CS 619 Introduction to OO Design and Development

GoF Patterns (Part 3)

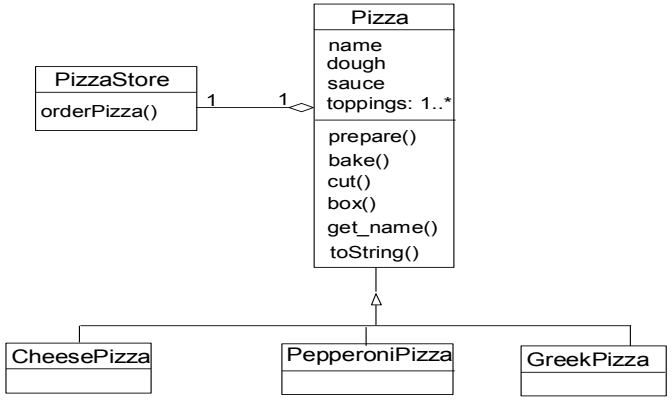
Fall 2014

Simple Factory Pattern: Example

Suppose that you are required to develop a system that accepts orders for pizzas. There are three types of pizzas, namely, cheese, Greek, and pepperoni. The pizzas differ according to the dough used, the sauce used and the toppings.

Draw a class diagram for the system.

Class Diagram



Problems with the design?


```
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;
}
```

How to Deal with Changes

- Programming to implementation like makes such changes difficult.
- Creating a **SimpleFactory** to encapsulate the code that changes will make the design more flexible.
- Remove the code that creates a pizza – forms a simple factory.


```
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;
}
```

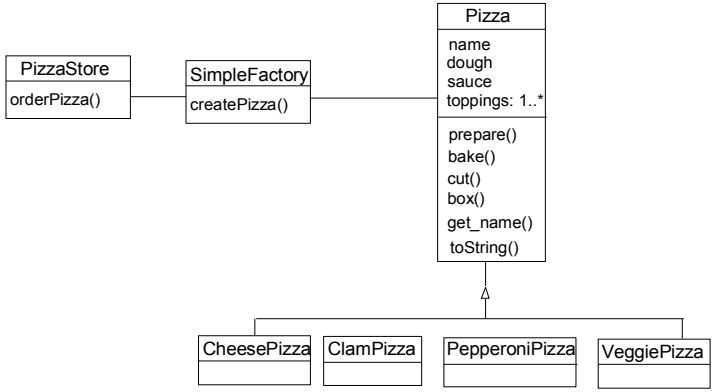
Encapsulate!


```
public class 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;
   }
}
```

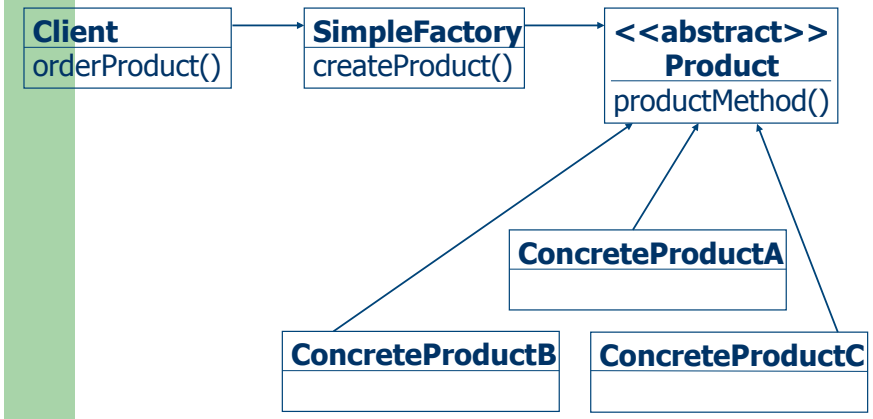
The Simple Factory Pattern

```
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;
  }
}
```


Example: Revised Class Diagram



Simple Factory Pattern (not GoF)

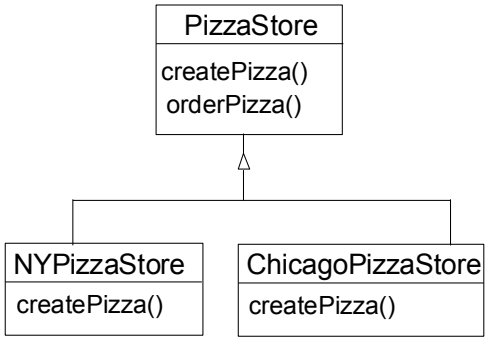


More changes

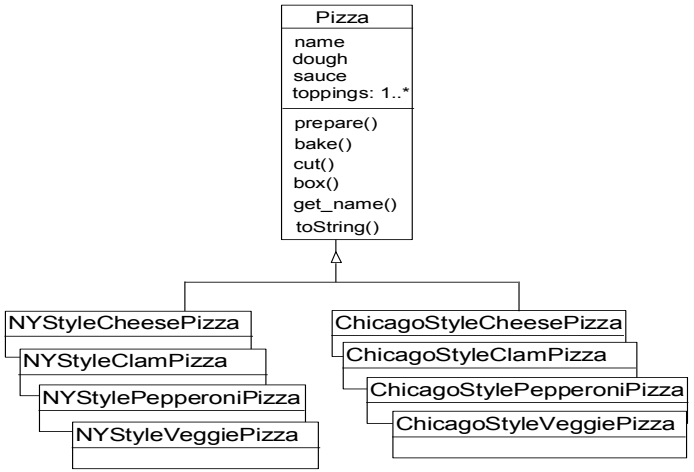
Franchises in different parts of the country are now adding their own special touches to the pizza. For example, customers at the franchise in New York like a thin base, with tasty sauce and little cheese. However, customers in Chicago prefer a thick base, rich sauce and a lot of cheese.

You need to extend the system to cater for this.

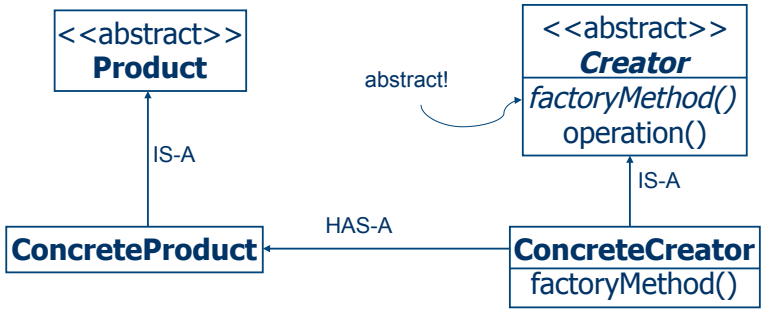
Creator Class



Product Class



Factory Method Pattern



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

Abstract Creator: the PizzaStore

```
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;
    }
}
```

Add a new store (the concrete creator)

```
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;
  }
}
```


The Factory Method Pattern

```
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");

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


The Factory Method Pattern

- 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.**
- Use Factory Method when:
  - A class can't anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates
  - Classes delegate to one of several helper subclasses, and you want to localize knowledge about which is the delegate

More Ingredience

- We want to list the exact list of ingredients for each concrete pizza. For example :
  - Chicago Cheese Pizza : Plum tomato Sauce, Mozzarella, Parmesan, Oregano;
  - New York Cheese Pizza : Marinara Sauce, Reggiano, Garlic.
- Each "style" uses a different set of ingredients,
- We could change implement Pizza as in:

```
public class Pizza {
  Dough dough;
  Sauce sauce;
  Veggies veggies[];
  Cheese cheese;
  Pepperoni pepperoni;
  Clams clam;
    . . .
}
```

- and the constructor naturally becomes something like :

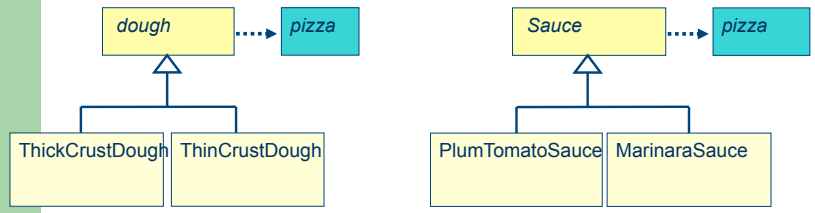
```
public Pizza(Dough d, Sauce s, Cheese c, Veggies
   v, Pepperoni p, Clams c)
{
  dough = d;
  sauce = s;
  veggies = v;
  cheese = c;
  pepperoni  = p;
  clam = c;
}
```


- but then:

```
public class NYPizzaStore extends PizzaStore {
  Pizza createPizza(String item) {
    if (item.equals("cheese")) {
      return new NYStyleCheesePizza(new ThinCrustDough(),
             new Marinara(), new Reggiano(), null, null );
    } else if (item.equals("veggie")) {
      return new NYStyleVeggiaPizza(new ThinCrustDough(),
            new Marinara(), new Reggiano(), mew Garlic() ,
           null);
    }
  }
```

This will cause a lot of maintenance headaches!
Imagine what happens when we create a new pizza!

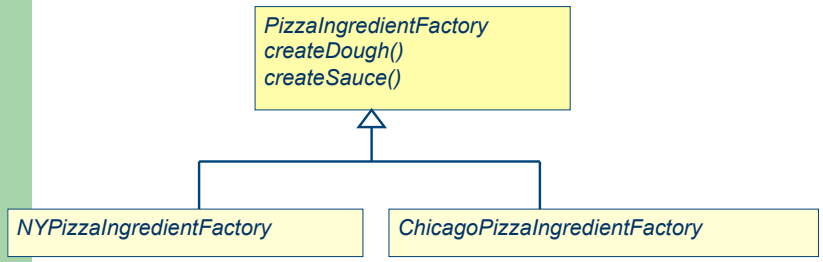
- How about this sulution?



This looks fine…but does it reflect our intention?
Would it really make sense to have a pizza, with Chicago Thick Crust + NY Marinara Sauce?
Model should not include any ”binding” between related products.

The Abstract Factory

- A ***Dough*** and a ***Sauce*** are not – as seen from a type point-of-view – related
- Would be somewhat artificial – or perhaps even impossible – to introduce a common base class
- However, we can enforce the binding through a shared factory class!



- We know that we have a certain set of ingredients that are used for New York..yet we have to keep repeating that set with each constructor. Can we define this unique set just once?
- After all we are creating concrete instances of dough, ingredients etc. :
  - let's use the factory of ingredients!

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


- We then "program to the interface" by implementing different concrete ingredients factories. For example here are the ingredients used in New York pizza style:

```
public class NYPizzaIngredientFactory : PizzaIngredientFactory {
   public Dough createDough() {
        return new ThinCrustDough();
   }
   public Sauce createSauce() {
        return new MarinaraSauce();
   }
   public Cheese createCheese() {
        return new ReggianoCheese();
   }
   //...
}
```


- Our Pizza class will remain an abstract class:

```
public abstract class Pizza {
   string name;
   Dough dough;
   Sauce sauce;
   Cheese cheese;
   Veggies veggies[];

  abstract void Prepare(); //now abstract
  public virtual string Bake() {
    Console.WriteLine("Bake for 25 minutes at 350 \n");
  }
  public virtual string Cut() {
    Console.WriteLine("Cutting the pizza into diagonal slices");
  }
 // ...
 }
```


- and our concrete Pizza (the client of PizzaIngredientFactory) simply become:

```
public class CheesePizza : Pizza {
   PizzaIngredientFactory ingredientFactory;
   public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
   }
   void prepare() {
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
   }
}
```

No need to instanciate an ingredient; the creation of the ingredients is delegated to a factory.

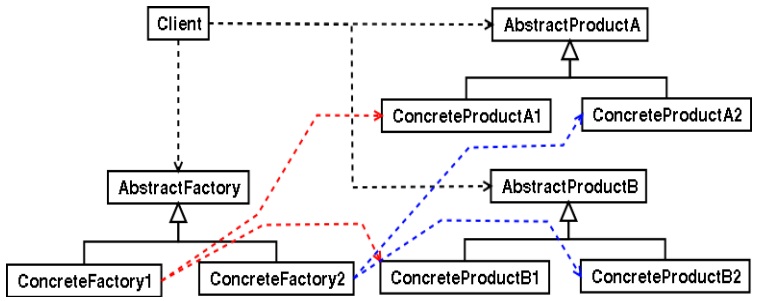
```
public class ClamPizza : Pizza {
   PizzaIngredientFactory ingredientFactory;
   public ClamPizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
   }
   void prepare() {
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
   }
}
```

No need to instanciate an ingredient; the creation of the ingredients is delegated to a factory.

- finally we must have our concrete Pizza Store:

```
public class NYPizzaStore : PizzaStore {
    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
                new NYPizzaIngredientFactory();
        if (item.equals("cheese")) {
                pizza = new CheesePizza(ingredientFactory);
        } else if (item.equals("veggie")) {
                pizza = new VeggiePizza(ingredientFactory);
        } else if (item.equals("clam")) {
                pizza = new ClamPizza(ingredientFactory);
        } else if (item.equals("pepperoni")) {
                pizza = new PepperoniPizza(ingredientFactory);
        }
        return pizza;
    }
}
```


Abstract Factory: Class Diagram



Abstract Factory: Participants

- *AbstractFactory*
  - declares interface for operations that create abstract products
- *ConcreteFactory*
  - implements operations to create concrete products
- *AbstractProduct*
  - declares an interface for a type of product object
- *ConcreteProduct*
  - defines the product object created by concrete factory
  - implements the AbstractProduct interface
- *Client*
  - uses **only** interfaces of ***AbstractFactory /AbstractProduct***

Abstract Factory – intent and context

- provides an interface for creating families of related or dependent objects without specifying their concrete classes
- use *AbstractFactory* when
  - a system should be independent of how its products are created, composed, represented
  - a system should be configured with one or 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

Abstract Factory Summary

- The creator gives you an interface for writing objects.
- The creator specifies the “factory method”, e.g. the createPizza()
- Other methods generally included in the creator are methods that operate on the product produced by the creator, e.g. orderPizza().
- Only subclasses implement the factory method.