

Factory

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 14

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD 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

Two Factory Patterns

- We'll use an example from Head First Design Patterns to introduce the concepts of (Simple Factory), Factory, and Abstract Factory

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”

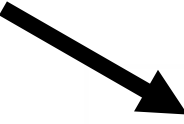
Encapsulate Creation Code

- 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

```

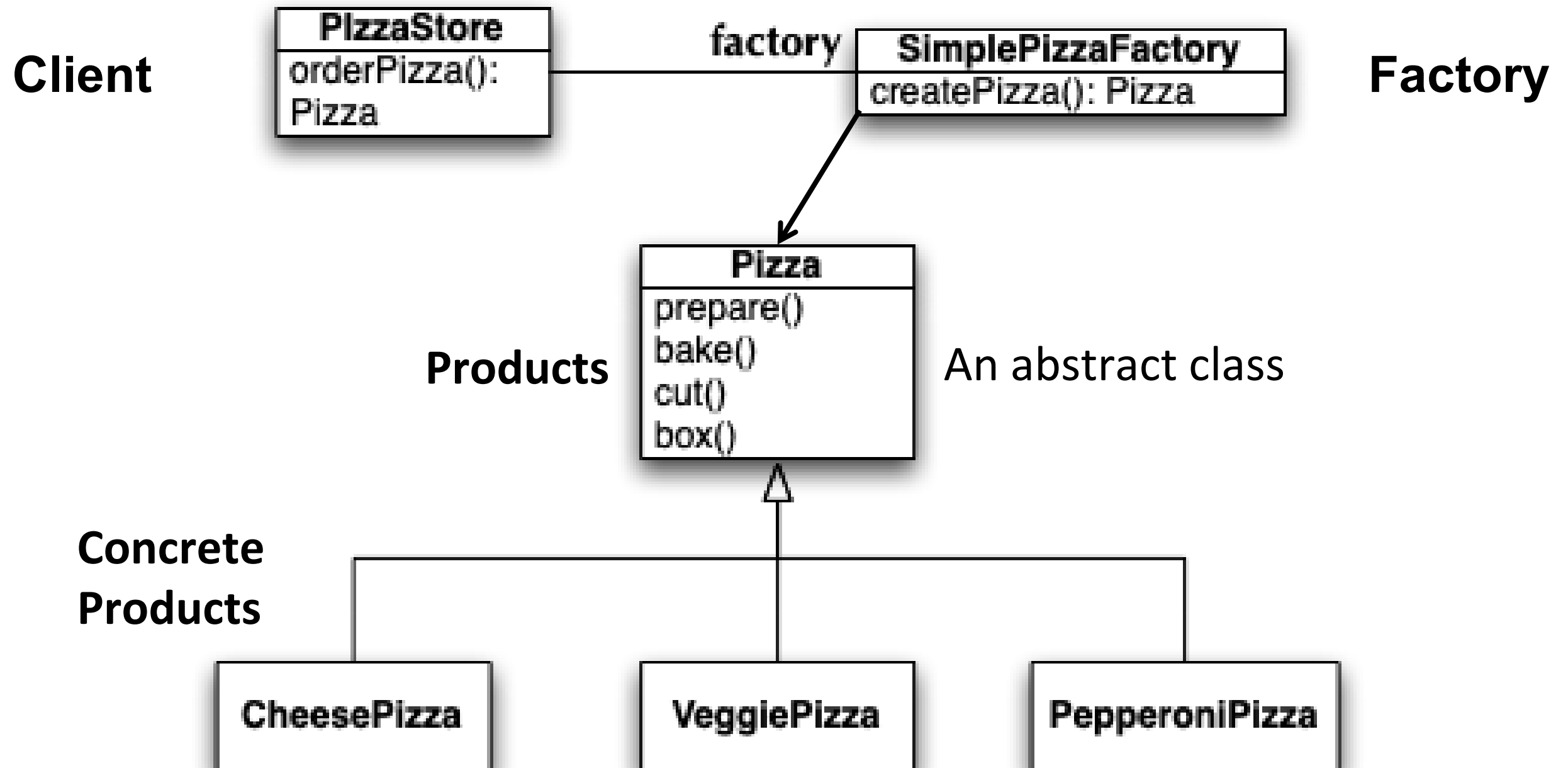


```

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

Implement an Interface...

- In the simple factory Pizza approach, the Concrete Products, like CheesePizza, extend the abstract Pizza class to get the factory method that Pizza has a reference to
- So when we say the Concrete Product “implements an interface” it doesn’t mean “a class that implements a Java Interface using the implements keyword”
- “Implements an interface” means, the concrete class is implementing a method from the supertype (which could be a class or an interface)

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

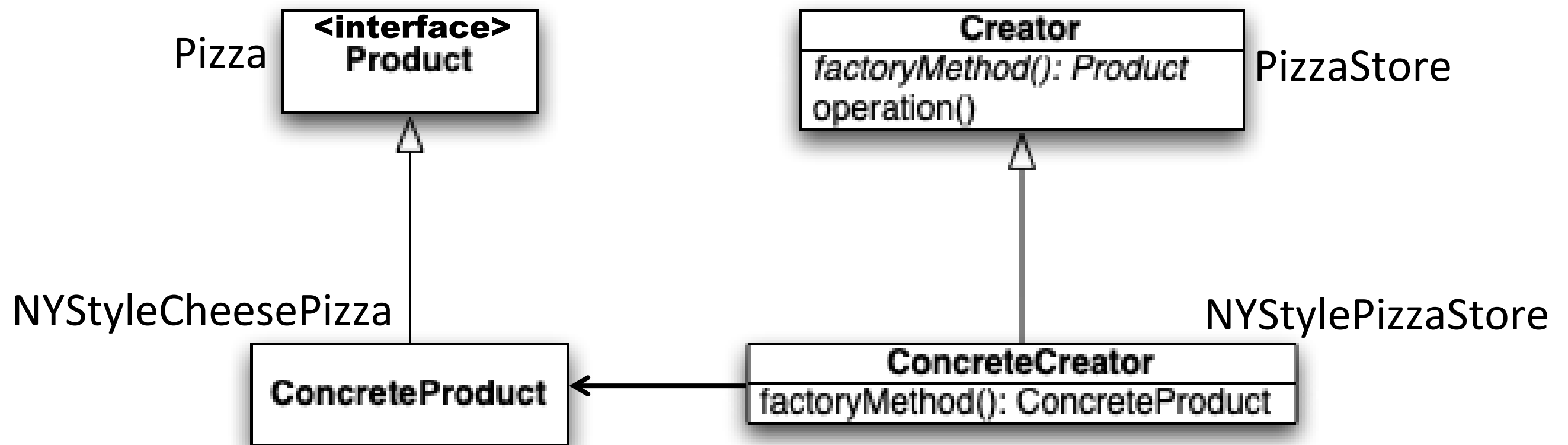
```
1 public class NYPizzaStore extends PizzaStore {  
2     public Pizza createPizza(String type) {  
3         if (type.equals("cheese")) {  
4             return new NYCheesePizza();  
5         } else if (type.equals("greek")) {  
6             return new NYGreekPizza();  
7         } else if (type.equals("pepperoni"))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

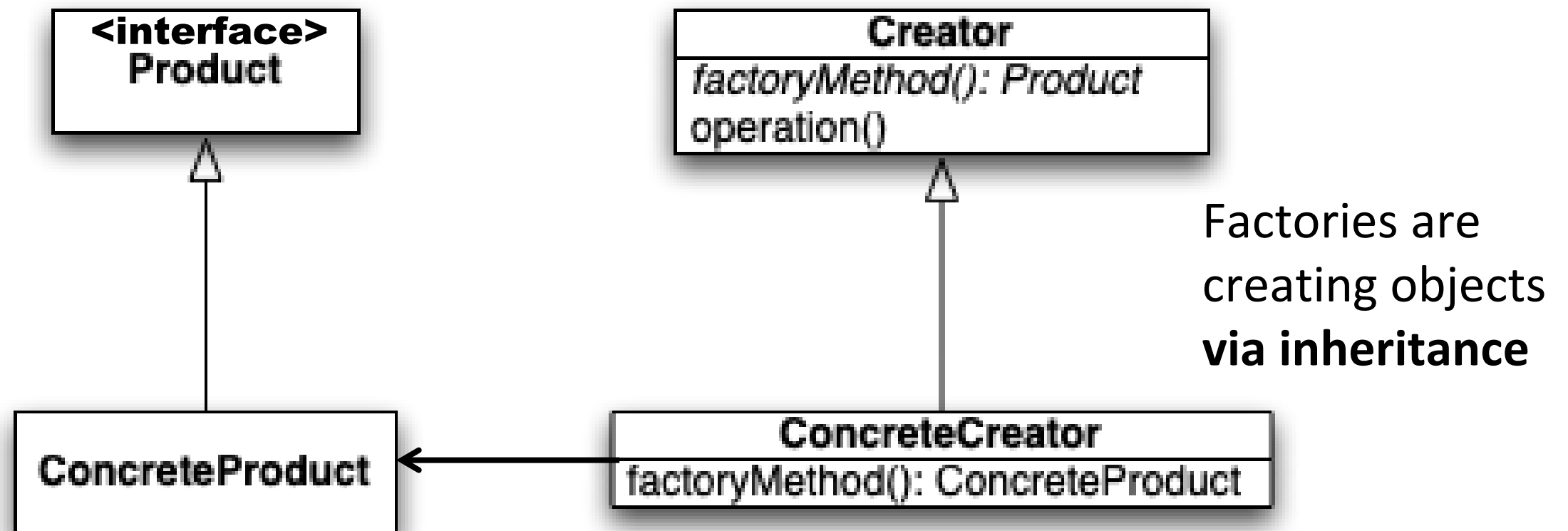
Factory Method: Definition and Structure

- 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

Factory Method: Specifics

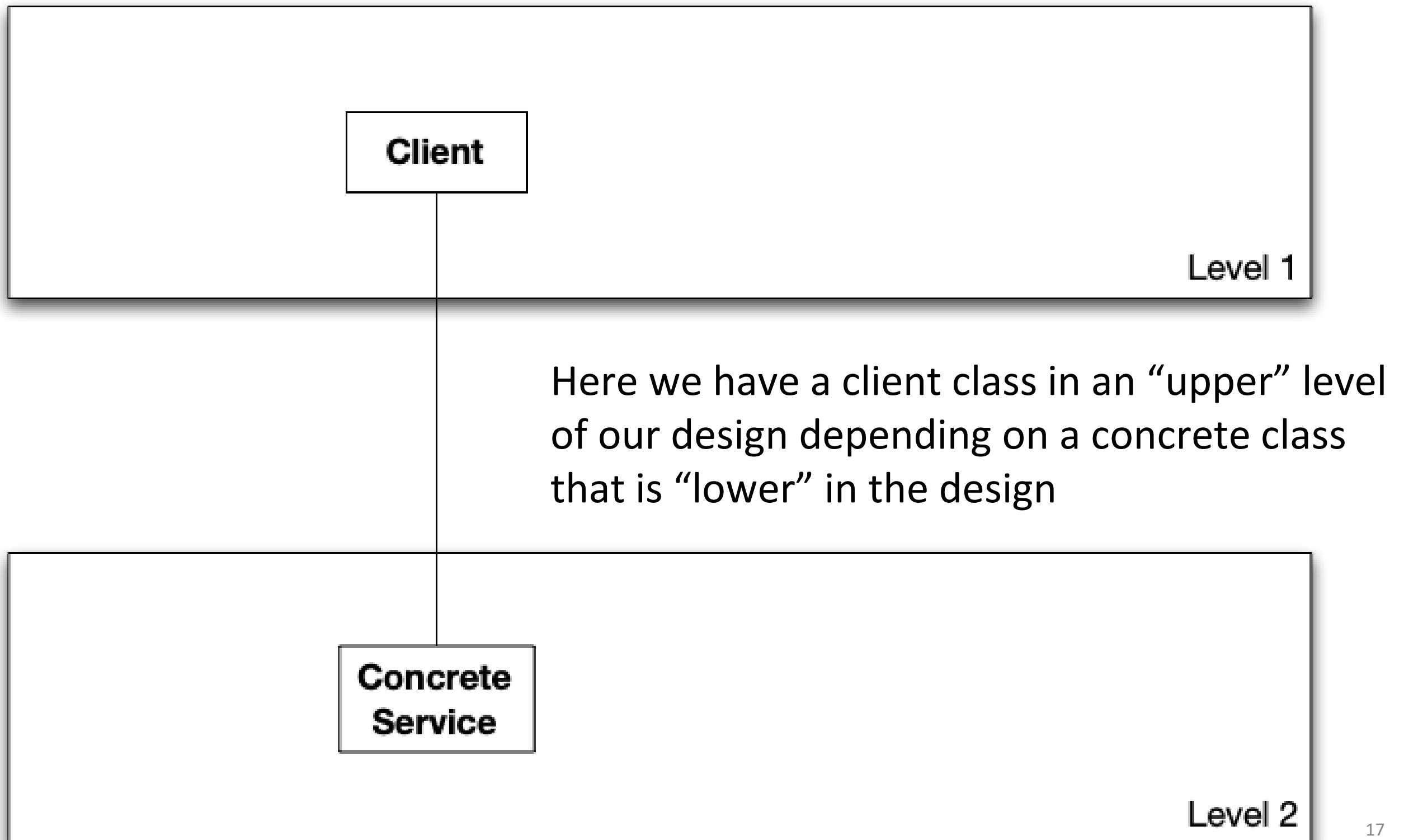


- Product declares an interface which is common to all objects that can be produced by the creator and its subclasses
- Concrete Products are different implementation of the Product Interface
- Creator declares the factory method that returns product objects
 - It could be an abstract method, making Concrete Creators implement local versions or
 - It could have a base version that returns a default product type
- Concrete Creators override the factory method to return different product types
- An aside – the “created” products could come from a cache, an object pool, or some other source, it doesn’t have to be new instances.

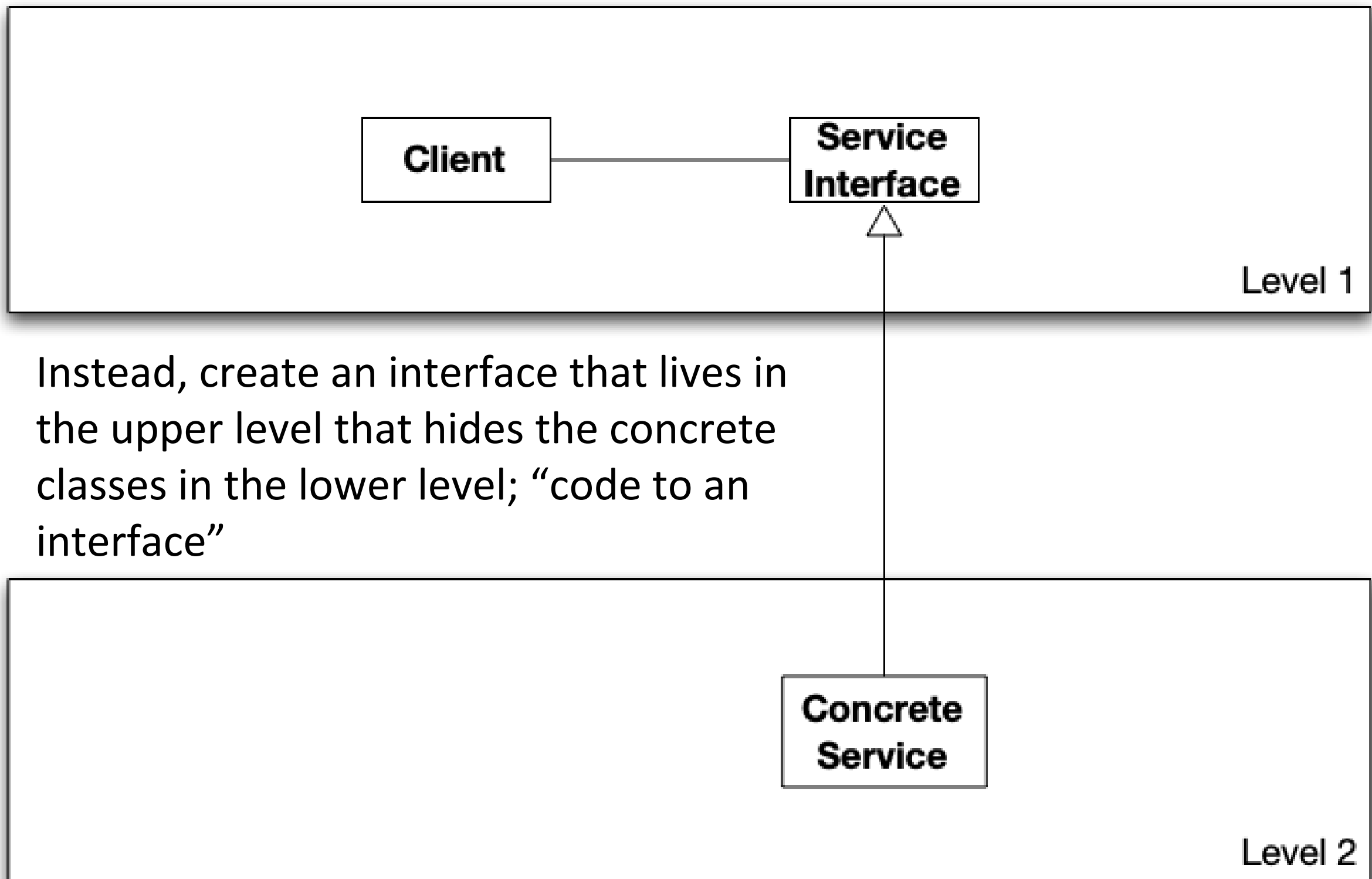
Dependency Inversion Principle (I)

- 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



Guidelines for Dependency Inversion Principle

- No variable should hold a reference to a concrete class
- No class should derive from a concrete class
 - Derive from interfaces or abstract classes
- No method should override an implemented method of base classes
 - Override abstract methods
 - If it's implemented in the base class, it's meant to be used by subclasses
- Remember, these are guidelines – not rules

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 (a poor design) 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

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)

Not what we want...

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...

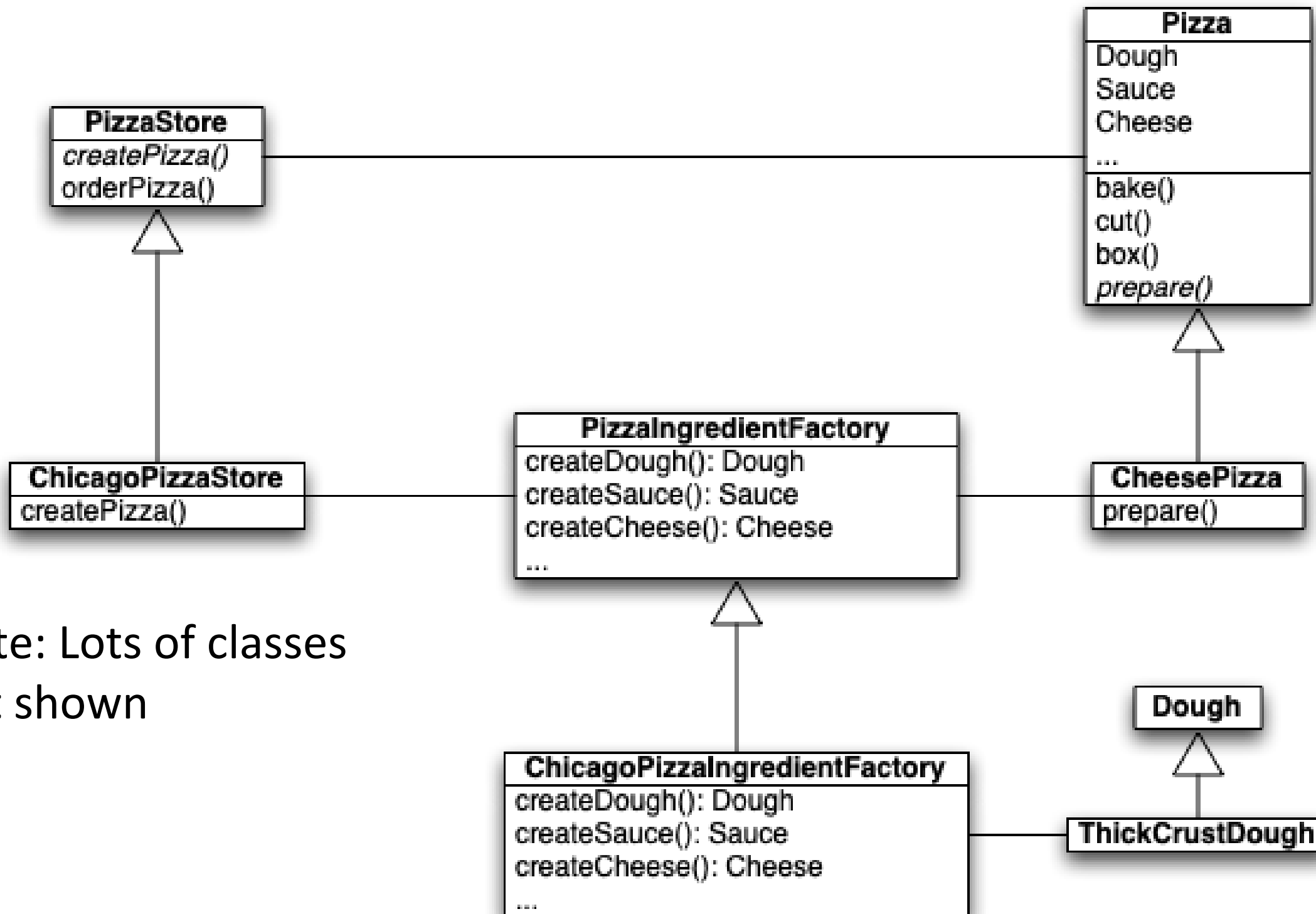
```
1 public class ChicagoPizzaStore extends PizzaStore {
2
3     protected Pizza createPizza(String item) {
4         Pizza pizza = null;
5         PizzaIngredientFactory ingredientFactory =
6         new ChicagoPizzaIngredientFactory();
7
8         if (item.equals("cheese")) {
9
10            pizza = 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
18            ...
19        }
20    }
21 }
```

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)

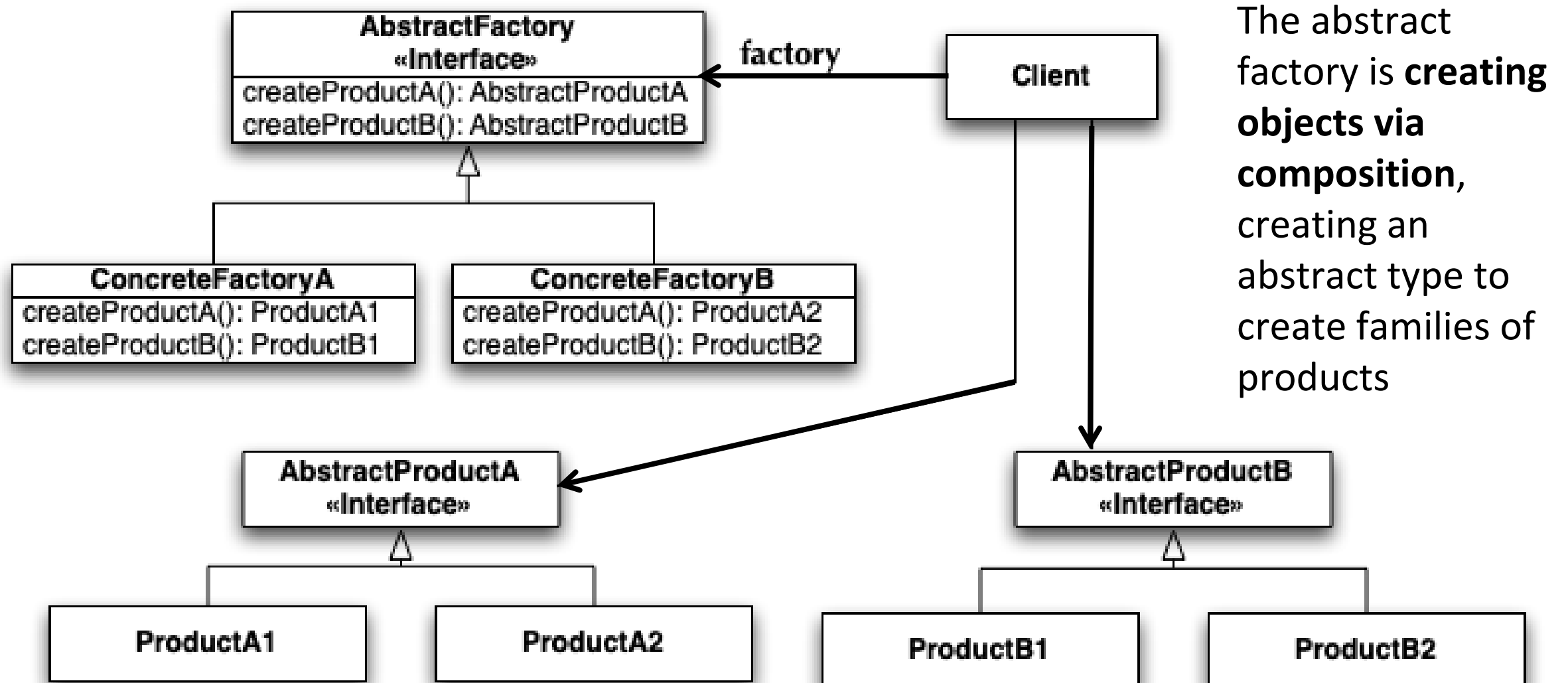
Partial Class Diagram of Abstract Factory Solution



Note: Lots of classes
not shown

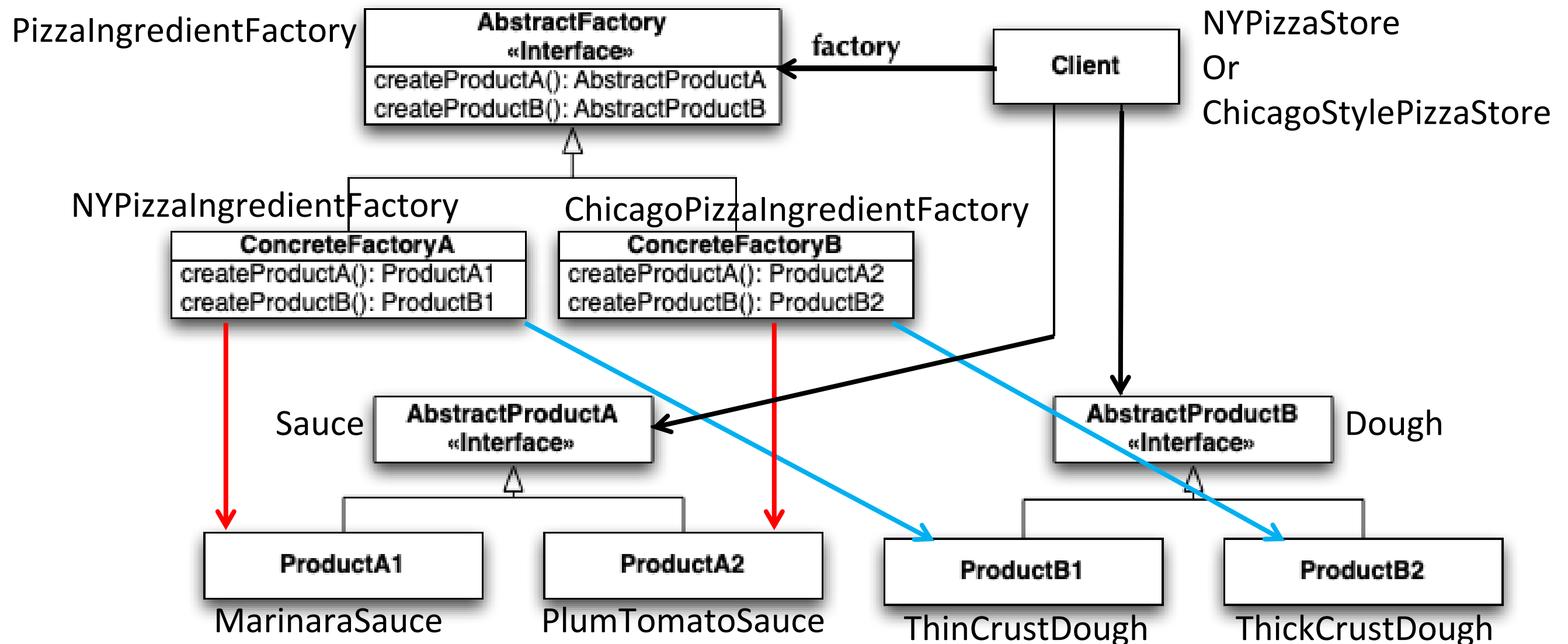
Abstract Factory: Definition and Structure

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



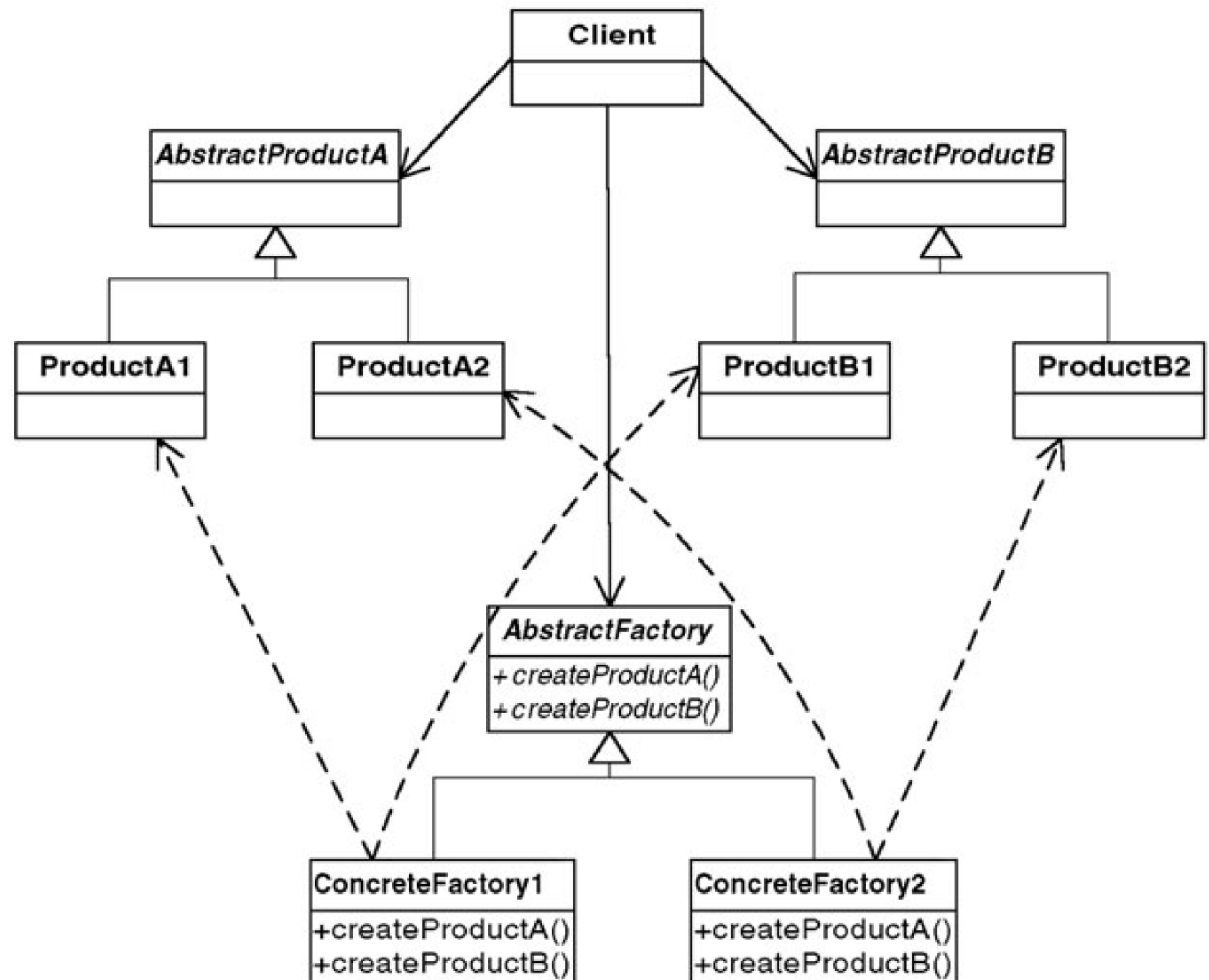
Abstract Factory: Definition and Structure

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



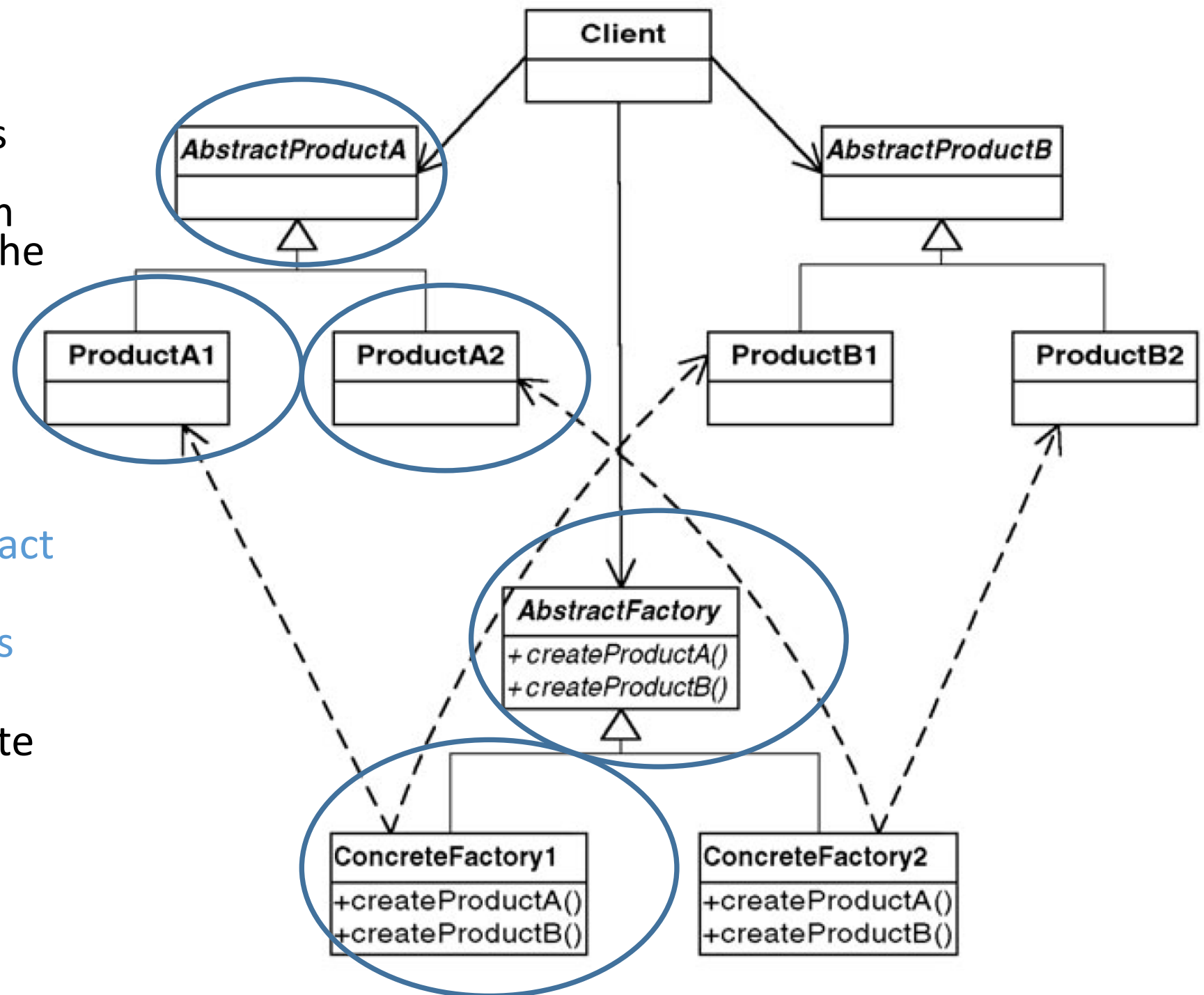
Abstract Factory: Another View

- 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



Abstract Factory: Another View

- 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
- The elements of an abstract factory that creates concrete products is a set of factory patterns embedded in the abstract factory...
- Factories create objects through inheritance
- Abstract Factories create through object composition



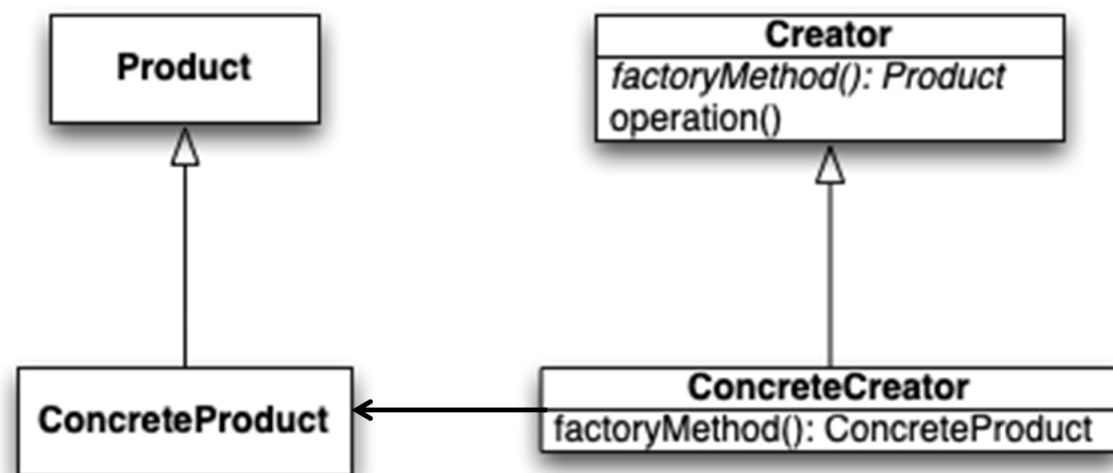
Summary

- We learned about Simple Factory (sort of a pattern)
 - Moving instantiation out to a delegated function for simple object creation
- We learned about Factory
 - Defines an interface for creating an object, but lets subclasses decide which class to instantiate (clients are decoupled from concrete types)
 - Factory lets a class defer instantiation to subclasses
 - Factories create **objects** through **inheritance**
- We learned about Abstract Factory
 - It enables the creation of families of objects
 - Makes sure clients create products that belong together
 - Hides (abstracts) the specific objects created from the clients that use them
 - Abstract Factories create **families of objects** via **object composition**
- Dependency Inversion Principle: Avoid dependencies on concrete types and strive for abstractions

Factory in Python

Classes:

- Creator
- ConcreteCreator1, 2
- Product
- ConcreteProduct1, 2



```
from abc import ABC, abstractmethod
```

```
class Creator(ABC):
```

```
# The Creator class declares the factory method
# that is supposed to return an object of a Product class.
```

```
@abstractmethod
```

```
def factory_method(self):
```

```
    pass
```

```
def some_operation(self) -> str:
```

```
# Call the factory method to create a Product object.
```

```
product = self.factory_method()
```

```
# Now, use the product.
```

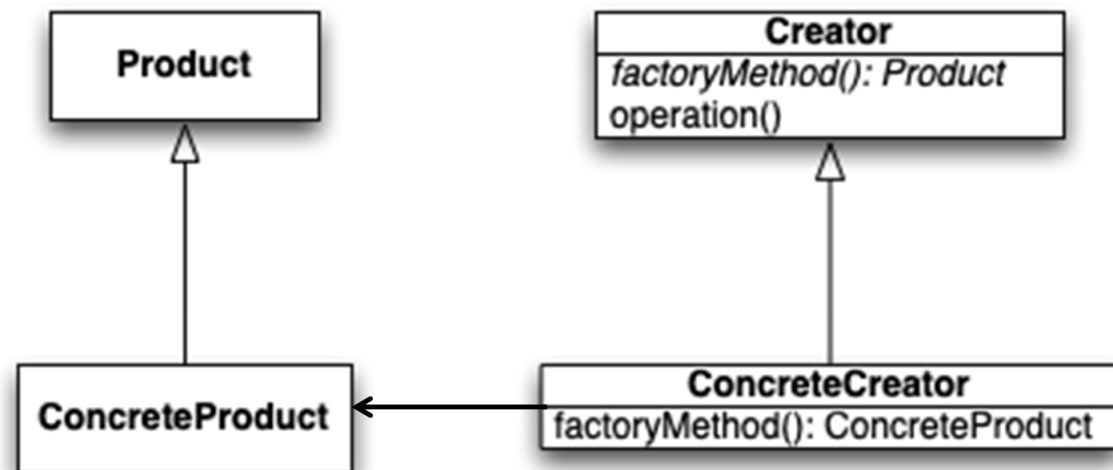
```
result = f"Creator: The same creator's code has  
just worked with {product.operation()}"
```

```
return result
```

Factory in Python

Classes:

- Creator
- ConcreteCreator1, 2
- Product
- ConcreteProduct1, 2



```
class ConcreteCreator1(Creator):
    def factory_method(self) -> ConcreteProduct1:
        return ConcreteProduct1()
```

```
class ConcreteCreator2(Creator):
    def factory_method(self) -> ConcreteProduct2:
        return ConcreteProduct2()
```

```
class Product(ABC):
    @abstractmethod
    def operation(self) -> str:
        pass
```

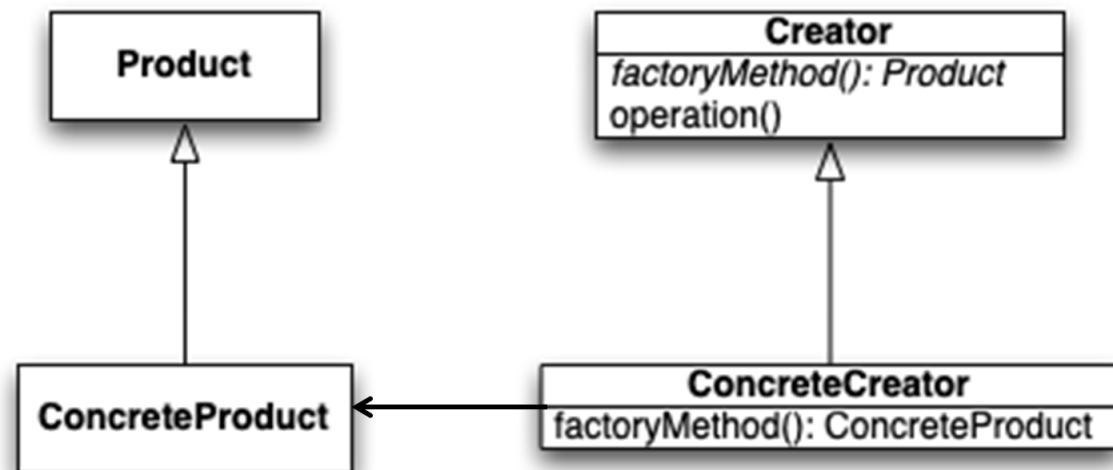
```
class ConcreteProduct1(Product):
    def operation(self) -> str:
        return "{Result of the ConcreteProduct1}"
```

```
class ConcreteProduct2(Product):
    def operation(self) -> str:
        return "{Result of the ConcreteProduct2}"
```

Factory in Python

Classes:

- Creator
- ConcreteCreator1, 2
- Product
- ConcreteProduct1, 2



```
def client_code(creator: Creator) -> None:
```

```
# The client code works with an instance
# of a concrete creator, albeit through
# its base interface. As long as the client
# keeps working with the creator via
# the base interface, you can pass it any
# creator's subclass.
```

```
print(f"Client: I'm not aware of the creator's class,
      but it still works.\n"
      f"{creator.some_operation()}", end="")
```

```
if __name__ == "__main__":
```

```
    print("App: Launched with the ConcreteCreator1.")
```

```
    client_code(ConcreteCreator1())
```

```
    print("\n")
```

```
    print("App: Launched with the ConcreteCreator2.")
```

```
    client_code(ConcreteCreator2())
```

App: Launched with the ConcreteCreator1.

Client: I'm not aware of the creator's class, but it still works.

Creator: The same creator's code has just worked with {Result of the ConcreteProduct1}

App: Launched with the ConcreteCreator2.

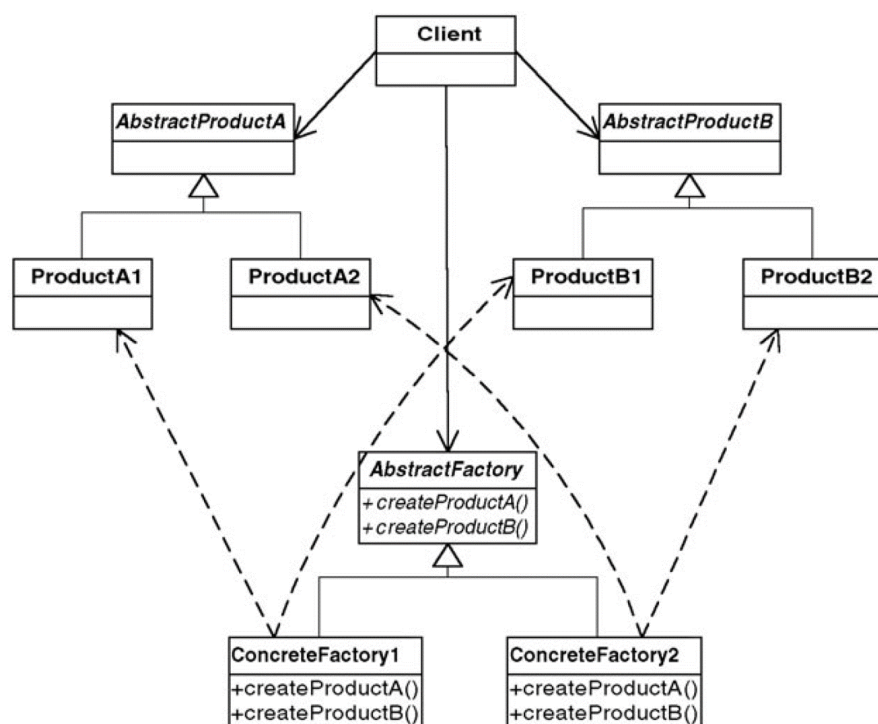
Client: I'm not aware of the creator's class, but it still works.

Creator: The same creator's code has just worked with {Result of the ConcreteProduct2}

Abstract Factory in Python

Classes:

- AbstractFactory
- ConcreteFactory1, 2
- AbstractProductA, B
- ConcreteProductA1, A2, B1, B2



```
class AbstractFactory(ABC):
```

```
    @abstractmethod
```

```
    def create_product_a(self) -> AbstractProductA:
```

```
        pass
```

```
    @abstractmethod
```

```
    def create_product_b(self) -> AbstractProductB:
```

```
        pass
```

```
class ConcreteFactory1(AbstractFactory):
```

```
    def create_product_a(self) -> ConcreteProductA1:
```

```
        return ConcreteProductA1()
```

```
    def create_product_b(self) -> ConcreteProductB1:
```

```
        return ConcreteProductB1()
```

```
class ConcreteFactory2(AbstractFactory):
```

```
    def create_product_a(self) -> ConcreteProductA2:
```

```
        return ConcreteProductA2()
```

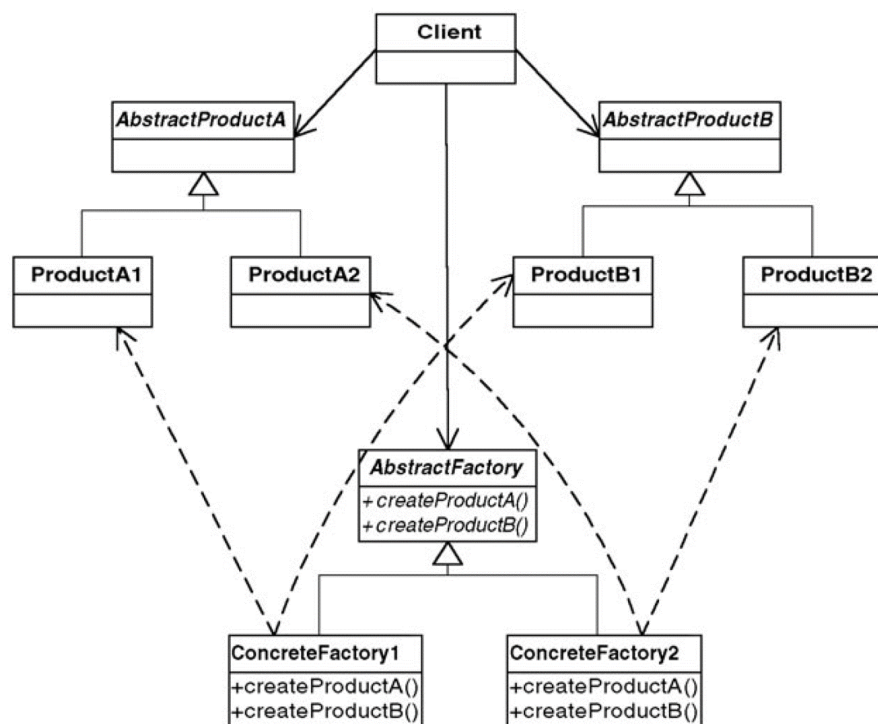
```
    def create_product_b(self) -> ConcreteProductB2:
```

```
        return ConcreteProductB2()
```

Abstract Factory in Python

Classes:

- AbstractFactory
- ConcreteFactory1, 2
- AbstractProductA, B
- ConcreteProductA1, A2, B1, B2



```
class AbstractProductA(ABC):
```

```
    @abstractmethod
```

```
    def useful_function_a(self) -> str:
```

```
        pass
```

```
class ConcreteProductA1(AbstractProductA):
```

```
    def useful_function_a(self) -> str:
```

```
        return "The result of the product A1."
```

```
class ConcreteProductA2(AbstractProductA):
```

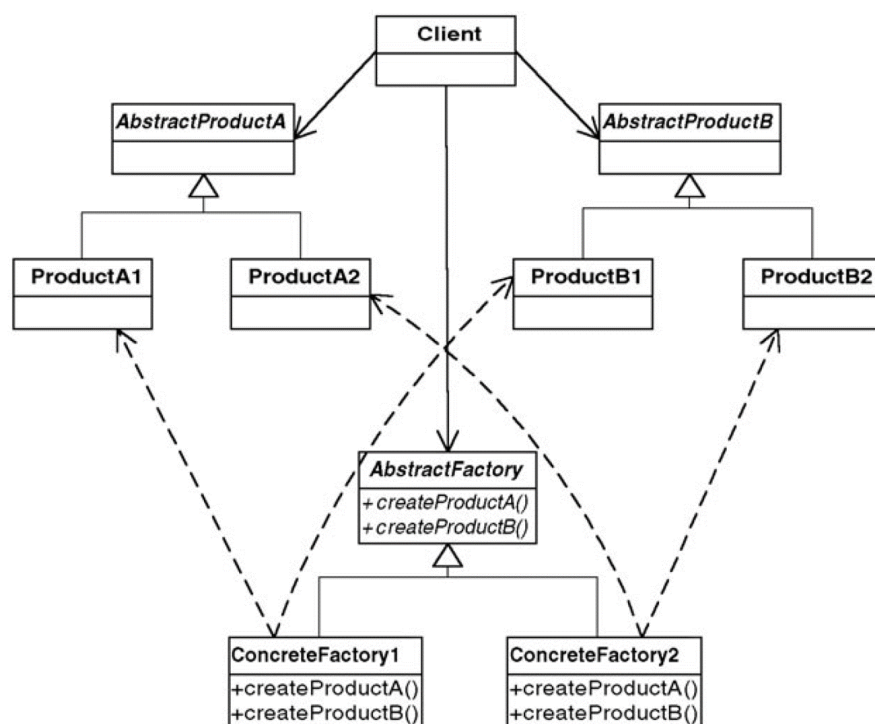
```
    def useful_function_a(self) -> str:
```

```
        return "The result of the product A2."
```

Abstract Factory in Python

Classes:

- AbstractFactory
- ConcreteFactory1, 2
- AbstractProductA, B
- ConcreteProductA1, A2, B1, B2



```
class AbstractProductB(ABC):
```

```
    @abstractmethod
```

```
    def useful_function_b(self) -> None:
```

```
        pass
```

```
    @abstractmethod
```

```
    def another_useful_function_b(self, collaborator: AbstractProductA) -> None:
```

```
        pass
```

```
class ConcreteProductB1(AbstractProductB):
```

```
    def useful_function_b(self) -> str:
```

```
        return "The result of the product B1."
```

```
    def another_useful_function_b(self, collaborator: AbstractProductA) -> str:
```

```
        result = collaborator.useful_function_a()
```

```
        return f"The result of the B1 collaborating with the ({result})"
```

```
class ConcreteProductB2(AbstractProductB):
```

```
    def useful_function_b(self) -> str:
```

```
        return "The result of the product B2."
```

```
    def another_useful_function_b(self, collaborator: AbstractProductA):
```

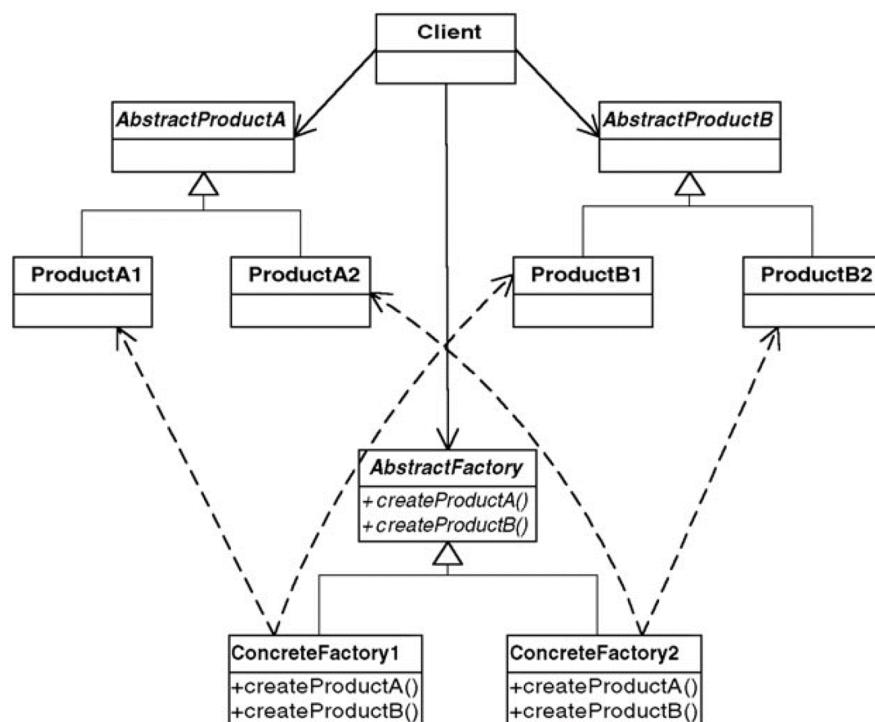
```
        result = collaborator.useful_function_a()
```

```
        return f"The result of the B2 collaborating with the ({result})"
```

Abstract Factory in Python

Classes:

- AbstractFactory
- ConcreteFactory1, 2
- AbstractProductA, B
- ConcreteProductA1, A2, B1, B2



```
def client_code(factory: AbstractFactory) -> None:
    product_a = factory.create_product_a()
    product_b = factory.create_product_b()
    print(f"{product_b.useful_function_b()}")
    print(f"{product_b.another_useful_function_b(product_a)}", end="")
```

```
if __name__ == "__main__":
    print("Client: Testing client code with the first factory type:")
    client_code(ConcreteFactory1())
    print("\n")
    print("Client: Testing the same client code with the second factory type:")
    client_code(ConcreteFactory2())
```

Client: Testing client code with the first factory type:
The result of the product B1.

The result of the B1 collaborating with the (The result of the product A1.)

Client: Testing the same client code with the second factory type:

The result of the product B2.

The result of the B2 collaborating with the (The result of the product A2.)

Next steps

- Project 1 in grading, Project 2 is due noon Wed 9/30
- Graduate Topic Submission in review
- Graduate Topic Outline is due noon Wed 9/30
- Quiz 3 due today, Quiz 4 open this weekend
- New discussion topic up to comment on
- Coming up: finish Decorator, then Factory, Singleton patterns
 - You'll want to look at the textbook Chapter 3-5 for readings supporting those lectures.
- If you need help! Office hours, Piazza, e-mail – don't be afraid to ask, it's what we're here for!