

~~Strategy & Bridge & Factory~~

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 10A — 02/25/19

Task number one

- If you have your note card from the first class, please place it in front of you...
- If not...

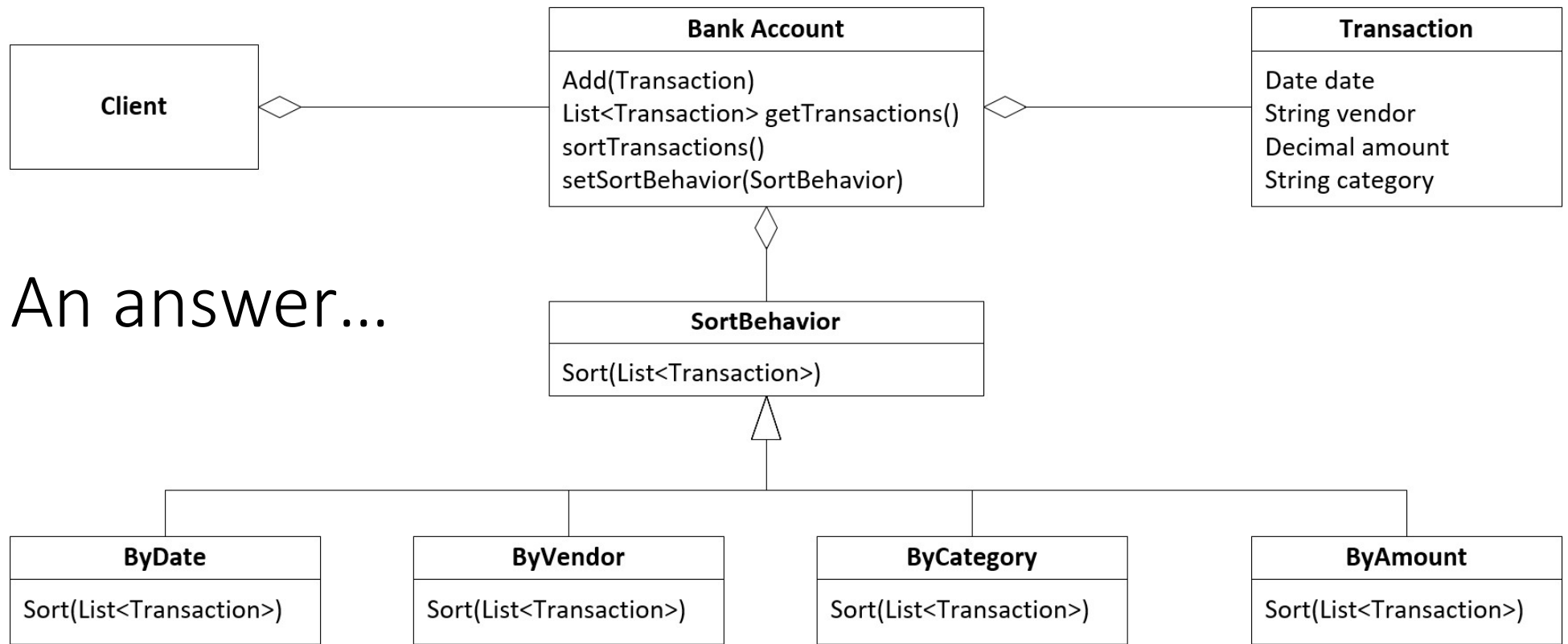
- Get a marker and a piece of card stock
- Fold it in half to make a little tent
- Write the name you'd like to be called on that
- Face the name towards me
- Try to remember to bring that to class for the next several weeks

Acknowledgement & Materials Copyright

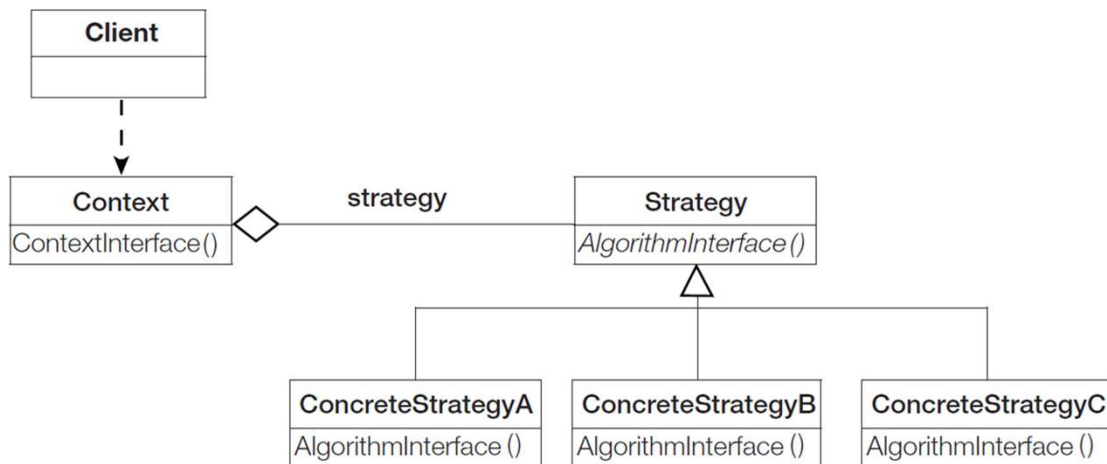
- Dr. Ken Anderson is a Professor of the Department of Computer Science and the Associate Dean for Education for the College of Engineering & Applied Science
- Ken taught this OOAD class on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class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

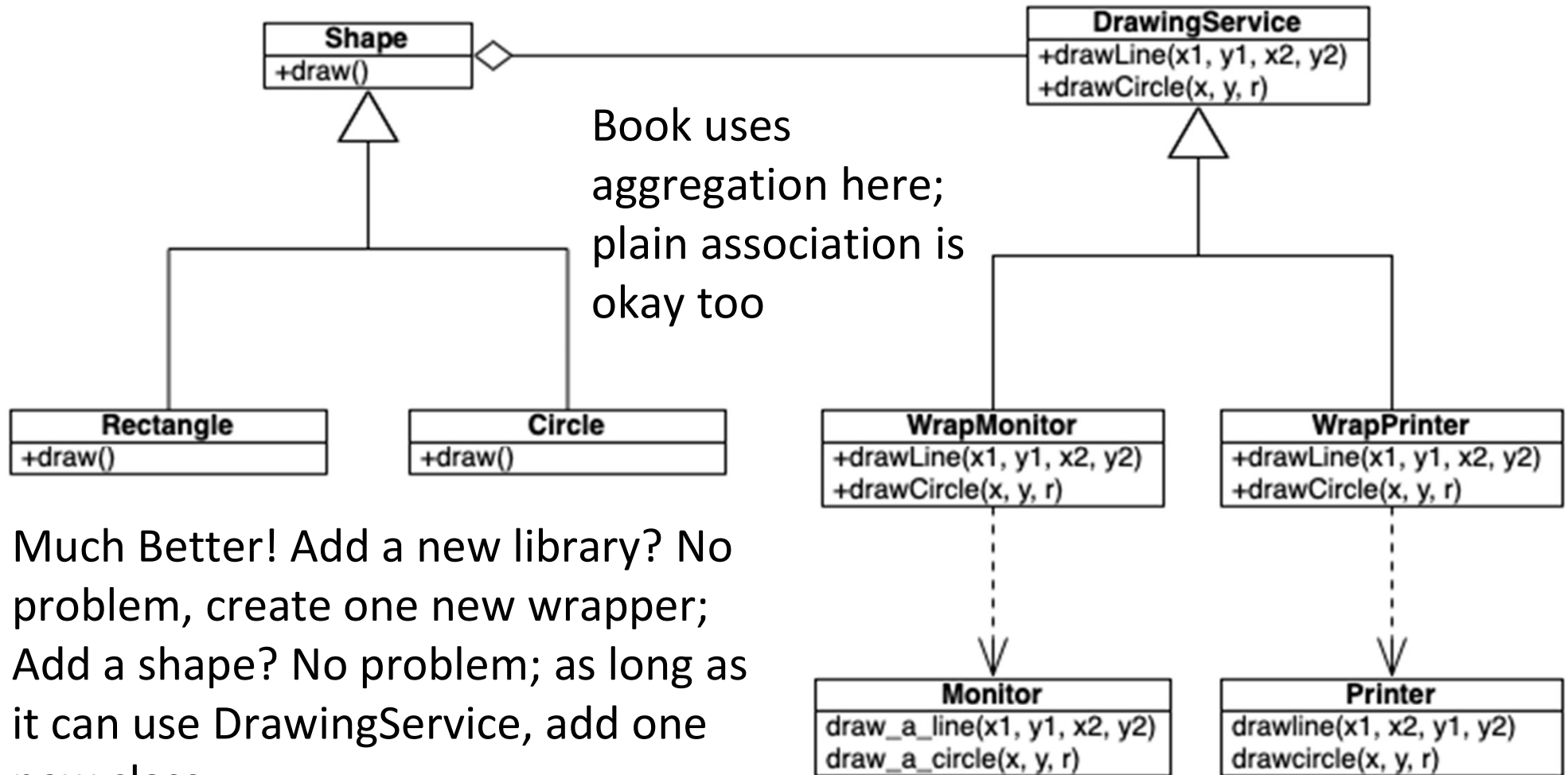
Goals of the Lecture

- Cover the material in Chapters 9, 10 & 11 of our textbook
 - Did The Strategy Pattern
 - Did The Bridge Pattern
 - Now, The Abstract Factory Pattern



Strategy
Pattern from
the Textbook

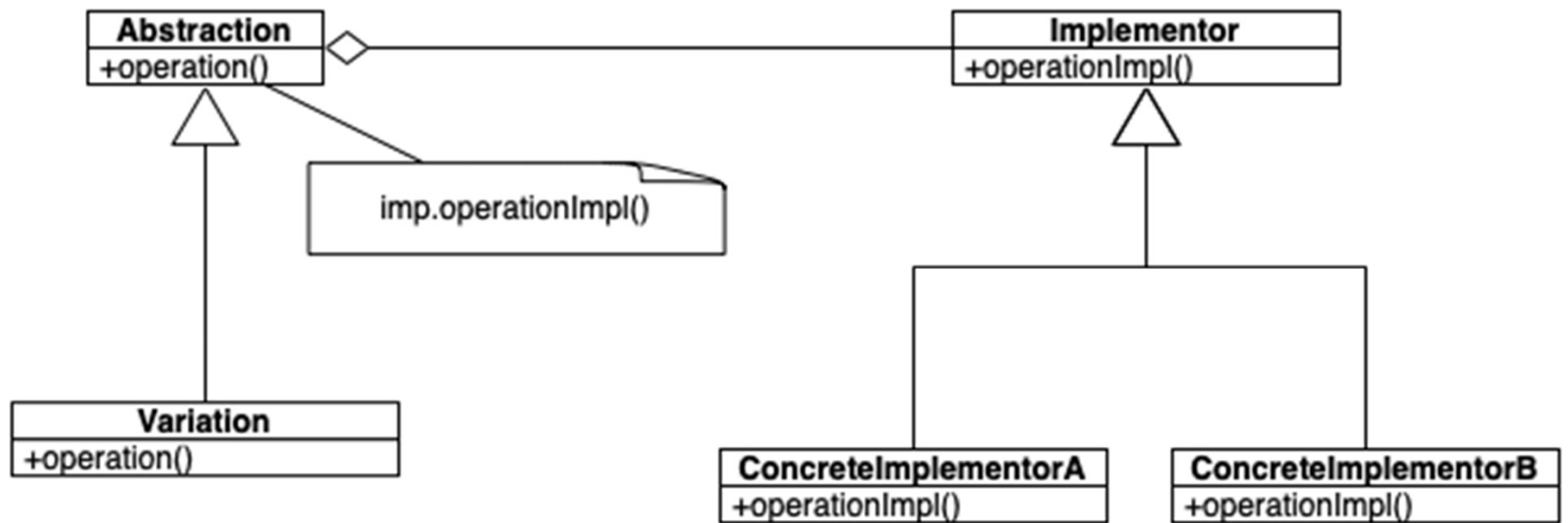




Much Better! Add a new library? No problem, create one new wrapper; Add a shape? No problem; as long as it can use DrawingService, add one new class

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

The Structure of the Bridge Pattern



Two Factory Patterns

- We'll 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 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**

PizzaStore Example

- We have a pizza store program that wants to separate the process of creating a pizza from the process of preparing/ordering a pizza
- Initial Code: mixes the two processes (see next slide)

```

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 earlier


Preparation

Note: excellent example of “coding to an interface”

Encapsulate Creation Code

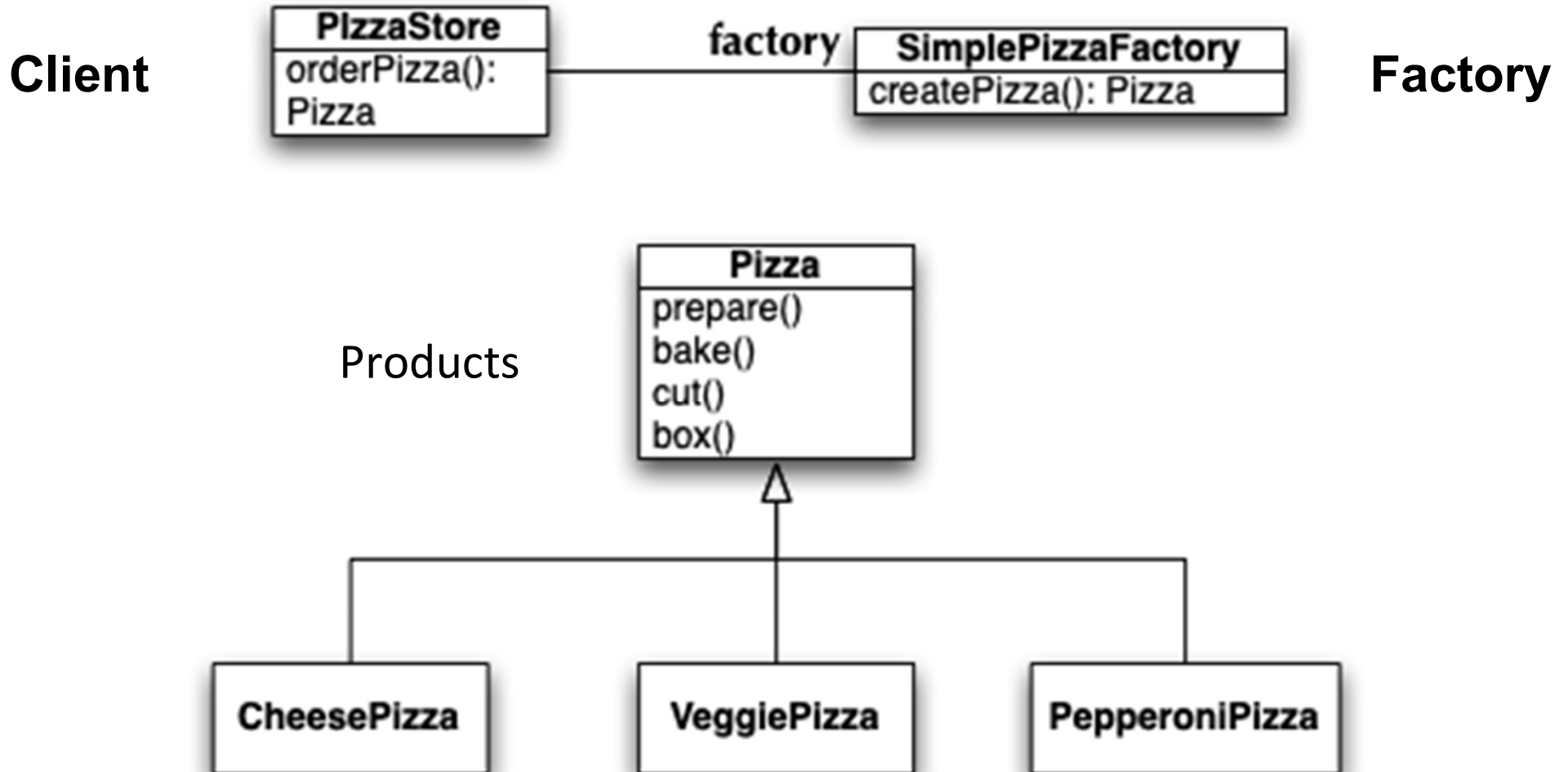
- A simple way to encapsulate this code is to put it in a separate class
 - That new class depends on the concrete classes, but those dependencies no longer impact the preparation code
 - See example next slide

```
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
```

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

- 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 needs its own factory to 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
```

Beautiful Abstract Base Class!

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.

New York Pizza Store

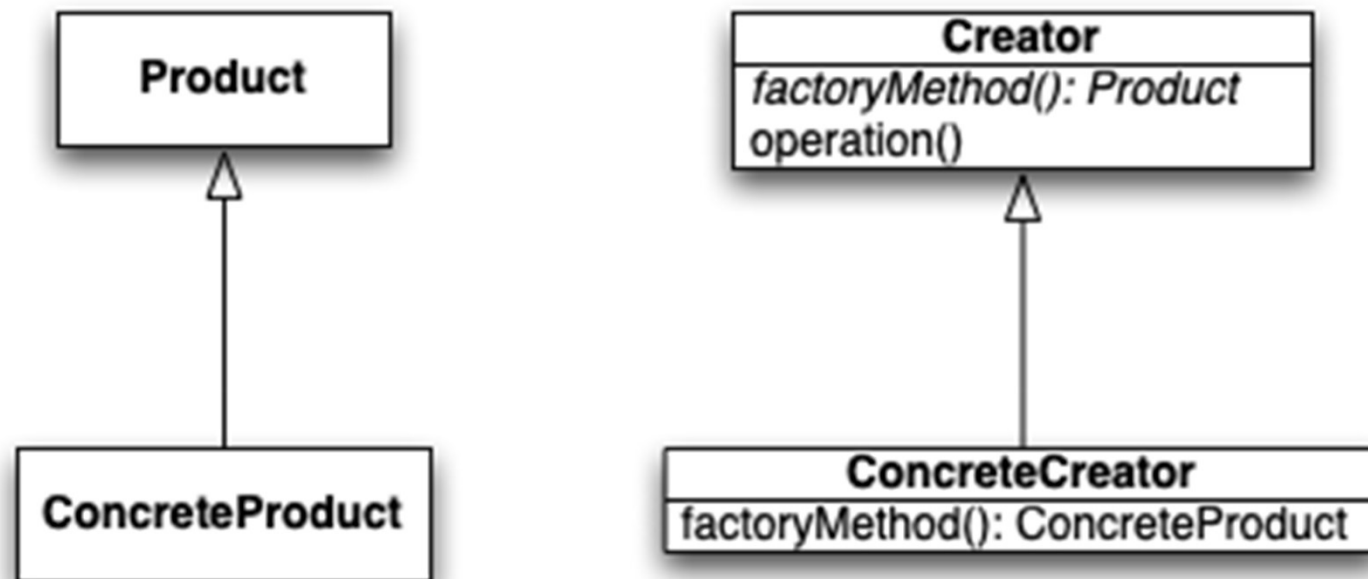
```
1 public class NYPizzaStore extends PizzaStore {  
2     public Pizza createPizza(String type) {  
3         if (type.equals("cheese")) {  
4             return new NYCheesePizza();  
5         } else if (type.equals("greek")) {  
6             return new NYGreekPizza();  
7         } else if (type.equals("pepperoni")) {  
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

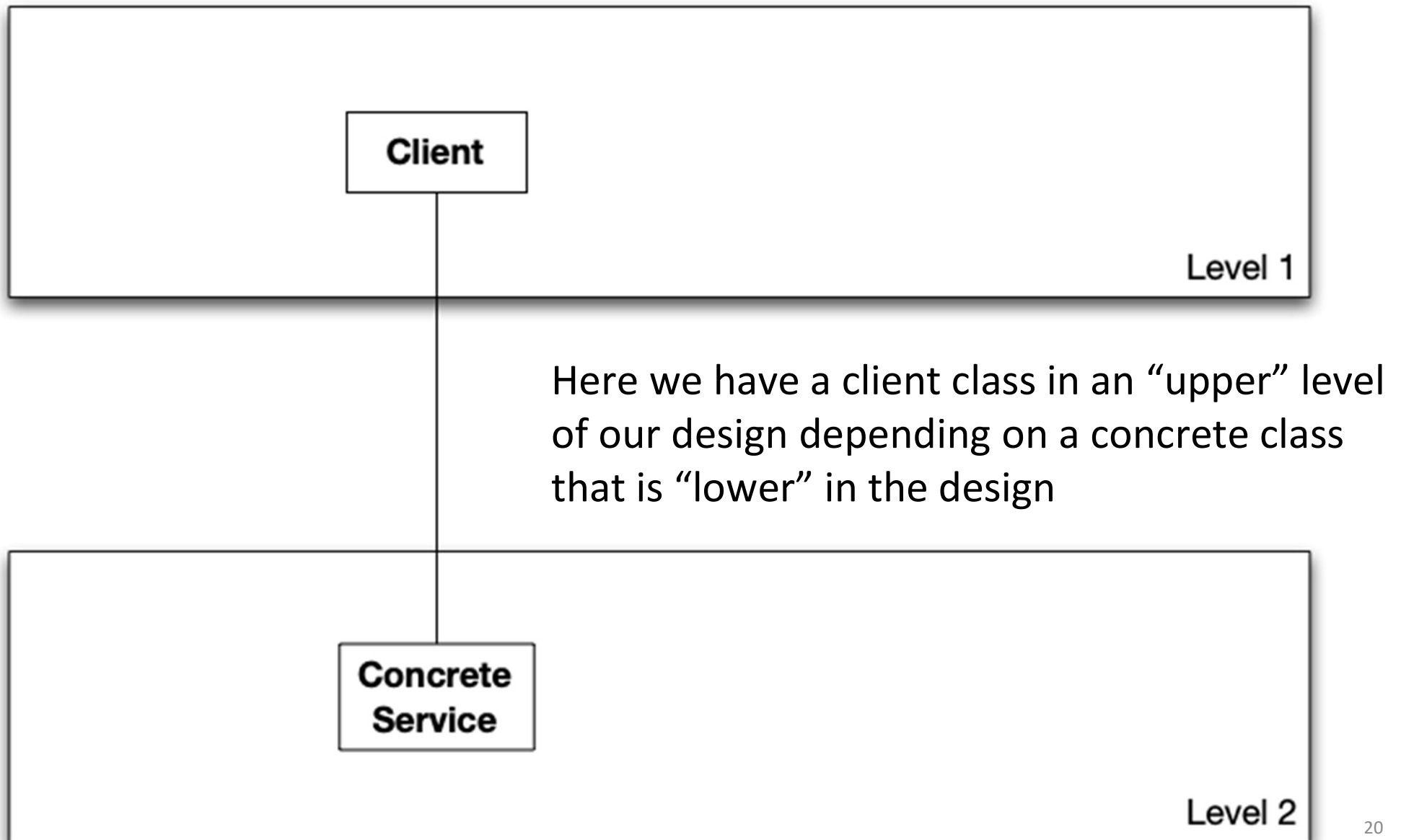


Factory Method leads to the creation of parallel class hierarchies; ConcreteCreators produce instances of ConcreteProducts that are operated on by Creators via the Product interface

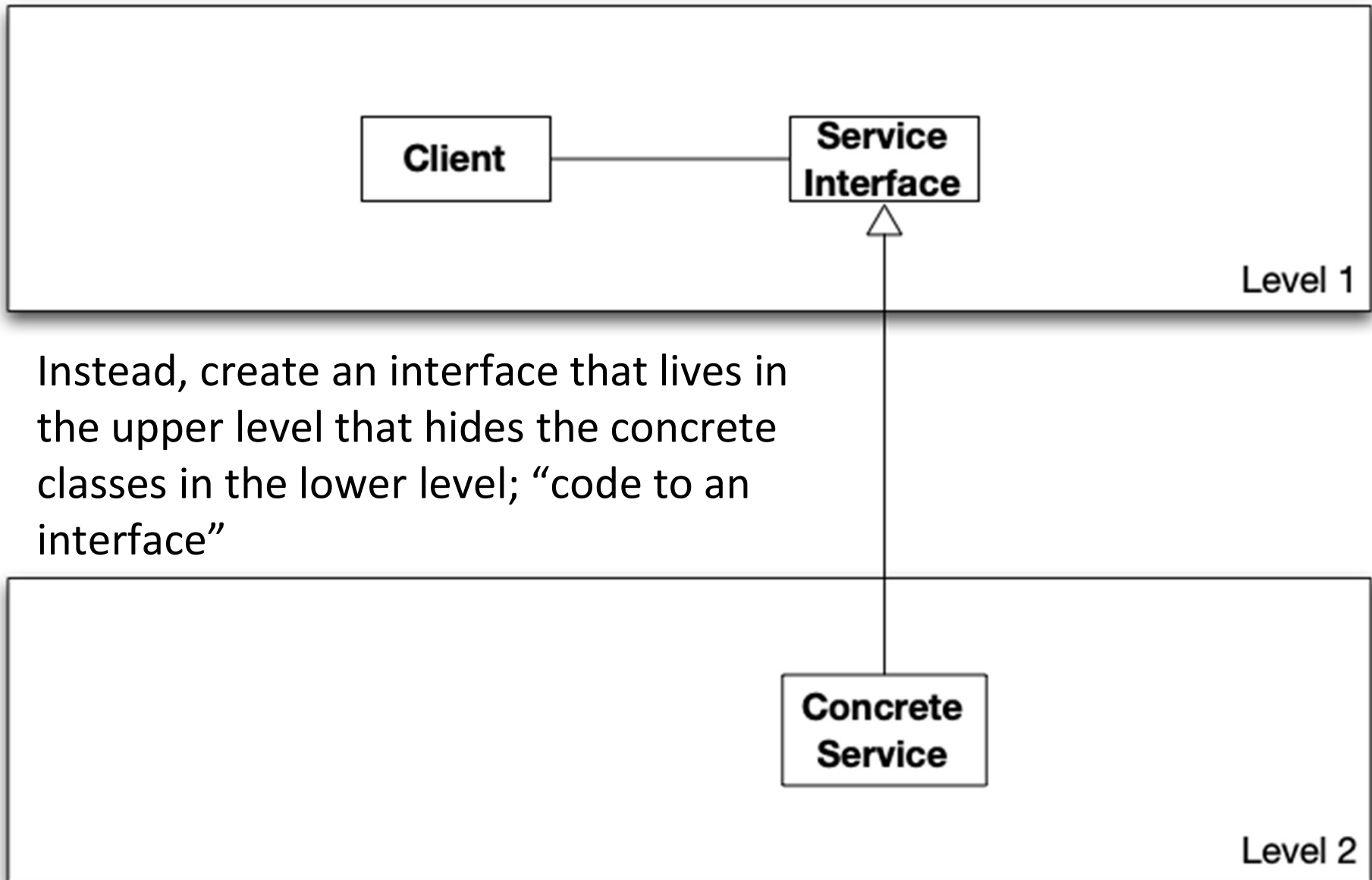
Dependency Inversion Principle (I)

- Factory Method is one way of following the dependency inversion principle – one of the OO principles
 - “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



Dependency Inversion Principle: Pictorially



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.

Java Implementation

- 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

Java code examples for the three Pizza examples of the Factory patterns are on Canvas under Files, Class Lectures, 10-Code Examples, Factory

Example: Dependent Pizza Store

```
public class DependentPizzaStore {
    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if ...
        }
    }
}
```

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

Example: Pizza Store

```
public abstract class PizzaStore {
```

```
    abstract Pizza createPizza(String item);
```

```
    public Pizza orderPizza(String type) {
```

```
        Pizza pizza = createPizza(type);
```

```
        System.out.println("--- Making a " + pizza.getName() + " ---");
```

```
        pizza.prepare();
```

```
        pizza.bake();
```

```
        pizza.cut();
```

```
        pizza.box();
```

```
        return 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?

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)

```
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...

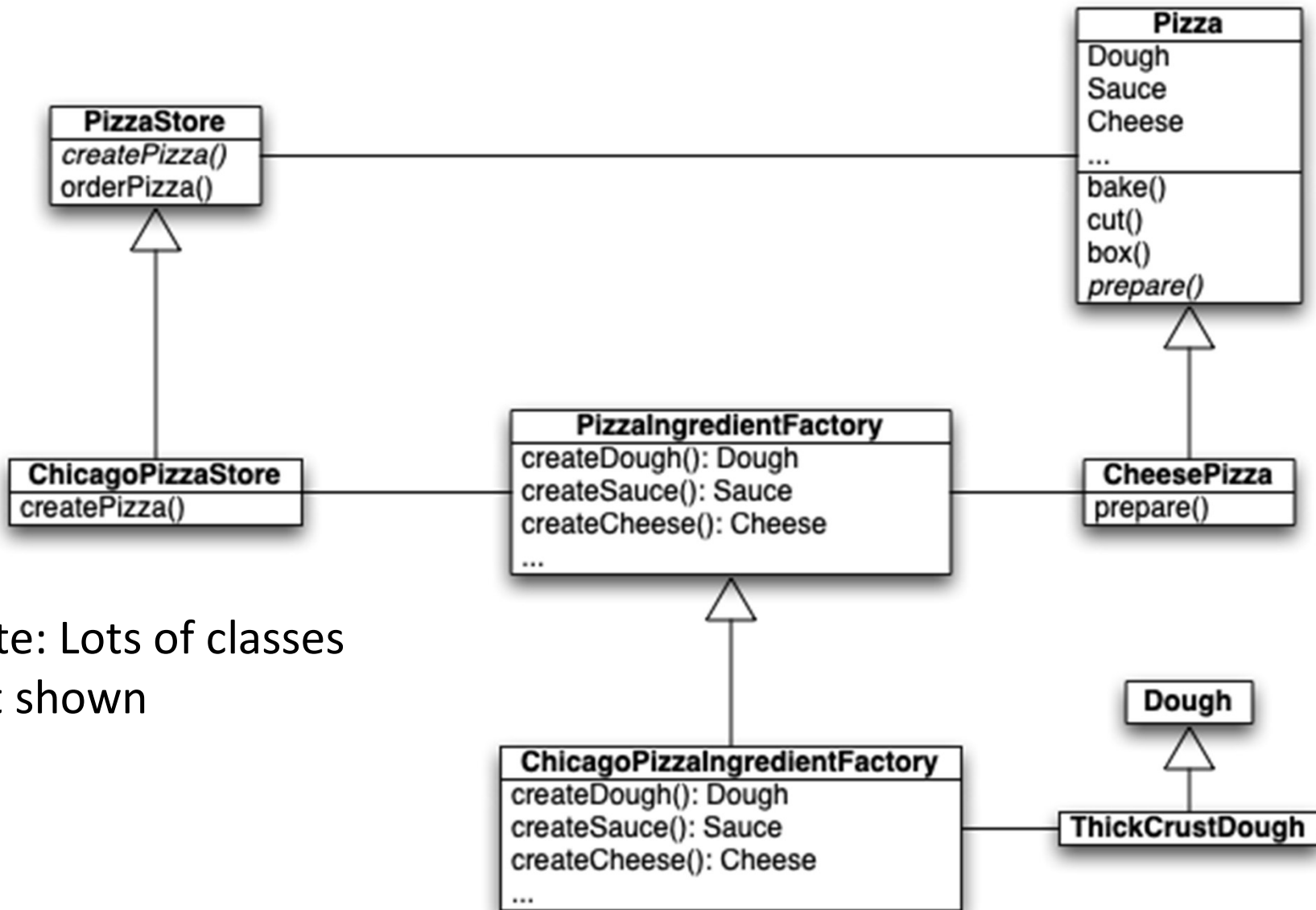
```
1 public class ChicagoPizzaStore extends PizzaStore {  
2  
3     protected Pizza createPizza(String item) {  
4         Pizza pizza = null;  
5         PizzaIngredientFactory ingredientFactory =  
6         new ChicagoPizzaIngredientFactory();  
7  
8         if (item.equals("cheese")) {  
9  
10            pizza = new CheesePizza(ingredientFactory);  
11            pizza.setName("Chicago Style Cheese Pizza");  
12  
13        } else if (item.equals("veggie")) {  
14  
15            pizza = new VeggiePizza(ingredientFactory);  
16            pizza.setName("Chicago Style Veggie Pizza");  
17  
            ...  
        }  
    }  
}
```

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?

- We created an ingredient factory interface to allow for the creation of a family of ingredients for a particular pizza
- This abstract factory gives us an interface for creating a family of products
 - The factory interface decouples the client code from the actual factory implementations that produce context-specific sets of products
- 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)

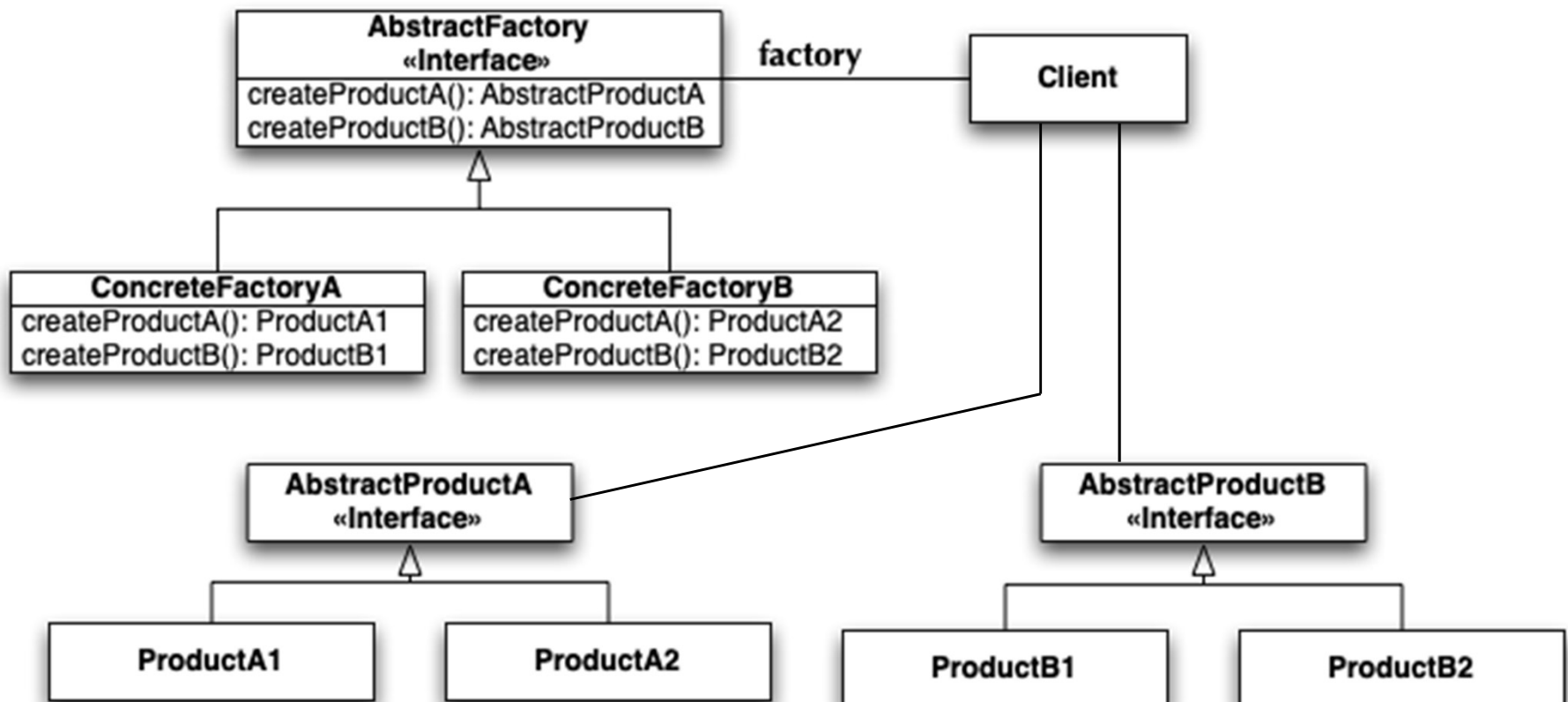
Partial Class Diagram of Abstract Factory Solution



Note: Lots of classes
not shown

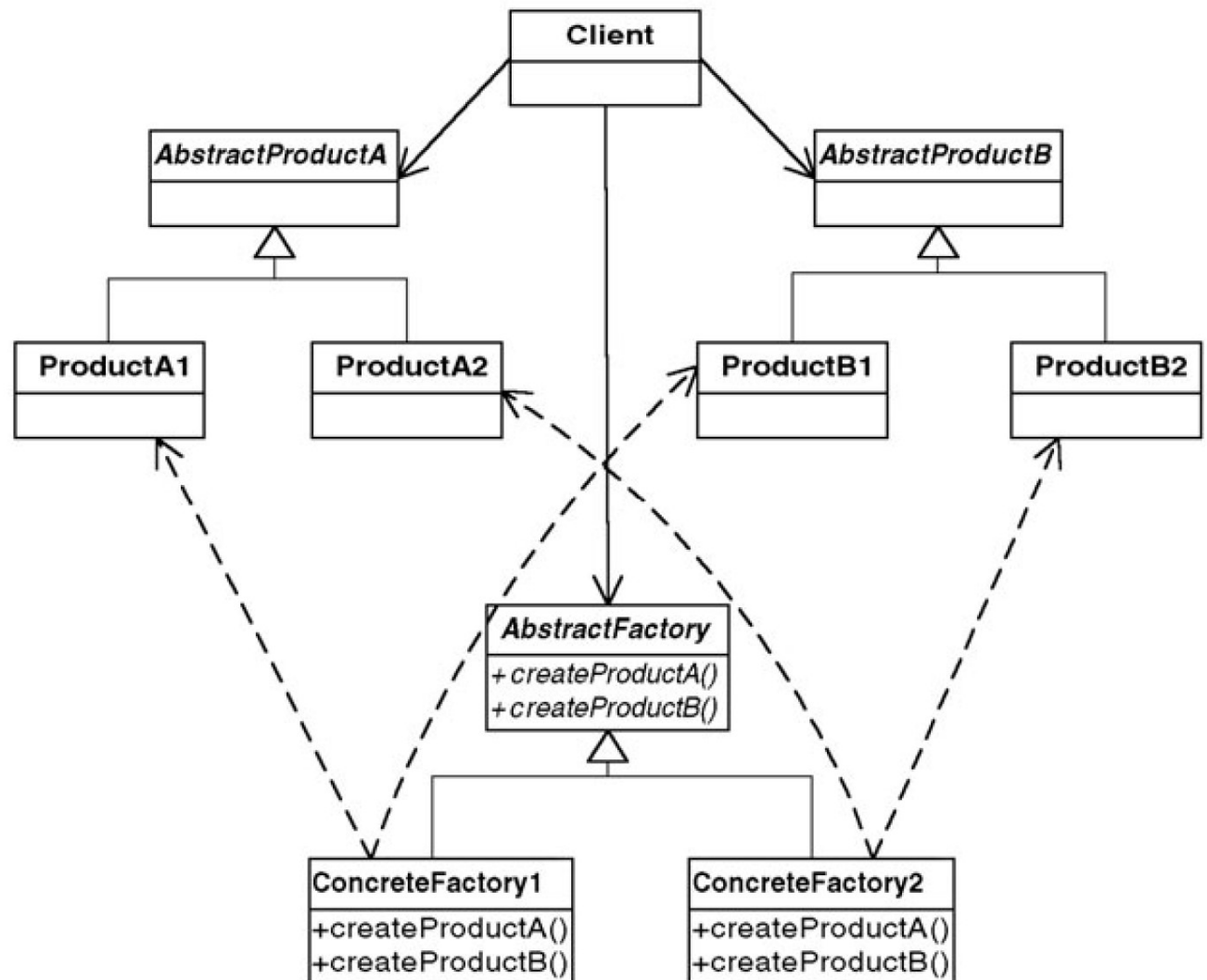
Abstract Factory: Definition and Structure

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



Abstract Factory: Another View

- From the textbook
- Problem: you need to instantiate families of related objects
- Solution: Take the rules for performing instantiation away from the client that's using the objects
- Take a look at the book's walk through this problem for Display and Print Drivers, including Java examples



Java code examples for the three Pizza examples of the Factory patterns are on Canvas under Files, Class Lectures, 10-Code Examples, Factory

Summary

- We recalled Strategy and Bridge
- We learned about Factory - defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory lets a class defer instantiation to subclasses
- We learned about Abstract Factory - it enables the creation of families of objects while hiding the specific objects created from the clients that use them
- Dependency Inversion Principle: Avoid dependencies on concrete types and strive for abstractions.

EXAM MATERIAL STOPS HERE

- This is the end of material that will be present on the midterm exam
- The lecture Friday 3/1 on OO in Java will not be part of the exam coverage

Possible topics for mid-term (I)

- Structured vs. Functional vs. OO Design & Programming
- Functional Decomposition
- Abstraction
- Encapsulation
- Classes, methods, attributes, instances, objects
- Inheritance (Is-A, superclass to subclass)
- Polymorphism
- Requirements (functional, non-functional, constraints)
- The “V”
- Cohesion & Coupling
- Conceptual, Specification, Implementation Perspectives
- Accessibility (Public, Protected, Private)
- Constructors/Destructors
- Abstract Classes
- Interfaces
- Collaborations
- UML Class, Sequence, State, Activity, Use Case Diagrams
- The WAVE rule for Use Cases
- Association & Multiplicity
- Aggregation, Composition, and Existence Dependency
- Qualification
- Delegation (Has-A)
- Design by Contract
- Object Identity and Equality

Possible topics for mid-term (II)

- Origins of design patterns
- Elements/features of patterns
- Benefits of learning and using patterns
- Multiple Inheritance issues and heuristics
- Proper design perspectives for objects, encapsulation, inheritance
- Commonality and Variability analysis
- OO Design and Agile
- OO Principles (9 from Head First Design Patterns, next page)
- Specific OO Patterns – UML, problem addressed, examples of use (when and why)
 - Strategy
 - Adapter
 - Façade
 - Bridge
 - Factory
 - Abstract Factory
- Code on Exam
 - No code writing
 - May ask you to read code
 - May ask for pseudo code
 - UML diagrams highly likely

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
- A class should have only one reason to change

Next Steps

- Optional additional material
 - Dr. Anderson's lecture on Strategy, Bridge, Abstract Factory
 - 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
 - Wednesday 2/27 – Recitation w/Manjunath (optional)
 - Good time to review anything you're fuzzy on for the midterm...
 - Friday 3/1 – Lecture: OO in Java, Homework 4: Design for Semester Project
 - Friday material is not on midterm
 - No Quiz prior to Exam
 - Monday 3/4 – Midterm Exam
- Things that are due
 - Grad Presentation Outline is due Mon 2/25 at 11 AM
 - Quiz 5 due Wed 2/27 11 AM, up on Saturday
 - Class Semester Project topic Canvas submission is due Friday 3/1 11 AM
 - Homework 3 is due Friday 3/1 11 AM
 - Distance students – be sure I have your proctor info
 - Accommodations for extended time exams – email me for room/time info