Decorator and Factory

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 20 — 10/30/2008

Lecture Goals

- Cover Material from Chapters 3 and 4 of the Design Patterns Textbook
    - Decorator Pattern
    - Factory and Abstract Factory Pattern

Decorator Pattern

- The Decorator Pattern provides a powerful mechanism for adding new behaviors to an object at run-time
  - The mechanism is based on the notion of “wrapping” which is just a fancy way of saying “delegation” but with the added twist that the delegator and the delegate both implement the same interface
    - You start with object A that implements abstract type X
    - You then create object B that also implements abstract type X
    - You pass A into B’s constructor and then pass B to A’s client
    - The client thinks its talking to A but its actually talking to B
    - B’s methods augment A’s methods to provide new behavior

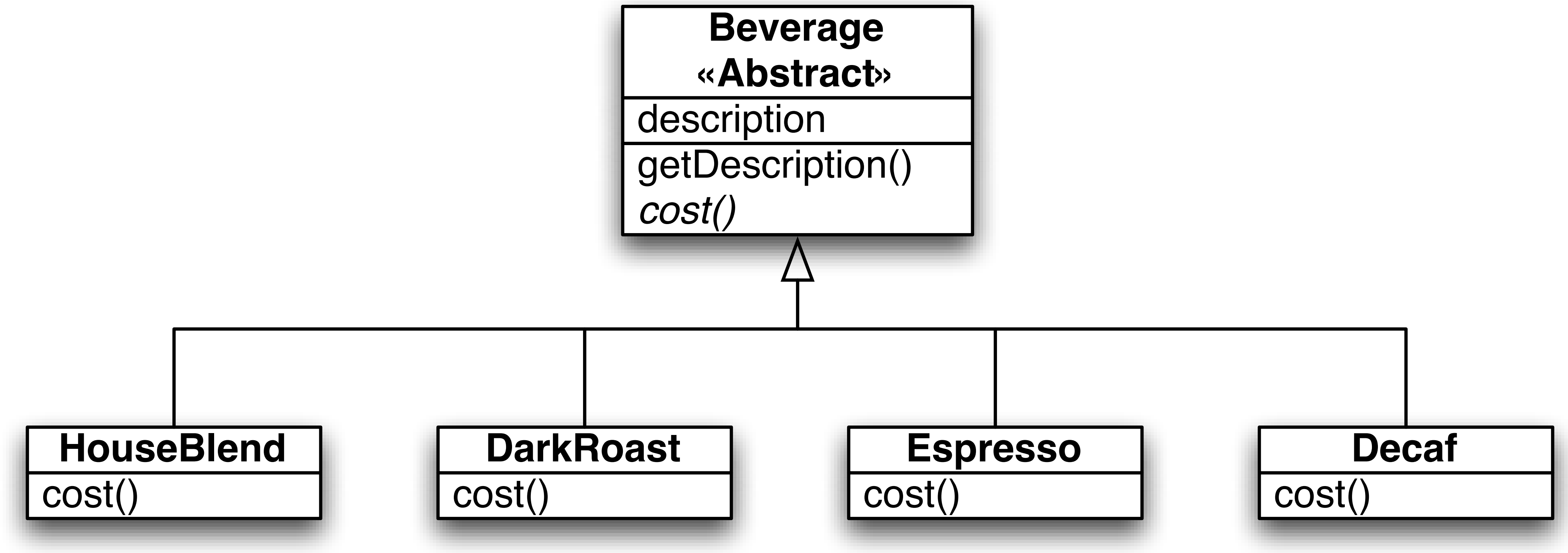
Why? Open-Closed Principle

- The decorator pattern provides yet another way in which a class's runtime behavior can be extended without requiring modification to the class
- This supports the goal of the open-closed principle:
    - Classes should be open for extension but closed to modification
        - Inheritance is one way to do this, but composition and delegation are more flexible (and Decorator takes advantage of delegation)
- Chapter 3's "Starbuzz Coffee" example clearly demonstrates why inheritance can get you into trouble and why delegation/composition provides greater run-time flexibility

Starbuzz Coffee

- Under pressure to update their “point of sale” system to keep up with their expanding set of beverage products
  - Started with a Beverage abstract base class and four implementations: HouseBlend, DarkRoast, Decaf, and Espresso
    - Each beverage can provide a description and compute its cost
  - But they also offer a range of condiments including: steamed milk, soy, and mocha
    - These condiments **alter** a beverage’s description and cost
      - “Alter” is a key word here since it provides a hint that we might be able to use the Decorator pattern

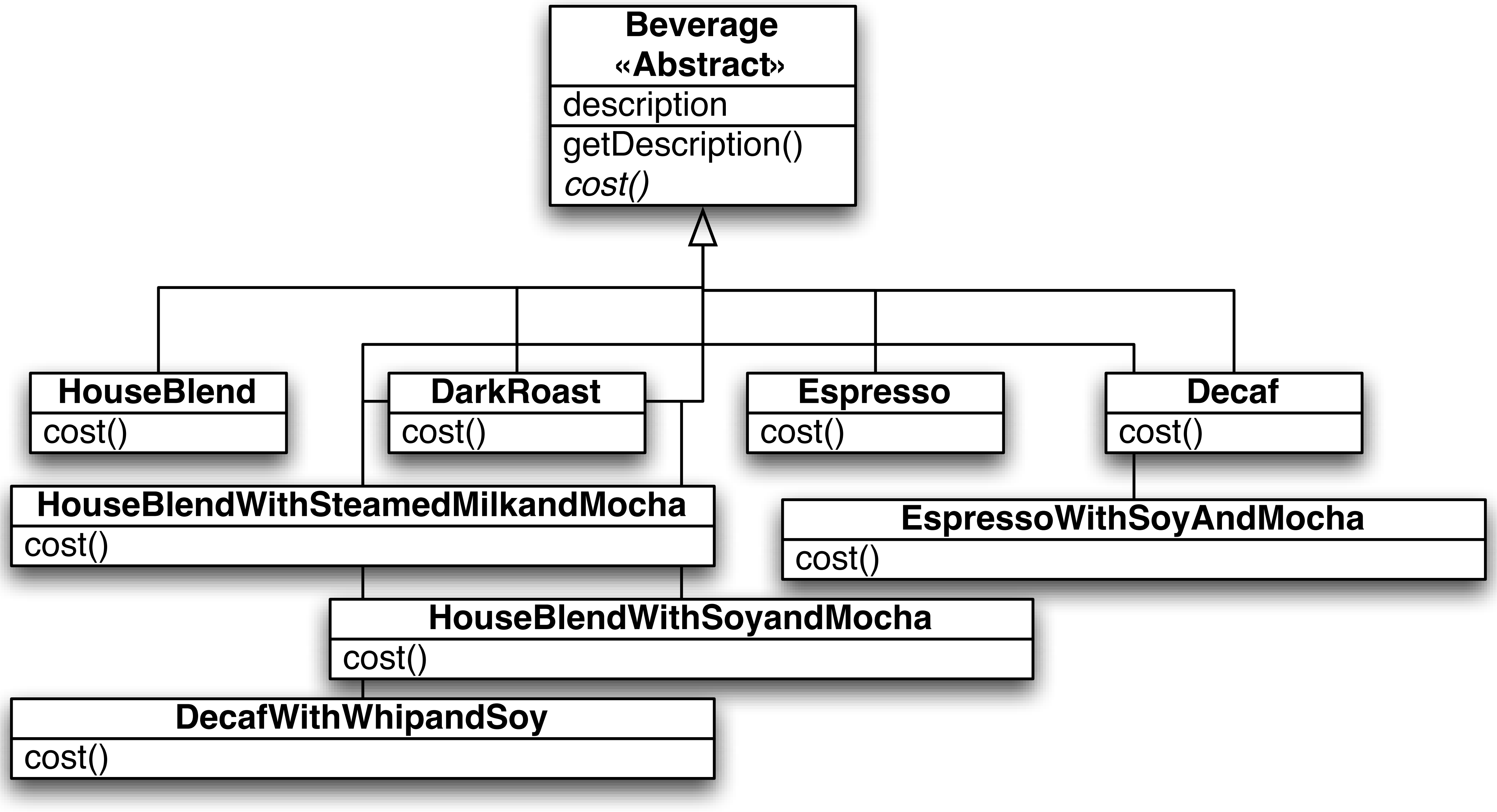
Initial Starbuzz System



With inheritance on your brain, you may add condiments to this design in one of two ways

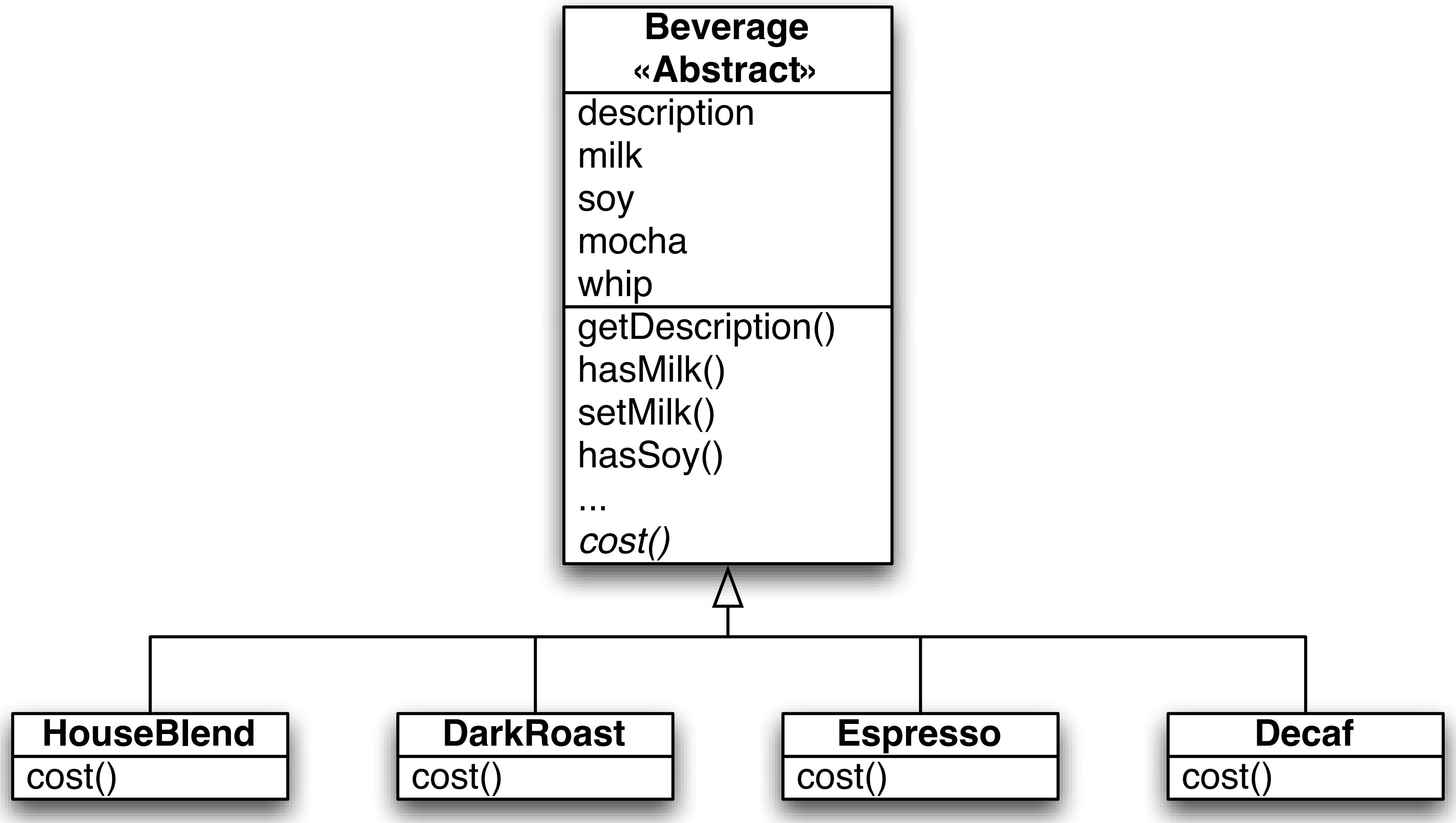
1. One subclass per combination of condiment (wont work in general but especially not in Boulder!)
2. Add condiment handling to the Beverage superclass

One Subclass per Combination



This is incomplete, but you can see the problem…
(see page 81 for a more complete picture)

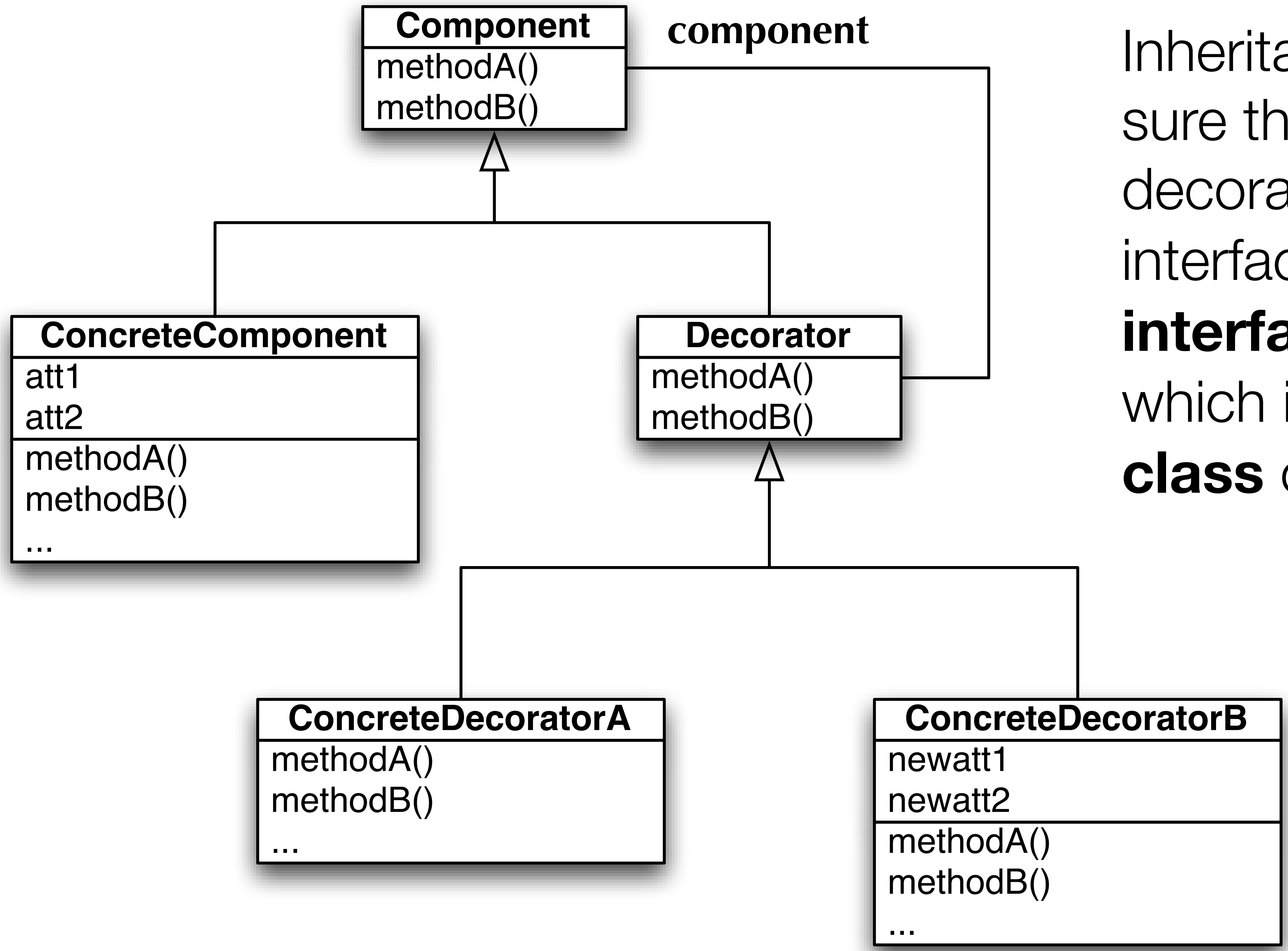
Let Beverage Handle Condiments



Houston, we have a problem…

1. This assumes that all concrete Beverage classes need these condiments
2. Condiments may vary (old ones go, new ones are added, price changes, etc.), shouldn't they be encapsulated some how?
3. How do you handle “double soy” drinks with boolean variables?

Decorator Pattern: Definition and Structure

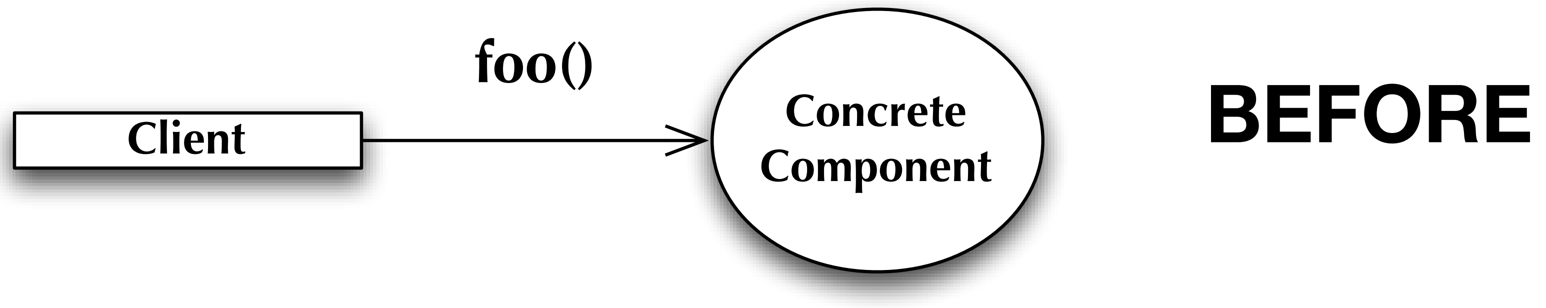


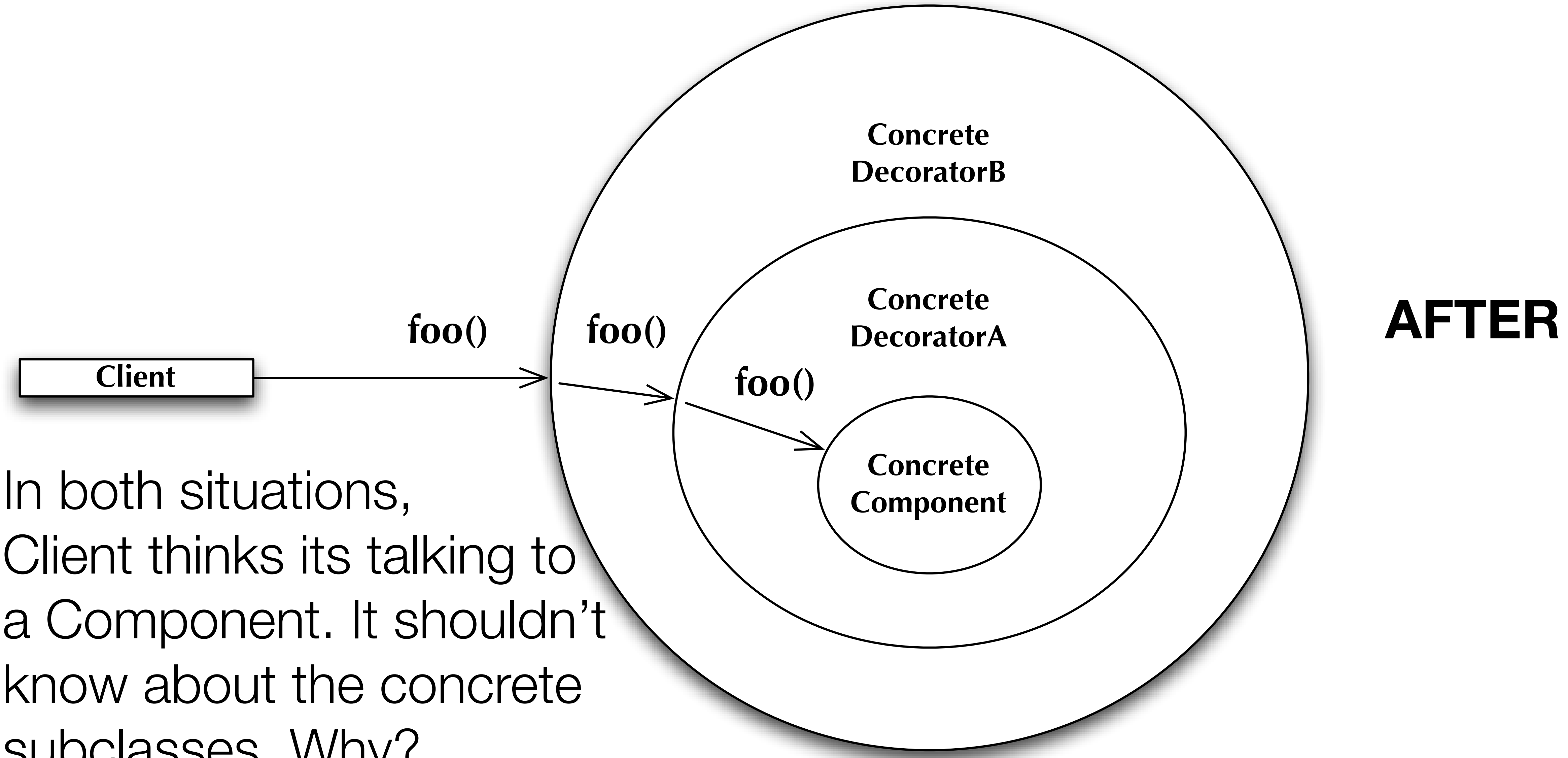
Inheritance is used to make sure that components and decorators **share** the same interface: namely the **public interface of Component** which is either an **abstract class** or an **interface**

At run-time, concrete decorators **wrap** concrete components and/or other concrete decorators

The object to be wrapped is typically passed in **via the constructor**

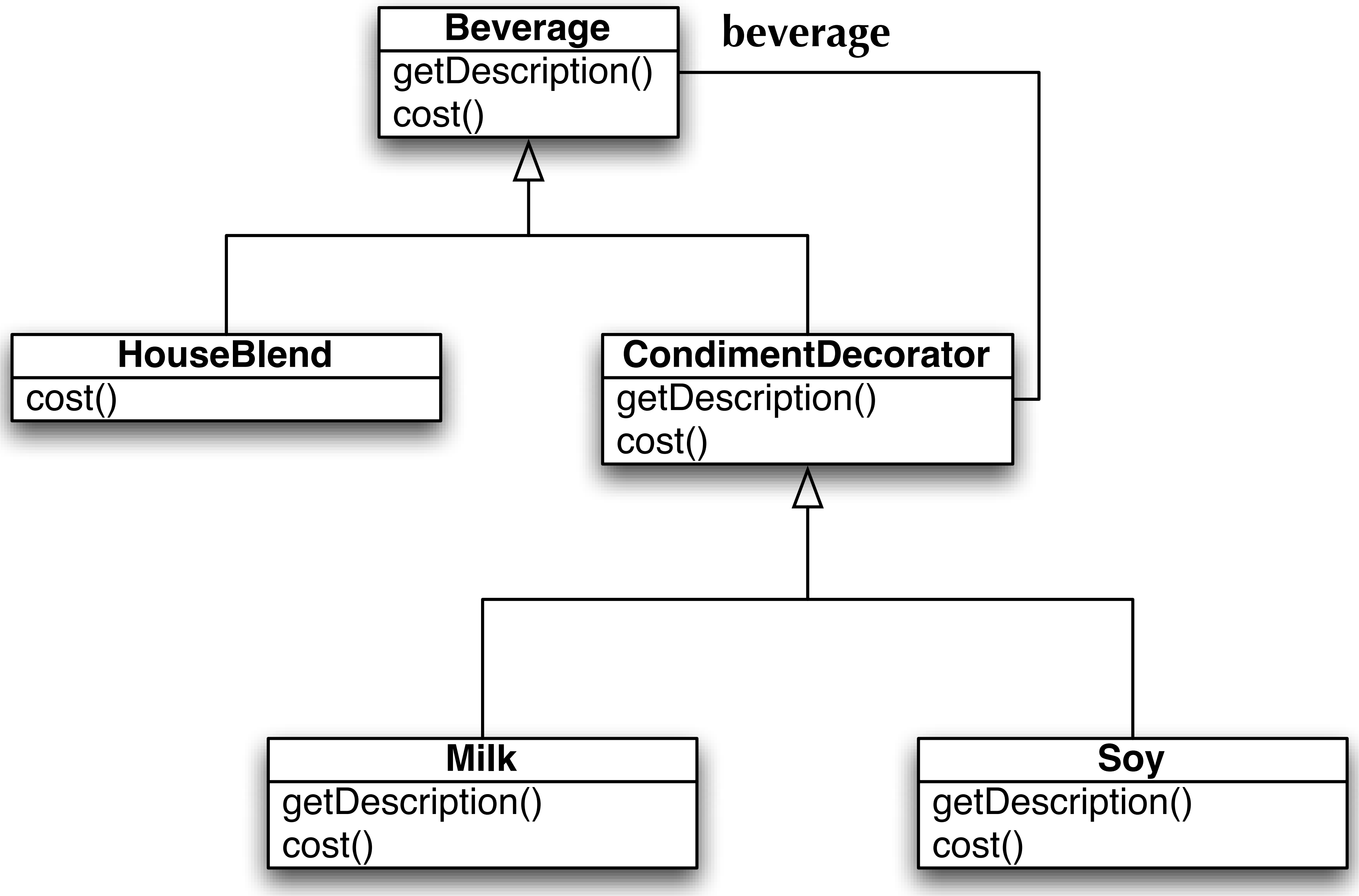
Client Perspective





In both situations,
Client thinks its talking to
a Component. It shouldn't
know about the concrete
subclasses. Why?

StarBuzz Using Decorators (Incomplete)



Demonstration

- Starbuzz Example
- Use of Decorator Pattern in java.io package
    - InputStream == Component
    - FilterInputStream == Decorator
    - FileInputStream, StringBufferInputStream, etc. == ConcreteComponent
    - BufferedInputStream, LineNumberInputStream, etc. == ConcreteDecorator

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”, in this case “Duck”, 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 open-closed principle and 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

```java
public class PizzaStore {

    Pizza orderPizza(String type) {

        Pizza pizza;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek")) {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        }

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

}
```

Creation

Creation code has all the same problems as the code on the previous slide

Preparation

Note: excellent example of "coding to an interface"

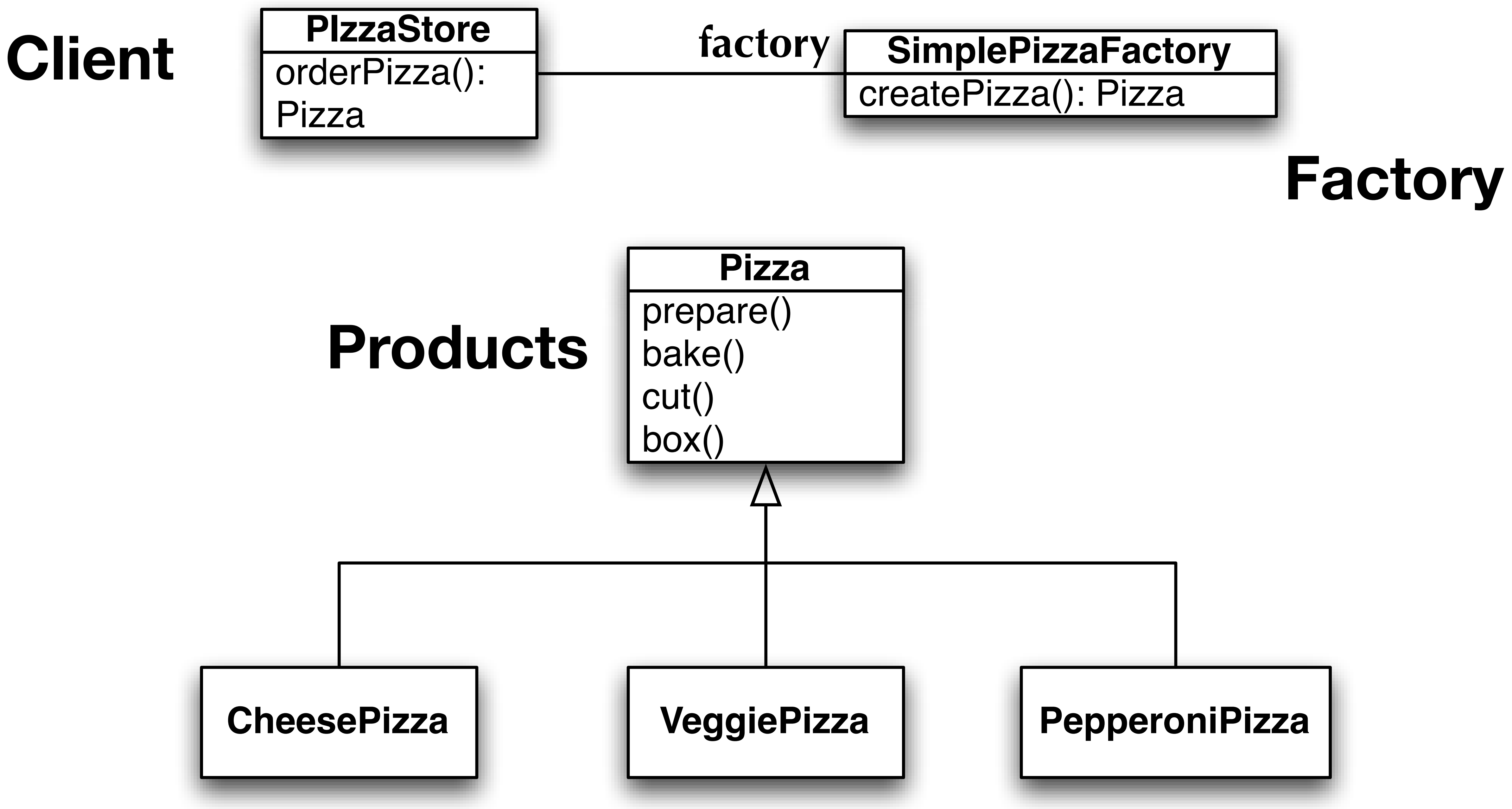
Encapsulate Creation Code

- 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

```java
public class PizzaStore {

    private SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {

        Pizza pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

}

```

```java
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new CheesePizza();
        } else if (type.equals("greek")) {
            return new GreekPizza();
        } else if (type.equals("pepperoni")) {
            return new PepperoniPizza();
        }
    }

}

```

Class Diagram of New Solution



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

```java
public abstract class PizzaStore {

    protected abstract createPizza(String type);

    public Pizza orderPizza(String type) {

        Pizza pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

}

```

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

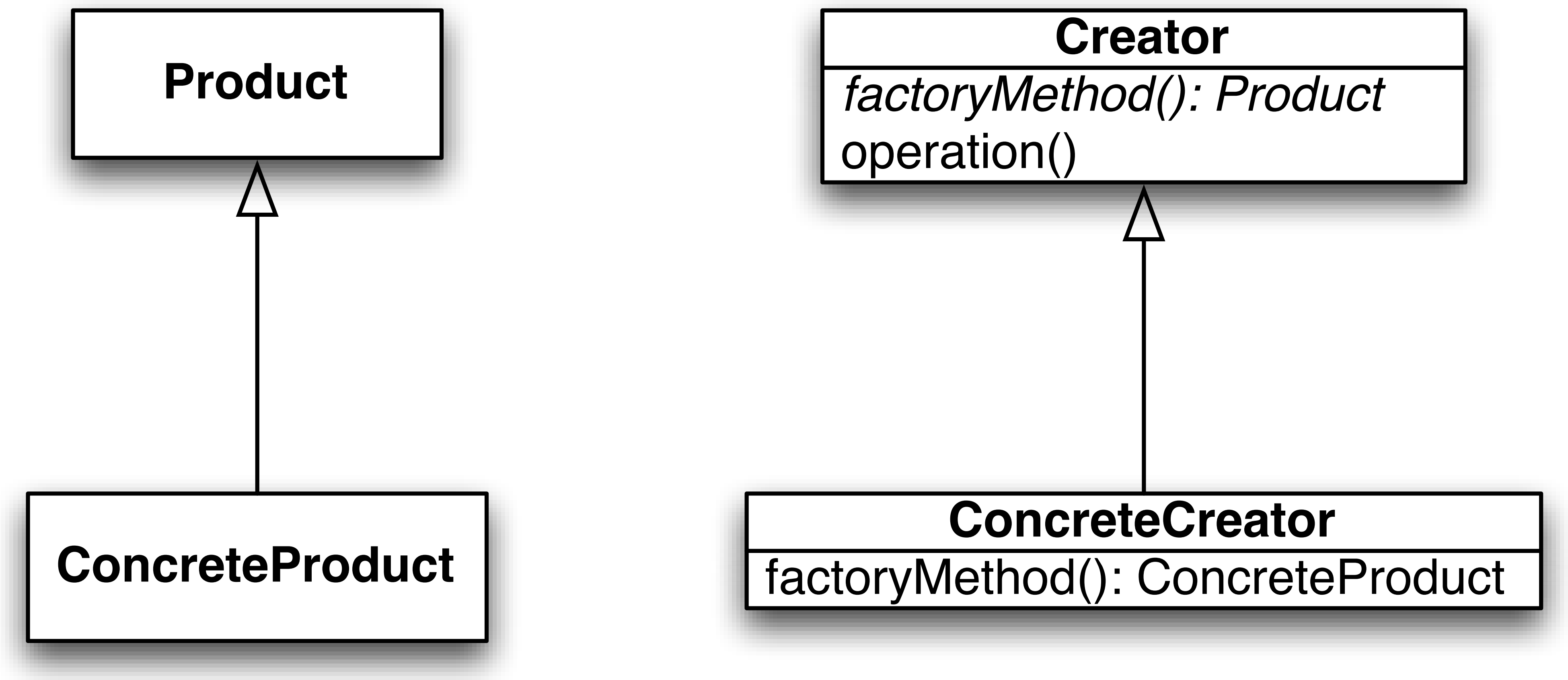
```java
public class NYPizzaStore extends PizzaStore {
    public Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new NYCheesePizza();
        } else if (type.equals("greek")) {
            return new NYGreekPizza();
        } else if (type.equals("pepperoni")) {
            return new NYPepperoniPizza();
        }
        return null;
    }
}

```

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 Creator's via the Product interface

Dependency Inversion Principle

- 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.

Dependency Inversion Principle: How To?

- To achieve the dependency inversion principle in your own designs, follow these GUIDELINES
  - No variable should hold a reference to a concrete class
  - No class should derive from a concrete class
  - No method should override an implemented method of its base classes
- These are guidelines because if you were to blindly follow these instructions, you would never produce a system that could be compiled or executed
  - Instead use them as instructions to help optimize your design
- And remember, not only should low-level classes depend on abstractions, but high-level classes should to… this is the very embodiment of “code to an interface”

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

```java
public interface PizzaIngredientFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();

}

```

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

Second, We implement a Region-Specific Factory

```
public class ChicagoPizzaIngredientFactory
    implements PizzaIngredientFactory
{

    public Dough createDough() {
        return new ThickCrustDough();
    }

    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }

    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
                              new Spinach(),
                              new Eggplant() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FrozenClams();
    }
}

```

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)

```java
public abstract class Pizza {
    String name;

    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
```

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

Within Pizza Subclasses… (II)

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}

```

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?

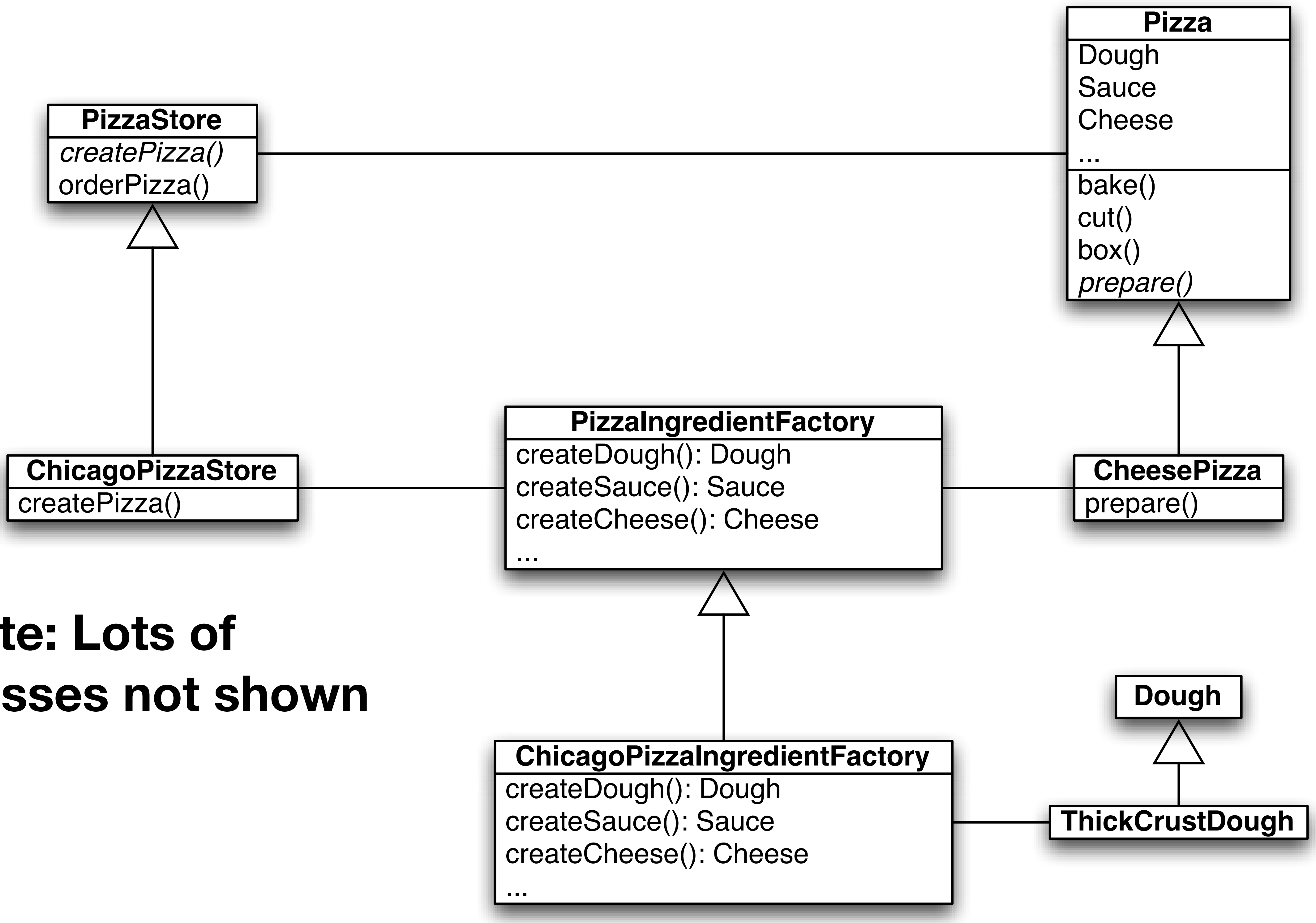
1. We created an ingredient factory interface to allow for the creation of a family of ingredients for a particular pizza

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



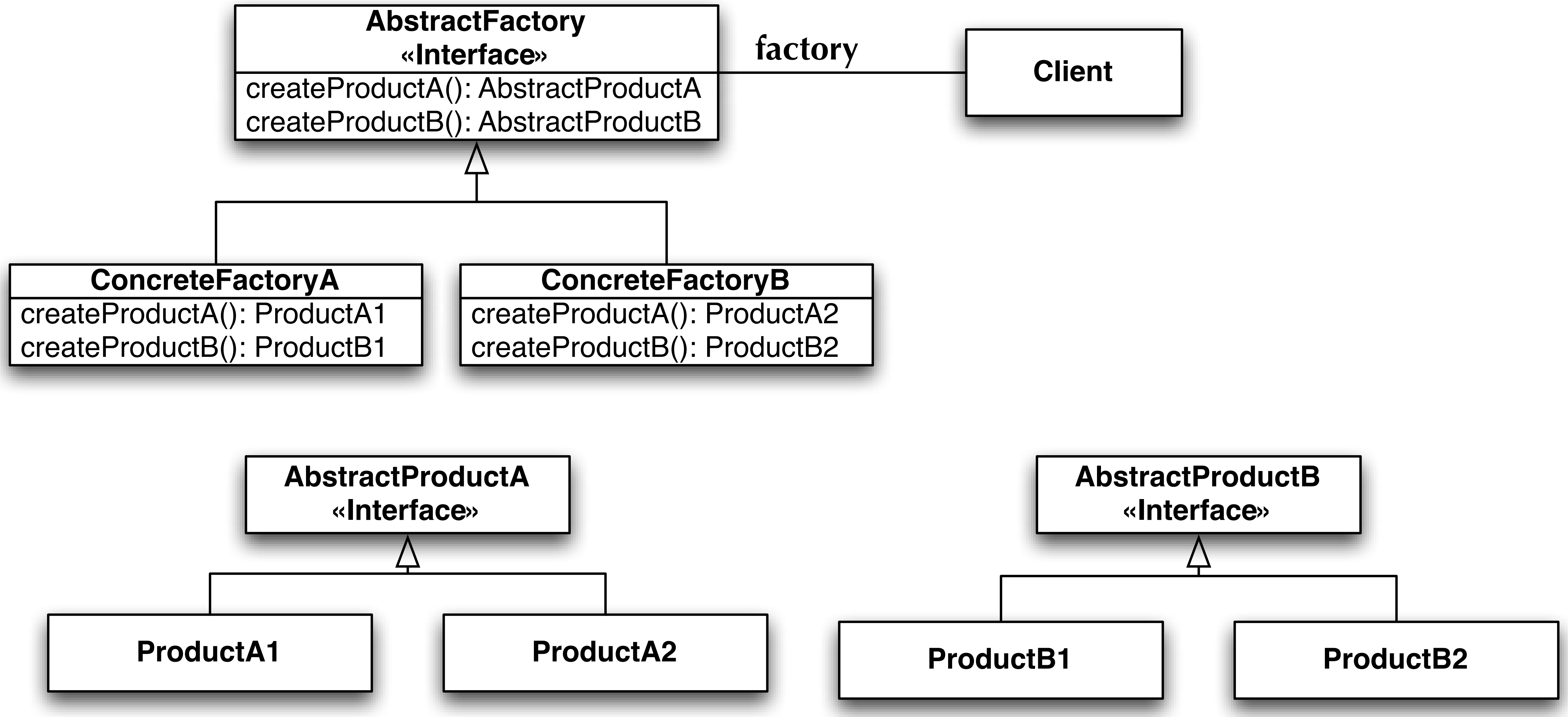
Note: Lots of classes not shown

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

- Decorator
  - Way to implement open-closed principle that
    - makes use of inheritance to share an interface between a set of components and a set of decorators
    - makes use of composition and delegation to dynamically wrap decorators around components at run-time
- All factories encapsulate object creation
  - Simple Factory is not a true pattern but is simple to understand/implement
  - Factory Method relies on inheritance: object creation occurs in subclasses
  - Abstract Factory relies on composition: object creation occurs in concrete factories
- Both can be used to shield your applications from concrete classes
  - And both help apply the dependency inversion principle in your designs

Coming Up Next

- Lecture 21: Singleton, Command & Adaptor Patterns
    - Read Chapters 5 — 7 of the Design Patterns Textbook
- Lecture 22: Template Methods, Iterator & Composite
    - Read Chapters 8 — 10 of the Design Patterns Textbook