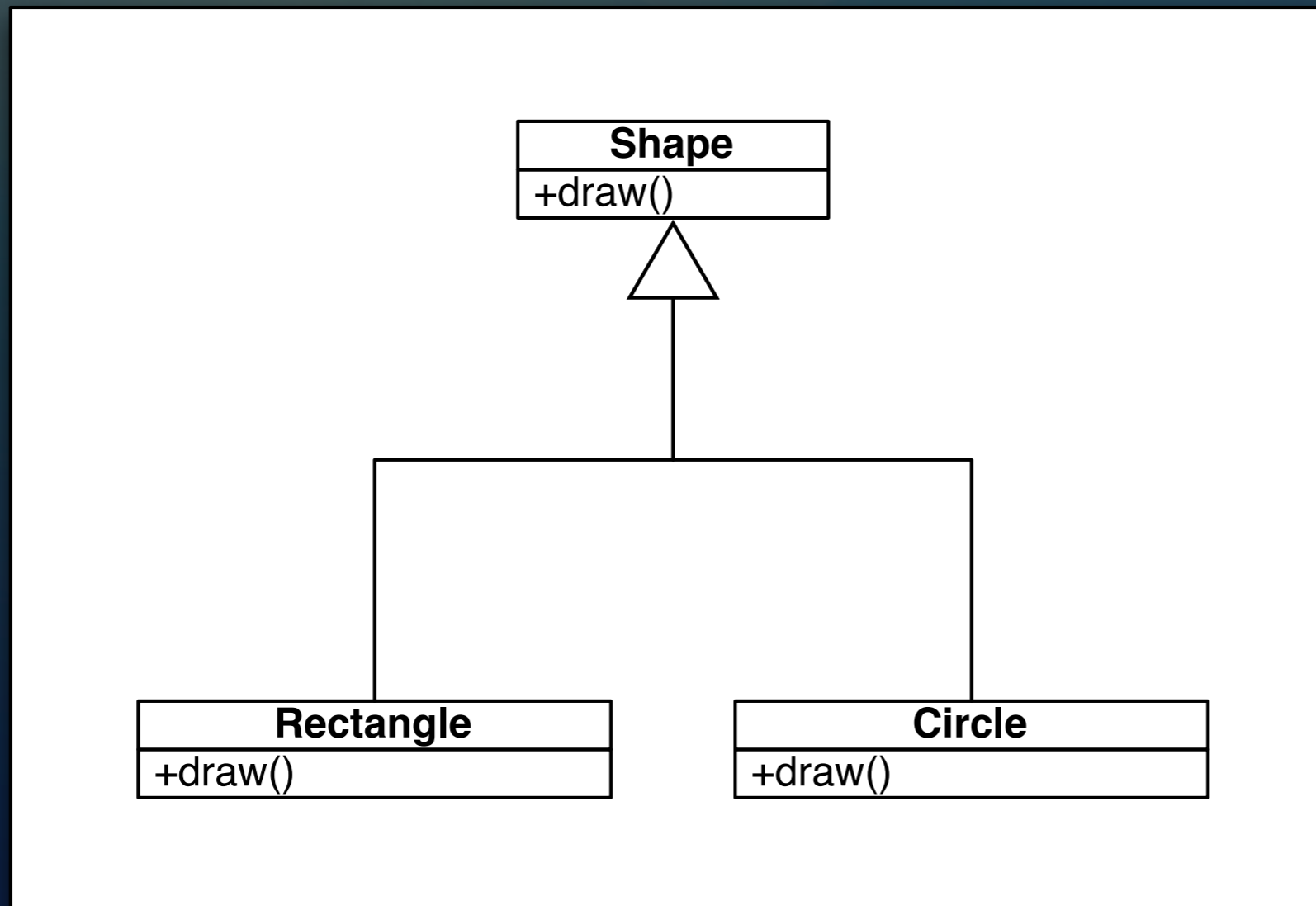
Circle
+draw()

We want to be able to create collections of rectangles and circles and then tell the collection to draw itself and have it work regardless of the medium

Problem: Try adding another shape or library

Client

Shape
+draw()

Rectangle
+draw()
#drawLine()

Circle
+draw()
#drawCircle()

MonitorRect
#drawLine()

PrinterRect
#drawLine()

MonitorCircle
#drawCircle()

PrinterCircle
#drawCircle()

Monitor
draw_a_line(x1, y1, x2, y2)
draw_a_circle(x, y, r)

Printer
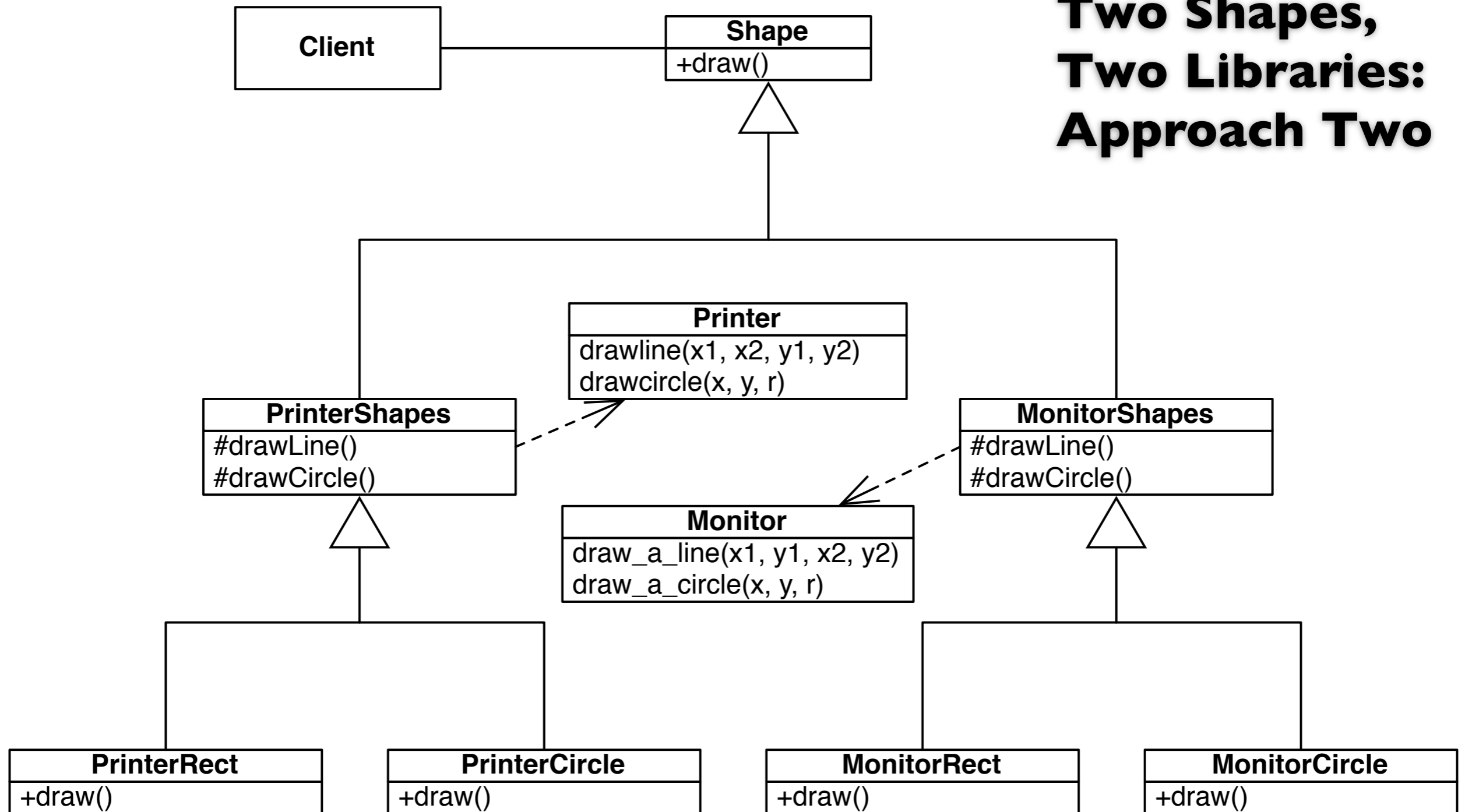drawline(x1, x2, y1, y2)
drawcircle(x, y, r)
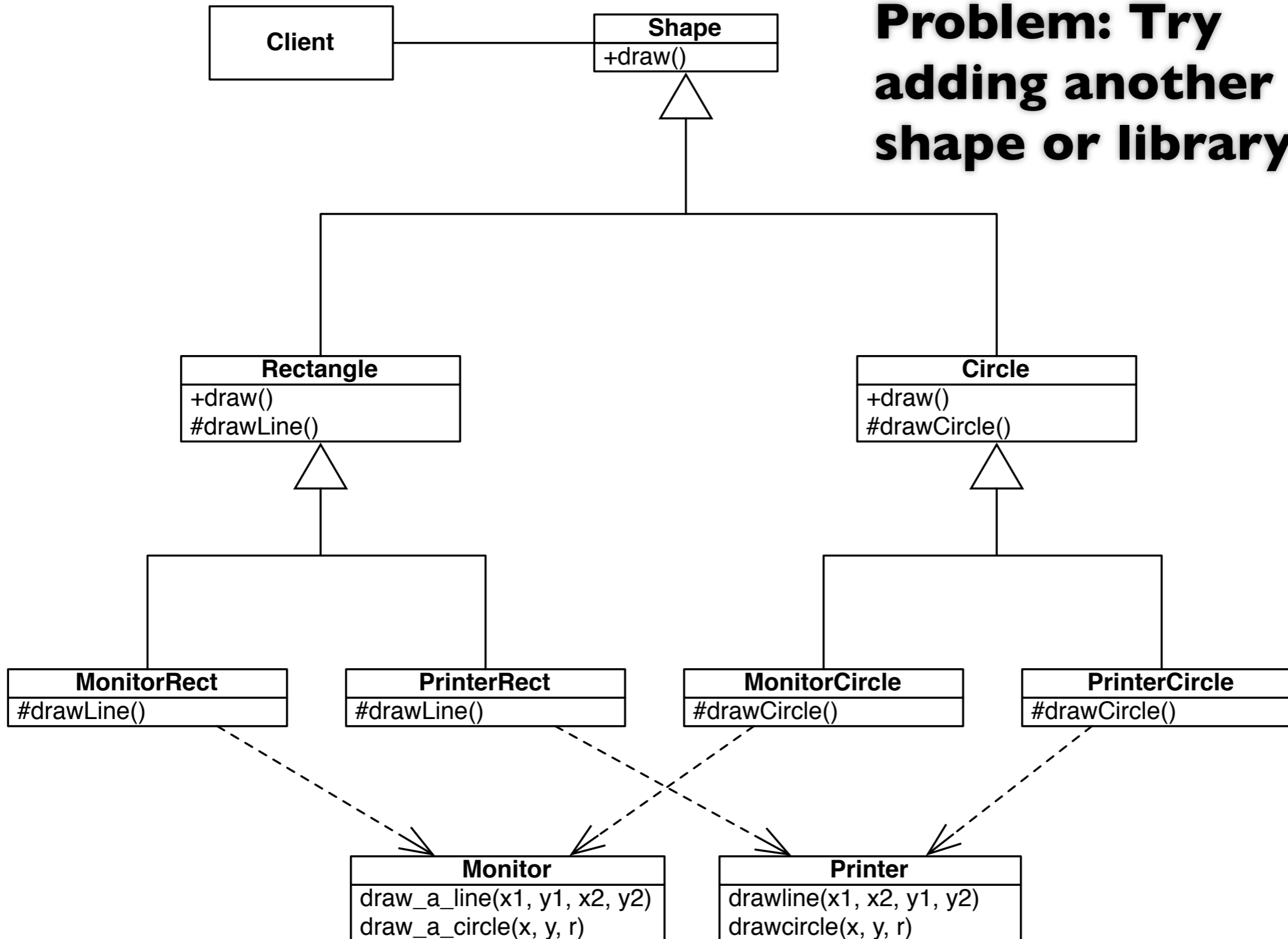
Emphasis of Problem (I)

- We are using inheritance to specialize for implementation

  - And, surprise, surprise, we encounter the combinatorial subclass program once again

    - 2 shapes, 2 libraries: 4 subclasses

    - 3 shapes, 3 libraries: 9 subclasses

    - 100 shapes, 10 libraries: 1000 subclasses

  - Not good!

Emphasis of Problem (II)

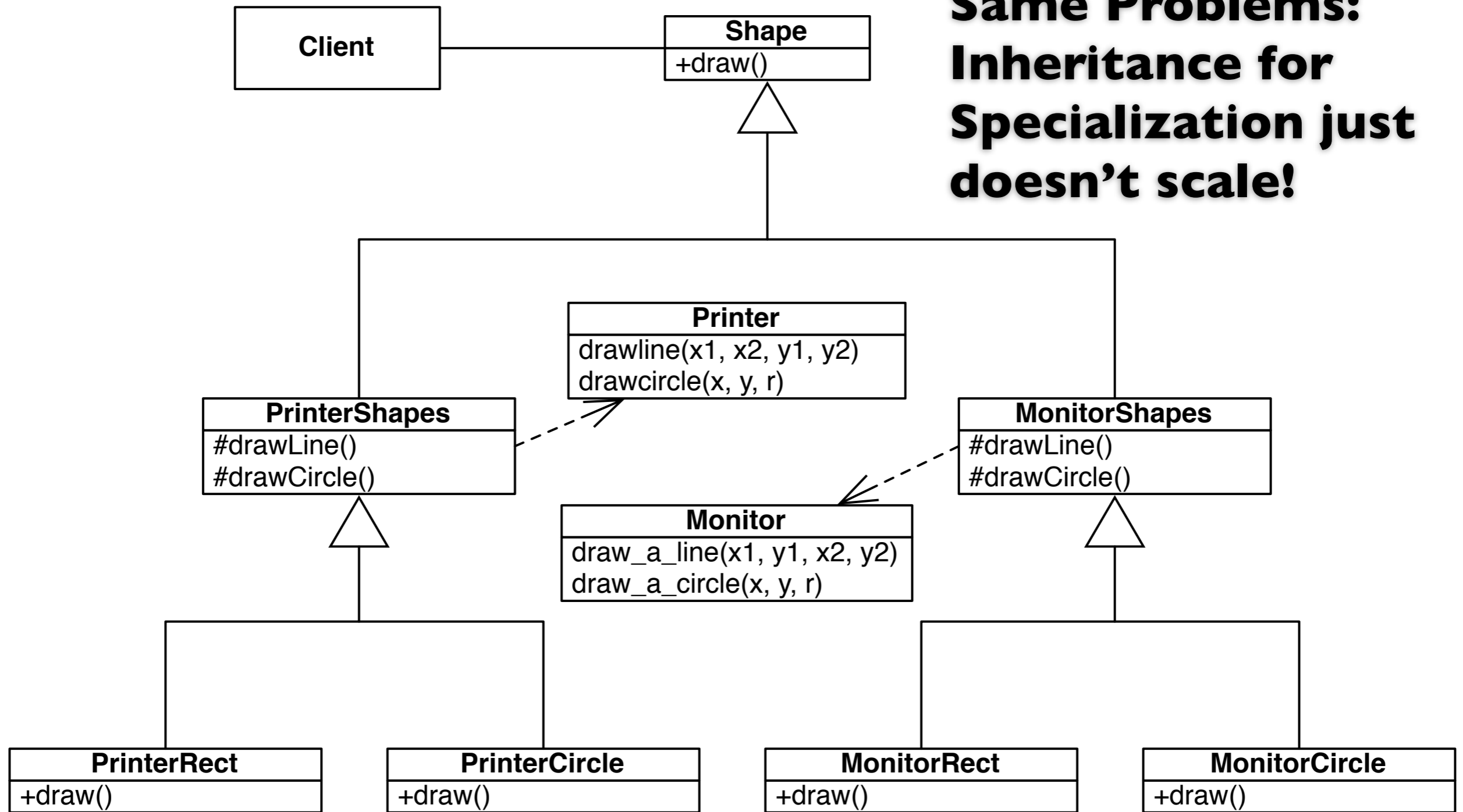- Is there redundancy (duplication) in this design?

  - Yes, each subclass method is VERY similar

- Tight Coupling

  - You bet… each subclass highly dependent on the drawing libraries

    - change a library, change a lot of subclasses

- Strong Cohesion? Not completely, shapes need to know about their drawing libraries; no single location for drawing

Two Shapes, Two Libraries: Approach Two

Same Problems: Inheritance for Specialization just doesn't scale!

Client

Shape
+draw()

Printer
drawline(x1, x2, y1, y2)
drawcircle(x, y, r)

PrinterShapes
#drawLine()
#drawCircle()

MonitorShapes
#drawLine()
#drawCircle()

Monitor
draw_a_line(x1, y1, x2, y2)
draw_a_circle(x, y, r)

PrinterRect
+draw()

PrinterCircle
+draw()

MonitorRect
+draw()

MonitorCircle
+draw()

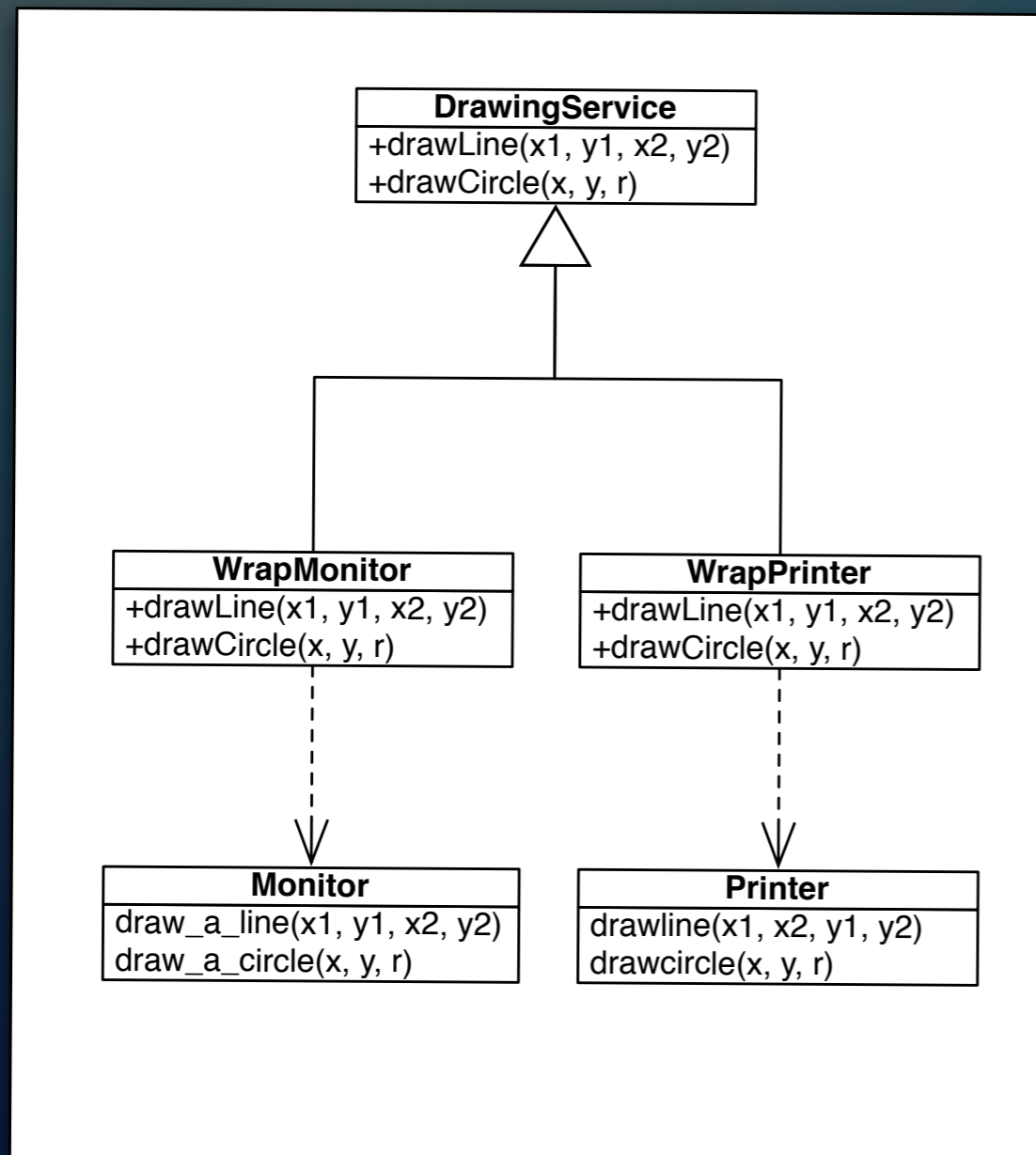# Finding a Solution

- Our book offers two strategies to find the right solution

    - Find what varies and encapsulate it

    - Favor delegation over inheritance (book: aggregation)

- What varies?

    - Shapes and Drawing Libraries

- We've seen two approaches to using inheritance

    - But neither worked, let's try delegation instead

What varies? Shapes
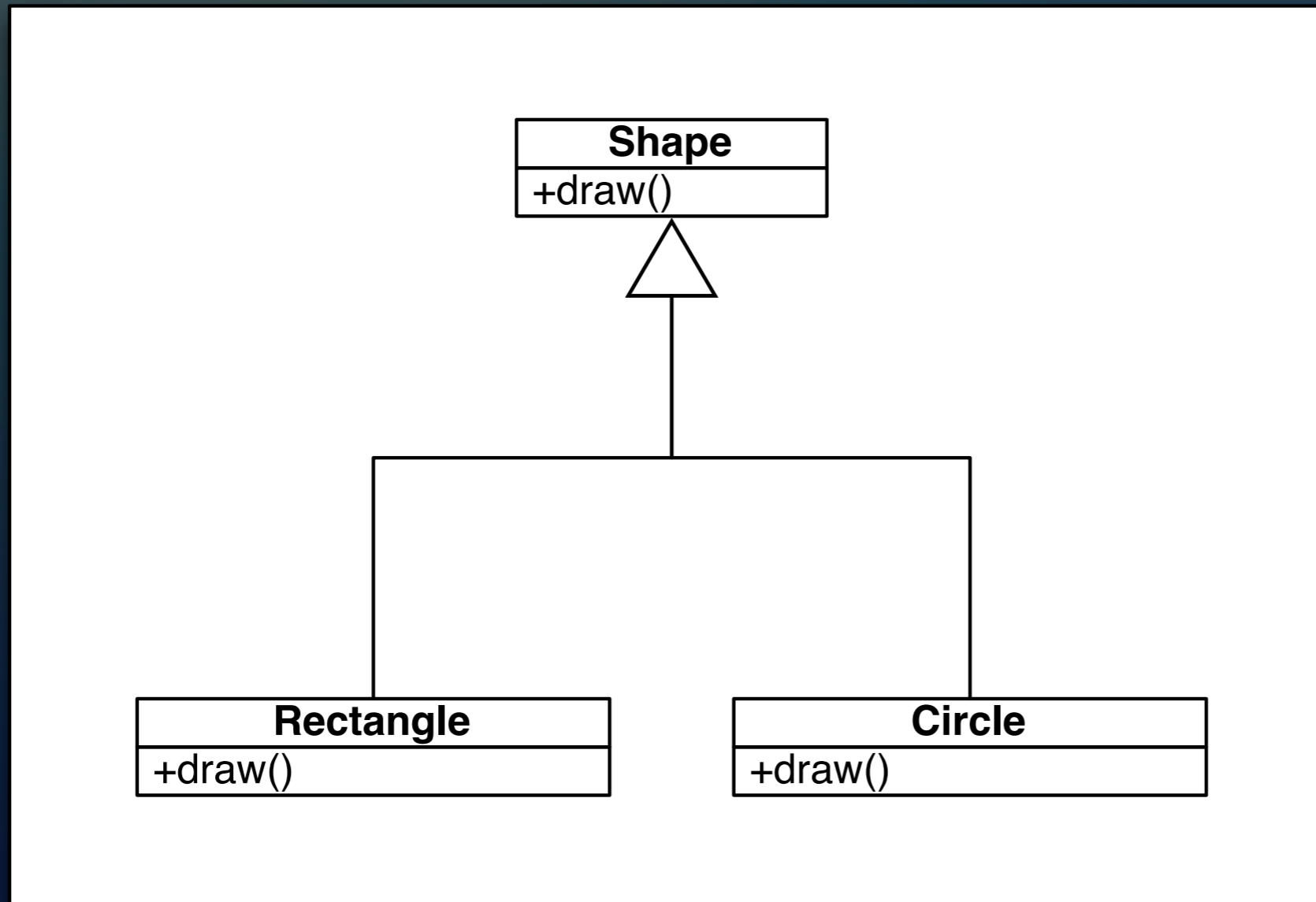
What varies? Drawing Libraries

DrawingService
+drawLine(x1, y1, x2, y2)
+drawCircle(x, y, r)

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
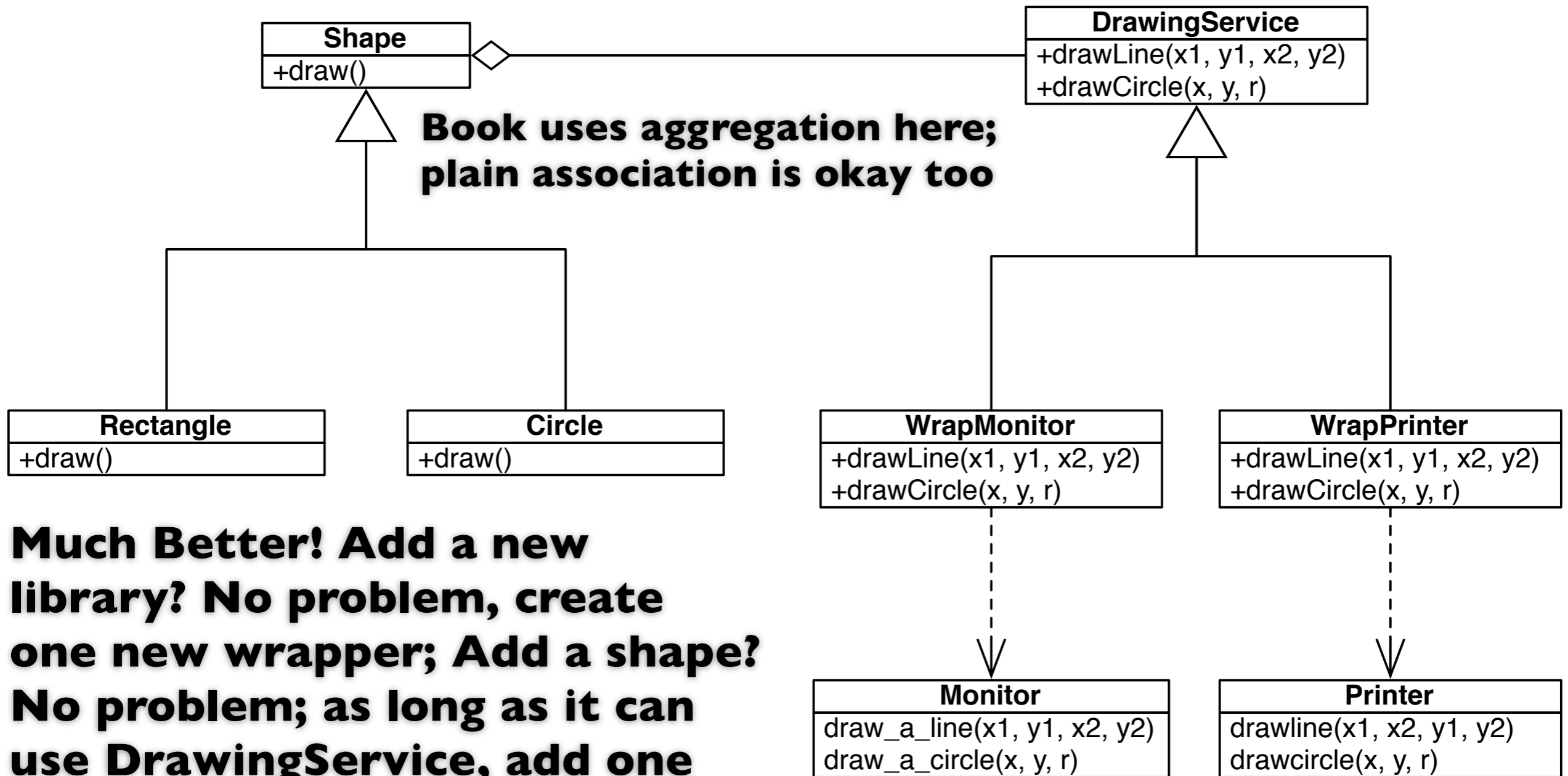drawline(x1, x2, y1, y2)
drawcircle(x, y, r)

One abstract service which defines a *uniform* interface

The next level down consists of classes that wrap each library behind the uniform interface

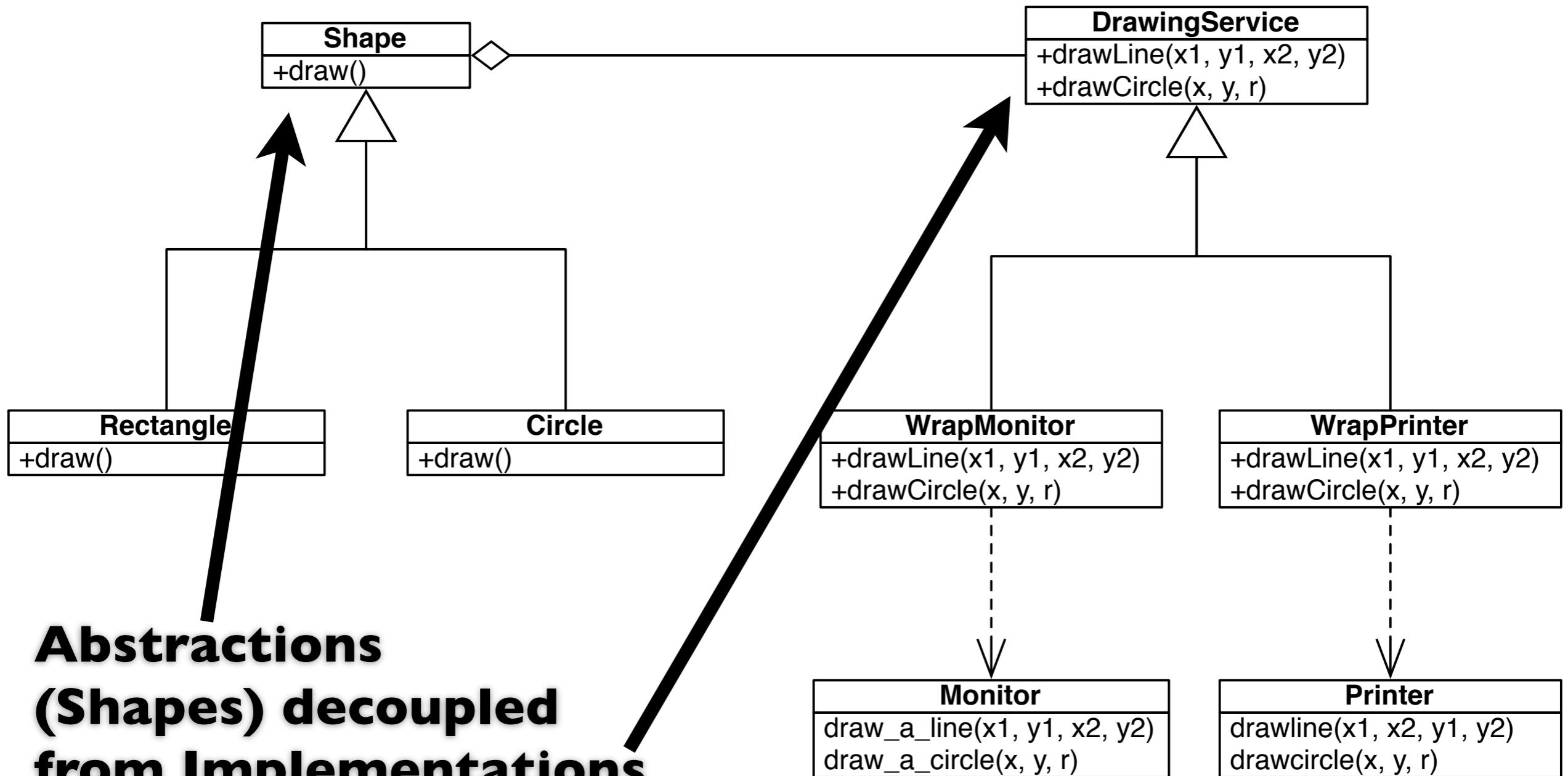# Favor delegation

- Two choices

    - DrawingLibrary delegates to Shape

        - That doesn't sound right

    - Shape delegates to DrawingLibrary

        - That sounds better

- So…

Shape
+draw()

DrawingService
+drawLine(x1, y1, x2, y2)
+drawCircle(x, y, r)

Book uses aggregation here; plain association is okay too

Rectangle
+draw()

Circle
+draw()

WrapMonitor
+drawLine(x1, y1, x2, y2)
+drawCircle(x, y, r)

WrapPrinter
+drawLine(x1, y1, x2, y2)
+drawCircle(x, y, r)

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

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)

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)

The Structure of the Bridge Pattern
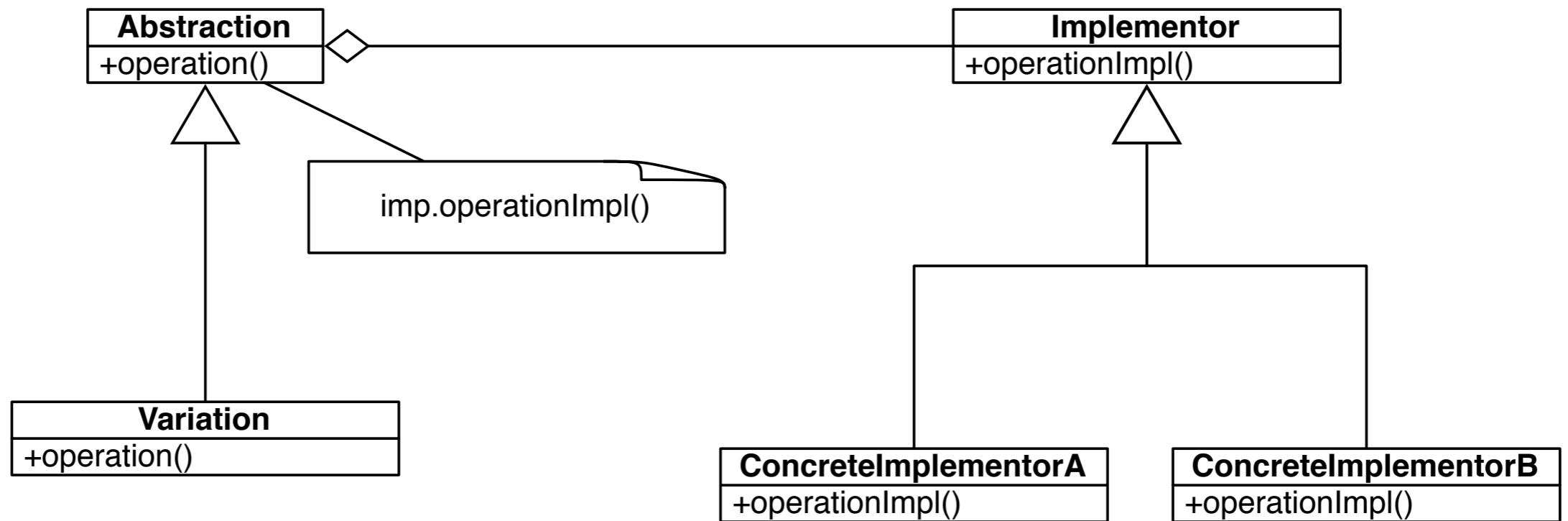
Two Factory Patterns

- I'm going to use an example from Head First Design Patterns to introduce the concept of Abstract Factory

  - This example uses Factory Method, so you get to be introduced early to this pattern

    - (Our textbook holds off on Factory Method until Chapter 23!)

- You still need to read chapter 11 of our textbook and be familiar with its material!

Factory Pattern: The Problem With "New"

- Each time we invoke the "new" command to create a new object, we violate the "Code to an Interface" design principle

- Example

    - Duck duck = new DecoyDuck()

- Even though our variable's type is set to an "interface", the class that contains this statement depends on "DecoyDuck"

- In addition, if you have code that checks a few variables and instantiates a particular type of class based on the state of those variables, then the containing class depends on each referenced concrete class

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

Obvious Problems:
needs to be recompiled each time a dep. changes;
add new classes, change this code;
remove existing classes, change this code

This means that this code violates the "encapsulate what varies" design principle

PizzaStore Example

- We have a pizza store program that wants to separate the process of creating a pizza with the process of preparing/ordering a pizza

- Initial Code: mixed the two processes

```
1  public class PizzaStore {
2
3      Pizza orderPizza(String type) {
4
5          Pizza pizza;
6
7          if (type.equals("cheese")) {
8              pizza = new CheesePizza();
9          } else if (type.equals("greek")) {
10             pizza = new GreekPizza();
11         } else if (type.equals("pepperoni")) {
12             pizza = new PepperoniPizza();
13         }
14
15         pizza.prepare();
16         pizza.bake();
17         pizza.cut();
18         pizza.box();
19
20         return pizza;
21     }
22
23  }
24
```

Creation

Creation code has all the same problems as the code on the previous slide
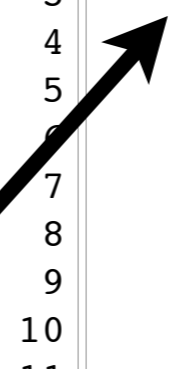
Preparation
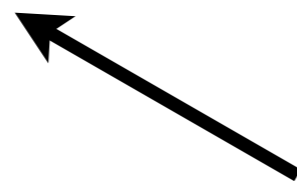
Note: excellent example of "coding to an interface"
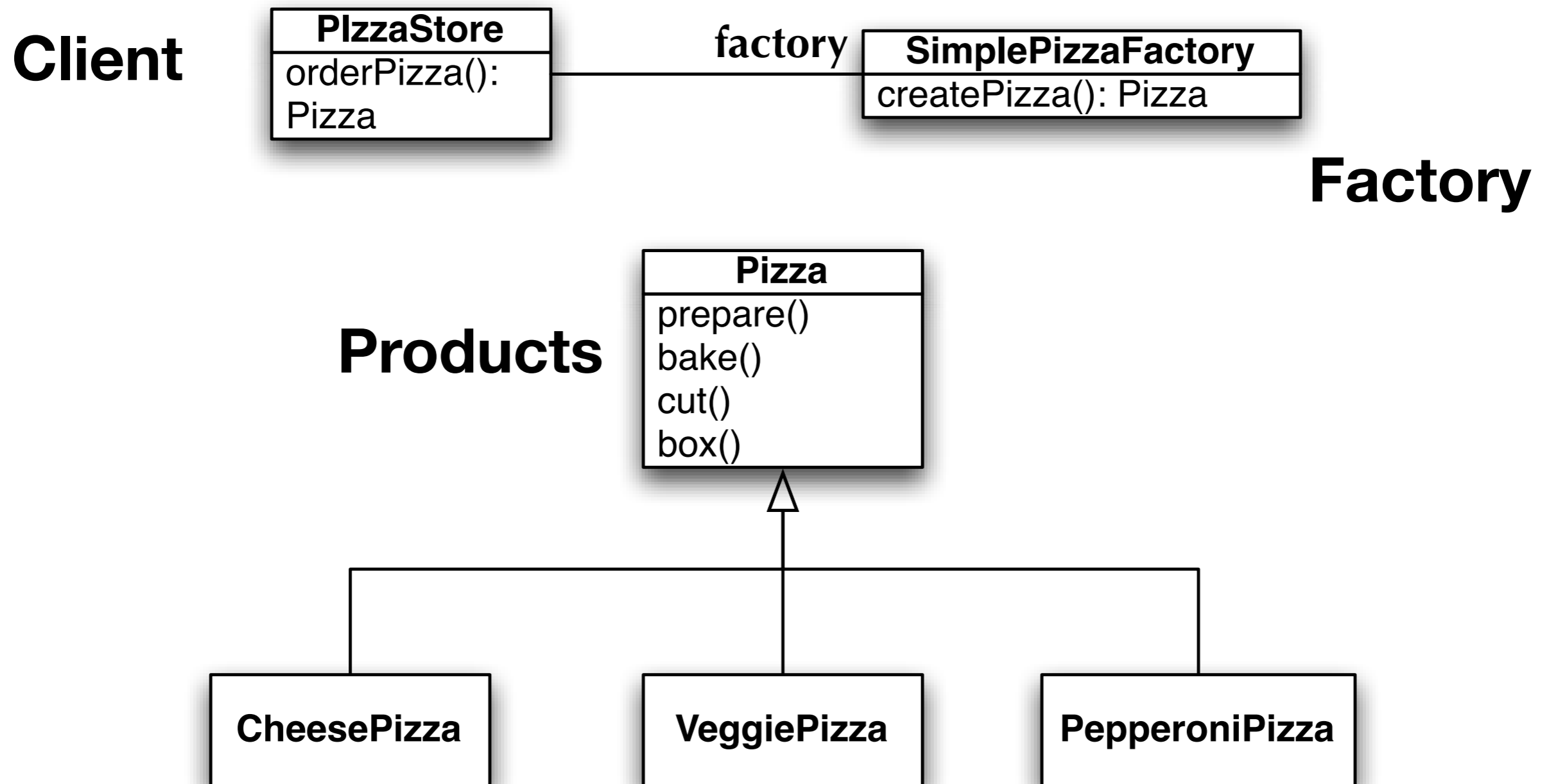
26

Encapsulate Creation Code

- A simple way to encapsulate this code is to put it in a separate class

  - That new class depends on the concrete classes, but those dependencies no longer impact the preparation code

```
 1  public class PizzaStore {
 2
 3      private SimplePizzaFactory factory;
 4
 5      public PizzaStore(SimplePizzaFactory factory) {
 6          this.factory = factory;
 7      }
 8
 9      public Pizza orderPizza(String type) {
10
11          Pizza pizza = factory.createPizza(type);
12
13          pizza.prepare();
14          pizza.bake();
15          pizza.cut();
16          pizza.box();
17
18          return pizza;
19      }
20
21  }
22
```

```
 1  public class SimplePizzaFactory {
 2
 3      public Pizza createPizza(String type) {
 4          if (type.equals("cheese")) {
 5              return new CheesePizza();
 6          } else if (type.equals("greek")) {
 7              return new GreekPizza();
 8          } else if (type.equals("pepperoni")) {
 9              return new PepperoniPizza();
10          }
11      }
12
13  }
14
```

27

Class Diagram of New Solution

Client

PIzzaStore
orderPizza(): Pizza

factory

SimplePizzaFactory
createPizza(): Pizza

Factory

Products

Pizza
prepare() bake() cut() box()

CheesePizza

VeggiePizza

PepperoniPizza

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

- To demonstrate the factory method pattern, the pizza store example evolves
  - to include the notion of different franchises
  - that exist in different parts of the country (California, New York, Chicago)
- Each franchise will need its own factory to create pizzas that match the proclivities of the locals
  - However, we want to retain the preparation process that has made PizzaStore such a great success
- The Factory Method Design Pattern allows you to do this by
  - placing abstract, "code to an interface" code in a superclass
  - placing object creation code in a subclass
- PizzaStore becomes an abstract class with an abstract createPizza() method
- We then create subclasses that override createPizza() for each region

New PizzaStore Class

```
 1  public abstract class PizzaStore {
 2
 3      protected abstract createPizza(String type);
 4
 5      public Pizza orderPizza(String type) {
 6
 7          Pizza pizza = createPizza(type);
 8
 9          pizza.prepare();
10          pizza.bake();
11          pizza.cut();
12          pizza.box();
13
14          return pizza;
15      }
16
17  }
18
```

Factory Method

This class is a (very simple) OO framework. The framework provides one service "prepare pizza".

The framework invokes the createPizza() factory method to create a pizza that it can prepare using a well-defined, consistent process.

A "client" of the framework will subclass this class and provide an implementation of the createPizza() method.

Any dependencies on concrete "product" classes are encapsulated in the subclass.

Beautiful Abstract Base Class!

New York Pizza Store
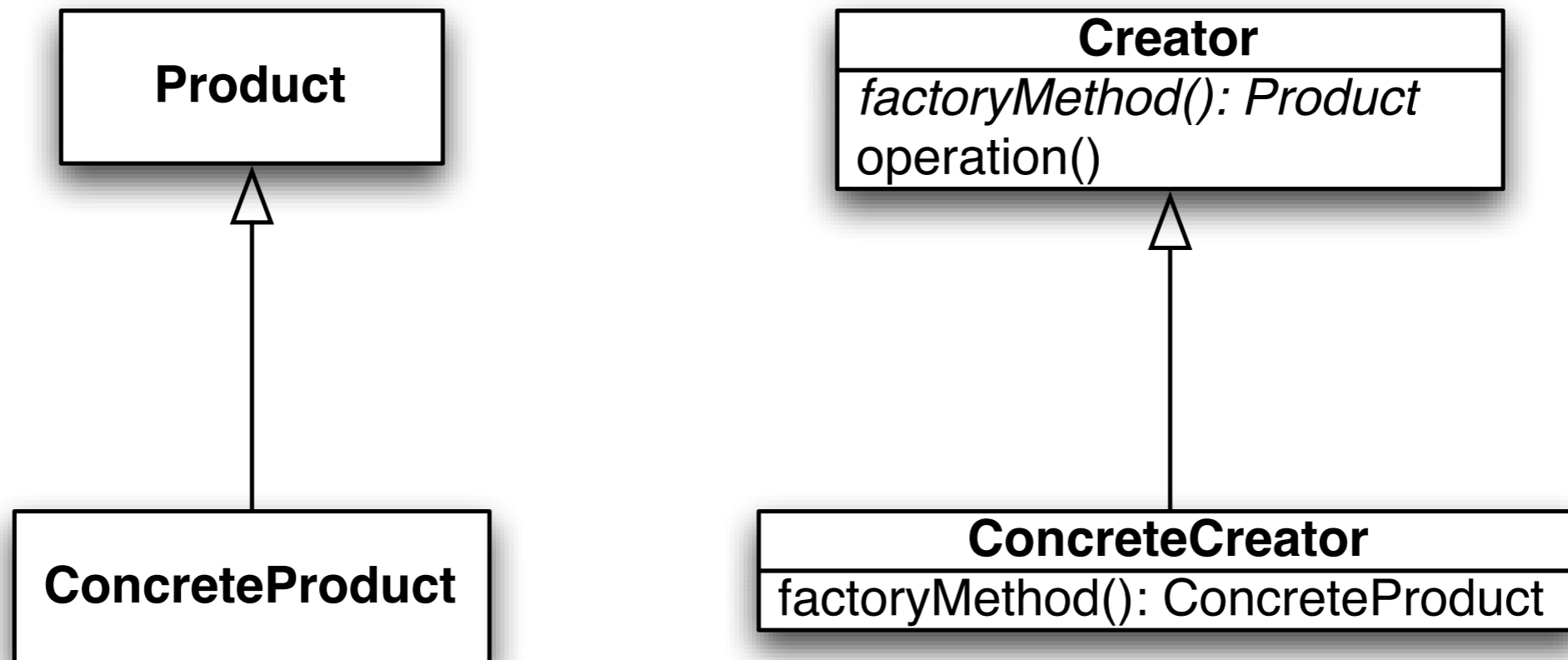
```
 1  public class NYPizzaStore extends PizzaStore {
 2      public Pizza createPizza(String type) {
 3          if (type.equals("cheese")) {
 4              return new NYCheesePizza();
 5          } else if (type.equals("greek")) {
 6              return new NYGreekPizza();
 7          } else if (type.equals("pepperoni")) {
 8              return new NYPepperoniPizza();
 9          }
10          return null;
11      }
12  }
13
```

Nice and Simple. If you want a NY-Style Pizza, you create an instance of this class and call orderPizza() passing in the type. The subclass makes sure that the pizza is created using the correct style.

If you need a different style, create a new subclass.

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

Product

Creator
factoryMethod(): Product
operation()

ConcreteProduct

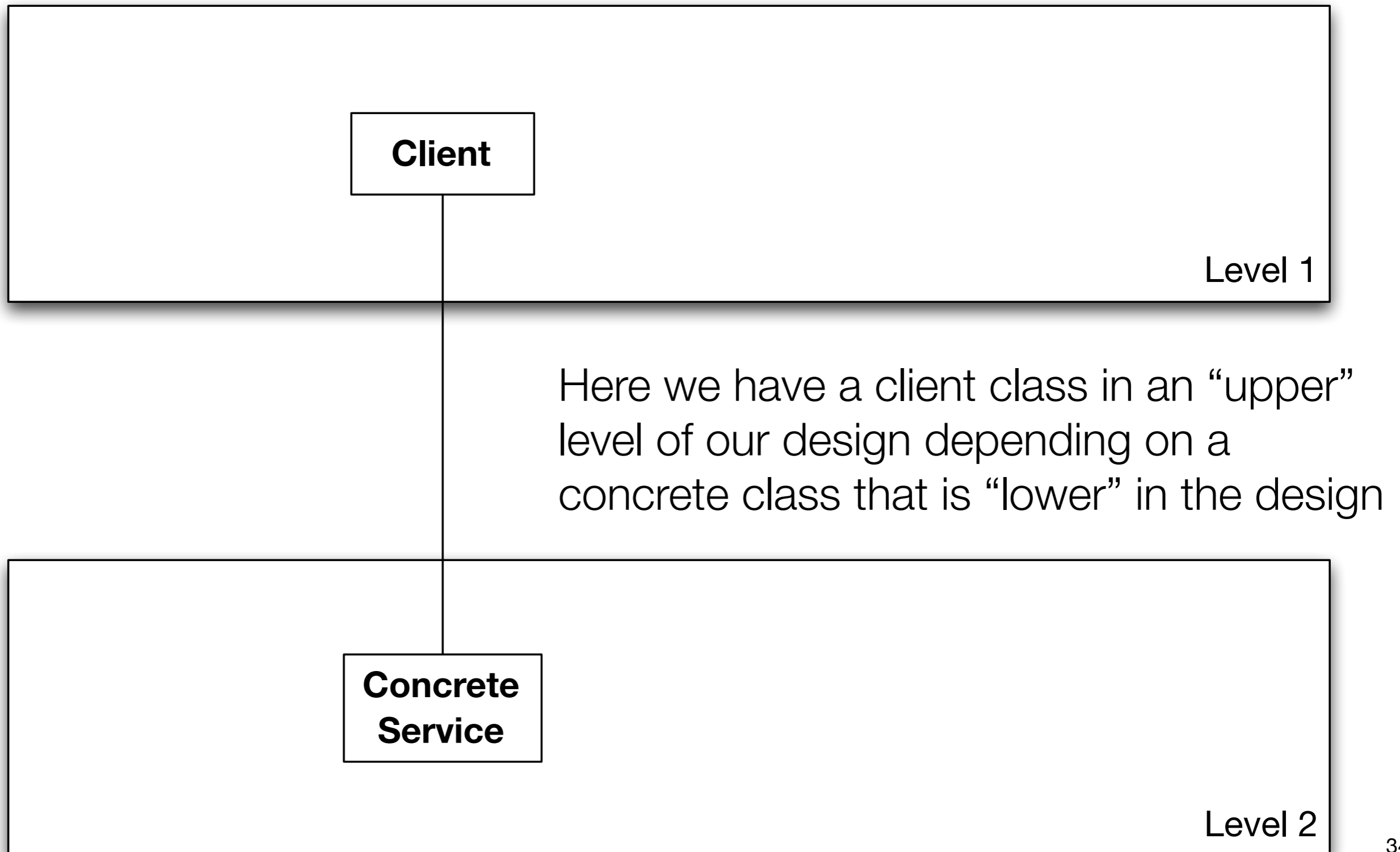| ConcreteCreator |
| --- |
| factoryMethod(): ConcreteProduct |

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

Dependency Inversion Principle (I)

- Factory Method is one way of following the **dependency inversion principle**

  - "Depend upon abstractions. Do not depend upon concrete classes."

- Normally "high-level" classes depend on "low-level" classes;

  - Instead, they BOTH should depend on an abstract interface

Dependency Inversion Principle: Pictorially

Client

Level 1

Here we have a client class in an "upper" level of our design depending on a concrete class that is "lower" in the design

Concrete Service

Level 2

Dependency Inversion Principle: Pictorially

Instead, create an interface that lives in the upper level that hides the concrete classes in the lower level; "code to an interface"
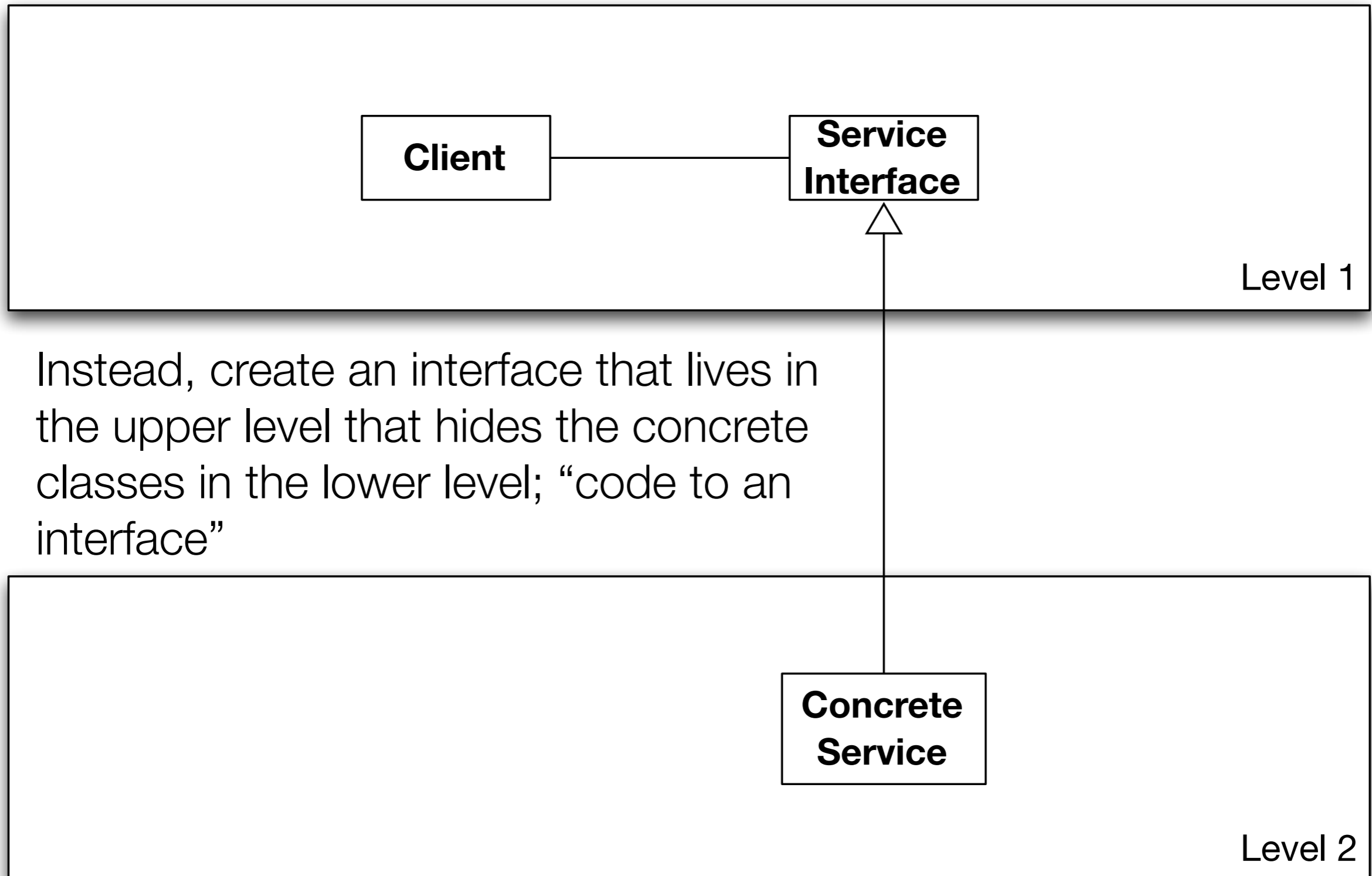
Dependency Inversion Principle (II)

- Factory Method is one way of following the **dependency inversion principle**

  - "Depend upon abstractions. Do not depend upon concrete classes."

- Normally "high-level" classes depend on "low-level" classes;

  - Instead, they BOTH should depend on an abstract interface

- DependentPizzaStore depends on eight concrete Pizza subclasses

  - PizzaStore, however, depends on the Pizza interface

    - as do the Pizza subclasses

- In this design, PizzaStore (the high-level class) no longer depends on the Pizza subclasses (the low level classes); they both depend on the abstraction "Pizza". Nice.

Demonstration

- Lets look at some code

  - The FactoryMethod directory of this lecture's example source code contains an implementation of the pizza store using the factory method design pattern

    - It even includes a file called "DependentPizzaStore.java" that shows how the code would be implemented without using this pattern

  - DependentPizzaStore is dependent on 8 different concrete classes and 1 abstract interface (Pizza)

  - PizzaStore is dependent on just the Pizza abstract interface (nice!)

    - Each of its subclasses is only dependent on 4 concrete classes

      - furthermore, they shield the superclass from these dependencies

Moving On

- The factory method approach to the pizza store is a big success allowing our company to create multiple franchises across the country quickly and easily

    - But, bad news, we have learned that some of the franchises

        - while following our procedures (the abstract code in PizzaStore forces them to)

        - are skimping on ingredients in order to lower costs and increase margins

    - Our company's success has always been dependent on the use of fresh, quality ingredients

        - so "Something Must Be Done!" ®

Abstract Factory to the Rescue!

- We will alter our design such that a factory is used to supply the ingredients that are needed during the pizza creation process

  - Since different regions use different types of ingredients, we'll create region-specific subclasses of the ingredient factory to ensure that the right ingredients are used

  - But, even with region-specific requirements, since we are supplying the factories, we'll make sure that ingredients that meet our quality standards are used by all franchises

    - They'll have to come up with some other way to lower costs. ☺

First, We need a Factory Interface

```
 1  public interface PizzaIngredientFactory {
 2
 3      public Dough createDough();
 4      public Sauce createSauce();
 5      public Cheese createCheese();
 6      public Veggies[] createVeggies();
 7      public Pepperoni createPepperoni();
 8      public Clams createClam();
 9
10  }
11
```

Note the introduction of more abstract classes: Dough, Sauce, Cheese, etc.

Second, We implement a Region-Specific Factory

```
1   public class ChicagoPizzaIngredientFactory
2       implements PizzaIngredientFactory
3   {
4
5       public Dough createDough() {
6           return new ThickCrustDough();
7       }
8
9       public Sauce createSauce() {
10          return new PlumTomatoSauce();
11      }
12
13      public Cheese createCheese() {
14          return new MozzarellaCheese();
15      }
16
17      public Veggies[] createVeggies() {
18          Veggies veggies[] = { new BlackOlives(),
19                                new Spinach(),
20                                new Eggplant() };
21          return veggies;
22      }
23
24      public Pepperoni createPepperoni() {
25          return new SlicedPepperoni();
26      }
27
28      public Clams createClam() {
29          return new FrozenClams();
30      }
31  }
32
```

This factory ensures that quality ingredients are used during the pizza creation process…

… while also taking into account the tastes of people who live in Chicago

But how (or where) is this factory used?

41

Within Pizza Subclasses… (I)

```
 1  public abstract class Pizza {
 2      String name;
 3
 4      Dough dough;
 5      Sauce sauce;
 6      Veggies veggies[];
 7      Cheese cheese;
 8      Pepperoni pepperoni;
 9      Clams clam;
10
11      abstract void prepare();
12
13      void bake() {
14          System.out.println("Bake for 25 minutes at 350");
15      }
16
17      void cut() {
```

First, alter the Pizza abstract base class to make the prepare method abstract…

Within Pizza Subclasses… (II)

```java
 1  public class CheesePizza extends Pizza {
 2      PizzaIngredientFactory ingredientFactory;
 3
 4      public CheesePizza(PizzaIngredientFactory ingredientFactory) {
 5          this.ingredientFactory = ingredientFactory;
 6      }
 7
 8      void prepare() {
 9          System.out.println("Preparing " + name);
10          dough = ingredientFactory.createDough();
11          sauce = ingredientFactory.createSauce();
12          cheese = ingredientFactory.createCheese();
13      }
14  }
15
```

Then, update Pizza subclasses to make use of the factory! Note: we no longer need subclasses like NYCheesePizza and ChicagoCheesePizza because the ingredient factory now handles regional differences
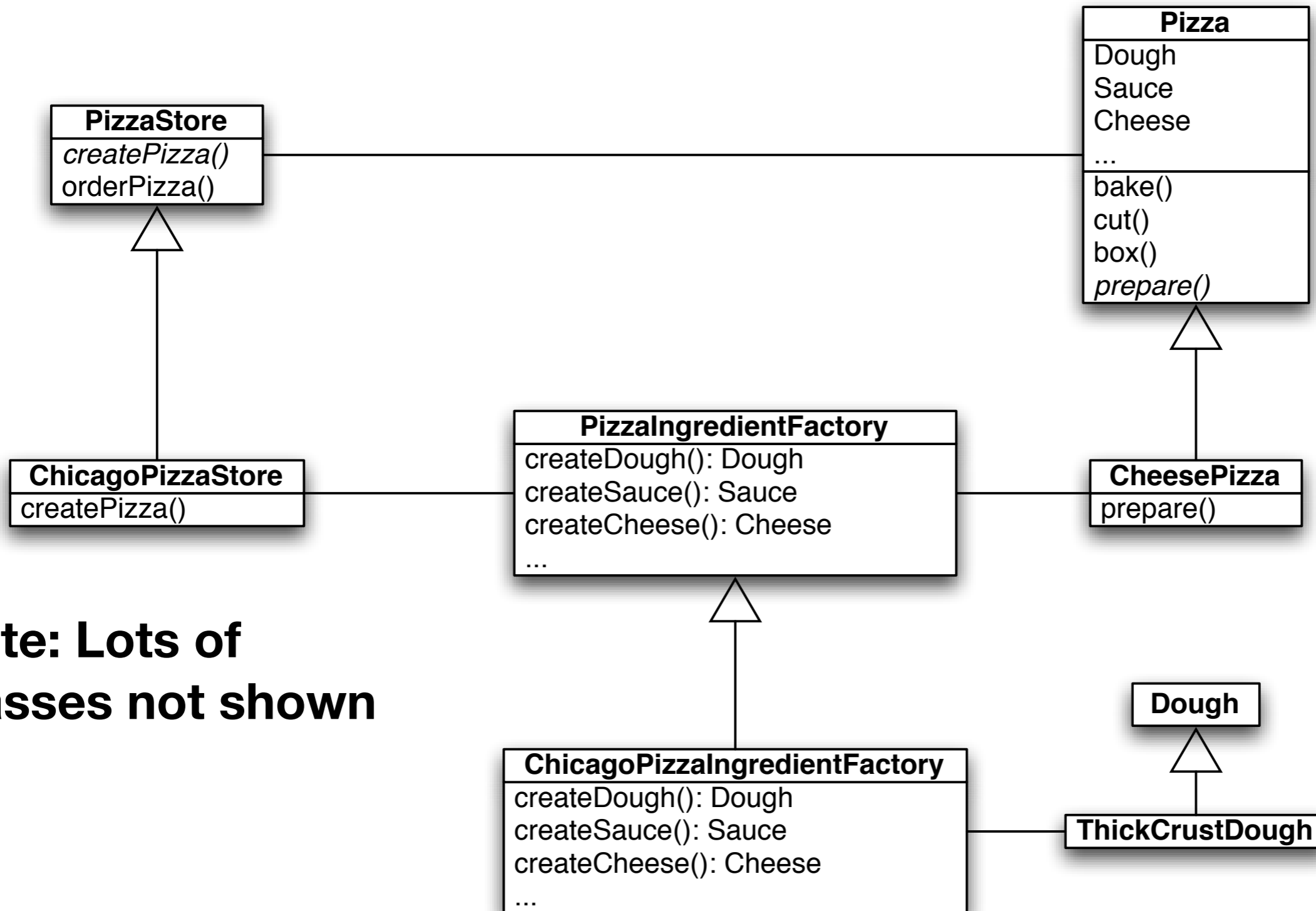
One last step…

```java
public class ChicagoPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
        new ChicagoPizzaIngredientFactory();

        if (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("Chicago Style Cheese Pizza");

        } else if (item.equals("veggie")) {

            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("Chicago Style Veggie Pizza");
                        ...
```

We need to update our PizzaStore subclasses to create the appropriate ingredient factory and pass it to each Pizza subclass in the createPizza factory method.

Summary: What did we just do?

1. We created an ingredient factory interface to allow for the creation of a family of ingredients for a particular pizza

2. This abstract factory gives us an interface for creating a family of products

   2.1.The factory interface decouples the client code from the actual factory implementations that produce context-specific sets of products

3. Our client code (PizzaStore) can then pick the factory appropriate to its region, plug it in, and get the correct style of pizza (Factory Method) with the correct set of ingredients (Abstract Factory)

Class Diagram of Abstract Factory Solution

Pizza
Dough
Sauce
Cheese
...
bake()
cut()
box()
prepare()

PizzaStore
createPizza()
orderPizza()

ChicagoPizzaStore
createPizza()

PizzaIngredientFactory
createDough(): Dough
createSauce(): Sauce
createCheese(): Cheese
...

CheesePizza
prepare()

Note: Lots of classes not shown

ChicagoPizzaIngredientFactory
createDough(): Dough
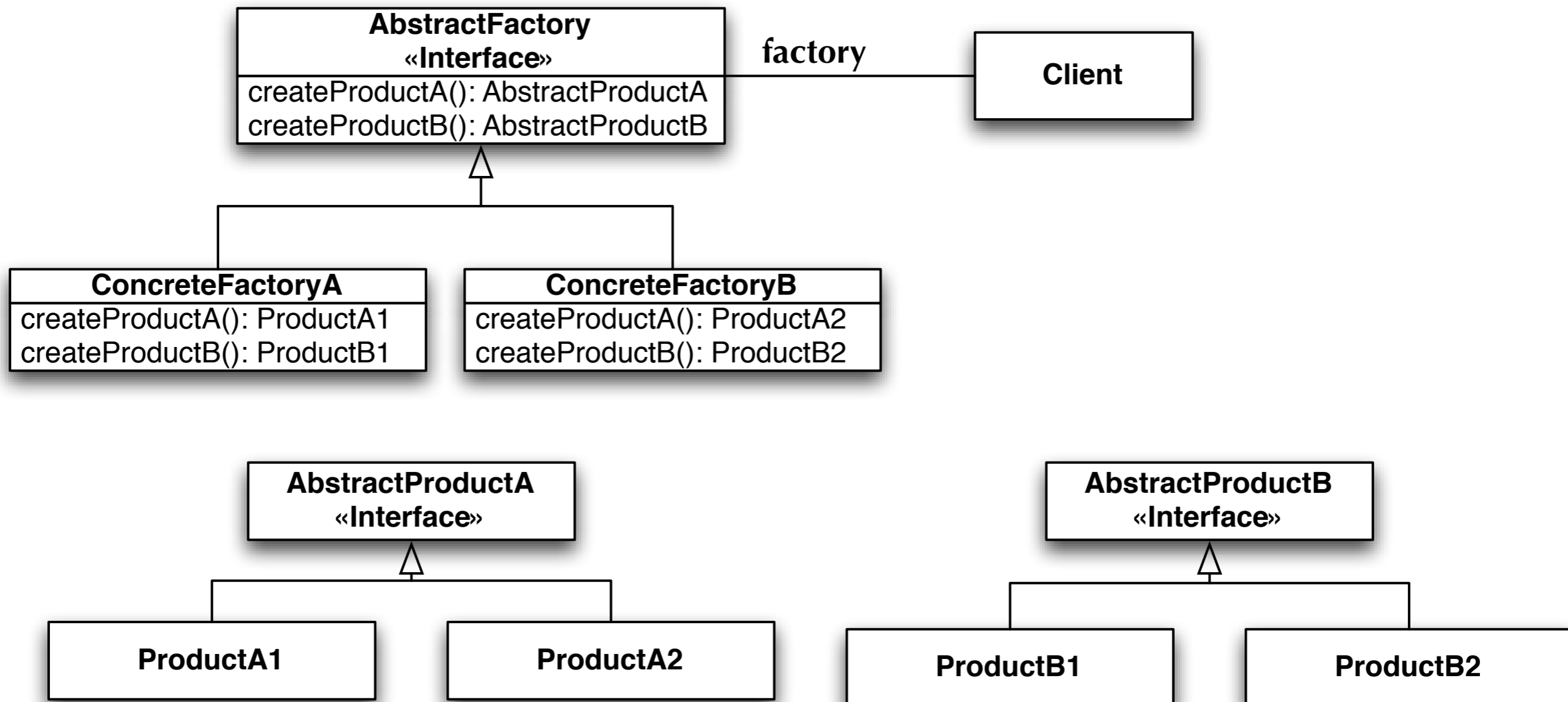createSauce(): Sauce
createCheese(): Cheese
...

Dough

ThickCrustDough

46

Demonstration

- Lets take a look at the code

Abstract Factory: Definition and Structure

- The abstract factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes

AbstractFactory
«Interface»
createProductA(): AbstractProductA
createProductB(): AbstractProductB

factory — **Client**

ConcreteFactoryA
createProductA(): ProductA1
createProductB(): ProductB1

ConcreteFactoryB
createProductA(): ProductA2
createProductB(): ProductB2

AbstractProductA
«Interface»

AbstractProductB
«Interface»

ProductA1

ProductA2

ProductB1

ProductB2

Wrapping Up

- We reviewed Strategy and worked on applying it

- We learned about Bridge and saw how it allows a set of abstractions to make use of multiple implementations in a scalable way

- We learned about Abstract Factory and how it enables the creation of families of objects while hiding the specific objects created from the clients that use them

Coming Up Next

- Lecture 10: Introduction to Java

- Lecture 11: Introduction to Android