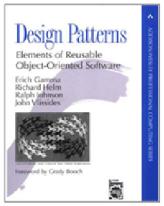


Design Pattern Definitions from the GoF Book

The Abstract Factory Pattern

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Creational Patterns

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

Structural Patterns

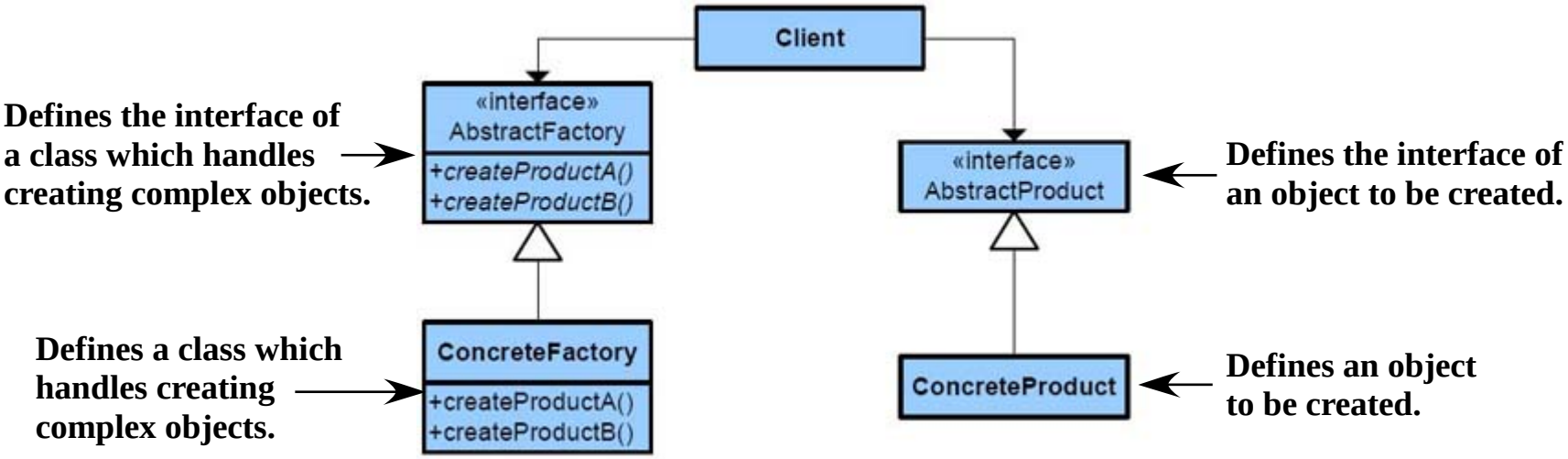
- **The Decorator Pattern**
  Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **The Adapter Pattern**
- **The Facade Pattern**
- **The Composite Pattern**
- **The Proxy Pattern**
- **The Bridge Pattern**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

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.



Design Patterns: The Abstract Factory

We have the same product families for all of our pizzas, but different implementations based on the region.

Product Family: Dough

- ThinCrustDough
- VeryThinCrustDough
- ThickCrustDough

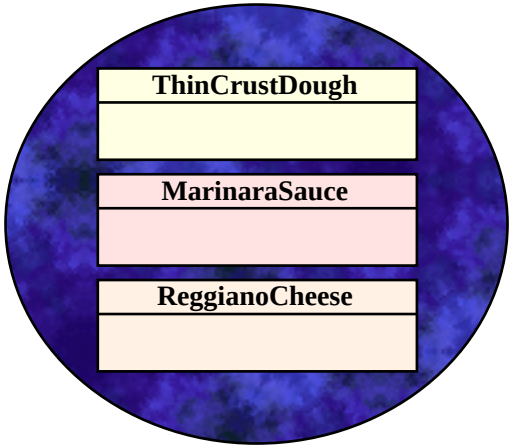
Product Family: Sauce

- PlumTomatoSauce
- MarinaraSauce
- BruschettaSauce

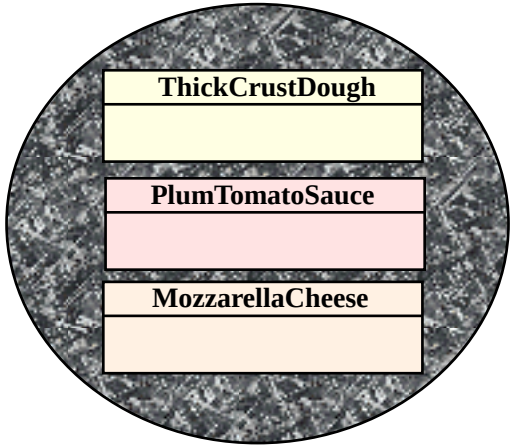
Product Family: Cheese

- MozzarellaCheese
- ReggianoCheese
- GoatCheese

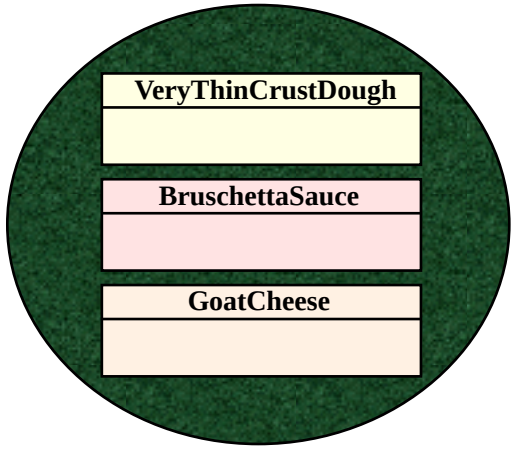
And others like veggies and meats



New York



Chicago



California

Design Patterns: The Abstract Factory

PizzaIngredientFactory

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

This becomes our abstract interface for building the various regional factories...

...like this NYPizzaIngredientFactory

```
Dough NYPizzaIngredientFactory::createDough()
{
    return new ThinCrustDough();
}

Sauce NYPizzaIngredientFactory::createSauce()
{
    return new MarinaraSauce();
}

Cheese NYPizzaIngredientFactory::createCheese()
{
    return new ReggianoCheese();
}
```

```
Pepperoni NYPizzaIngredientFactory::createPepperoni()
{
    return new SlicedPepperoni();
}

Veggies NYPizzaIngredientFactory::createVeggiesi()
{
    Veggies veggies[] = {new Garlic(), new Onion(),
          new Mushroom(), new RedPepper()};
     return veggies;
}

Clams NYPizzaIngredientFactory::createClams()
{
    return new FreshClams();
}
```

When they are created each PizzaShop sub-class instantiates the appropriate PizzaIngredientFactory

Design Patterns: The Abstract Factory

Now let's rework the Pizza class.

Pizza

```
class Pizza
{
    public:
        Dough dough;
        Sauce sauce;
        Cheese cheese;
        Veggies veggies[];
        Pepperoni pepperoni;
        Clams clams;

      virtual void prepare();
      void bake();
      void cut();
      void box();
      void setName(string name);
      string getName();
}
```

And each of its sub-classes like this CheesePizza.

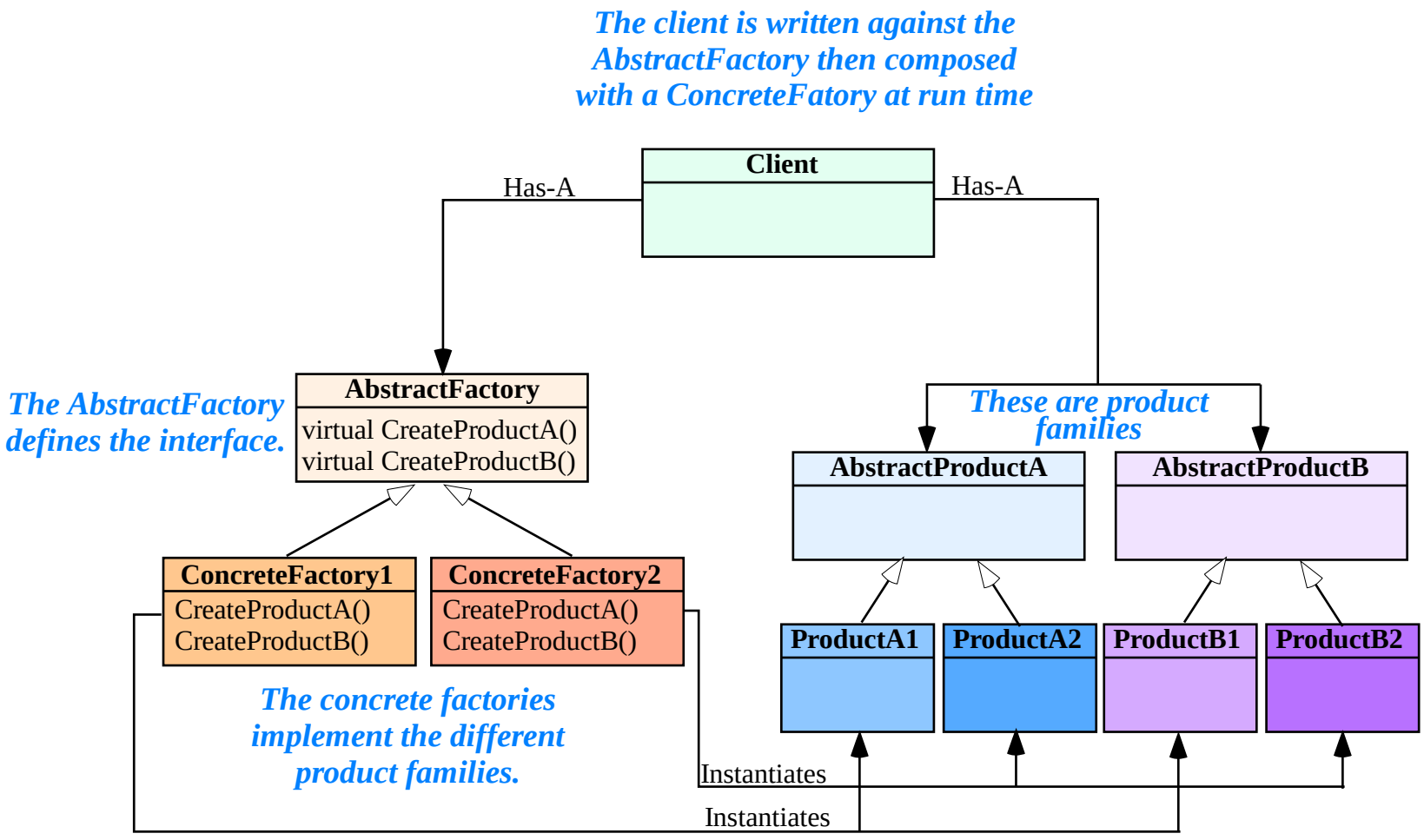
CheesePizza

```
class CheesePizza::Pizza
{
   private:
      PizzaIngredientFactory ingredientFactory;

    public:
       CheesePizza(PizzaIngredientFactory
                                  ingredientFactory)
       {
          this->ingredientFactory = ingredientFactory;
       }

       void prepare()
       {
          cout << "Preparing " << name;
          dough=ingredientFactory.createDough();
          sauce=ingredientFactory.createSauce();
          cheese=ingredientFactory.createCheese();
        }
}
```

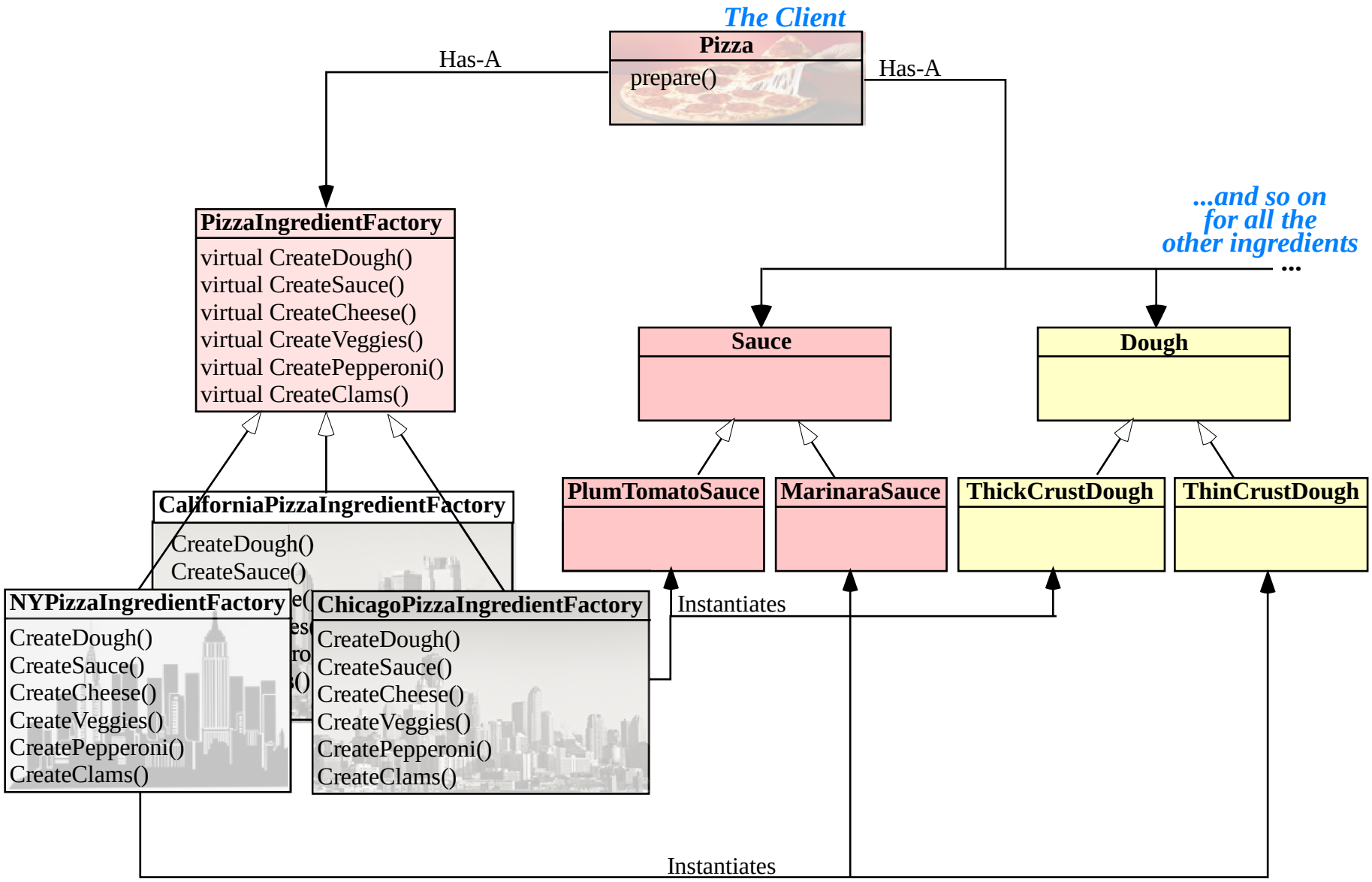
In the constructor we pass in the appropriate PizzaIngredientFactory for the region.

The Pizza doesn't care which PizzaIngredientFactory it uses.

Design Patterns: The Abstract Factory

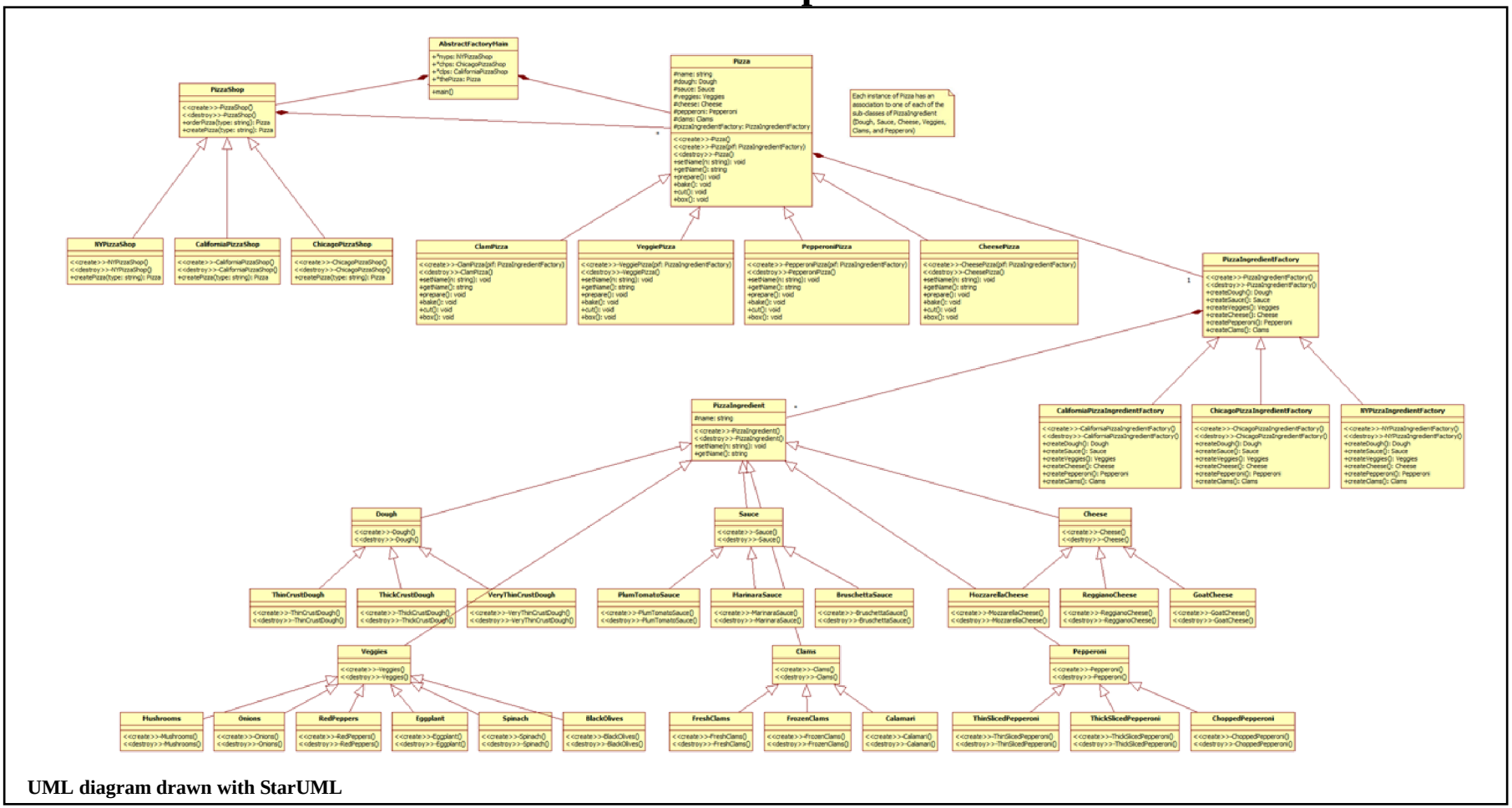


Design Patterns: The Abstract Factory



Design Patterns: Abstract Factory

Code Sample



UML diagram drawn with StarUML

AbstractFactorMain

"Customer" enters a PizzaShop (NY, Chi., or Cal.)

"Customer" orders a type of pizza (Cheese, Clam, Pepperoni, Veggie)

Calls PizzaShop->OrderPizza(type) which calls CreatePizza(type)

Creates the Pizza type passing it the appropriate ingredient factory

Pizza uses the ingredient factory to create its regional style

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