FACTORY PATTERN

Building Complex Objects

New is an implementation

- Calling “new” is certainly coding to an implementation
- In fact, it’s always related to a concrete class
- when there is code that makes use of lots of “new” concrete classes
  - *the code may have to be changed as “new” concrete classes are added.*
  - *the code will not be “closed for modification.”*
  - *to extend the code with “new” concrete classes, you’ll have to reopen it.*
- That’s fine when things are simple, but . . .

Look Out For Change

- When you have several related classes, that’s probably a good sign that they might change in the future

```
Duck duck;
if (picnic) {
   duck = new MallardDuck();
} else if (hunting) {
   duck = new DecoyDuck();
} else if (inBathTub) {
   duck  = new RubberDucky();
}
```

Design Principle

- What should you try to do with code that changes?
- example - a variety of types of pizzas

```
Pizza orderPizza() {
   Pizza pizza = new Pizza();

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

   return pizza;
}
```

```
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;
}
```

```
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();
    } else if (type.equals(“sausage”)) {
       pizza = new SausagePizza();
    } else if (type.equals(“veggie”)) {
       pizza = new VeggiePizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

```
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();
    } else if (type.equals(“sausage”)) {
       pizza = new SausagePizza();
    } else if (type.equals(“veggie”)) {
       pizza = new VeggiePizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Encapsulate!

```
public class SimplePizzaFactory {
   public Pizza createPizza(String type) {
     Pizza pizza;

     if (type.equals(“cheese”)) {
       pizza = new CheesePizza();
     } else if (type.equals(“pepperoni”)) {
       pizza = new PepperoniPizza();
     } else if (type.equals(“sausage”)) {
       pizza = new SausagePizza();
     } else if (type.equals(“veggie”)) {
       pizza = new VeggiePizza();
     }

     return pizza;
   }
}
```

Now orderPizza() is tidy

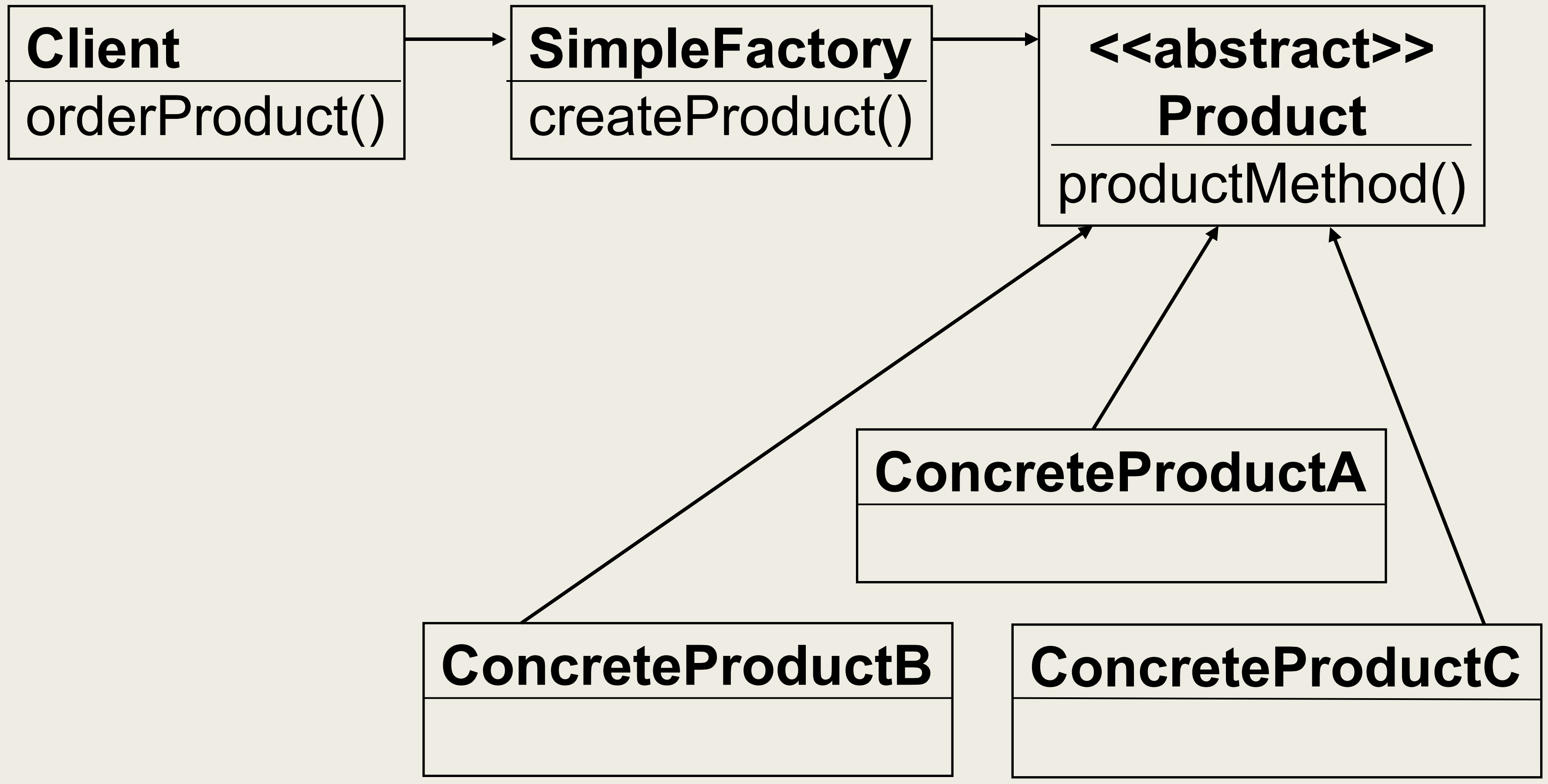
```
public class PizzaStore {
   SimplePizzaFactory factory;

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

   public Pizza orderPizza(String type) {
     Pizza pizza;
     pizza = factory.createPizza(type);

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

     return pizza;
   }
}
```

No **new** operator

Simple Factory

- Pull the code that builds the instances out and put it into a separate class
  - *Identify the aspects of your application that vary and separate them from what stays the same*

Simple Factory Pattern



Why would we do this?

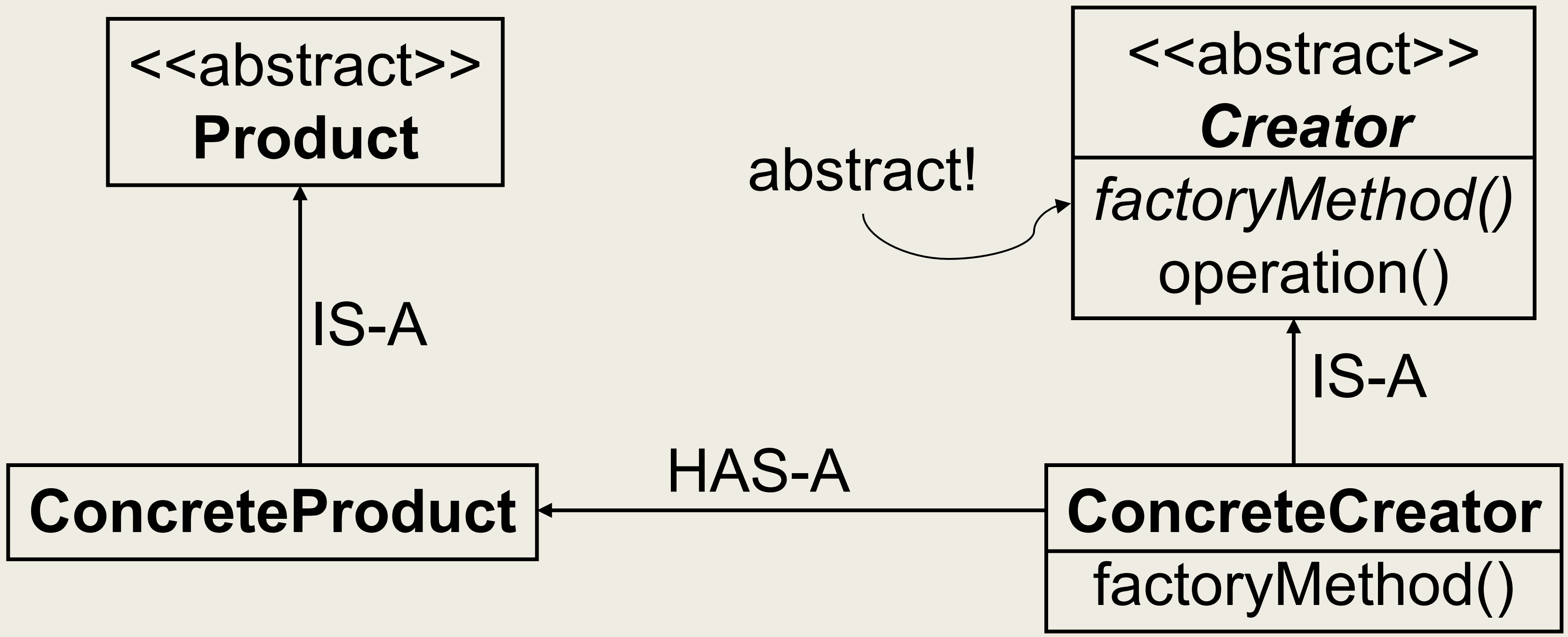
- Multiple clients needing same types of object
- Ensure consistent object initialization

The Factory Method

Definition:

The Factory Method Pattern defines an interface for creating an object, but lets the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Factory Method Pattern

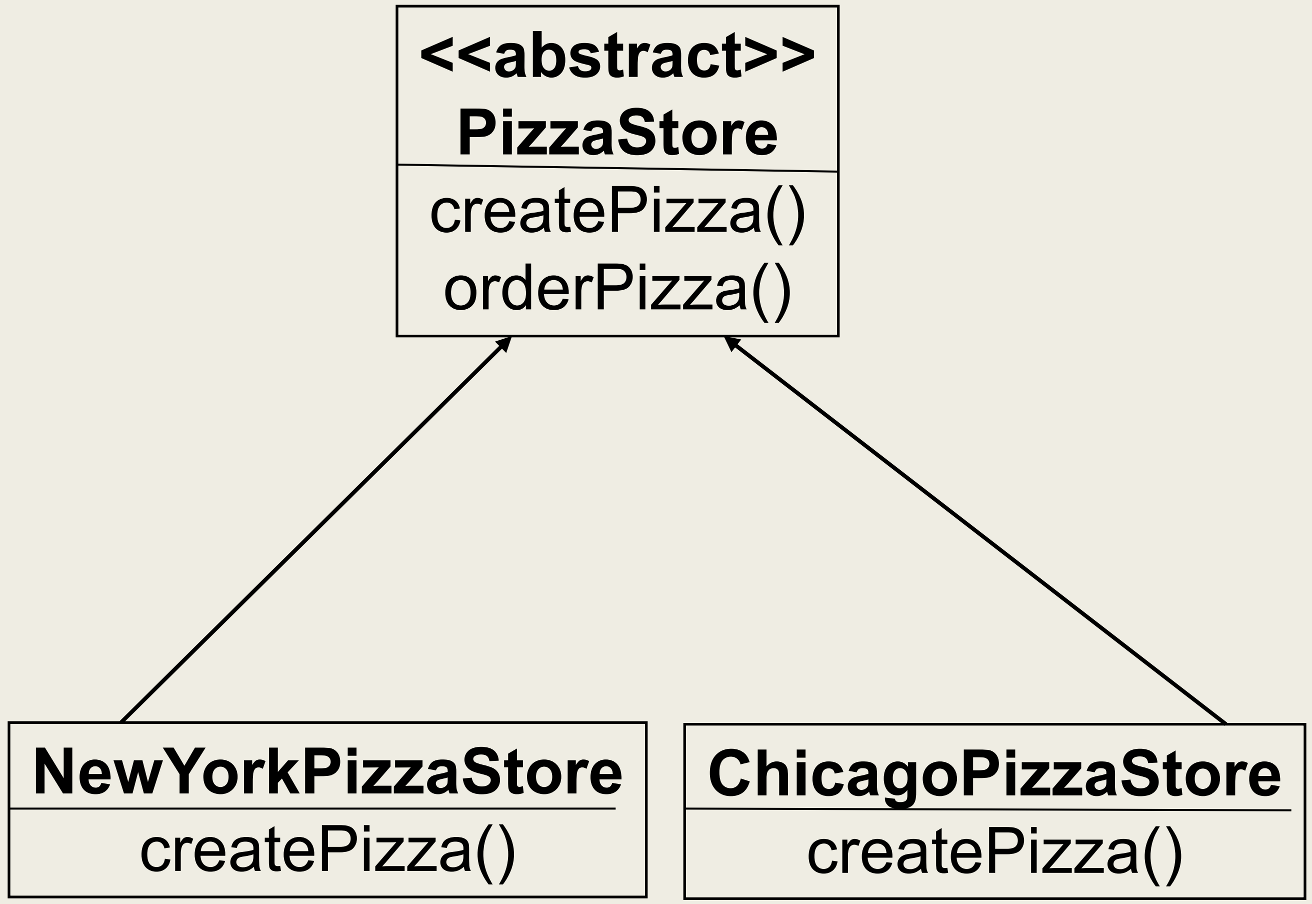


No more SimpleFactory **class**
Object creation is back in our class, but …
delegated to concrete classes

```
public abstract class PizzaStore {

   public Pizza orderPizza(String type) {
     Pizza pizza;
     pizza = createPizza(type);

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

     return pizza;
   }

   protected abstract Pizza createPizza(String type);
}
```

```
main() {
PizzaStore myPizzaStore =
    new NewYorkPizzaStore();
myPizzaStore.orderPizza(cheese);
}
```

Factory Method Pattern



Hard to Achieve Guidelines

Depend upon abstractions -- not concrete classes

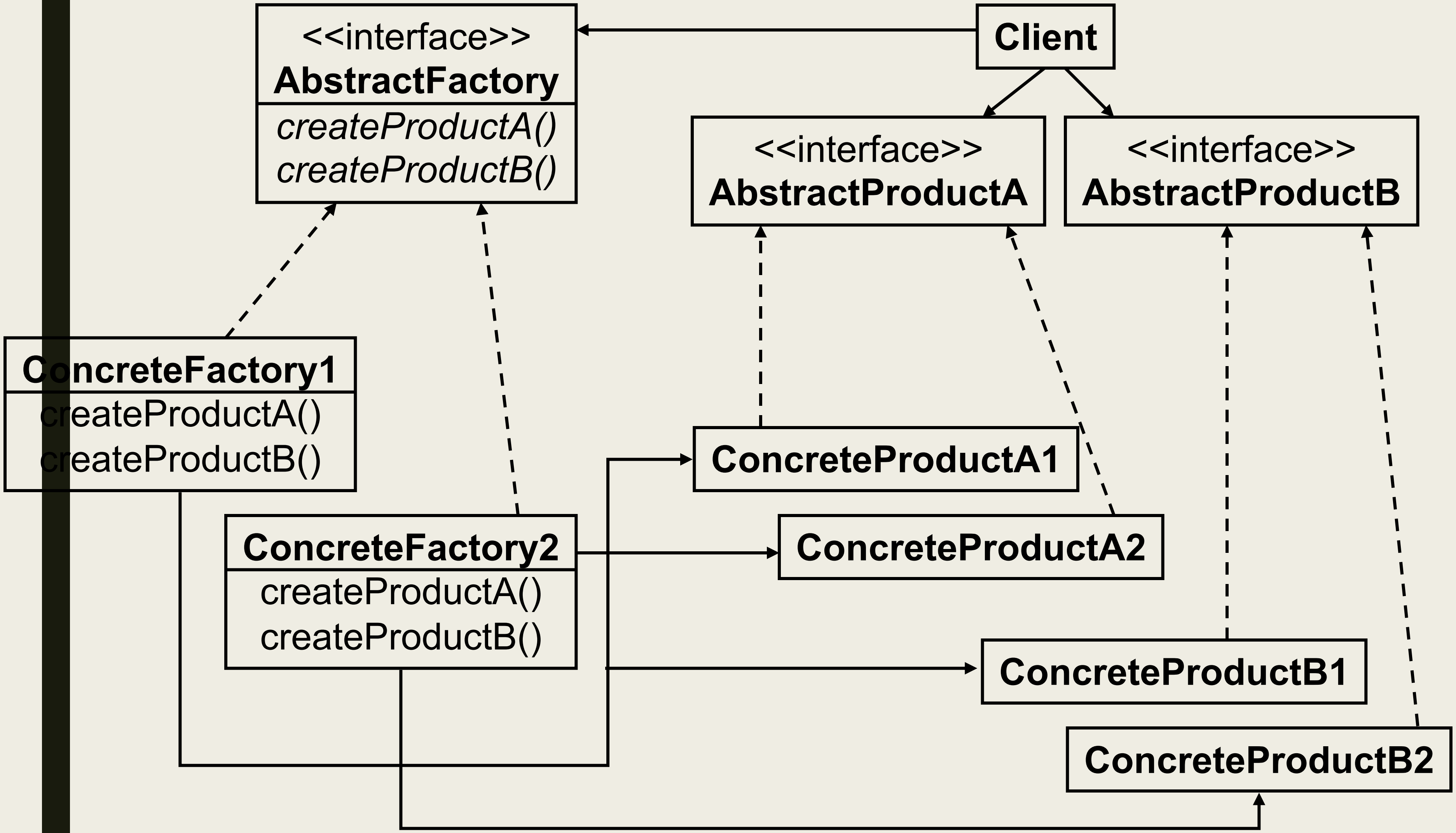
- No variable should hold a reference to a concrete class
- No class should derive from a concrete class
- No method should override an implemented method of an of its base classes.

The idea here is to stay clear of classes that are likely to change

What if there are categories of Products?

- Make the **factory** an abstract class as well
- Instead of subclasses for the **products**, subclasses for the **components** of the products
- The next diagram is overwhelming…

Abstract Factory Pattern



Abstract Factory Pattern

This design favors composition

handy when you need to ensure the quality of many similar, but slightly different versions of a class

Summary

- Simple Factory
  - *Use when you have only one family of factory*
- Factory Method Pattern
  - *Use when you have multiple families of factories*
- Abstract Factory
  - *Use when you have multiple families of product components*