Head First Design Patterns

Chapter 4

Factory Pattern

Calling the constructor

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

This type of code will often lead to problems when new types have to be added.

Another example in similar vein

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

Identifying the aspects that vary

- If the pizza shop decides to change the types of pizza it offers, the orderPizza method has to be changed.

Another example in similar vein

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

Part that varies.

Part that remains constant

Encapsulating object creation

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

SimplePizzaFactory

Building a simple pizza factory

```
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;
   }
}
```

Reworking the PizzaStore Class

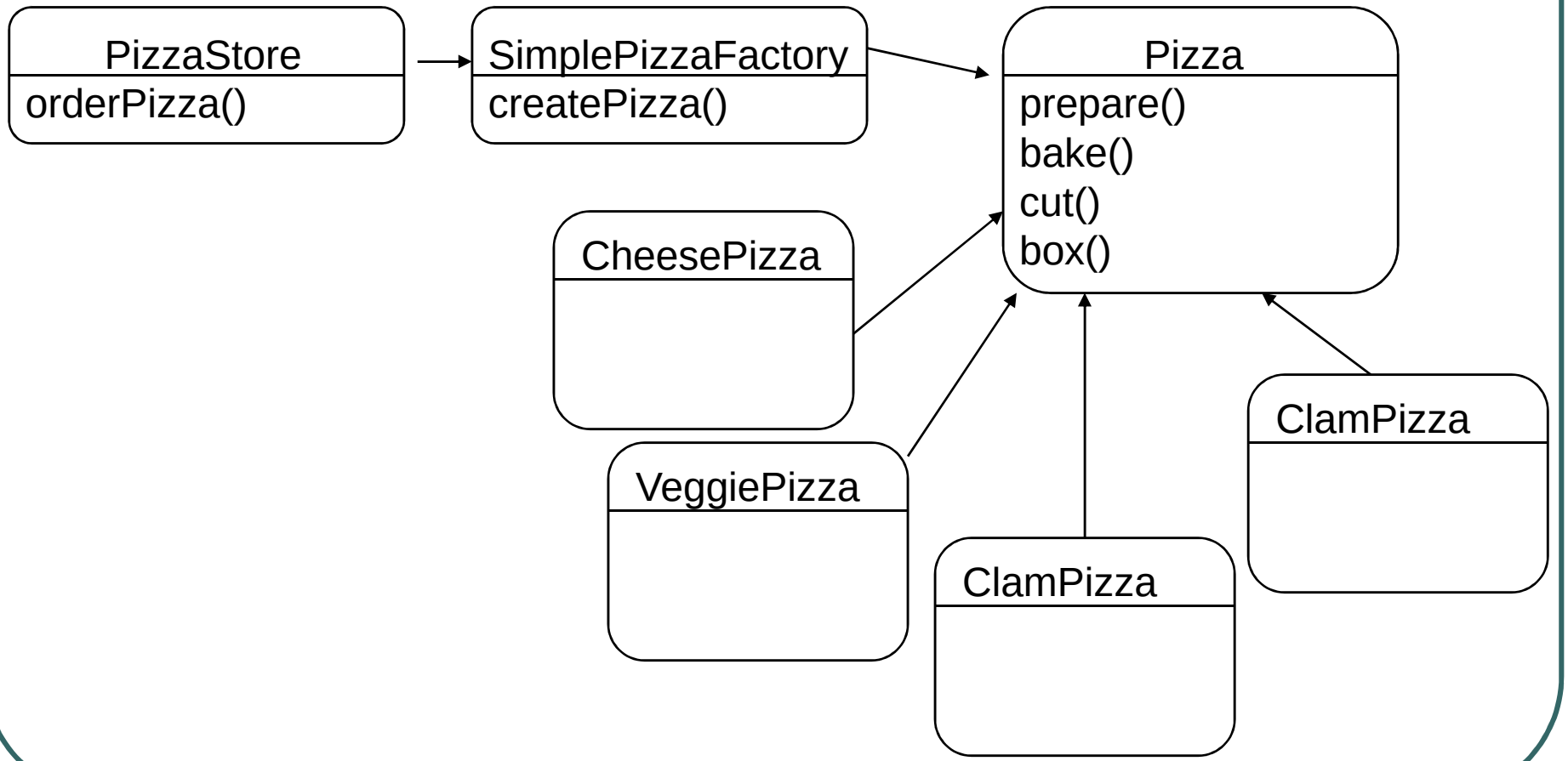
```
public class PizzaStore {
   SimplePizzaFactory factory;
   public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
   }
   public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
   }
}
```

Complete example for Simple Factory

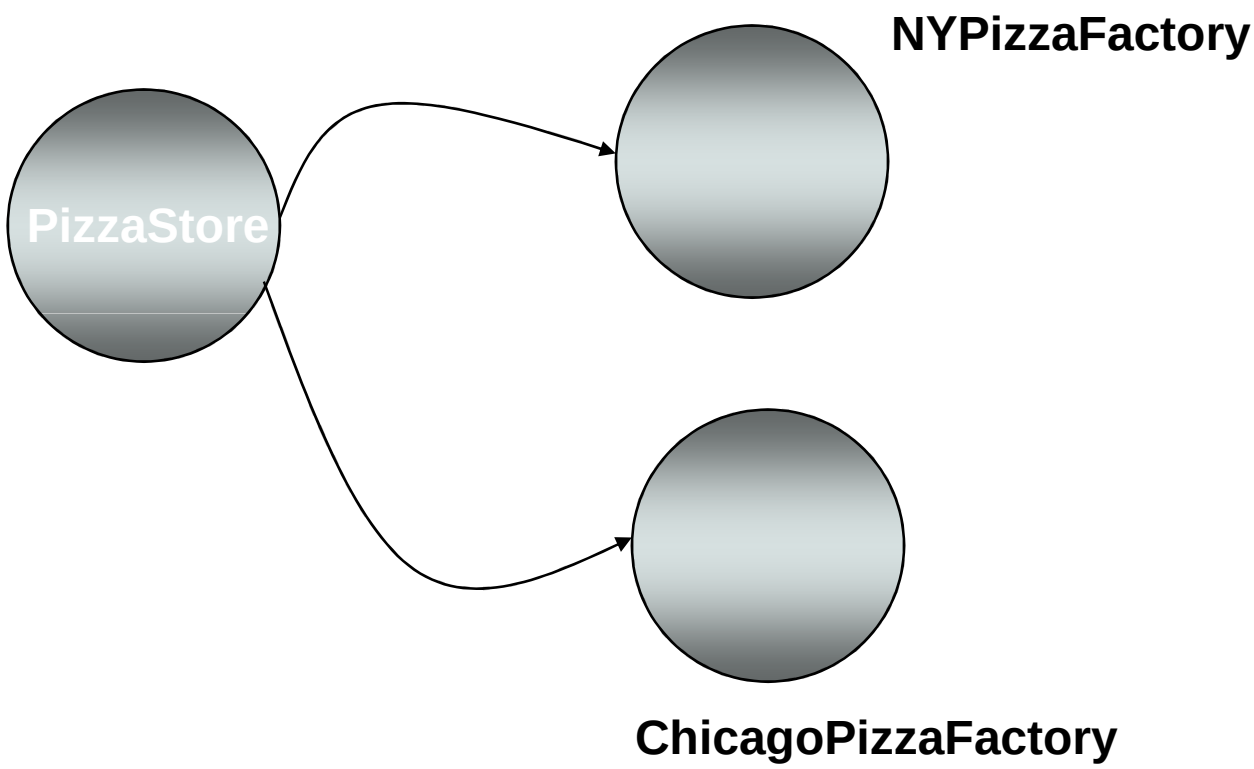
```
SimplePizzaFactory factory = new SimplePizzaFactory();
PizzaStore store = new PizzaStore(factory);
Pizza pizza = store.orderPizza("cheese");
```

- Look at Eclipse

Simple Factory Defined



Creating multiple factories



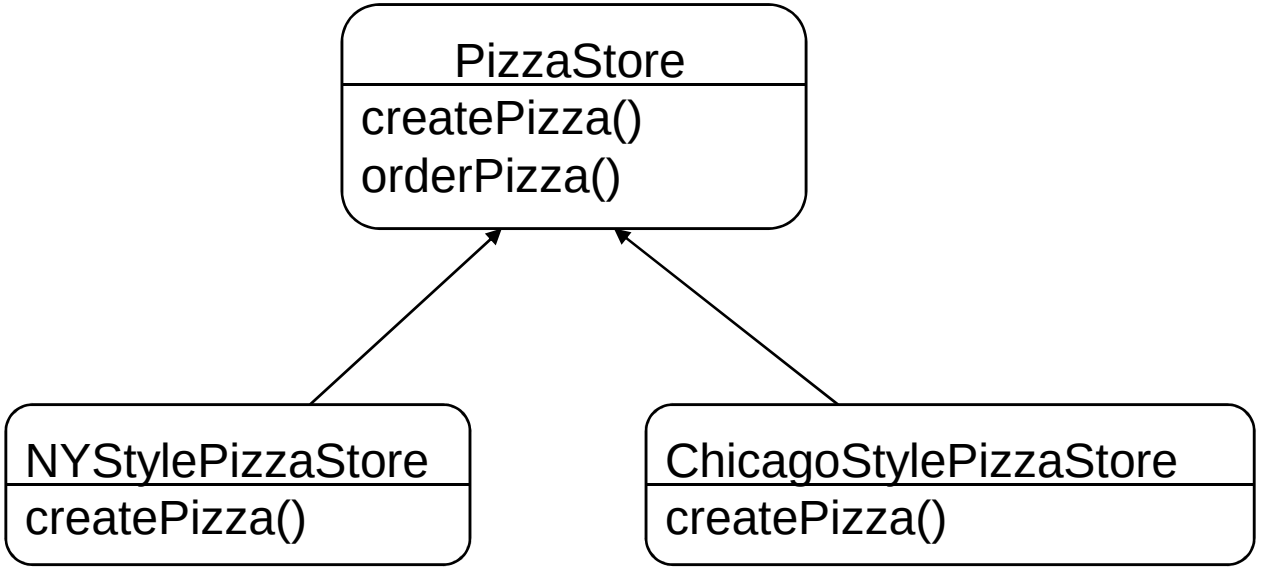
Creating multiple instances

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
Pizza pizza = nyStore.orderPizza("cheese");

ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
Pizza pizza = chicagoStore.orderPizza("cheese");
```

Alternate approach – Abstract method – a framework for the pizza store

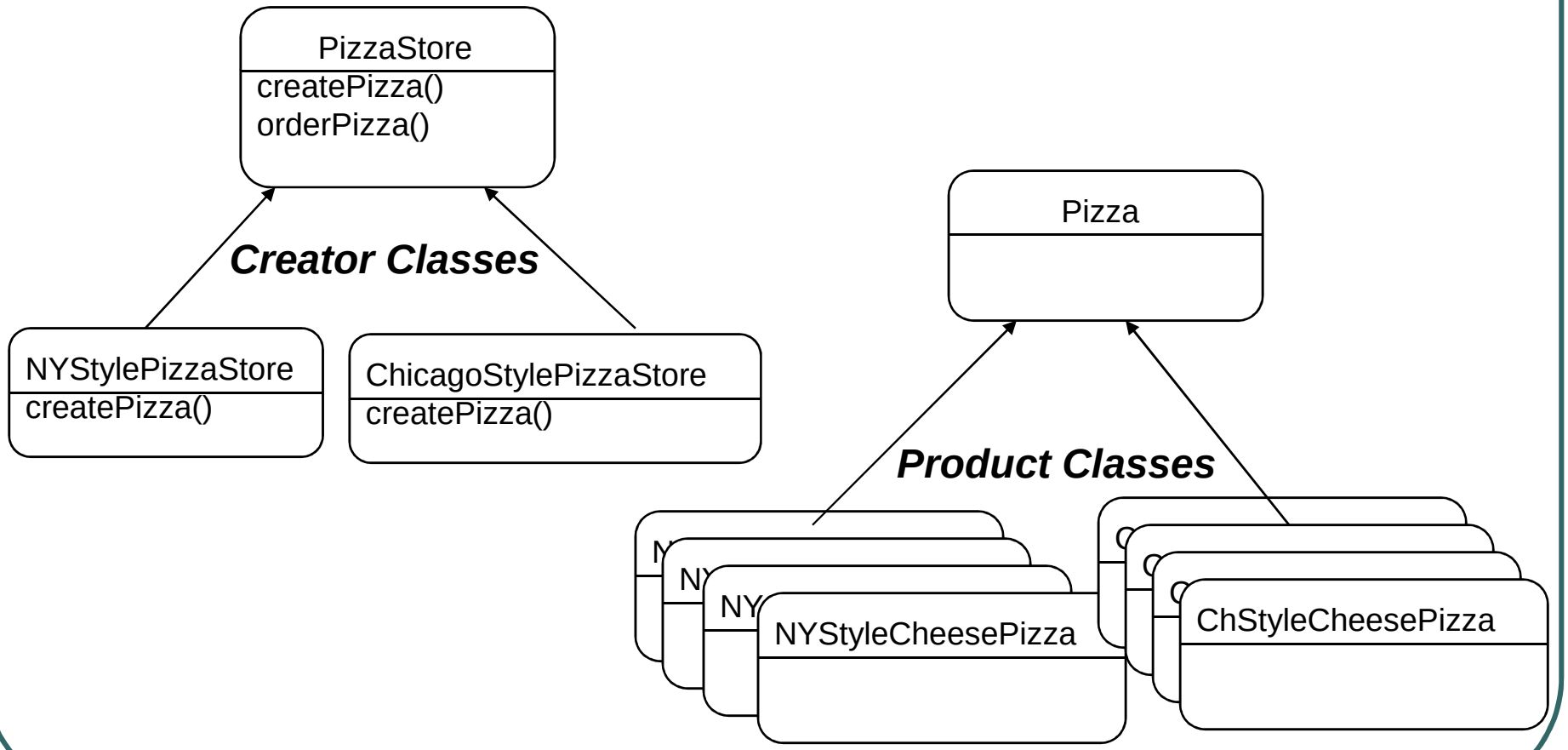
```
public abstract class PizzaStore {
   abstract Pizza createPizza(String item);

   public Pizza orderPizza(String type) {
      Pizza pizza = createPizza(type);
      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
   }
}
```

Allowing the subclasses to decide



A factory method handles object creation and encapsulates it in the subclass. This decouples the client code in the super class from the object creation that happens in the subclass.

Factory Method Pattern



Factory Method Pattern defined

The factory method pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclass.