



Factory Method Design Pattern

Topics

- 1) How can we prevent instantiation coupling when we **must instantiate new objects**?

Let's make Pizzas!

- We are opening a Pizza restaurant chain!
 - Pizza types (Cheese, Veggie, Hawaiian, ...)
 - Pizza styles (New York = thin; Chicago = deep dish)
- And, you know:
the requirements are going to change!



We have `new` Problems

- The problem with new:
 - new creates an object of a concrete type
 - new couples our code to a specific concrete class
- We want to depend on general types (“interfaces”), not concrete types.
- Solutions
 - If we need an object so we can do our job, use..
 - If our job is **creating new objects**, we can't use DI: we can..

`new` Problem: Code

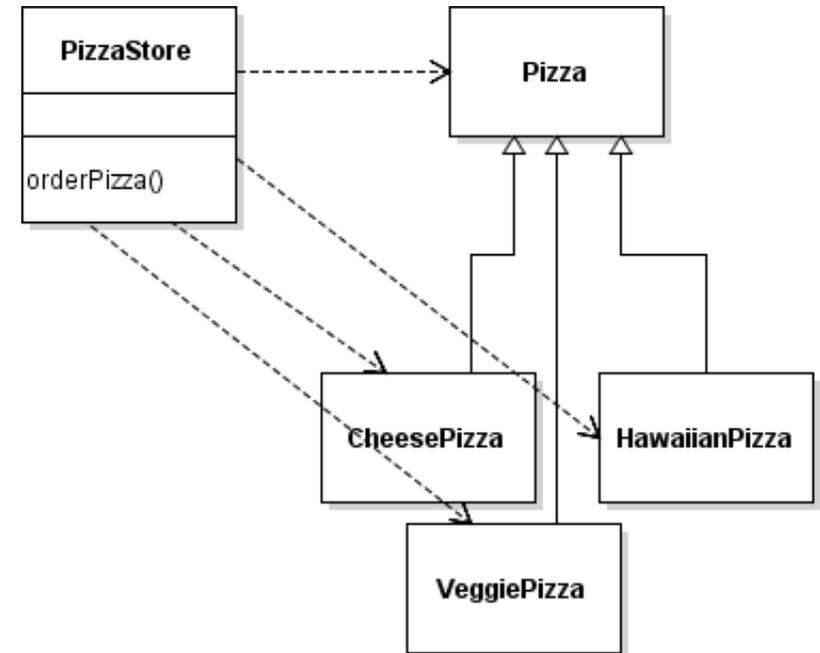
```
Pizza orderPizza(String type) {  
    Pizza pizza = null;  
    if (type == "Cheese") {  
        pizza = new CheesePizza();  
    } else if (type == "Hawaiian") {  
        pizza = new HawaiianPizza();  
    } else if (type == "Veggie") {  
        pizza = new VeggiePizza();  
    }  
    pizza.prepare();  
    pizza.bake();  
    pizza.box();  
    return pizza;  
}
```

- What changes when adding a new pizza type?
 - Which design principle does this violate?
- What type of coupling?
 - Couples high-level (pizza order) to low level classes (Cheese,)

Coupling

- ..

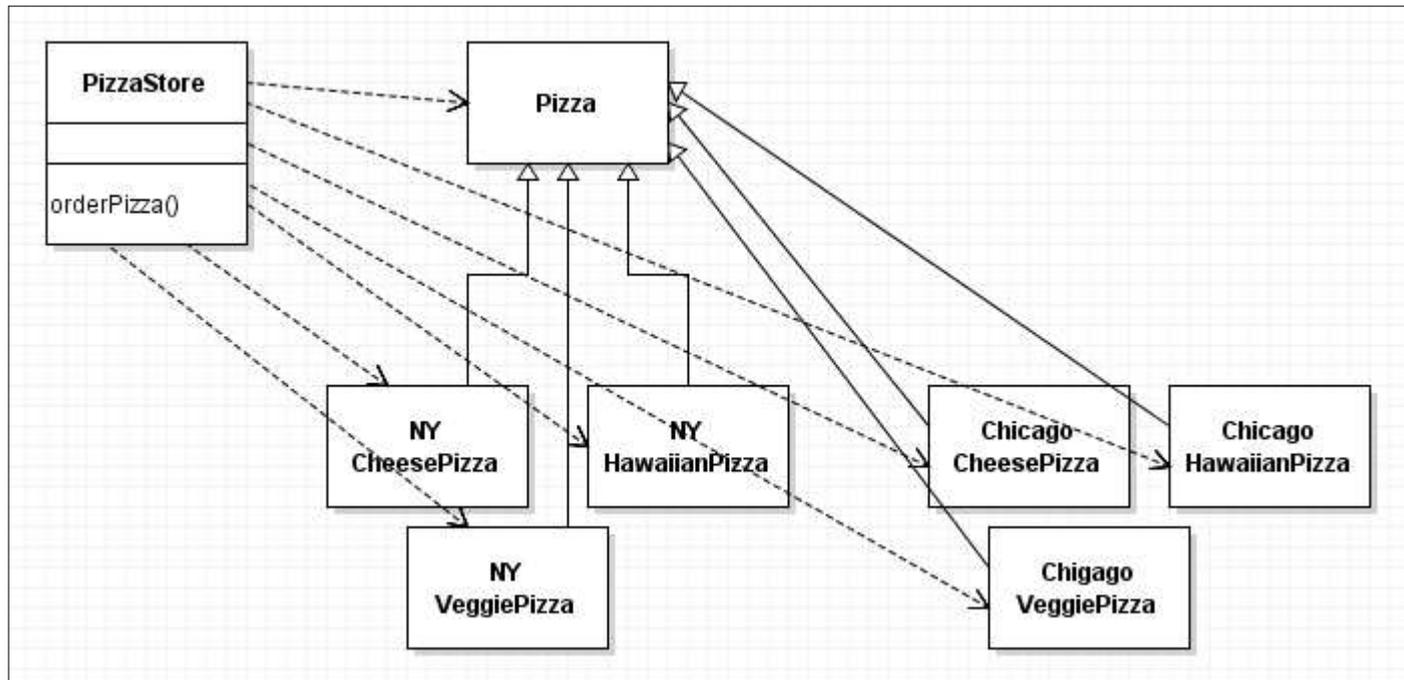
```
Pizza orderPizza(String type) {  
    Pizza pizza = null;  
    if (type.equals("Cheese")) {  
        pizza = new CheesePizza();  
    } else if (type.equals("Hawaiian")) {  
        pizza = new HawaiianPizza();  
    } else if (type.equals("Veggie")) {  
        pizza = new VeggiePizza();  
    }  
    pizza.prepare();  
    pizza.bake();  
    pizza.box();  
    return pizza;  
}
```



Factory Method

Creating families of objects

- What if we want to support creating NY or Chicago pizzas?
 - Ex: Want a NY Cheese, and a Chicago Cheese



Creating families of objects (code)

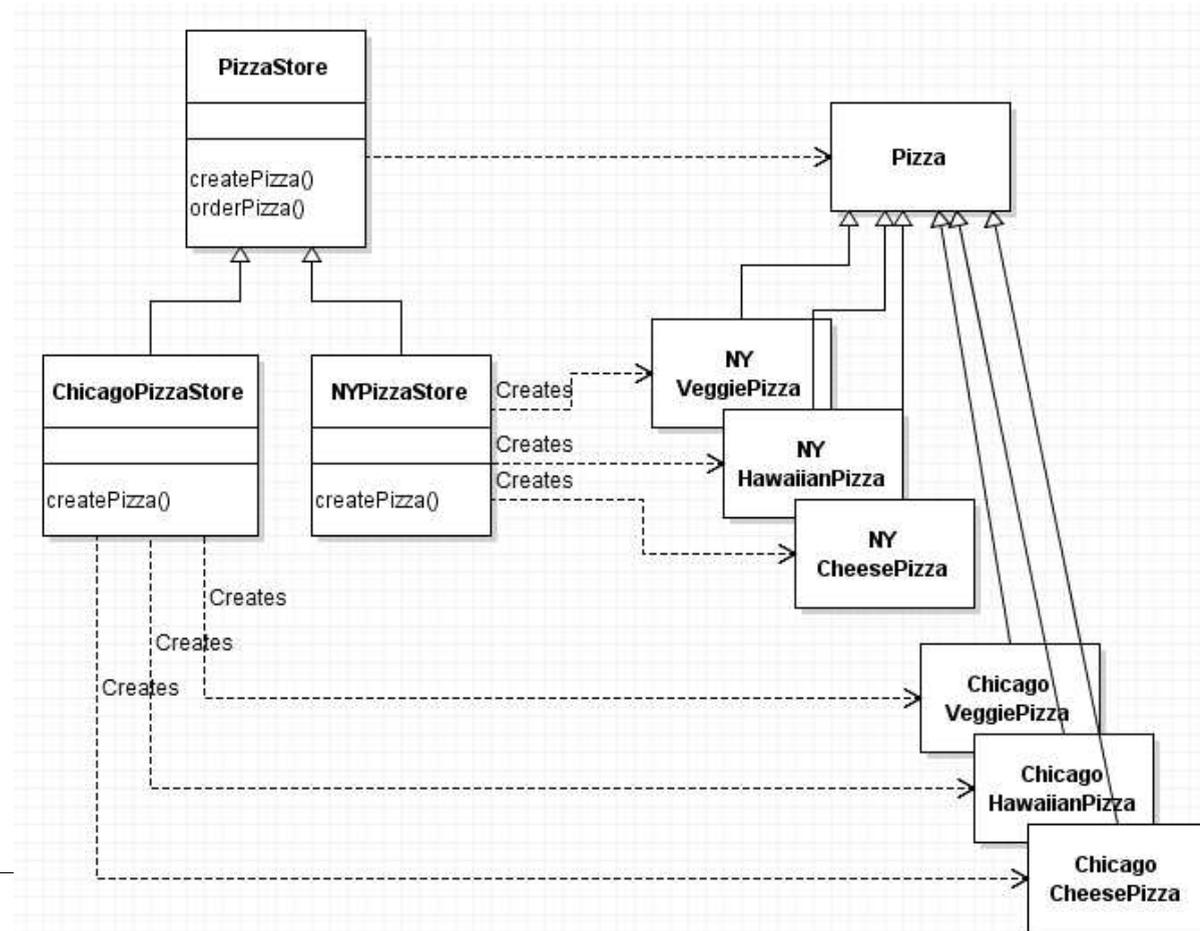
- It's ugly having PizzaStore instantiating all know styles and types of pizza.

```
Pizza orderPizza(String type, String style) {
    Pizza pizza = null;
    if (style == "NY") {
        if (type == "Cheese") {
            pizza = new NYCheesePizza();
        } else if (type == "Hawaiian") {
            pizza = new NYHawaiianPizza();
        } else if (type == "Veggie") {
            pizza = new NYVeggiePizza();
        }
    } else if (style == "Chicago") {
        if (type == "Cheese") {
            pizza = new ChicagoCheesePizza();
        } else if (type == "Hawaiian") {
            pizza = new ChicagoHawaiianPizza();
        } else if (type == "Veggie") {
            pizza = new ChicagoVeggiePizza();
        }
    }
    pizza.prepare();
    pizza.bake();
    pizza.box();
    return pizza;
}
```

- What can we do to clean this up?

Defer instantiation to derived class

- Encapsulate what Varies:
..
(derived classes).
- Base class
 - Does work with Pizza
 - Abstract method to create pizzas: createPizza()
- Derived class
 - Overrides createPizza() to instantiate the correct style of pizza



Factory Method Code

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

Abstract method
("Factory Method")
in base class

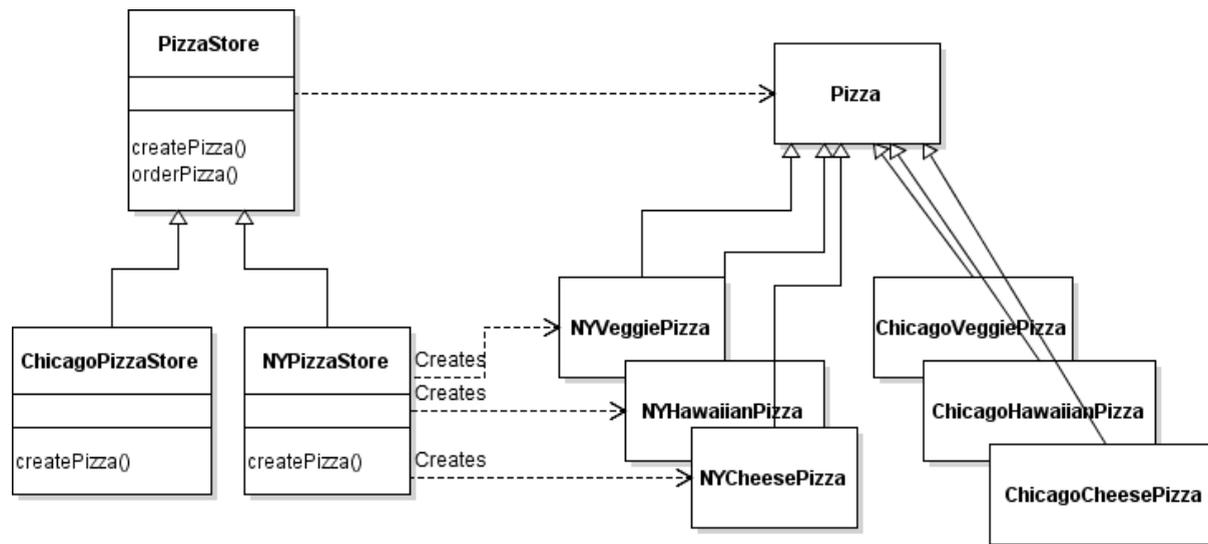
PizzaStore is a
framework for working
with any Pizza

Override factory method
in derived class

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

Factory Method Idea

- ..
 - derived classes instantiate different (families of) objects.
 - Base class
 - defines an abstract factory method for creating objects
 - Derived classes
 - overrides factory method to instantiate concrete types

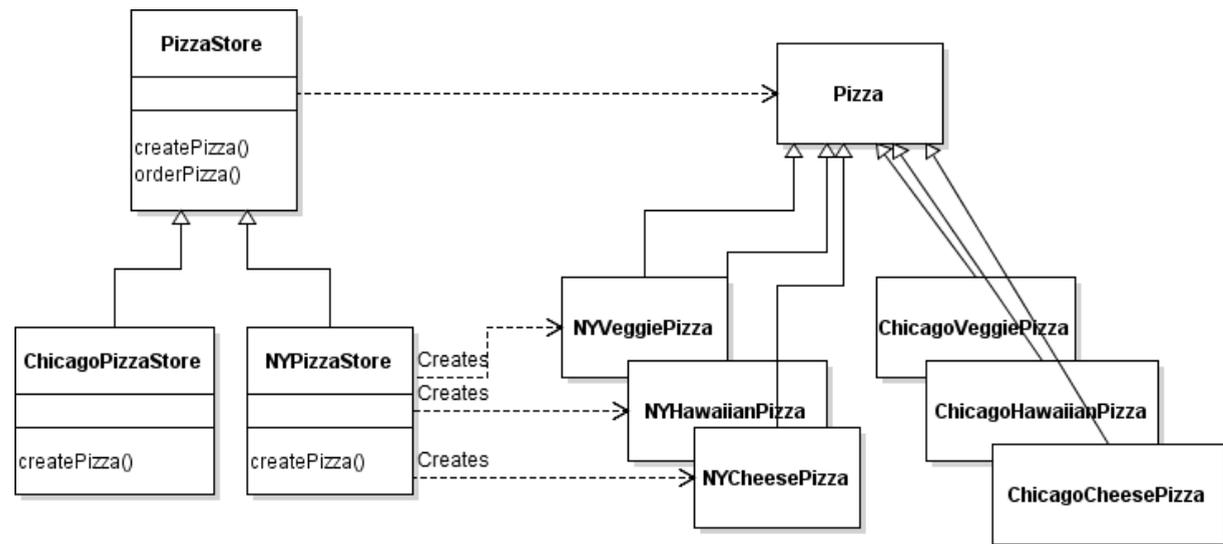


Benefits of Factory Method

- Satisfies..
and..
 - Adding a new pizza style adds we classes
- High-level class (PizzaStore) depends on an abstract type (Pizza), not a concrete implementation (NYVeggiePizza)

This is actually a
“Parameterized factory
method”:
The object is created
based on an argument.

Can apply this pattern
without arguments to the
factory method.



Exercise: Writing Client Code

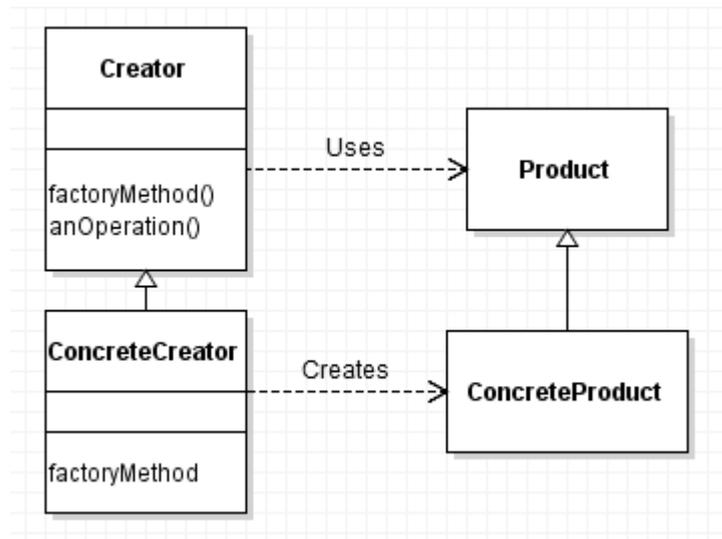
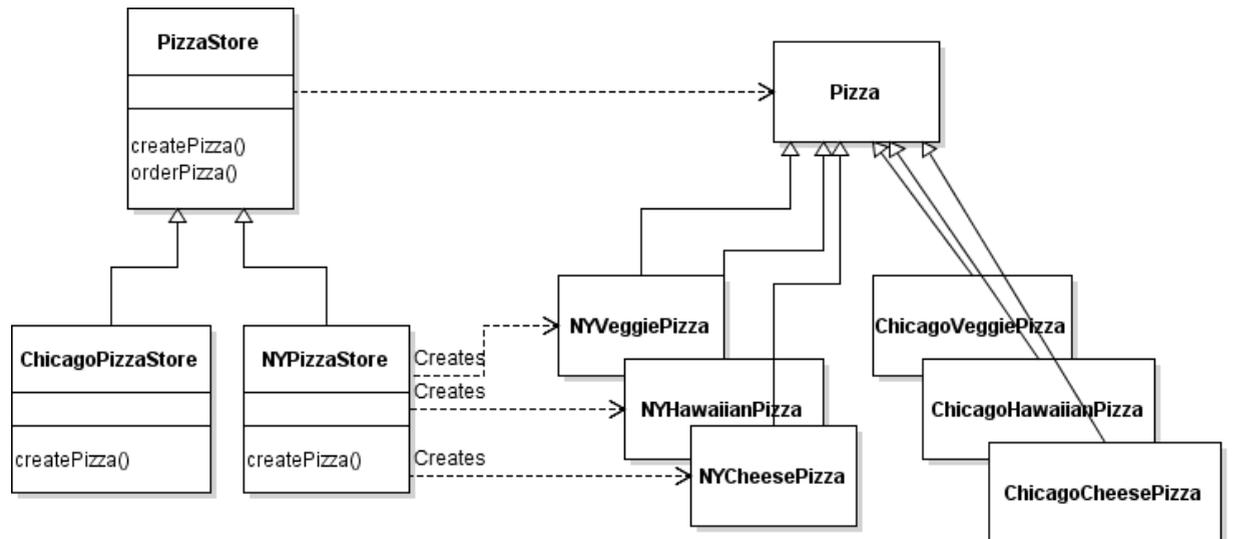
- Write client code which
 1. creates a `NYPizzaStore` and
 2. orders a Cheese pizza.
 - Trace with UML

Factory Method Design Pattern

- Factor Method Design Pattern:
 - Define an interface for creating an object (abstract function), but let subclasses decide which class to instantiate.
 - Factory Method design pattern..

- .. : all the PizzaStores

- .. : all the Pizza's



Drawbacks

- Parallel Hierarchies

- ..

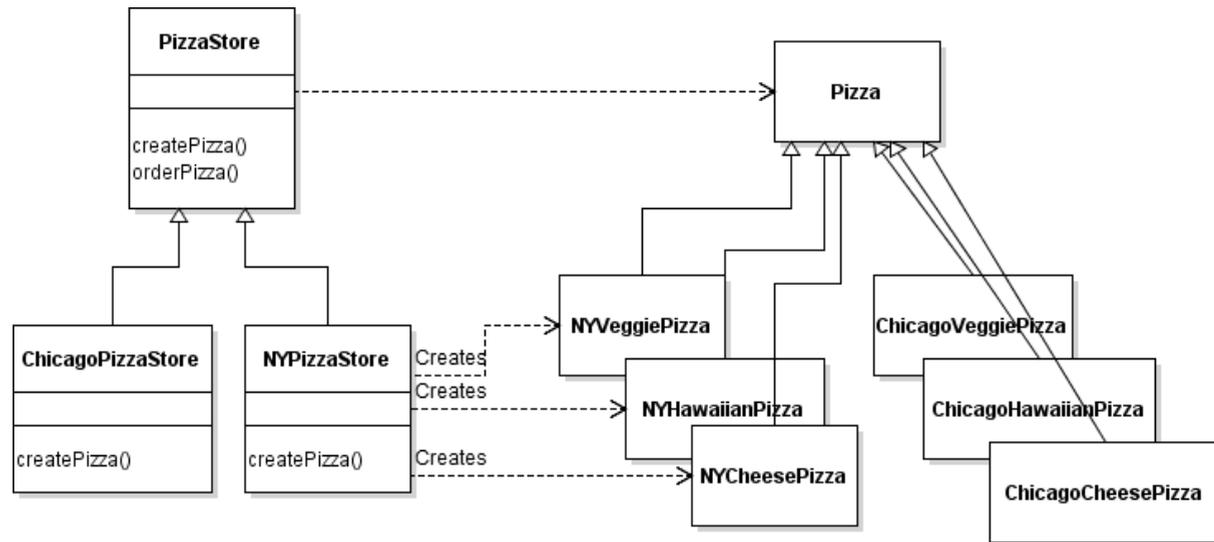
- Add a VancouverPizzaStore?

Adding a new class to creator hierarchy requires adding new classes to products

- Add a GreekPizza?

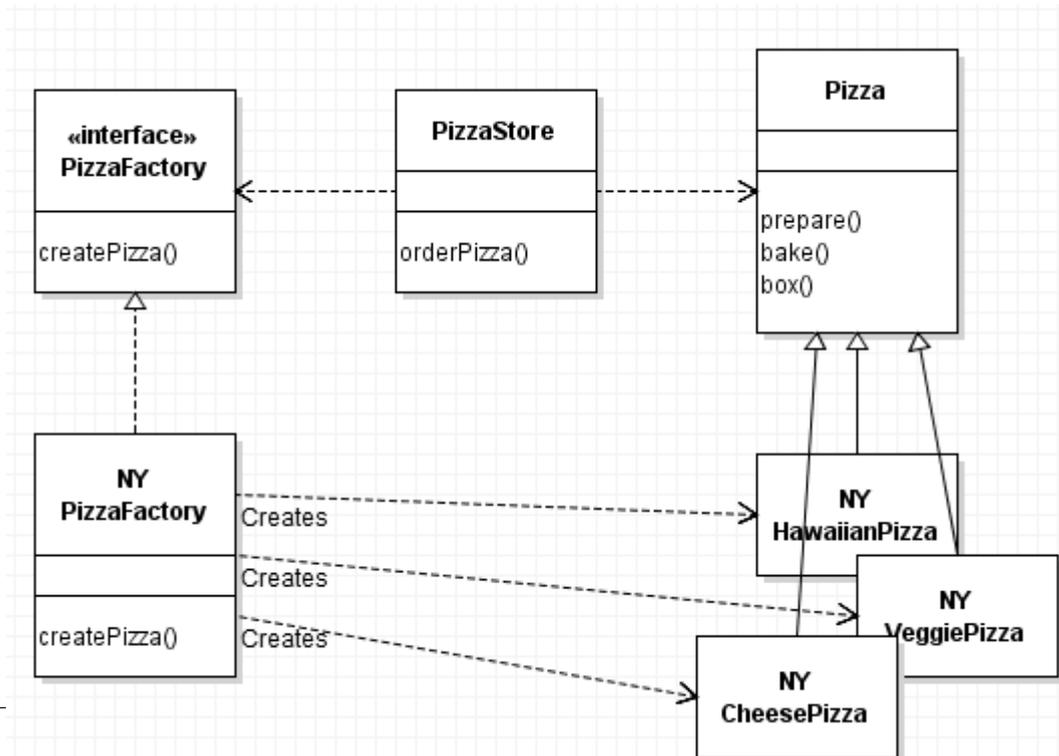
Adding a product class requires:

- changing all creators
 - creating matching products for each creator



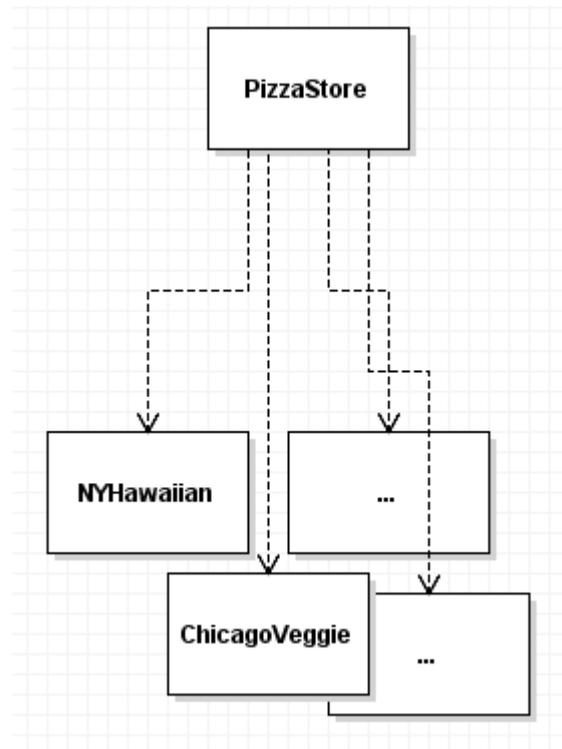
Drawbacks

- Inheritance is fixed at runtime:
 - Cannot..
- This is addressed by using the..
 - Define a separate object for instantiation (factory object)
 - PizzaStore has-a PizzaFactory (NYPizzaFactory,...)
 - Design Principle: Favour composition over inheritance

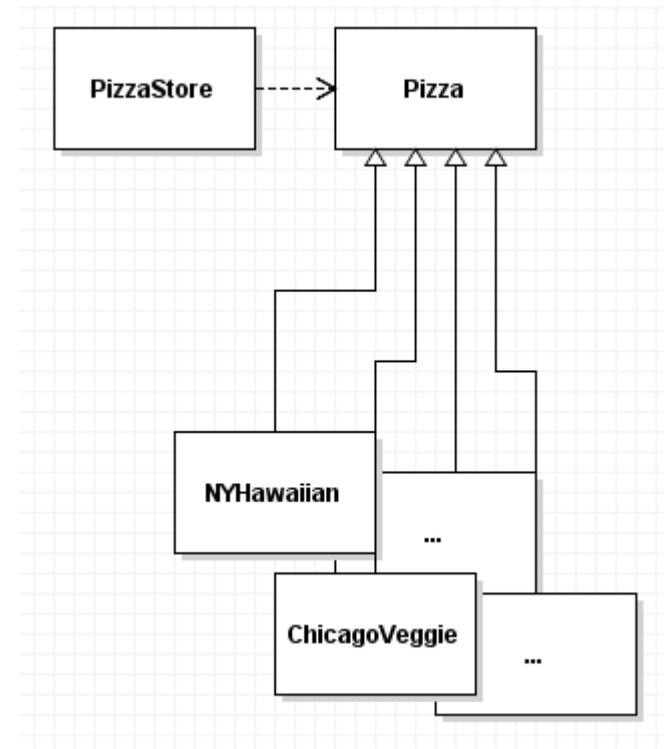


Dependency Inversion

- *Without* Factory Method
PizzaStore depends on..



- *With* Factory Method
PizzaStore and all concrete pizzas depends on..



Design Principle: Dependency Inversion

- Design Principle: Dependency Inversion
 - ..
Do not depend upon concrete classes.
- Similar to “Code to an interface, not an implementation” but this is stronger:
 - DIP: Have both high and low level classes
 - ..
 - “Code to interface” motivated by flexibility: ability to change object type later.
 - DIP motivated by cleaning up the dependencies from high to low and coupling
- We invert the dependency lines in the UML for PizzaStore

Summary

- Creating an object with new couples code to a concrete class.
- High-level code should not depend on concrete types: therefore it should not instantiate with new!
 - Dependency Injection:
For when we can be handed the objects we need.
 - Factory Method:
Delegate instantiation of concrete objects to a derived class (inheritance).
 - Abstract Factory Pattern:
Delegate instantiation of concrete objects to a factory object (composition).