Factory Patterns

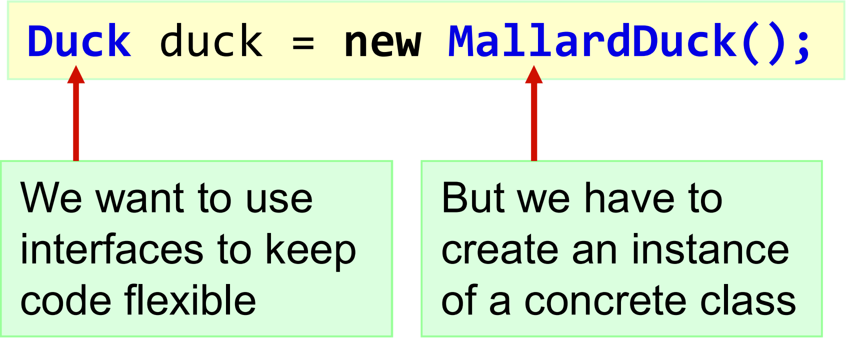
"Baking with OO Goodness"

The Constitution of Software Architects

- Encapsulate what varies
- Program through an interface not to an implementation
- Favor Composition over Inheritance
- Classes should be open for extension but closed for modification
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????

"new" = "concrete"

- Design Principle: ***"Program through an interface not to an implementation"***
- However, every time you do a "*new*" you need to deal with a "concrete" class, not an abstraction.



```
Duck duck = new MallardDuck();
```

We want to use interfaces to keep code flexible

But we have to create an instance of a concrete class

With a whole set of related concrete classes:

```
Duck duck;
if (picnic)
   duck = new MallardDuck();
else if (hunting)
   duck = new DecoyDuck();
else if (inBathtub)
   duck = new RubberDuck();
}
```

not "closed for modification"

What's wrong with this? What principle is broken here?

What can you do?

- **Principle**: Identify the aspects that vary and separate them from what stays the same.

- How might you take all the parts of your application that *instantiate concrete classes* and *separate or encapsulate them* from the rest of your application?

Identifying aspects that vary

- Order pizza in a pizza store in cutting edge Objectville!

```
public class PizzaStore {
   Pizza orderPizza() {
      Pizza pizza = new Pizza();

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

For flexibility it would be nice **if this wasn't concrete**, but we can't **instantiate abstract** classes!

Prepare the pizza

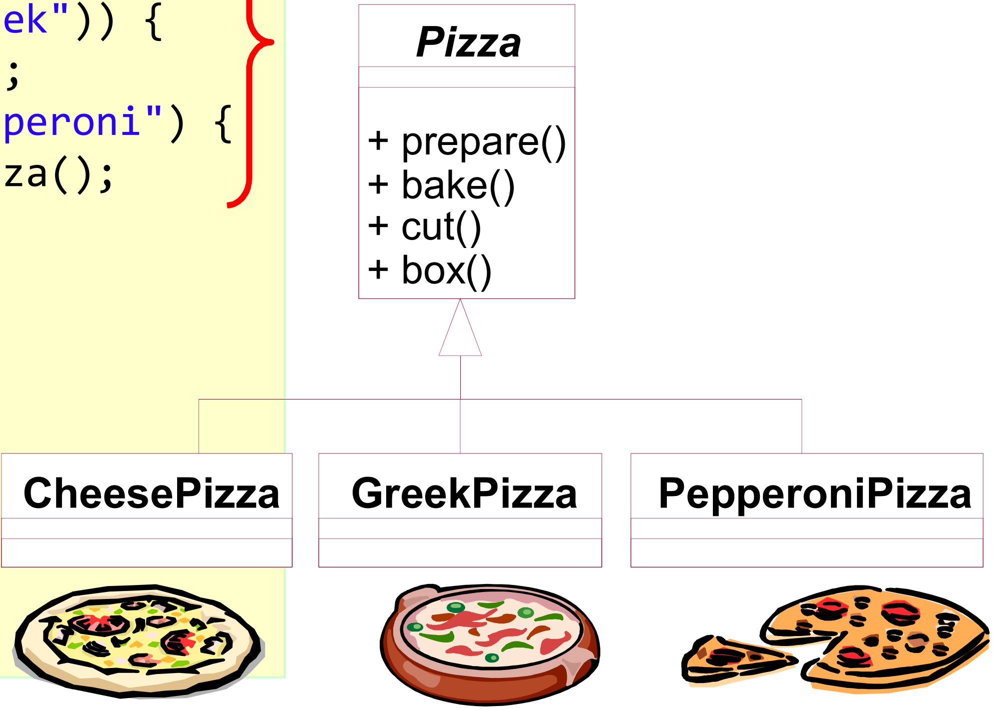
Identifying aspects that Vary

- But you need more than one type of pizza:

```
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;
   }
}
```

Pass in the **`type`** of **`Pizza`** to **`orderPizza()`**

Instantiate based on type of pizza

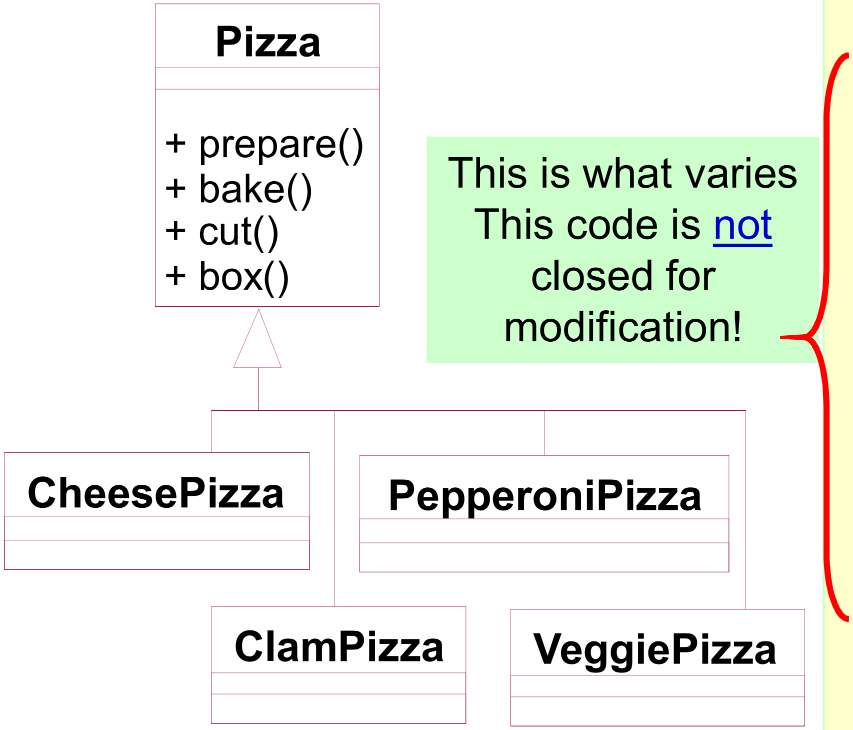


But the pressure is on to add more pizza types….

- Need to add a couple trendy pizzas to their menu: Clam and Veggie.
- Greek is not selling so well – so take it out!

- *What do you think would need to vary* and *what would stay constant*?

Modified `orderPizza()`



This is what varies
This code is not
closed for
modification!

```
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();
      } else if (type.equals("clam") {
         pizza = new ClamPizza();
      } else if (type.equals("veggie") {
         pizza = new VeggiePizza();
      }
      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
   }
}
```

This is what we expect
will stay the same

Encapsulating Object Creation

- Move the object creation out of the **orderPizza()** method.
- How?
  - Move the creation code into a special purpose object that is concerned with only creating pizzas

```java
public class PizzaStore {
   Pizza orderPizza(String type) {
      Pizza pizza;



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

pull it out

What's going to go here?

```java
if (type.equals("cheese") {
   pizza = new CheesePizza();
} else if (type.equals("pepperoni") {
    pizza = new PepperoniPizza();
} else if (type.equals("clam") {
    pizza = new ClamPizza();
} else if (type.equals("veggie") {
    pizza = new VeggiePizza();
}
```

We place this code into a separate object **SimplePizzaFactory**

Building a Simple Pizza Factory

Factories handle the details of the object creation.

```
public class SimplePizzaFactory {
   public Pizza createPizza(String type) {
      Pizza pizza = null;
      if (type.equals("cheese") {
         pizza = new CheesePizza();
      } else if (type.equals("pepperoni") {
         pizza = new PepperoniPizza();
      } else if (type.equals("clam") {
         pizza = new ClamPizza();
      } else if (type.equals("veggie") {
         pizza = new VeggiePizza();
      }
      return pizza;
   }
}
```

Here’s code we plucked out of the **orderPizza()** method.

Code is still parameterized by the type of pizza.

Could this method be made static?

Reworking the PizzaStore class

```
public class PizzaStore {
   private 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.cut();
      pizza.box();
      return pizza;
   }
}
```

PizzaStore has a reference to the factory

PizzaStore gets the factory passed in the constructor

new operator replaced by create method!

Why is this better?

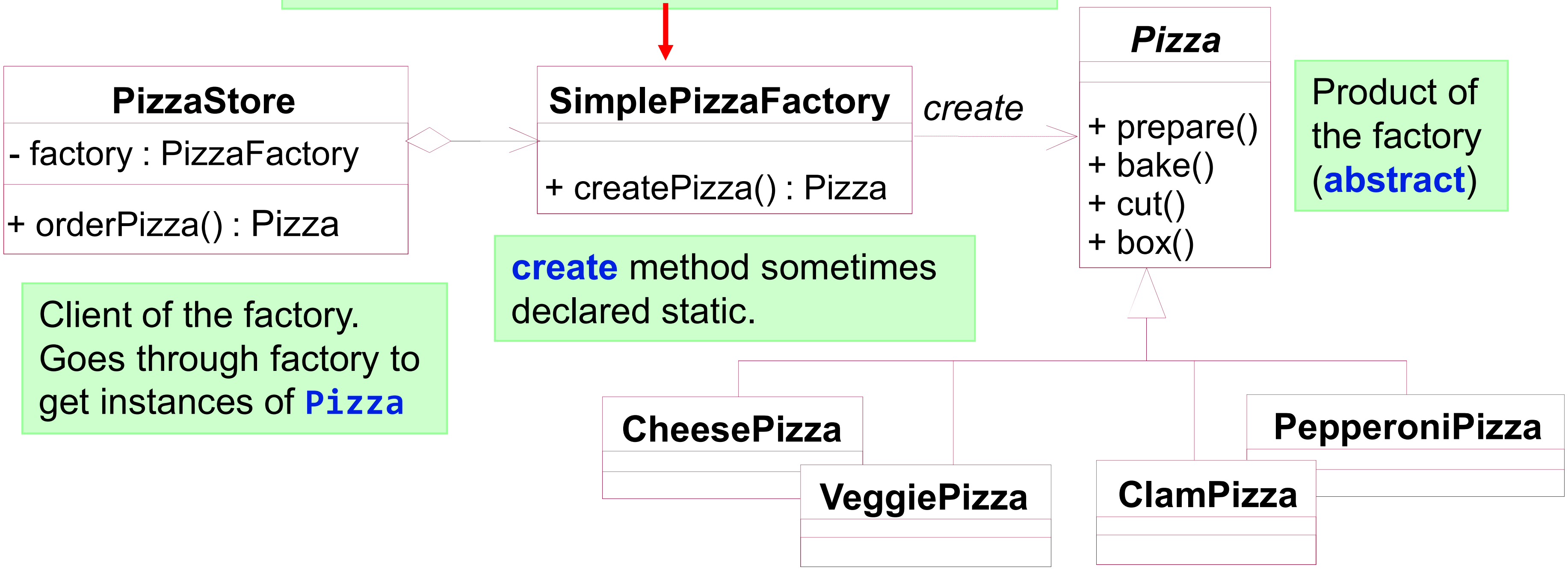
- ***`SimplePizzaFactory`*** *may* *have many more clients than* **`orderPizza()`** method
  - **`PizzaShopMenu`**, **`HomeDelivery`** etc.
- Client code *does not have any concrete classes* anymore!!

- **`SimpleFactory`** is not really a design pattern but the actual **`Factory`** patterns are based on it!

Simple Factory Pattern

The factory where we create pizzas.
This should be the only part of our application that refers to concrete **Pizza** classes.

Abstract class with some helpful implementations that can be overridden.



PizzaStore

- factory : PizzaFactory

+ orderPizza() : Pizza

SimplePizzaFactory

+ createPizza() : Pizza

create

Pizza

+ prepare()
+ bake()
+ cut()
+ box()

CheesePizza

VeggiePizza

ClamPizza

PepperoniPizza

Product of the factory (**abstract**)

Client of the factory.
Goes through factory to get instances of **Pizza**

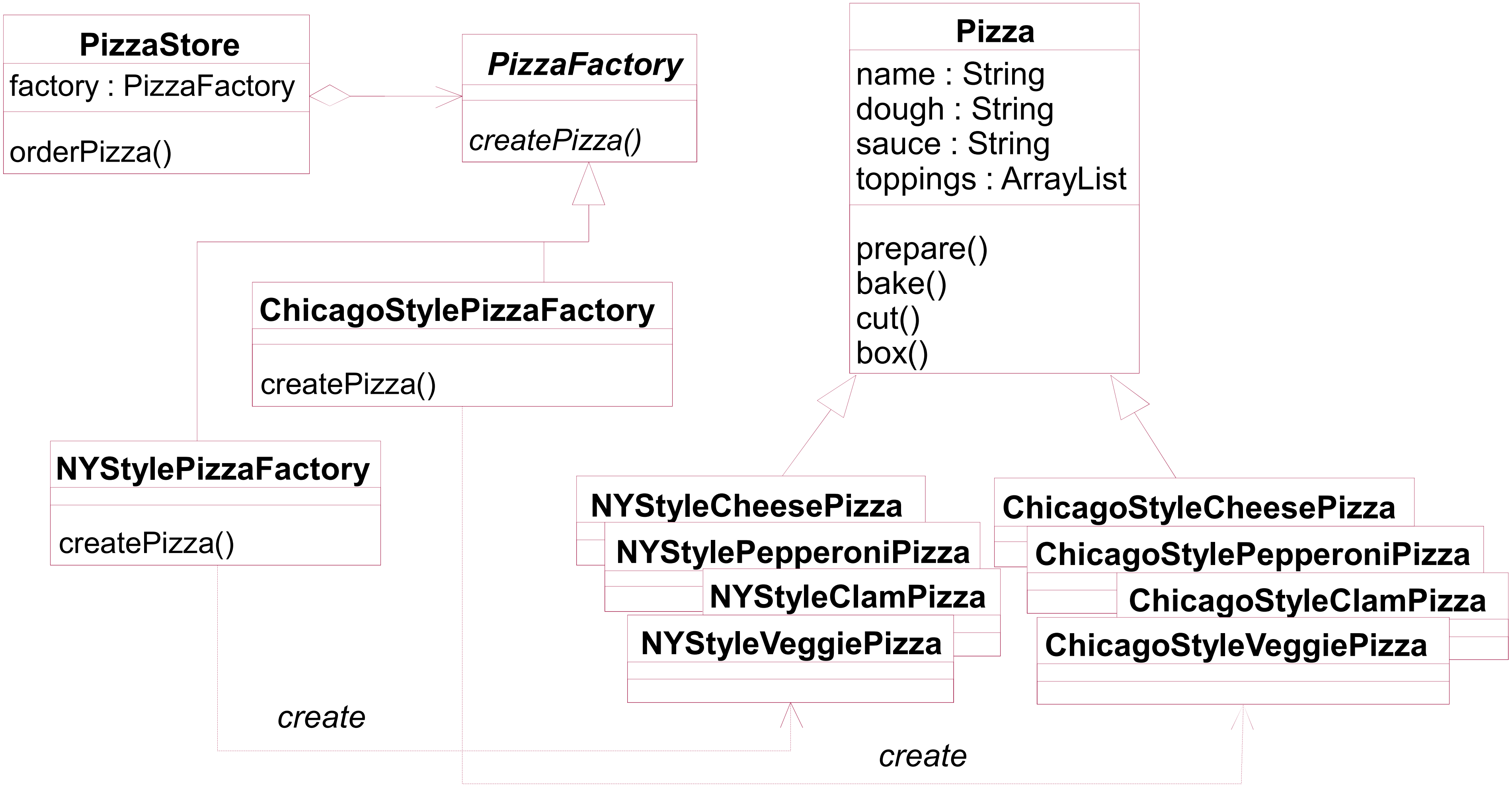
create method sometimes declared static.

Concrete products.
Each product needs to implement the **Pizza** interface.

Onwards with the Pizza Franchise

- Franchises in different cities
  - Must ensure quality of pizza
  - Must account for regional differences (NY, Chicago..)
- Want franchise store to leverage your **`PizzaStore`** code --> pizzas are prepared the same way
- New York needs a factory that makes New York style pizza
- Chicago needs a factory that makes Chicago style pizza
- One approach --> **Simple Factory**

Applying SimpleFactory



Applying Simple Factory Pattern

Here we create a factory for making **NY style pizza**

```
NYStylePizzaFactory nyFactory =
            new NYStylePizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nystore.orderPizza("Veggie");

ChicagoStylePizzaFactory cFactory = new ChicagoStylePizzaFactory();
PizzaStore cStore = new PizzaStore(cFactory);
cStore.orderPizza("Veggie");
```

Then we create a **PizzaStore** and pass it a reference to the **NY factory**

…and when we make pizzas we get **NY style pizzas**

Issues

- Franchises using your factory to create pizza, but using homegrown procedures for baking, cut the pizza, uses third-party boxes, etc.
- Yet, each franchise “needs room for adding own improvements”
  - You don’t want to know what they put on their pizza - detail that should be “exposed” only to the individual stores.

    Yet you want to have some control (quality control!)

What is needed is a framework that ties the ***store*** and ***pizza creation*** together, yet still allows for flexibility.

Factory Method pattern

A Framework

- Need a mechanism to “localize” all pizza making activities to the **PizzaStore** class and yet give franchises freedom to have their own regional style!

```
public abstract class PizzaStore {
   public Pizza orderPizza(String type) {

      Pizza pizza = createPizza(type);

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

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

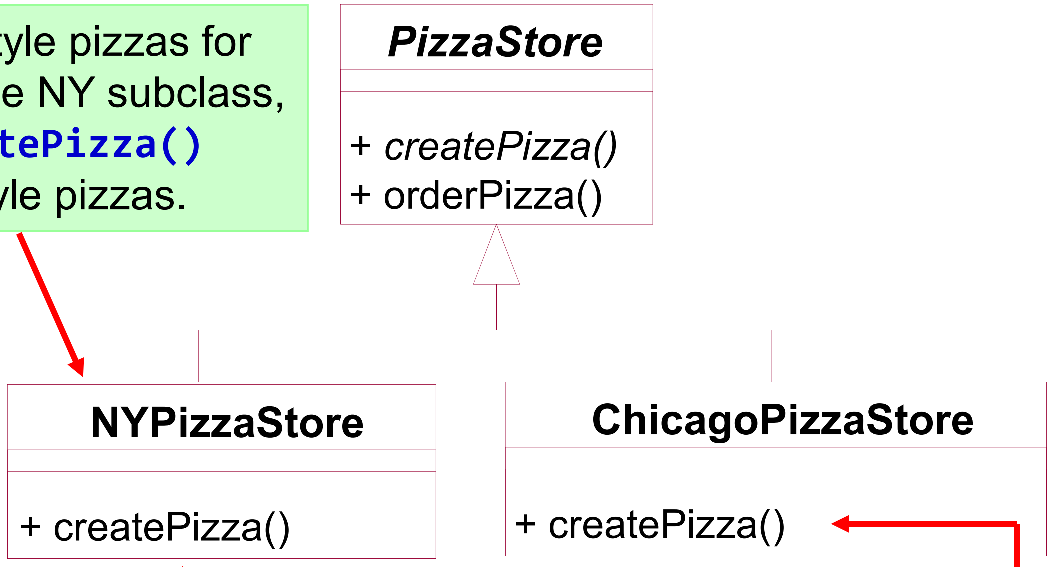
Now **createPizza** is back to being a call to a method in the **PizzaStore** rather than a Factory object!

Our **factory method** is now `abstract` in **PizzaStore**.

Allows each individual subclass to decide which Factory to invoke.
All subclasses MUST implement the createPizza method.

Allowing the subclasses to decide…

If franchise wants NY style pizzas for its customers, it uses the NY subclass, which has its own **createPizza()** method, creating NY style pizzas.



PizzaStore

+ *createPizza()*
+ orderPizza()

NYPizzaStore

+ createPizza()

ChicagoPizzaStore

+ createPizza()

Each subclass overrides the **createPizza()** method, while all subclasses make use of the **orderPizza()** method defined in the **PizzaStore**.

createPizza() returns a **Pizza** and the subclass is fully responsible for which concrete **Pizza** it instantiates

Let's make a PizzaStore

extends **PizzaStore**, so it inherits the orderPizza() method

```
public class NYPizzaStore extends PizzaStore {
   Pizza createPizza(String item) {
      if (item.equals("cheese")) {
         return new NYStyleCheesePizza();
      } else if (item.equals("veggie")) {
         return new NYStyleVeggiePizza();
      } else if (item.equals("clam")) {
         return new NYStyleClamPizza();
      } else if (item.equals("pepperoni")) {
         return new NYStylePepperoniPizza();
      } else return null;
   }
}
```

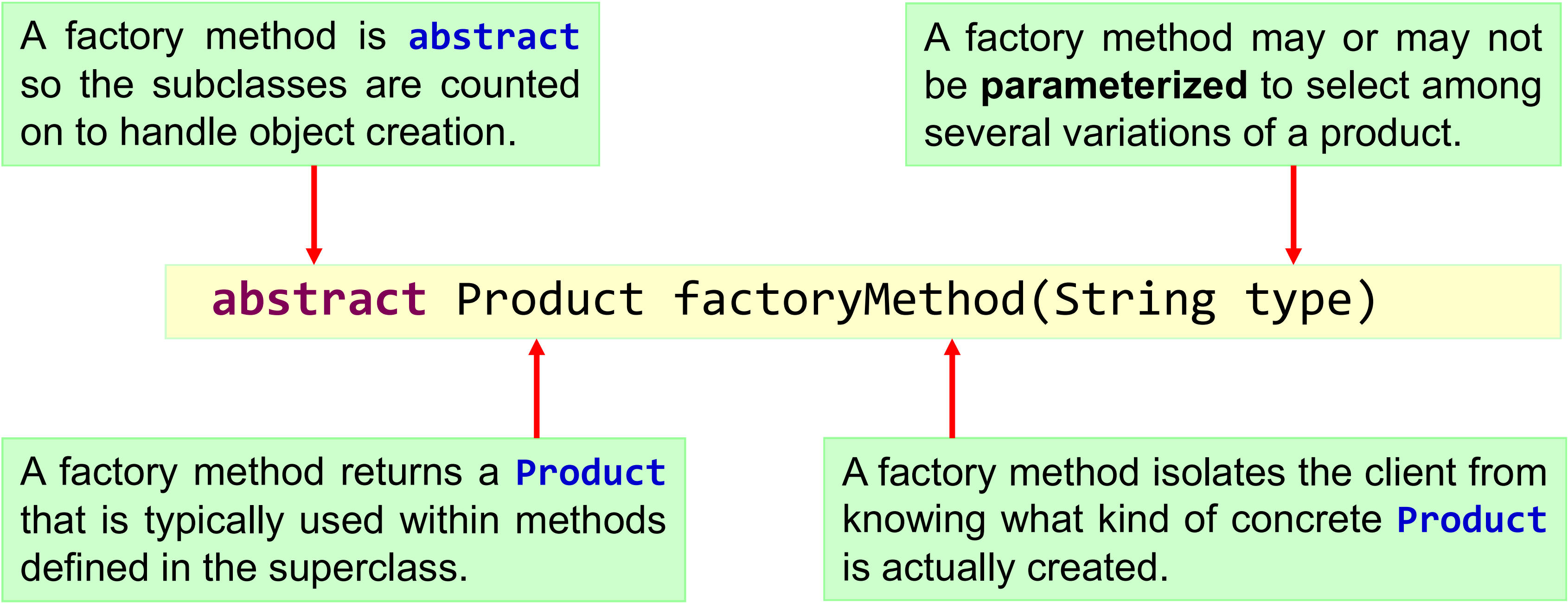
implement **createPizza()**, since it is abstract in **PizzaStore**

Here’s where the concrete classes are being created!

Note: **orderPizza()** method in superclass has no clue which **Pizza** we are creating. It only knows it can prepare, bake, cut and box it!

A Factory Method Up Close!

- A “Factory” method handles object creation and encapsulates it in a subclass.
  This decouples the client code in the superclass from the object creation code in the subclass.



A factory method is **abstract** so the subclasses are counted on to handle object creation.

A factory method may or may not be **parameterized** to select among several variations of a product.

```
abstract Product factoryMethod(String type)
```

A factory method returns a **Product** that is typically used within methods defined in the superclass.

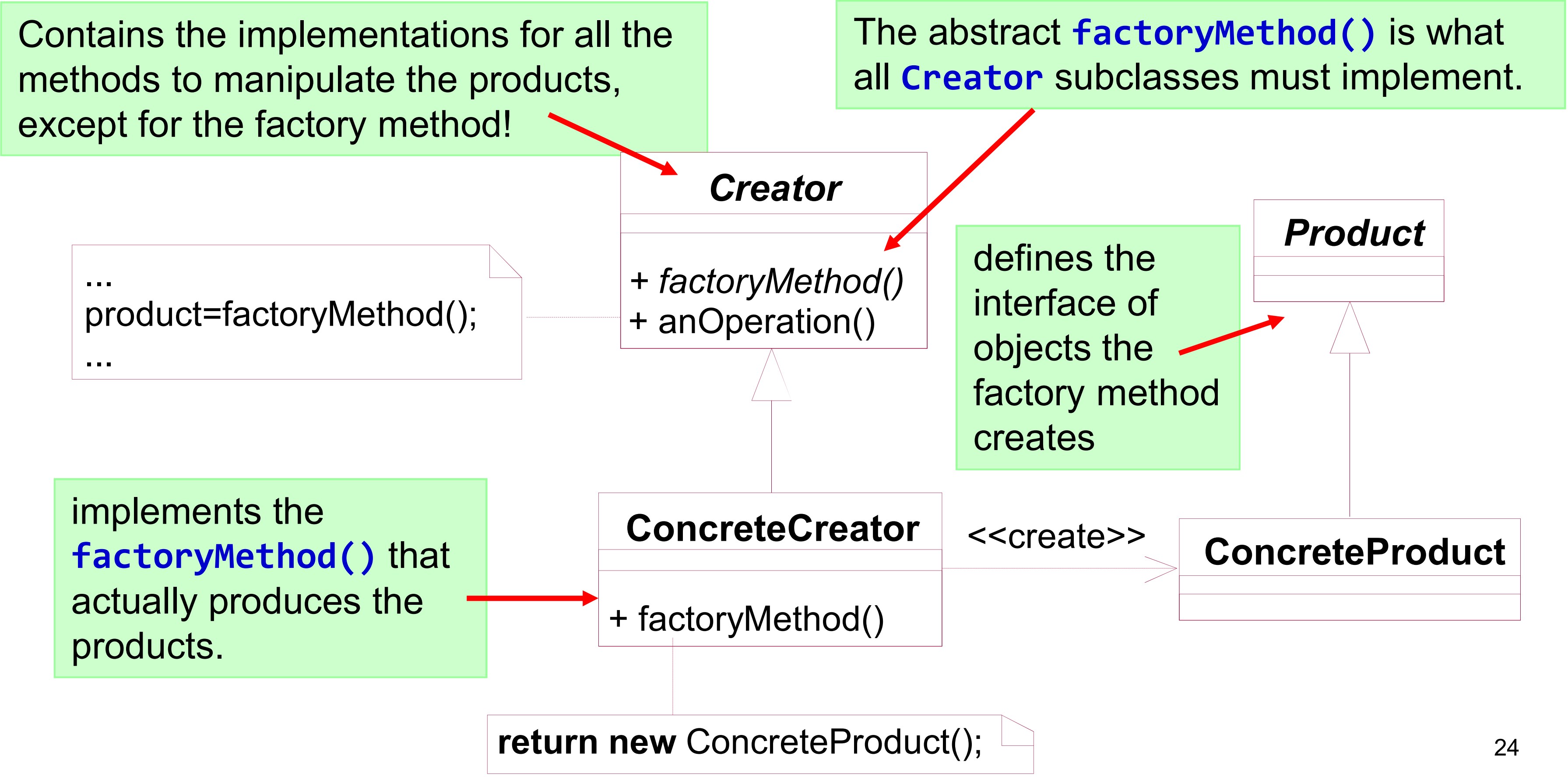
A factory method isolates the client from knowing what kind of concrete **Product** is actually created.

Factory Method Pattern

- All factory patterns encapsulate “object creation”
- Factory method pattern encapsulates object creation by letting subclasses decide what objects to create.
- Who are the players in this pattern?

Factory Method Pattern Defined

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



Contains the implementations for all the methods to manipulate the products, except for the factory method!

The abstract **factoryMethod()** is what all **Creator** subclasses must implement.

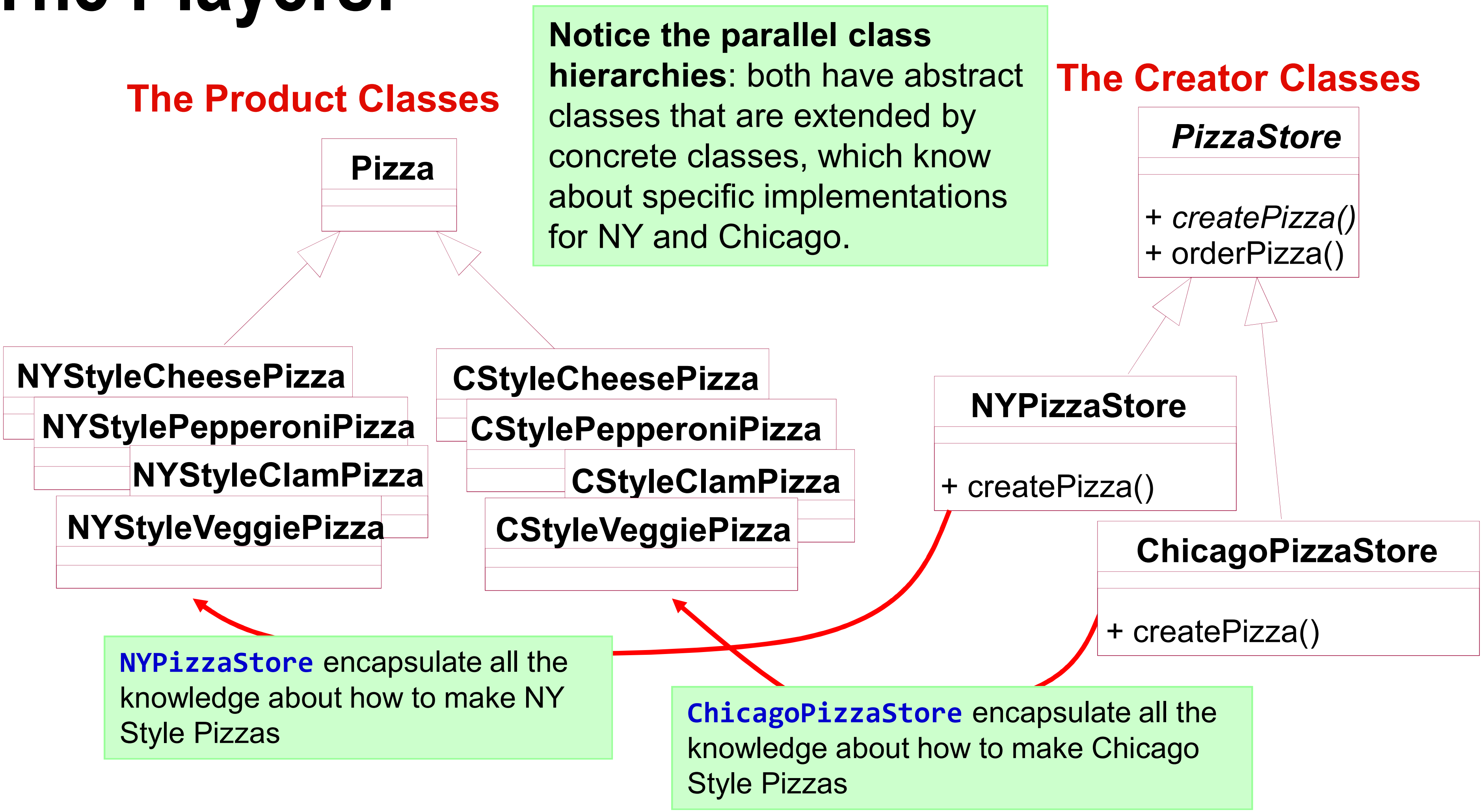
defines the interface of objects the factory method creates

implements the **factoryMethod()** that actually produces the products.

Advantages/Disadvantages

- ++
  - Eliminates the need to bind application-specific classes into your code
  - Provides hooks for subclassing.
    Creating objects inside a class with a factory method is always more flexible than creating an object directly.
  - This method gives subclasses a hook for providing an extended version of an object
  - Connects parallel heirarchies.
    Factory method localises knowledge of which classes belong together. Parallel class hierarchies result when a class delegates some of its responsibilities to a separate class.
- --
  - Clients might have to subclass the Creator class just to create a particular Concreate object.

The Players!



Notice the parallel class hierarchies: both have abstract classes that are extended by concrete classes, which know about specific implementations for NY and Chicago.

The Product Classes

Pizza

NYStyleCheesePizza

NYStylePepperoniPizza

NYStyleClamPizza

NYStyleVeggiePizza

CStyleCheesePizza

CStylePepperoniPizza

CStyleClamPizza

CStyleVeggiePizza

The Creator Classes

PizzaStore

+ *createPizza()*
+ orderPizza()

NYPizzaStore

+ createPizza()

ChicagoPizzaStore

+ createPizza()

`NYPizzaStore` encapsulate all the knowledge about how to make NY Style Pizzas

`ChicagoPizzaStore` encapsulate all the knowledge about how to make Chicago Style Pizzas

Factory method is key to encapsulating this knowledge!

Pizza class

```
public abstract class Pizza {
   protected String name;
   protected String dough;
   protected String sauce;
   protected ArrayList toppings = new ArrayList();
   void prepare() {
      System.out.println("Preparing " + name);
      System.out.println("Tossing dough...");
      System.out.println("Adding sauce...");
      System.out.println("Adding toppings: ");
      for (int i = 0; i < toppings.size(); i++) {
         System.out.println("   " + toppings.get(i));
      }
   }
   void bake() {
      System.out.println("Bake for 25 minutes at 350");
   }
   void cut() {
      System.out.println("Cutting the pizza into diagonal slices");
   }
   void box() {
      System.out.println("Place pizza in official PizzaStore box");
   }
   public String getName() {
      return name;
   }
}
```

Each pizza has a name, a type of dough, a type of sauce, and a set of toppings

Abstract class provides some basic defaults for baking, cutting and boxing

Preparation follows a number of steps in a particular sequence

New York and Chicago style cheese pizzas

```java
public class NYStyleCheesePizza extends Pizza {
   public NYStyleCheesePizza() {
      name = "NY Style Sauce and Cheese Pizza";
      dough = "Thin Crust Dough";
      sauce = "Marinara Sauce";
      toppings.add("Grated Reggiano Cheese");
   }
}
```

Each **Pizza** type has its own style sauce, dough and toppings

```java
public class ChicagoStyleCheesePizza extends Pizza {
   public ChicagoStyleCheesePizza() {
      name = "Chicago Style Deep Dish Cheese Pizza";
      dough = "Extra Thick Crust Dough";
      sauce = "Plum Tomato Sauce";
      toppings.add("Shredded Mozzarella Cheese");
   }
   void cut() {
      System.out.println("Cutting the pizza into square slices");
   }
}
```

The Chicago Pizza overides **cut()** method so that the pieces are cut into squares

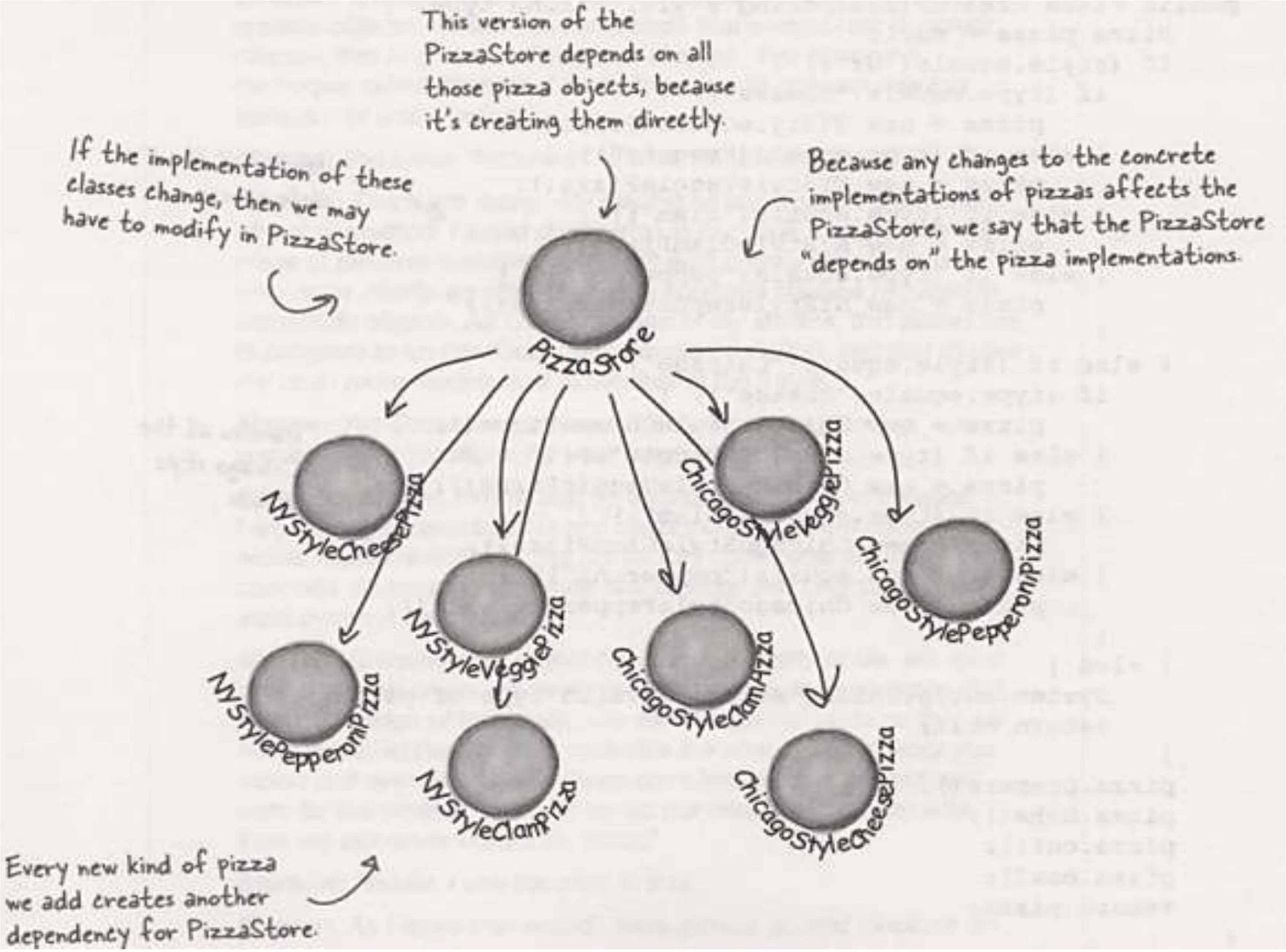
Test Drive

```
public class PizzaTestDrive {
   public static void main(String[] args) {

      PizzaStore nyStore = new NYPizzaStore();
      PizzaStore chicagoStore = new ChicagoPizzaStore();

      Pizza pizza = nyStore.orderPizza("cheese");
      System.out.println("Ethan ordered a " +
                          pizza.getName() + "\n");

      pizza = chicagoStore.orderPizza("cheese");
      System.out.println("Joel ordered a " +
                          pizza.getName() + "\n");

      // ...

   }
}
```

What we have learned from Factory Method

- First of all let’s take a look on what we tried to avoid

```
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 (style.equals("Chicago")) {
         if (type.equals("cheese")) {
            pizza = new ChicagoStyleCheesePizza();
         } else if (type.equals("veggie")) {
            pizza = new ChicagoStyleVeggiePizza();
         ...
      }
      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
   }
}
```



This version of the PizzaStore depends on all those pizza objects, because it's creating them directly.

If the implementation of these classes change, then we may have to modify in PizzaStore.

Because any changes to the concrete implementations of pizzas affects the PizzaStore, we say that the PizzaStore "depends on" the pizza implementations.

Every new kind of pizza we add creates another dependency for PizzaStore.

Design Principle

Dependency Invertion Principle

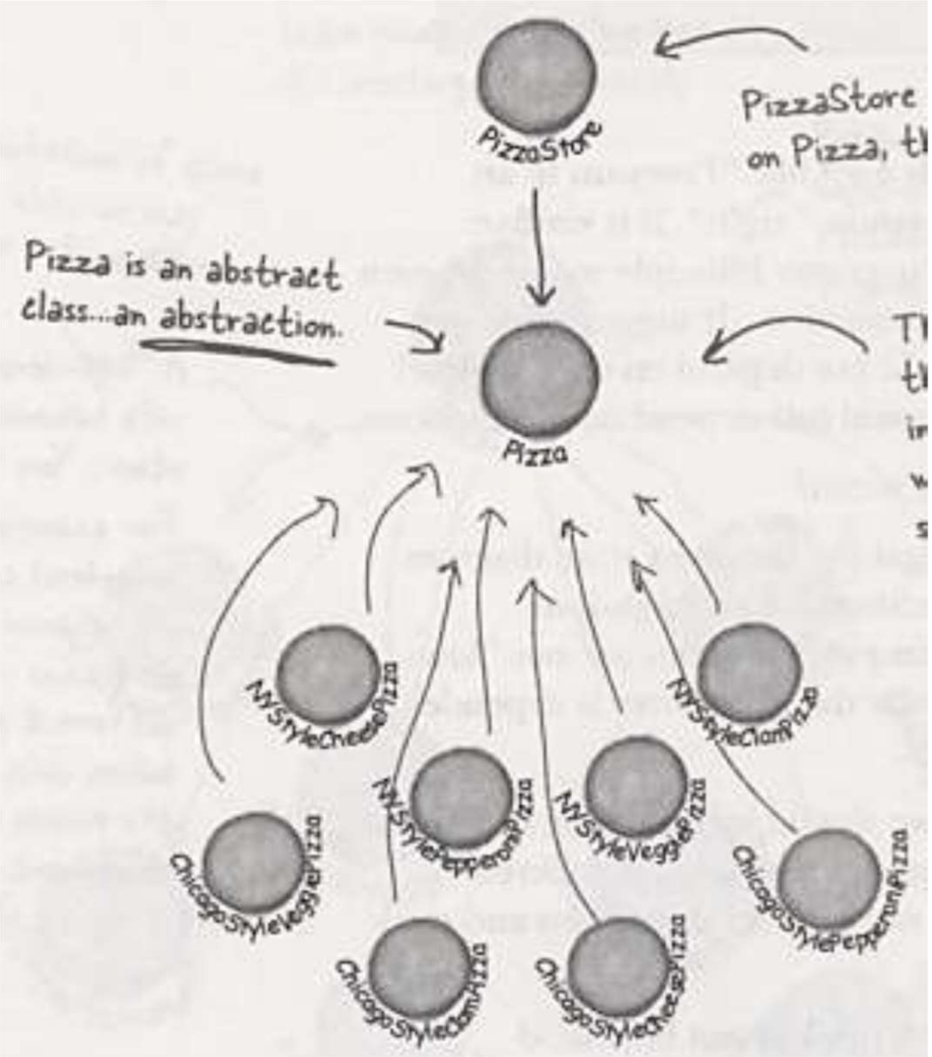
Depend upon abstractions.
Do not depend upon concrete classes.

DIP explained

Depend upon abstractions.
Do not depend upon concrete classes.

High-level components should not depend on low level components; they should both depend on abstractions

PizzaStore is ”*high level component*”
Pizzas are ”*low level components*”



Guidlines to follow the DIP

- **No variable should hold a reference to a concrete class**
  - If you use `new` you are holding a reference to a concrete class. Use a factory to get around that
- **No class should derive from a concrete class**
  - If you derive from a concrete class, you’re depending on a concrete class. Derive from an abstraction like an interface or an abstract class.
- **No method should override an implemented method of any of its base classes**
  - If you override an implemented method, then your base class wasn’t really an abstraction to start with.
    Those methods implemented in the base class are meant to be shared by all your subclasses.

Meanwhile, back at the PizzaStore….

- Things are going good, but you have learned that a few franchises are substituting inferior ingredients
- How are you going to ensure each factory is using quality ingredients?
  - You are going to build a factory that produces them and ships them to your franchises!
- One problem:
  - Franchises are located in different regions so what is red sauce in NY is not red sauce in Chicago!
- So: same product families (dough, cheese, sauce etc.) but different implementations based on region.

Building the Ingredient Factory

- **Ingredient factory**: creates each ingredient in the ingredient family (but does not handle regional differences yet!)

```
public interface PizzaIngredientFactory {
   public Dough createDough();
   public Sauce createSauce();
   public Cheese createCheese();
   public Veggies[] createVeggies();
   public Pepperoni createPepperoni();
   public Clams createClam();
}
```

For each ingredient we define a create method in our interface.

If we had some **common** “machinery” to implement in each instance of factory, we could have made this **abstract** instead.

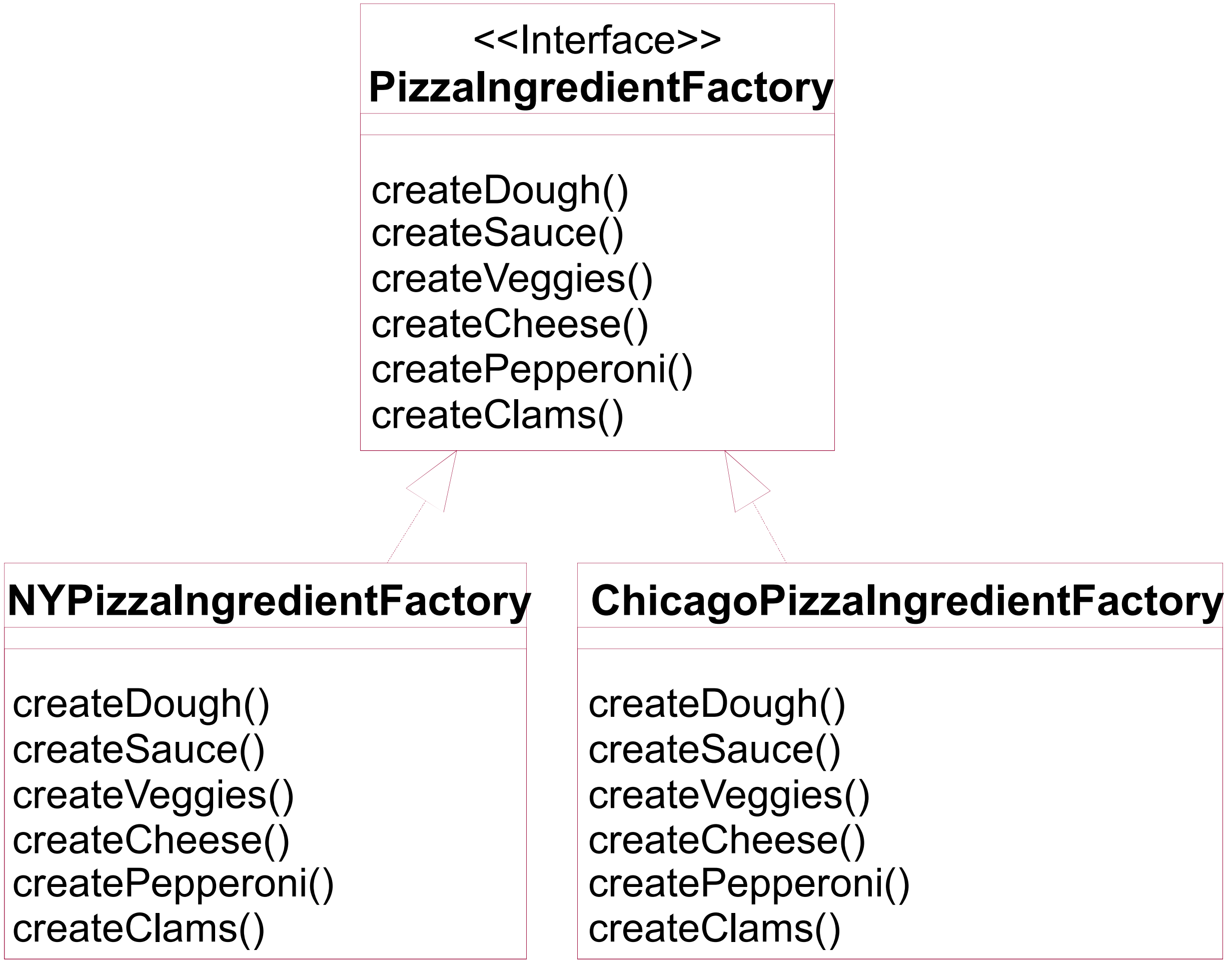
Lots of new classes here, one per ingredient

What to do next?

- Build a factory for each region.
  To do this, create a subclass of `PizzaIngredientFactory` that implements each create method.
- Implement a set of ingredient classes to be used with the factory, like `RedPeppers`, `ThickCrustDough`, etc. These classes can be shared among regions where appropriate.
- Then we still need to hook all this up by working our new ingredient factories into the old `PizzaStore` code.

Abstract factory pattern

Build a factory for each region



<<Interface>>
PizzaIngredientFactory

createDough()
createSauce()
createVeggies()
createCheese()
createPepperoni()
createClams()

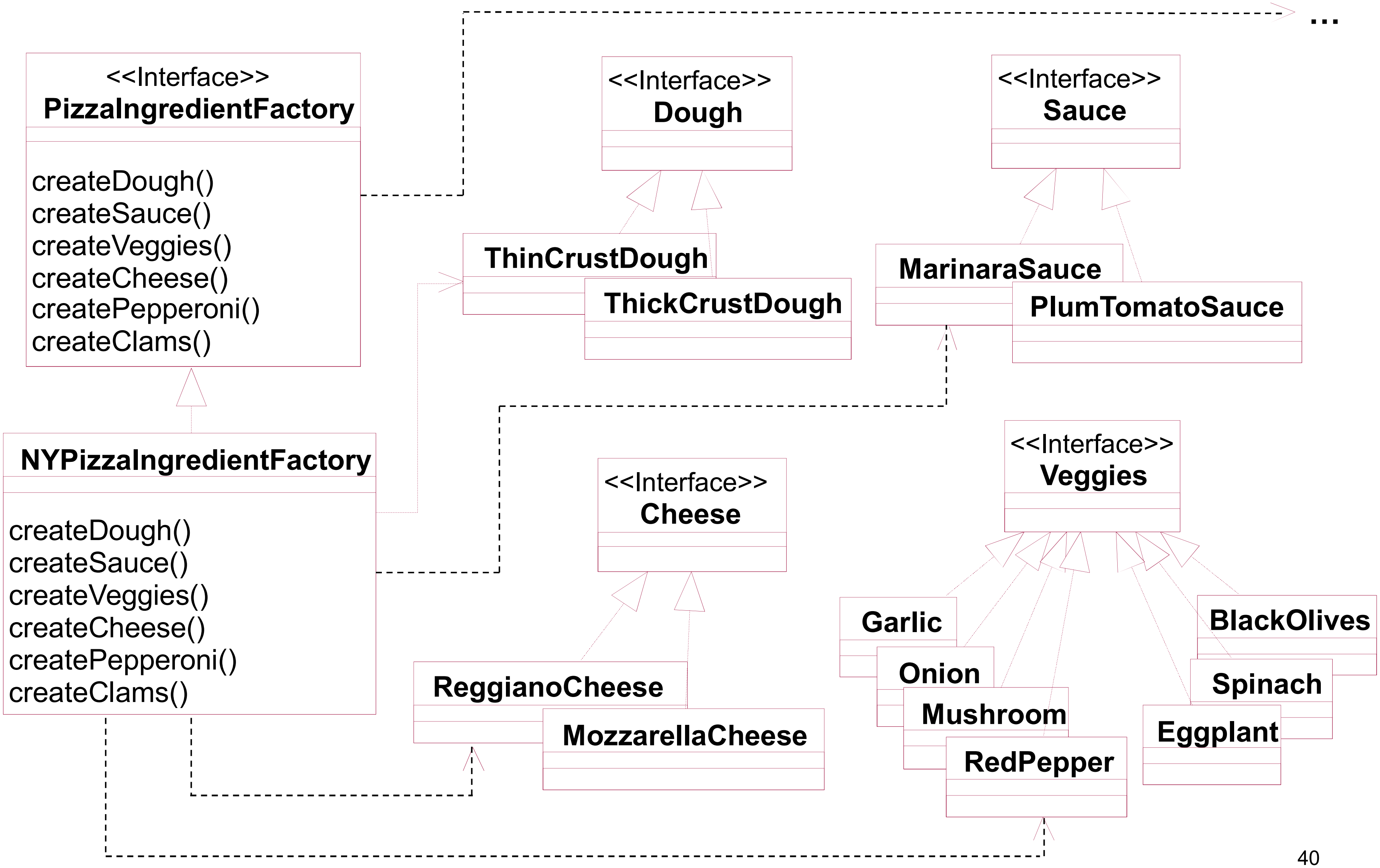
NYPizzaIngredientFactory

createDough()
createSauce()
createVeggies()
createCheese()
createPepperoni()
createClams()

ChicagoPizzaIngredientFactory

createDough()
createSauce()
createVeggies()
createCheese()
createPepperoni()
createClams()

Build a factory for New york



...

<<Interface>>
PizzaIngredientFactory

createDough()
createSauce()
createVeggies()
createCheese()
createPepperoni()
createClams()

NYPizzaIngredientFactory

createDough()
createSauce()
createVeggies()
createCheese()
createPepperoni()
createClams()

<<Interface>>
Dough

ThinCrustDough

ThickCrustDough

<<Interface>>
Sauce

MarinaraSauce

PlumTomatoSauce

<<Interface>>
Cheese

ReggianoCheese

MozzarellaCheese

<<Interface>>
Veggies

Garlic

Onion

Mushroom

RedPepper

BlackOlives

Spinach

Eggplant

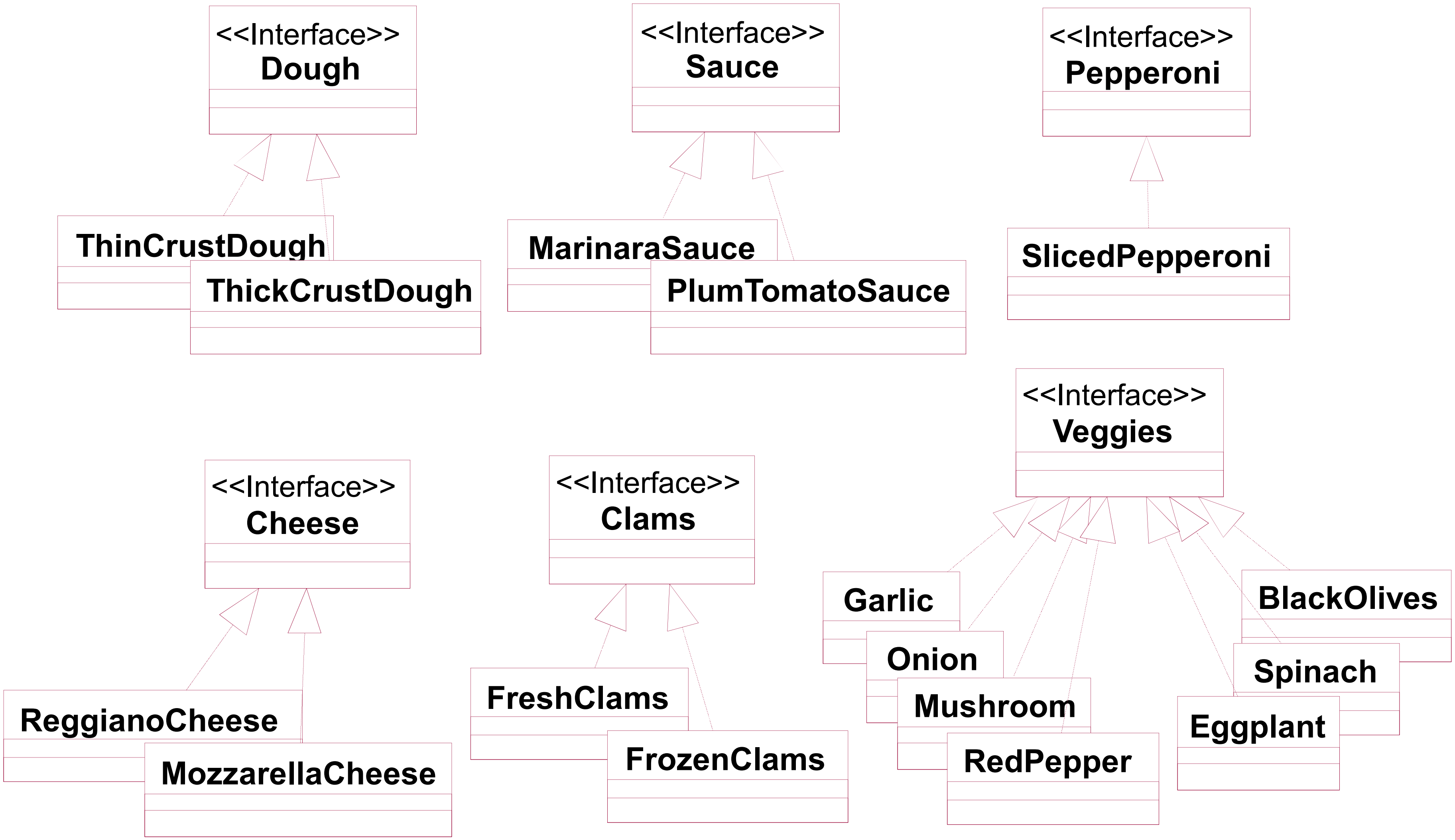
(1) The New York Ingredient Factory

```
public class NYPizzaIngredientFactory
         implements PizzaIngredientFactory {
   public Dough createDough() {
     return new ThinCrustDough();
   }
   public Sauce createSauce() {
      return new MarinaraSauce();
   }
   public Cheese createCheese() {
      return new ReggianoCheese();
   }
   public Veggies[] createVeggies() {
      Veggies veggies[] =
          { new Garlic(), new Onion(),
            new Mushroom(), new RedPepper() };
     return veggies;
   }
   // other ingredients
}
```

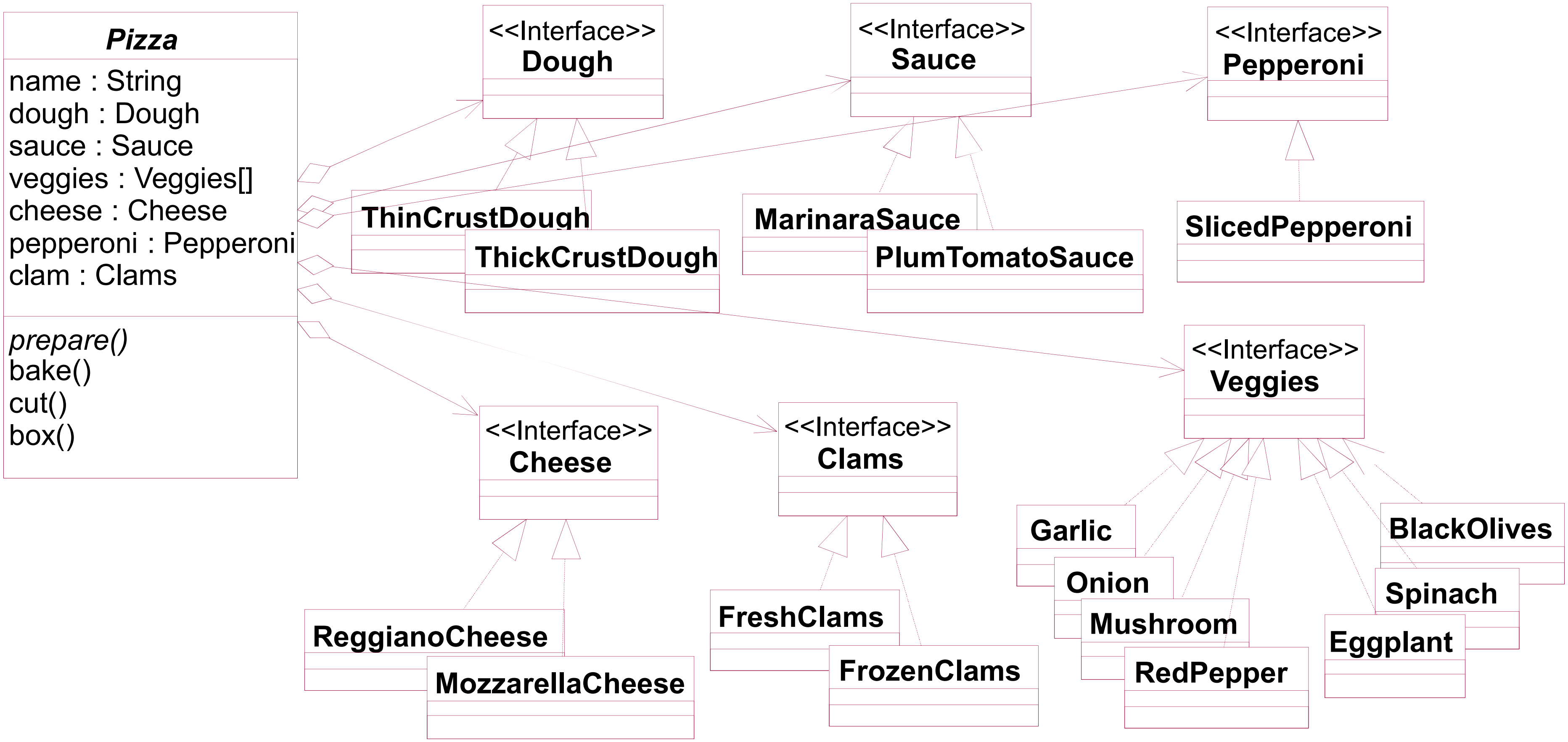
The NY ingredient factory implements the interface for all ingredient factories

For each ingredient in the ingredient family, we create the NY version.

Implement a set of ingredient classes



Reworking the Pizzas



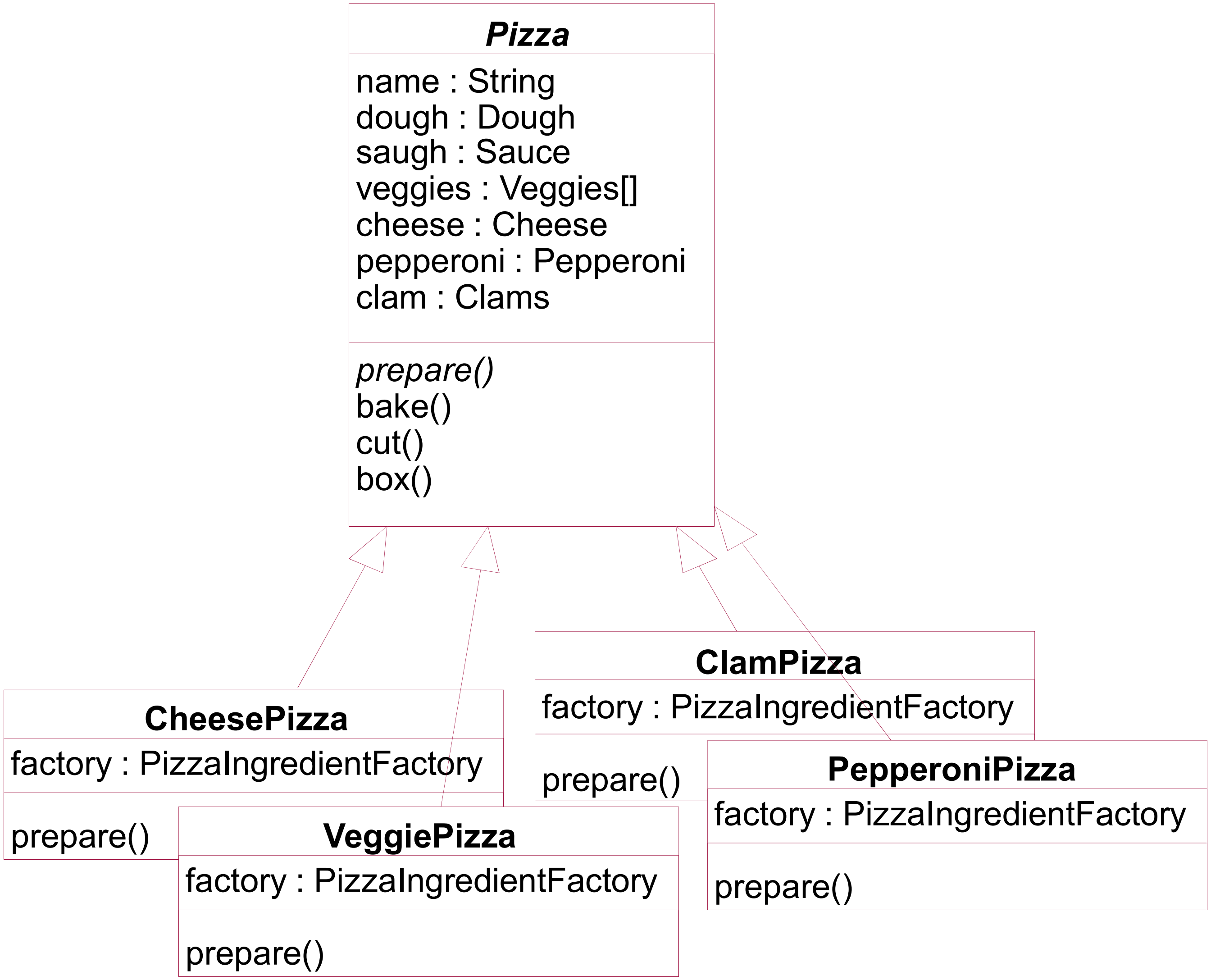
Reworking the Pizzas

```
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() {
      System.out.println("Cutting the pizza into diagonal slices");
   }
   void box() {
      System.out.println("Place pizza in official PizzaStore box");
   }
   void setName(String name) { this.name = name; }
   String getName() { return name; }
}
```

Each pizza holds **a set of ingredients** that are used in its prep.

The **`prepare()`** method is **`abstract`**. This is where we are going to **collect the ingredients** needed for the pizza which will come from the ingredient factory.

Hook pizza working ingredient factories



Pizza

name : String
dough : Dough
saugh : Sauce
veggies : Veggies[]
cheese : Cheese
pepperoni : Pepperoni
clam : Clams

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

CheesePizza

factory : PizzaIngredientFactory

prepare()

VeggiePizza

factory : PizzaIngredientFactory

prepare()

ClamPizza

factory : PizzaIngredientFactory

prepare()

PepperoniPizza

factory : PizzaIngredientFactory

prepare()

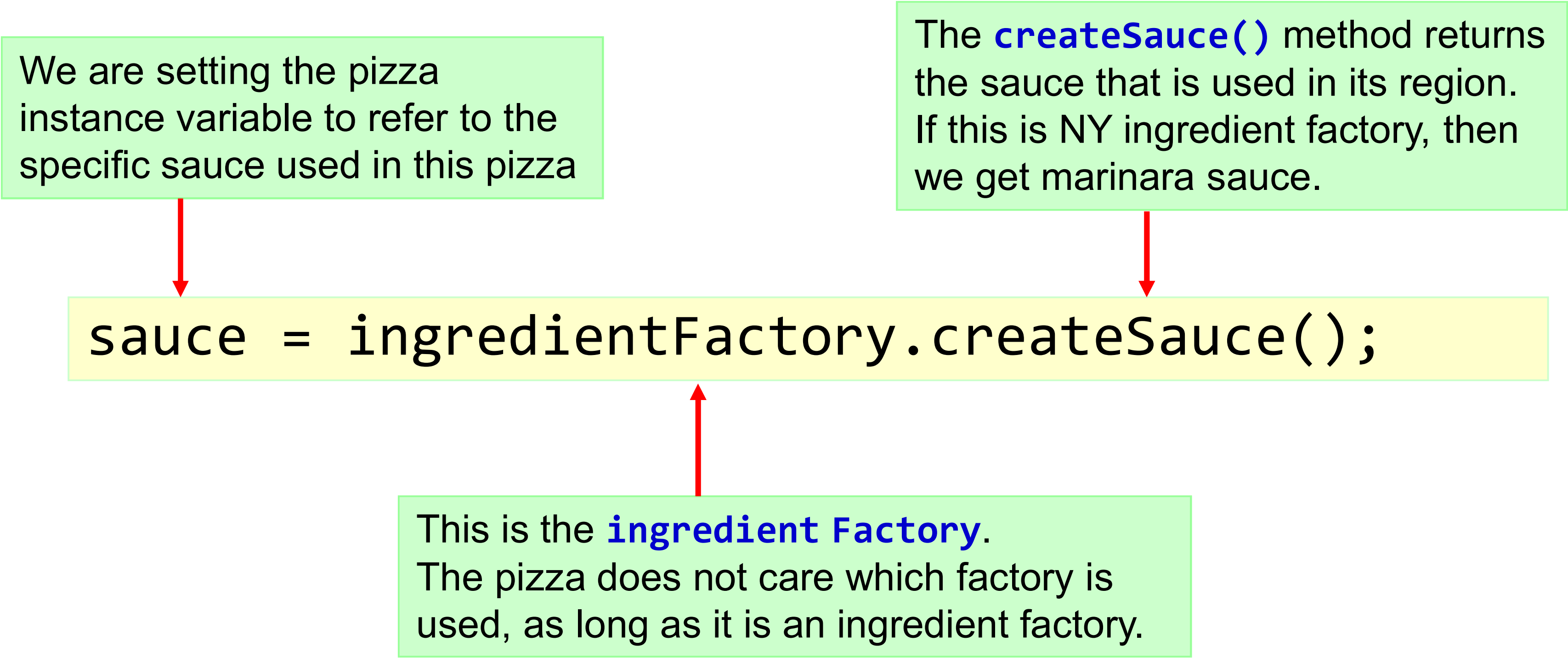
Reworking the Pizzas (cont)

```
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();
   }
}
```

To make a pizza now, we need a factory to provide the ingredients. So each class gets a factory passed into its constructor, and its stored in an instance variable.

The **prepare()** method steps through the creating a cheese pizza, and each time it needs an ingredient, it asks the factory to produce it.

Code Up Close!



We are setting the pizza instance variable to refer to the specific sauce used in this pizza

The **createSauce()** method returns the sauce that is used in its region. If this is NY ingredient factory, then we get marinara sauce.

```
sauce = ingredientFactory.createSauce();
```

This is the **ingredient Factory**.
The pizza does not care which factory is used, as long as it is an ingredient factory.

Revisiting the Pizza Store

```java
public class NYPizzaStore extends PizzaStore {

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

      if (item.equals("cheese")) {
         pizza = new CheesePizza(ingredientFactory);
         pizza.setName("New York Style Cheese Pizza");
      } else if (item.equals("veggie")) {
         pizza = new VeggiePizza(ingredientFactory);
         pizza.setName("New York Style Veggie Pizza");
      } // same for all other pizza types.
      return pizza;
   }
}
```

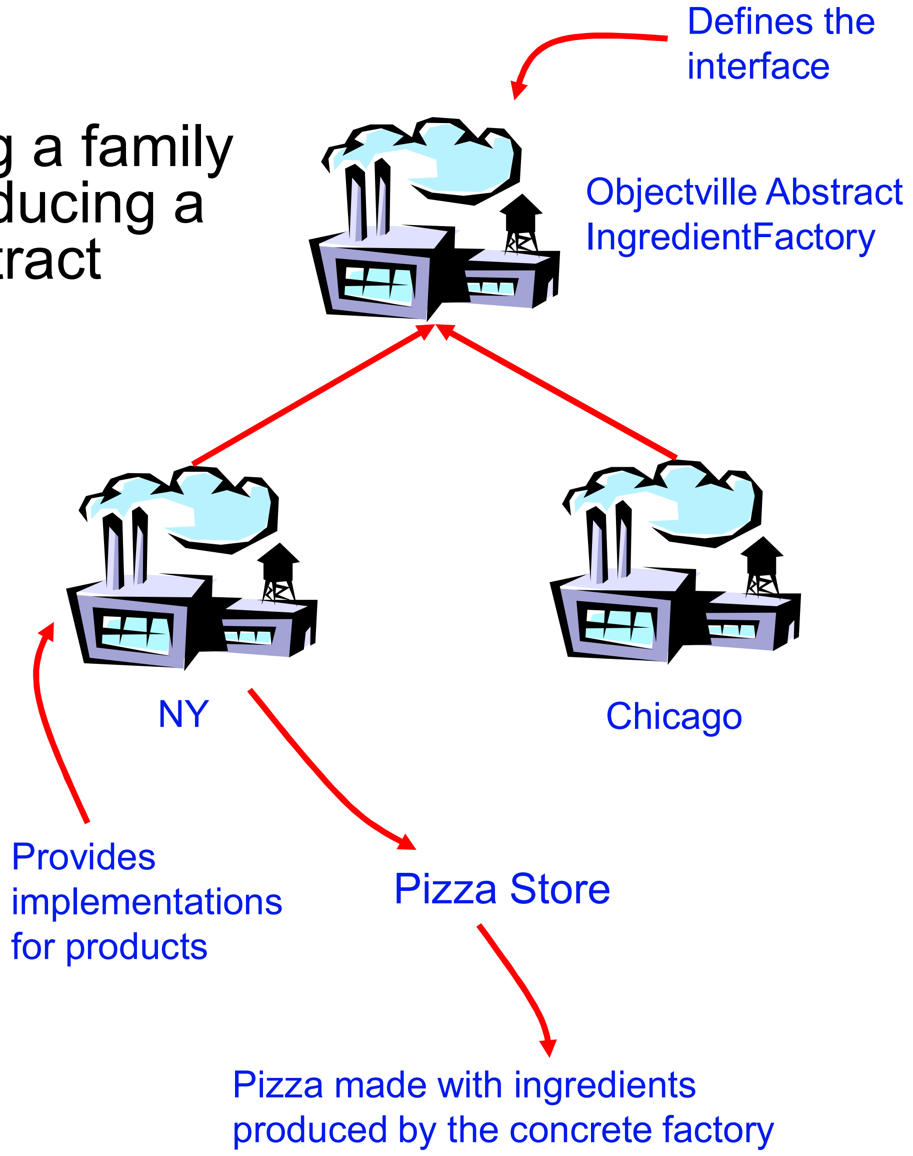
The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY style pizzas.

We now pass each pizza the factory that should be used to produce its ingredients

For each type of pizza we instantiate a new `Pizza` and give it the `factory` that it needs to get its ingredients.

What have we done?

- We provided a means of creating a family of ingredients for pizzas by introducing a new type of factory called -- Abstract Factory.
- Abstract Factory:
  - Gives an interface for creating a family of products.
  - By writing code that uses this interface we decouple our code from the actual factory that creates the products.
  - Allows us to implements a variety of factories that produce products meant for different contexts.
  - Decoupling --> enables substitution of different factories to get different behaviors.



Defines the interface

Objectville Abstract IngredientFactory

NY

Chicago

Provides implementations for products

Pizza Store

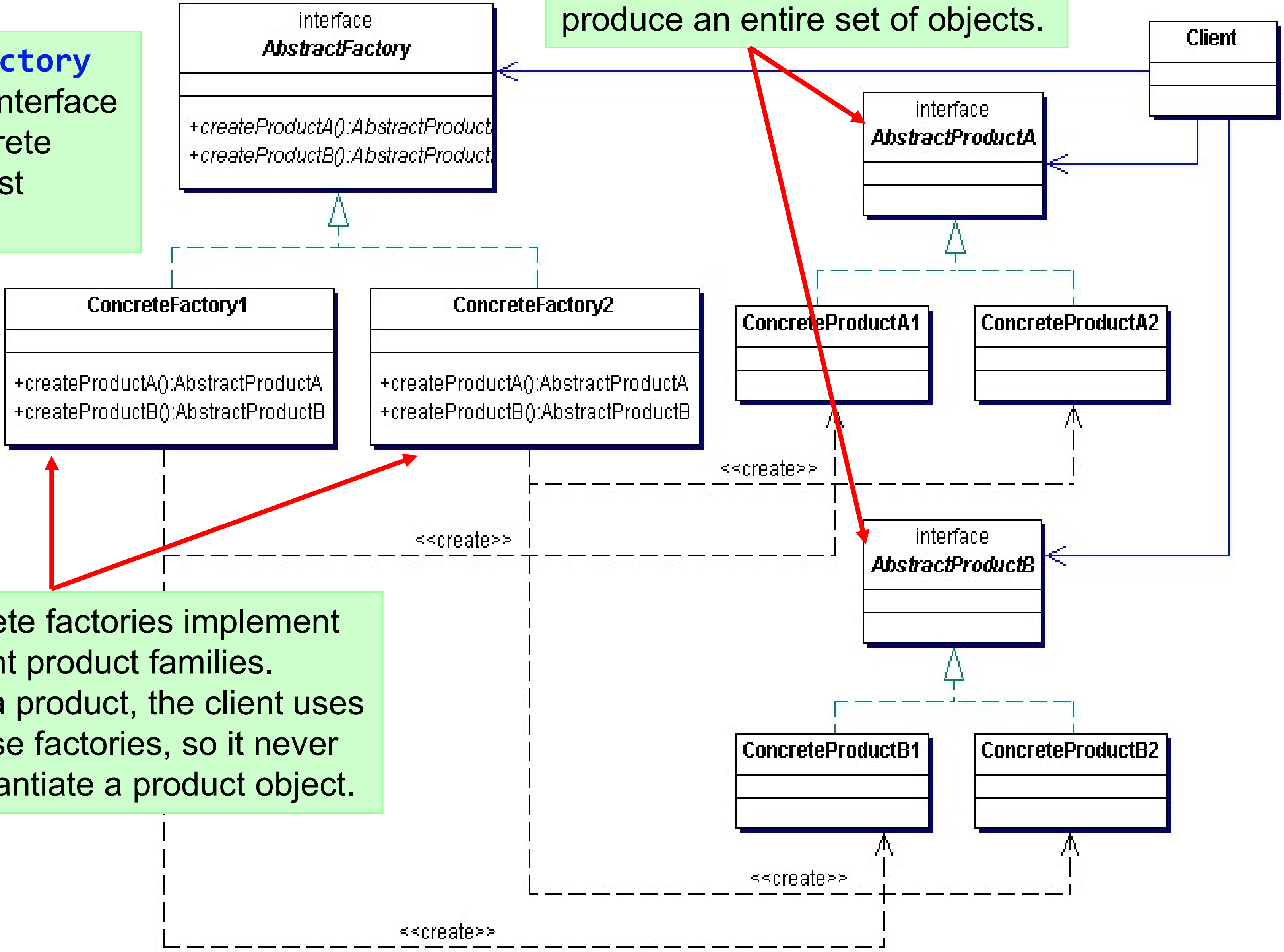
Pizza made with ingredients produced by the concrete factory

Abstract Factory Pattern

- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Applicability
  - When clients cannot anticipate groups of classes to instantiate
  - A system should be independent of how its products are created, composed, and represented.
  - A system should be configured with one of multiple families of products.
  - A family of related product objects is designed to be used together, and you need to enforce this constraint.
  - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure

AbstractFactory defines the interface that all concrete factories must implement,



This is the **product family**.
Each concrete factory can produce an entire set of objects.

The concrete factories implement the different product families.
To create a product, the client uses one of these factories, so it never has to instantiate a product object.

Summary

- All factories encapsulate object creation
- Simple factory, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes.
- Factory method relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects
- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.

Summary

- All factory methods promote loose coupling by reducing the dependency of your application on concrete classes.
- The intent of Factory method is to allow a class to defer instantiation to its subclasses.
- The intent of the Abstract Factory is to create families of related objects without having to depend on their concrete classes.
- The Dependency Inversion principle guides us to avoid dependencies on concrete types and to strive for abstractions.
- Factories are a powerful technique for coding to abstractions, not concrete classes.

How important is it to use Factories?

- Factories are powerful tools
  - Great benefit when trying to conform to DIP
- Heuristics for use:
  - Strict interpretation -- use factories for every volatile class
  - But don’t use factories for everything: too extreme
  - Initially - don’t start out using factories, add them in as you see the need for them
    - Might need to “spoof” objects during testing
- Remember: Factories are a complexity that can often be avoided in the early phases
  - They unnecessarily complicate designs!