A very dependent PizzaStore

Sharpen your pencil

Let's pretend you've never heard of an OO factory. Here's a version of the PizzaStore that doesn't use a factory; make a count of the number of concrete pizza objects this class is dependent on. If you added California style pizzas to this PizzaStore, how many objects would it be dependent on then?

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

Handles all the NY style pizzas

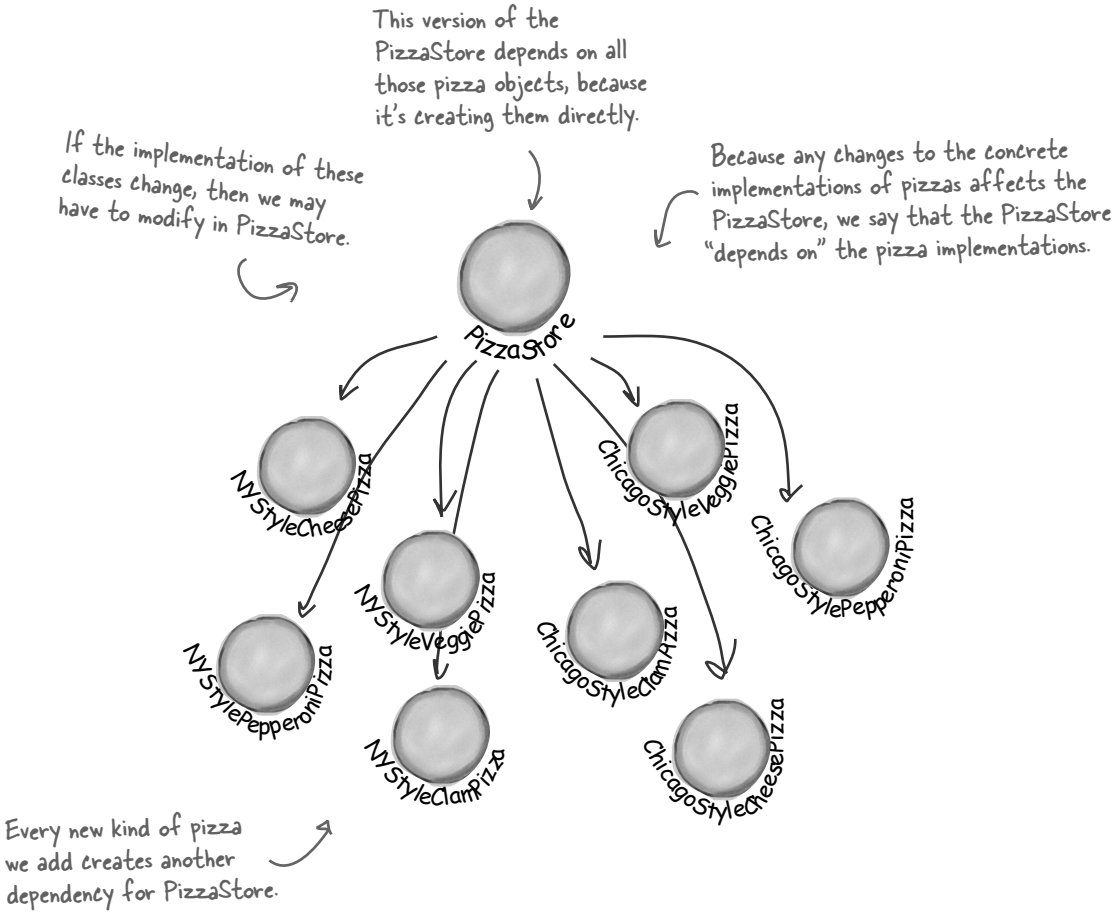
Handles all the Chicago style pizzas

You can write your answers here: _____ number _____ number with California too

Looking at object dependencies

When you directly instantiate an object, you are depending on its concrete class. Take a look at our very dependent PizzaStore one page back. It creates all the pizza objects right in the PizzaStore class instead of delegating to a factory.

If we draw a diagram representing that version of the PizzaStore and all the objects it depends on, here's what it looks like:



The Dependency Inversion Principle

It should be pretty clear that reducing dependencies to concrete classes in our code is a "good thing." In fact, we've got an OO design principle that formalizes this notion; it even has a big, formal name: *Dependency Inversion Principle*.

Yet another phrase you can use to impress the execs in the room! Your raise will more than offset the cost of this book, and you'll gain the admiration of your fellow developers.

Here's the general principle:

Design Principle

Depend upon abstractions. Do not depend upon concrete classes.

At first, this principle sounds a lot like "Program to an interface, not an implementation," right? It is similar; however, the Dependency Inversion Principle makes an even stronger statement about abstraction. It suggests that our high-level components should not depend on our low-level components; rather, they should *both* depend on abstractions.

A "high-level" component is a class with behavior defined in terms of other, "low level" components.

For example, PizzaStore is a high-level component because its behavior is defined in terms of pizzas – it creates all the different pizza objects, prepares, bakes, cuts, and boxes them, while the pizzas it uses are low-level components.

But what the heck does that mean?

Well, let's start by looking again at the pizza store diagram on the previous page. PizzaStore is our "high-level component" and the pizza implementations are our "low-level components," and clearly the PizzaStore is dependent on the concrete pizza classes.

Now, this principle tells us we should instead write our code so that we are depending on abstractions, not concrete classes. That goes for both our high level modules and our low-level modules.

But how do we do this? Let's think about how we'd apply this principle to our Very Dependent PizzaStore implementation...

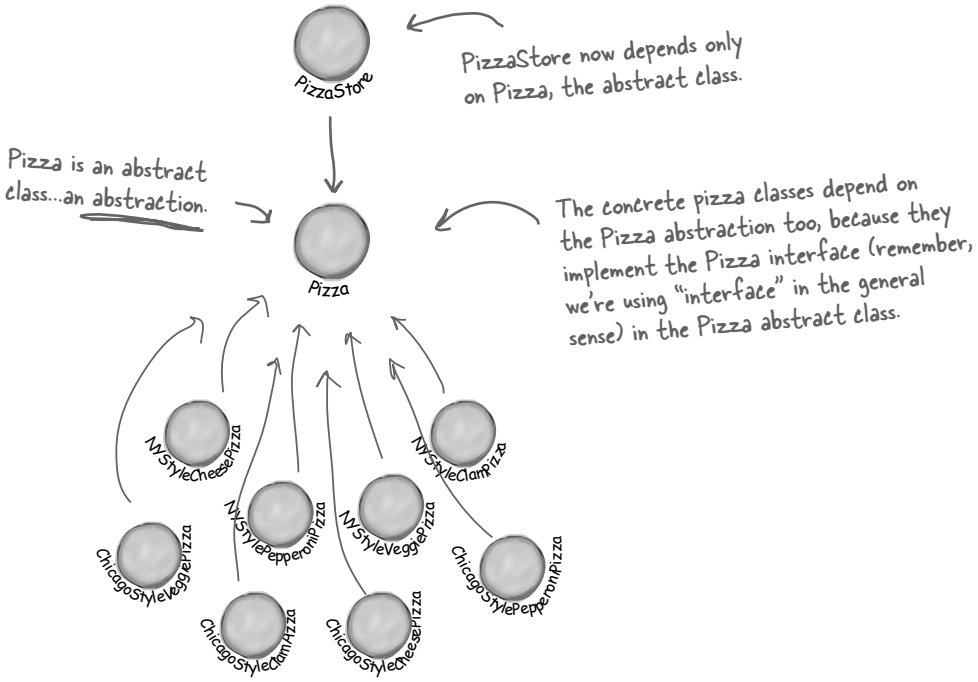
Applying the Principle

Now, the main problem with the Very Dependent PizzaStore is that it depends on every type of pizza because it actually instantiates concrete types in its orderPizza() method.

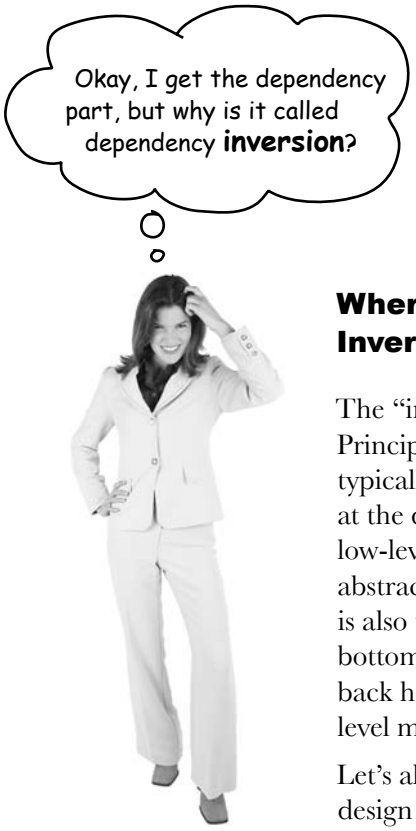
While we've created an abstraction, Pizza, we're nevertheless creating concrete Pizzas in this code, so we don't get a lot of leverage out of this abstraction.

How can we get those instantiations out of the orderPizza() method? Well, as we know, the Factory Method allows us to do just that.

So, after we've applied the Factory Method, our diagram looks like this:



After applying the Factory Method, you'll notice that our high-level component, the PizzaStore, and our low-level components, the pizzas, both depend on Pizza, the abstraction. Factory Method is not the only technique for adhering to the Dependency Inversion Principle, but it is one of the more powerful ones.



Where's the "inversion" in Dependency Inversion Principle?

The "inversion" in the name Dependency Inversion Principle is there because it inverts the way you typically might think about your OO design. Look at the diagram on the previous page, notice that the low-level components now depend on a higher level abstraction. Likewise, the high-level component is also tied to the same abstraction. So, the top-to-bottom dependency chart we drew a couple of pages back has inverted itself, with both high-level and low-level modules now depending on the abstraction.

Let's also walk through the thinking behind the typical design process and see how introducing the principle can invert the way we think about the design...

Inverting your thinking...

Okay, so you need to implement a PizzaStore. What's the first thought that pops into your head?

Right, you start at top and follow things down to the concrete classes. But, as you've seen, you don't want your store to know about the concrete pizza types, because then it'll be dependent on all those concrete classes!

Now, let's "invert" your thinking... instead of starting at the top, start at the Pizzas and think about what you can abstract.

Right! You are thinking about the abstraction *Pizza*. So now, go back and think about the design of the Pizza Store again.

Close. But to do that you'll have to rely on a factory to get those concrete classes out of your Pizza Store. Once you've done that, your different concrete pizza types depend only on an abstraction and so does your store. We've taken a design where the store depended on concrete classes and inverted those dependencies (along with your thinking).

A few guidelines to help you follow the Principle...

The following guidelines can help you avoid OO designs that violate the Dependency Inversion Principle:

- No variable should hold a reference to a concrete class.

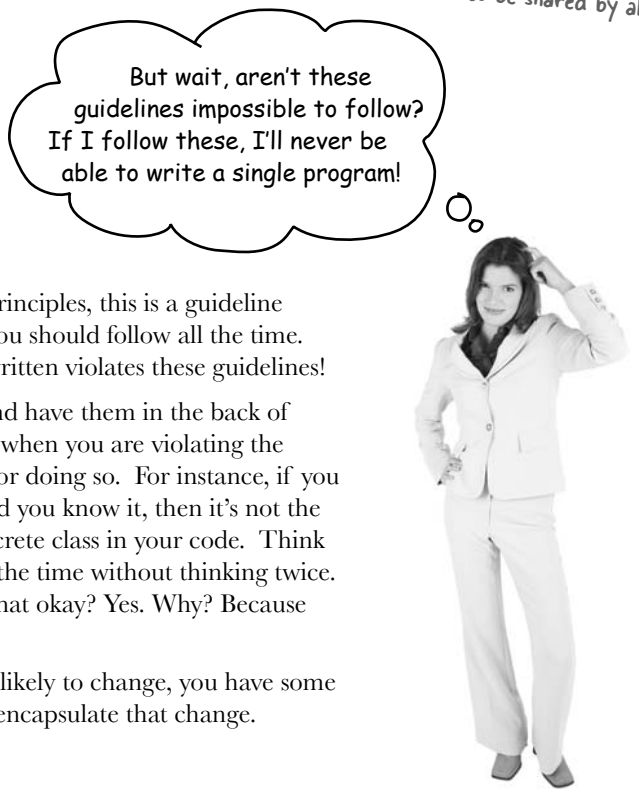
If you use **new**, you'll be holding a reference to a concrete class. Use a factory to get around that!

- No class should derive from a concrete class.

If you derive from a concrete class, you're depending on a concrete class. Derive from an abstraction, like an interface or an abstract class.

- No method should override an implemented method of any of its base classes.

If you override an implemented method, then your base class wasn't really an abstraction to start with. Those methods implemented in the base class are meant to be shared by all your subclasses.



You're exactly right! Like many of our principles, this is a guideline you should strive for, rather than a rule you should follow all the time. Clearly, every single Java program ever written violates these guidelines!

But, if you internalize these guidelines and have them in the back of your mind when you design, you'll know when you are violating the principle and you'll have a good reason for doing so. For instance, if you have a class that isn't likely to change, and you know it, then it's not the end of the world if you instantiate a concrete class in your code. Think about it; we instantiate String objects all the time without thinking twice. Does that violate the principle? Yes. Is that okay? Yes. Why? Because String is very unlikely to change.

If, on the other hand, a class you write is likely to change, you have some good techniques like Factory Method to encapsulate that change.