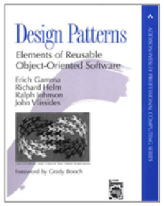


Design Pattern Definitions from the GoF Book

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 subclasses.

Creational Patterns

- **The Factory Method Pattern**
- **The Abstract Factory Pattern**
- **The Singleton Pattern**
- **The Builder Pattern**
- **The Prototype Pattern**

Structural Patterns

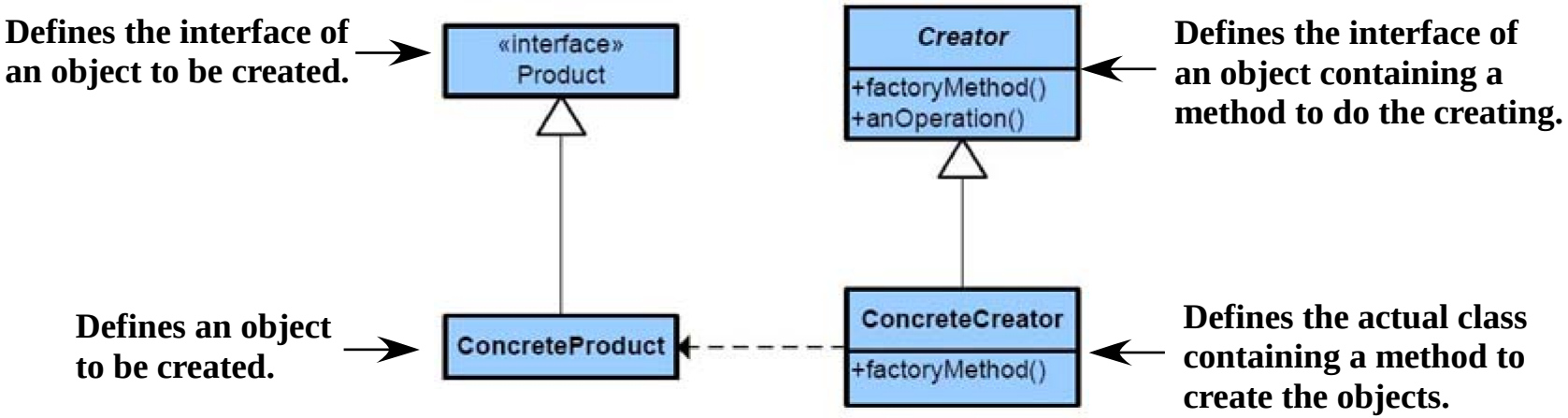
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**
- **The Flyweight Pattern**

Behavioral Patterns

- **The Strategy Pattern**
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **The Observer Pattern**
  Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatially.
- **The Command Pattern**
- **The Template Method Pattern**
- **The Iterator Pattern**
- **The State Pattern**
- **The Chain of Responsibility Pattern**
- **The Interpreter Pattern**
- **The Mediator Pattern**
- **The Memento Pattern**
- **The Visitor Pattern**

Design Patterns: The Factory Method
Quick Overview

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



Design Patterns: The Factory Method

Suppose you have several places in your code where you create ducks. Maybe the code looks like this...

```
Duck duck;

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

So what's wrong with using *new* here?

Design Patterns: The Factory Method


```
Pizza *PizzaShop::orderPizza()
{
   Pizza *pizza = new Pizza();

   pizza->prepare();
   pizza->bake();
   pizza->cut();
   pizza->box();

   return pizza;
}
```

A Typical Pizza Shop method

What if I want a special type of pizza?

```
Pizza *PizzaShop::orderPizza(string type)
{
   Pizza *pizza;

   if(type == “cheese”)
     pizza = new CheesePizza();
   else if(type == “greek”)
     pizza = new GreekPizza();
   else if(type == “pepperoni”)
     pizza = new PepperoniPizza()

   pizza->prepare();
   pizza->bake();
   pizza->cut();
   pizza->box();

   return pizza;
}
```

Cheese, Greek, and Pepperoni are subclasses of Pizza.

That’s better, but...

*Remember that old **Demon Change**?*

Design Patterns: The Factory Method

```
Pizza *PizzaShop::orderPizza(string type)
{
   Pizza *pizza;

   if(type == “cheese”)
      pizza = new CheesePizza();
   else if(type == “greek”)
      pizza = new GreekPizza();
   else if(type == “pepperoni”)
      pizza = new PepperoniPizza()

   pizza->prepare();
   pizza->bake();
   pizza->cut();
   pizza->box();

   return pizza;
}
```

Not good! This has to change every time we add a new type of pizza and everywhere in the code where pizzas are created.

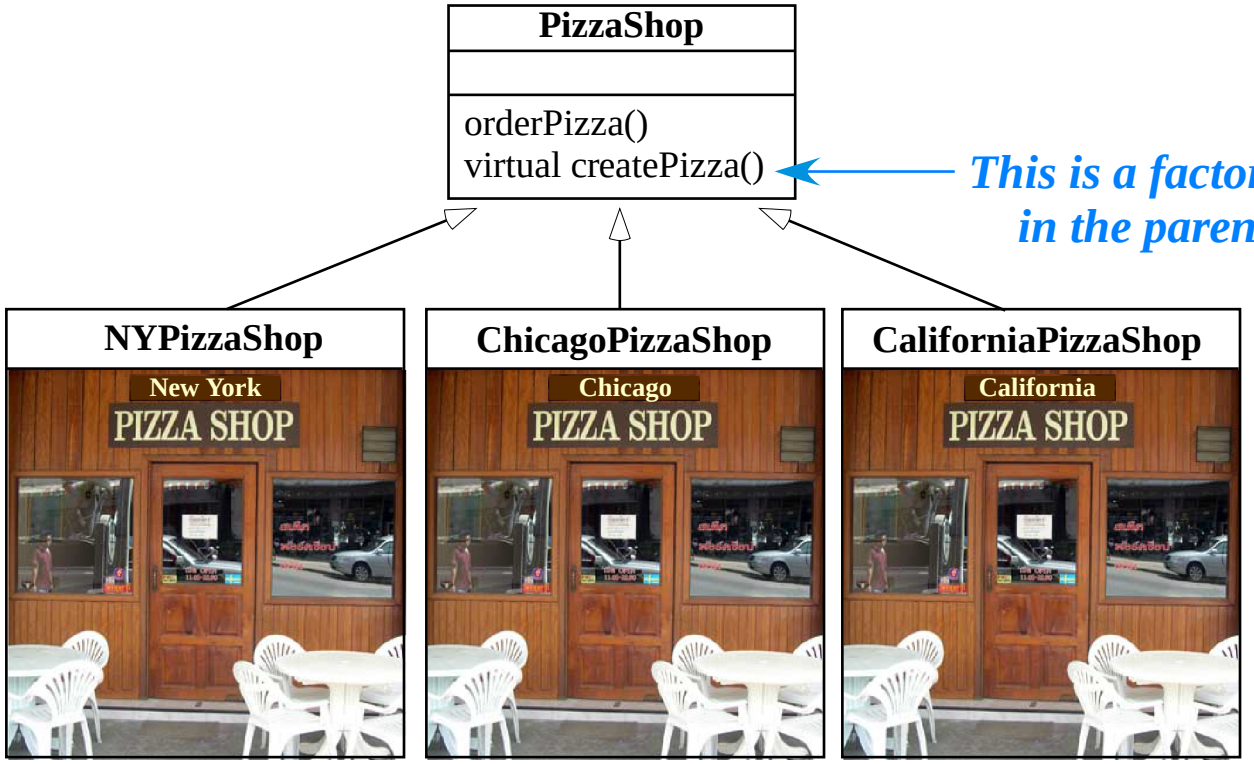
```
Pizza *PizzaShop::createPizza(string type)
{
    Pizza *pizza = NULL;

    if(type == “cheese”)
      pizza = new CheesePizza();
   else if(type == “greek”)
      pizza = new GreekPizza();
   else if(type == “pepperoni”)
      pizza = new PepperoniPizza();
   else if (type == “clam”)
      pizza = new ClamPizza();
   else if (type == “veggie”)
      pizza = new VeggiePizza();

   return pizza;
}
```

```
Pizza *PizzaShop::orderPizza(string type)
{
   Pizza *pizza;
   pizza = this->createPizza(type);
   // Same prepare, bake, cut, box goes here
   return pizza;
}
```

Now we’re cooking! ... Well, almost.

Not all pizzas are the same style, even if they are the same type.

Design Patterns: The Factory Method

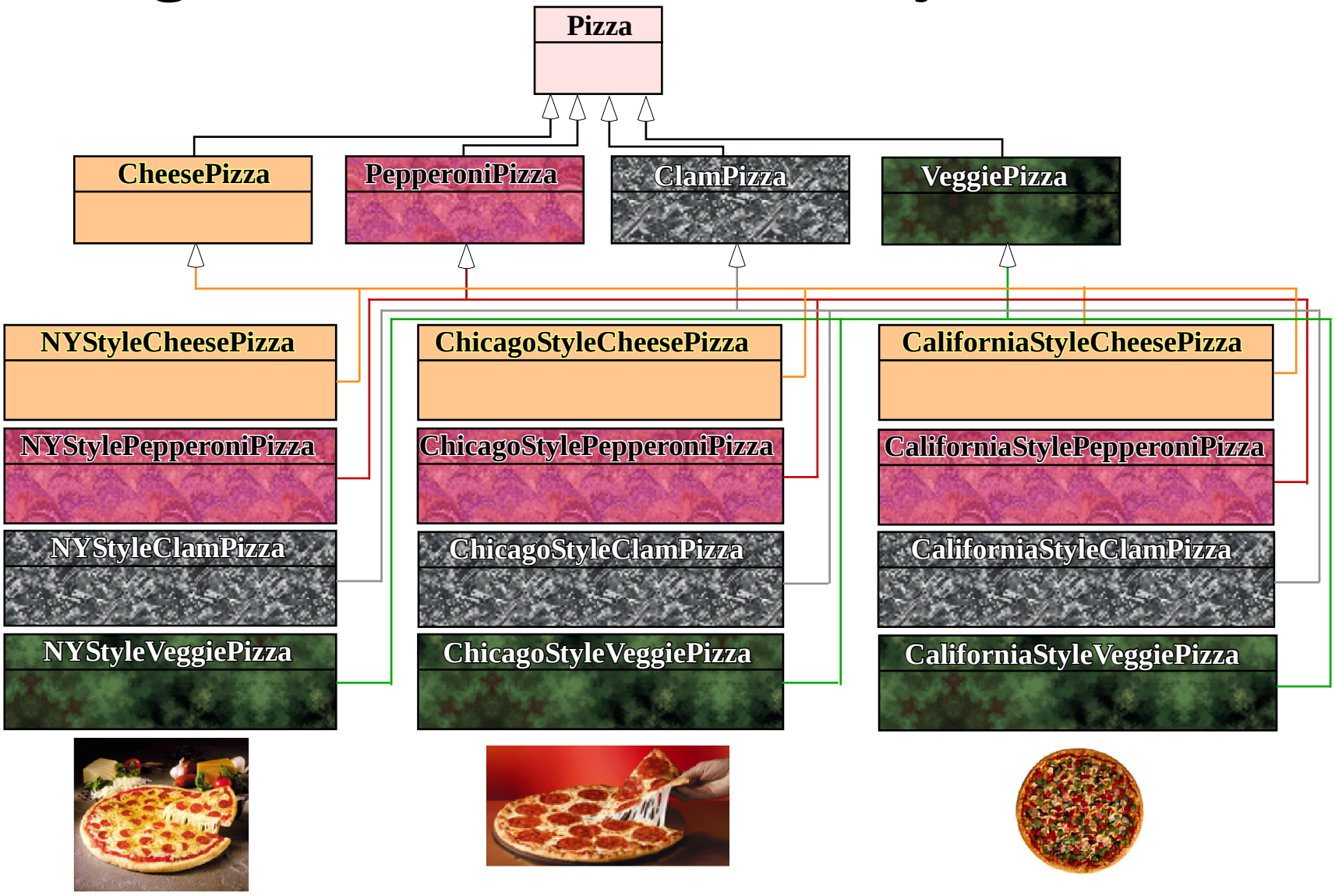



```
Pizza *NYPizzaShop::createPizza(string type)
{
    Pizza *pizza = NULL;

    if(type == “cheese”)
      pizza = new NYStyleCheesePizza();
    else if(type == “pepperoni”)
      pizza = new NYStylePepperoniPizza();
    else if (type == “clam”)
       pizza = new NYStyleClamPizza();
    else if (type == “veggie”)
       pizza = new NYStyleVeggiePizza();

    return pizza;
}
```

Each sub-class now must implement this method to create its’ particular regional style of pizza.

And we can take this another step...

Design Patterns: The Factory Method



Each PizzaShop sub-class encapsulates the knowledge to create its' particular regional style of pizza.

Design Patterns: The Factory Method

The VeryDependentPizzaShop

```
Pizza *VeryDependentPizzaShop::createPizza(string style, string type)
{
   Pizza *pizza;

   if(style == “NY”)
   {
      if(type == “cheese”)
        pizza = new NYCheesePizza();
      else if(type == “veggie”)
        pizza = new NYVeggiePizza();
      else if(type == “pepperoni”;
        pizza = new NYPepperoniPizza()
      else if(type == “clam”;
        pizza = new NYClamPizza()
   }
   else if(style == “Chicago”)
   {
      if(type == “cheese”)
        pizza = new ChicagoCheesePizza();
      else if(type == “veggie”)
        pizza = new ChicagoVeggiePizza();
      else if(type == “pepperoni”;
        pizza = new ChicagoPepperoniPizza()
      else if(type == “clam”;
        pizza = new ChicagoClamPizza()
   }
   // etc. for all the other “Styles”
   else
   {
      cout << “Error: Invalid type of pizza.”;
      return NULL;
   }
   if(pizza != NULL)
   {
      pizza->prepare();
      pizza->bake();
      pizza->cut();
      pizza->box();
   }
return pizza;
}
```

Unfortunately, this is an all too common example of how code can just keep growing and growing and growing as changes occur.

Design Principles

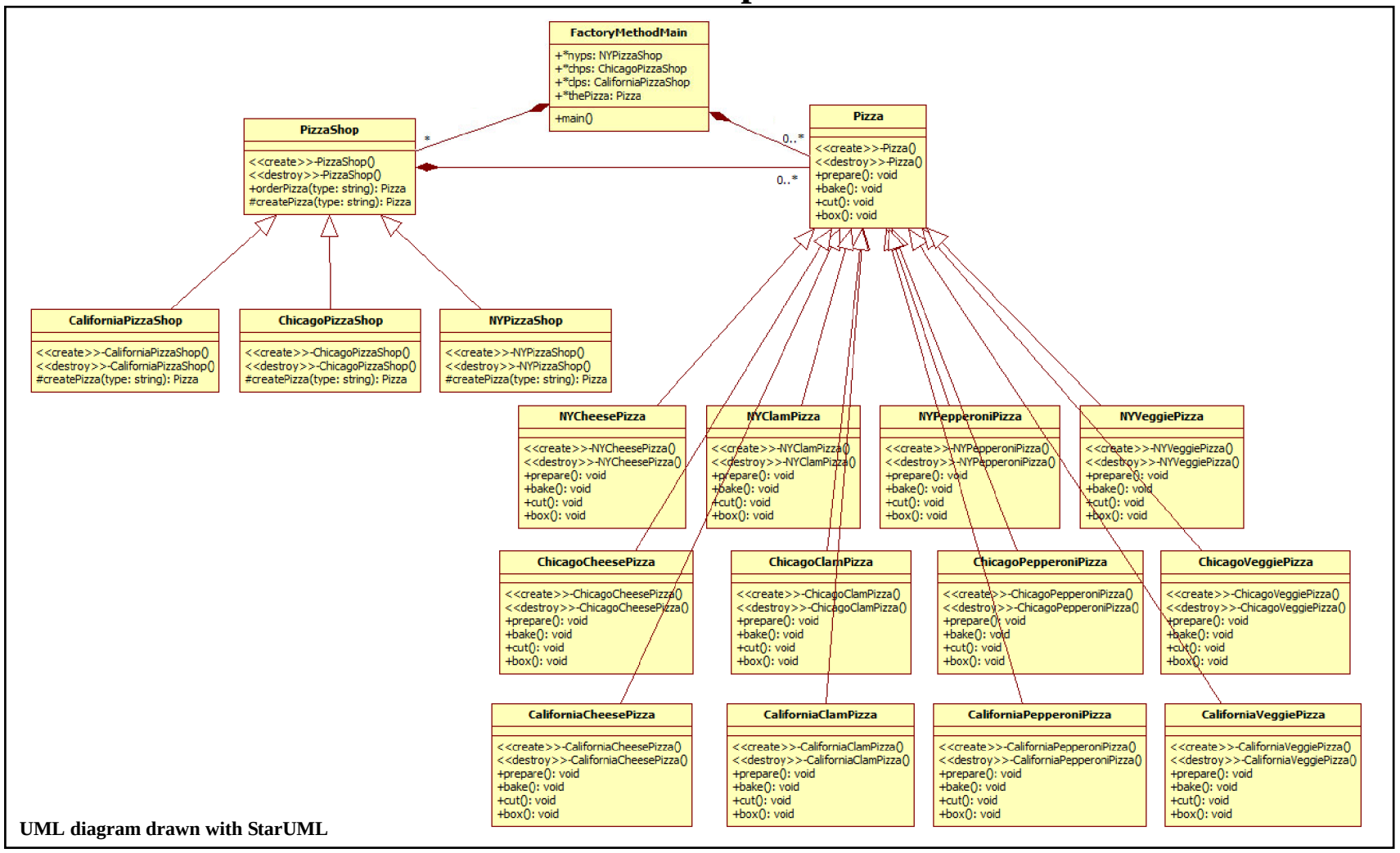
- ***Depend on abstractions. Do not depend on concrete classes.***

We can access any of the sub-classes of PizzaShop using the abstract interface.

We can access any type of pizza from any of the pizza shops through the abstract interfaces created for them.

Design Patterns: The Factory Method

Code Sample



UML diagram drawn with StarUML

FactoryMethodMain

Instantiates sub-classes of PizzaShop using pointers to Pizzashop
Calls createPizza in each PizzaShop ordering a type of pizza
Using its Factory Method each pizza shop creates its specialty pizza.

Let's look at the code and run the demonstration.