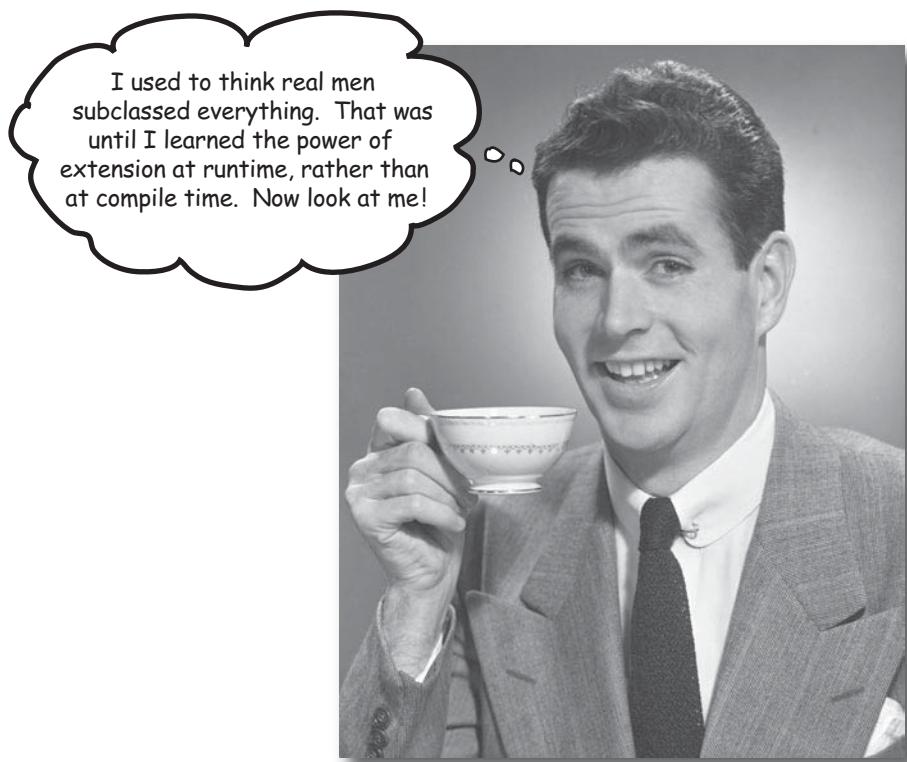


3 the Decorator Pattern

Decorating Objects



Just call this chapter “Design Eye for the Inheritance Guy.”

We'll re-examine the typical overuse of inheritance and you'll learn how to decorate your classes at runtime using a form of object composition. Why? Once you know the techniques of decorating, you'll be able to give your (or someone else's) objects new responsibilities *without making any code changes to the underlying classes.*

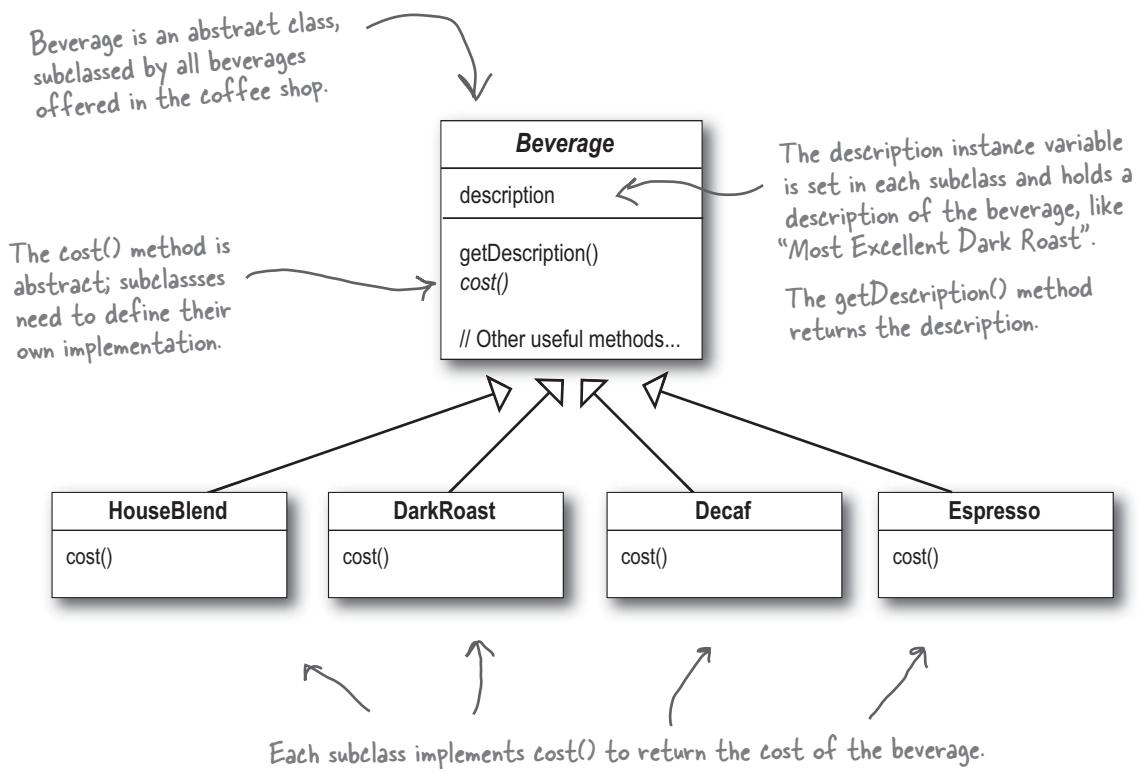
Welcome to Starbuzz Coffee

Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around. If you've seen one on your local corner, look across the street; you'll see another one.



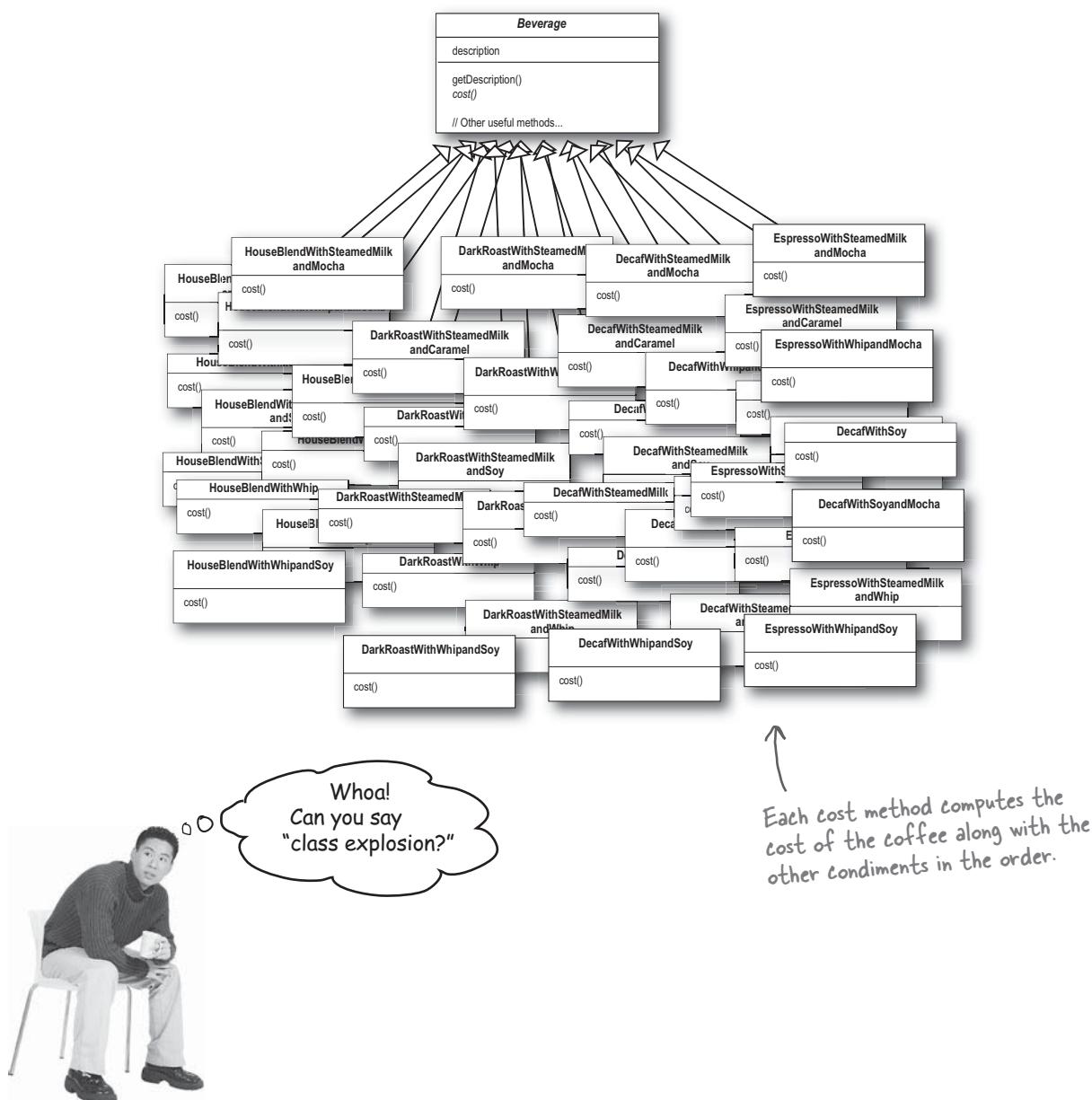
Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.

When they first went into business they designed their classes like this...



In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them built into their order system.

Here's their first attempt...



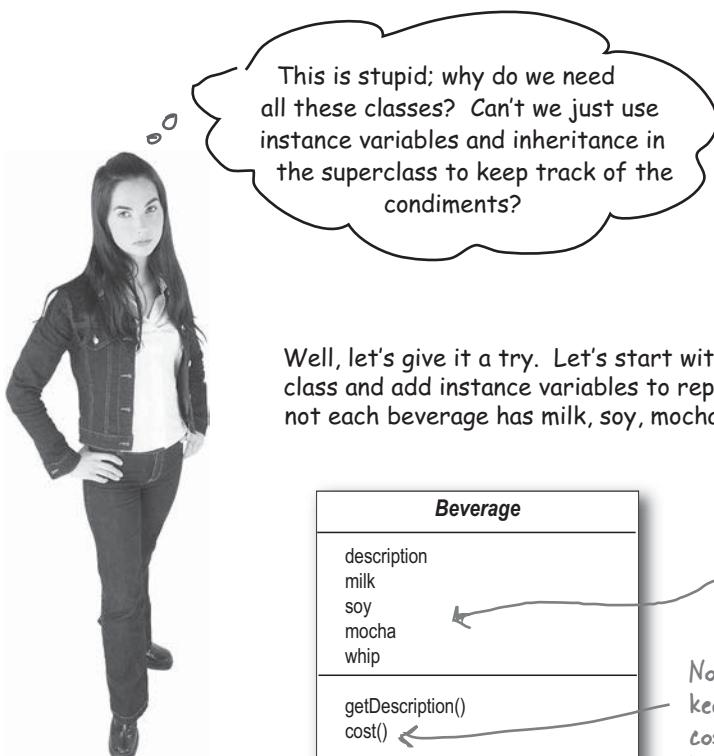
violating design principles



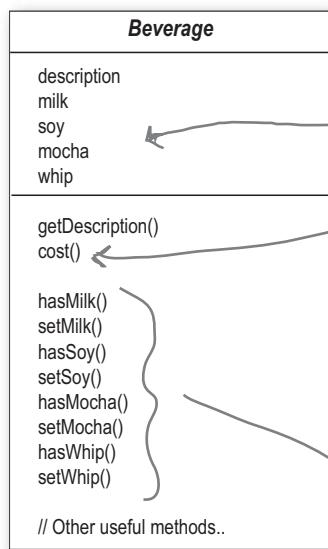
It's pretty obvious that Starbuzz has created a maintenance nightmare for themselves. What happens when the price of milk goes up? What do they do when they add a new caramel topping?

Thinking beyond the maintenance problem, which of the design principles that we've covered so far are they violating?

Hint: they're violating two of them in a big way!



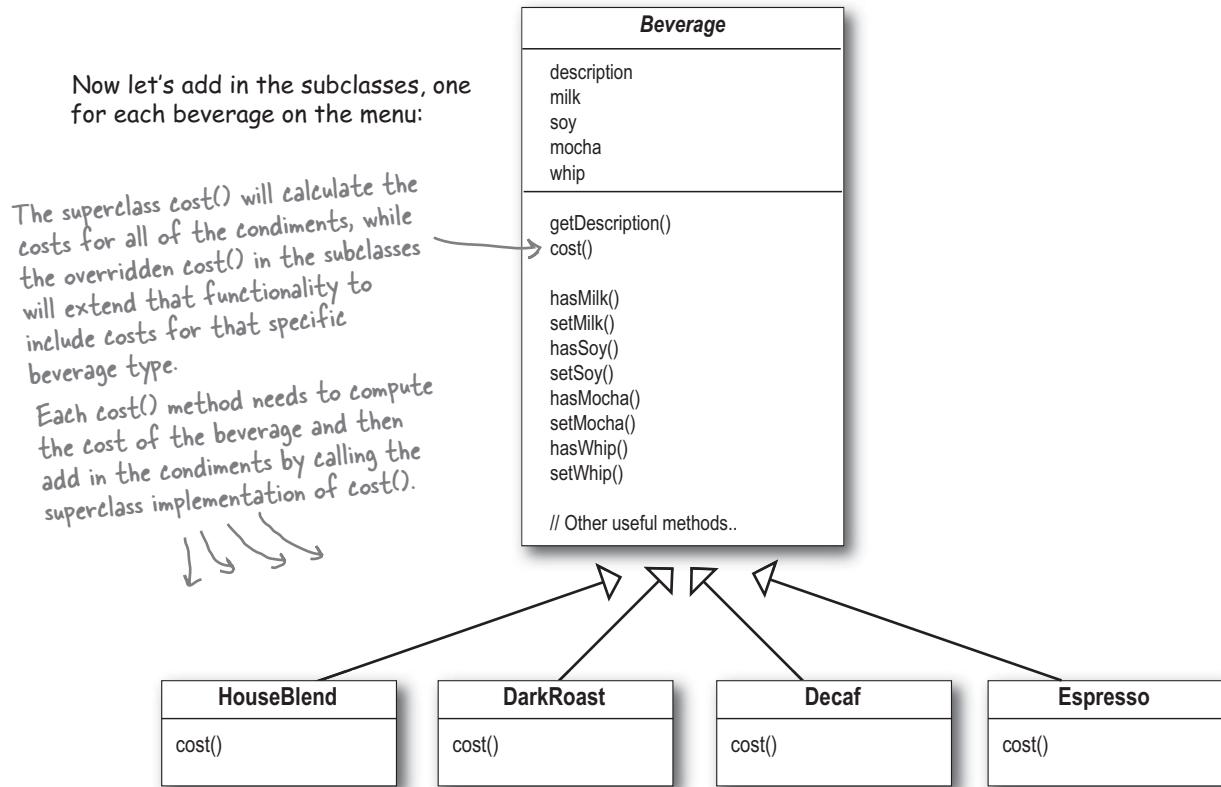
Well, let's give it a try. Let's start with the Beverage base class and add instance variables to represent whether or not each beverage has milk, soy, mocha and whip...



New boolean values for each condiment.

Now we'll implement `cost()` in `Beverage` (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override `cost()`, but they will also invoke the super version so that they can calculate the total cost of the basic beverage plus the costs of the added condiments.

These get and set the boolean values for the condiments.



Sharpen your pencil

Write the `cost()` methods for the following classes (pseudo-Java is okay):

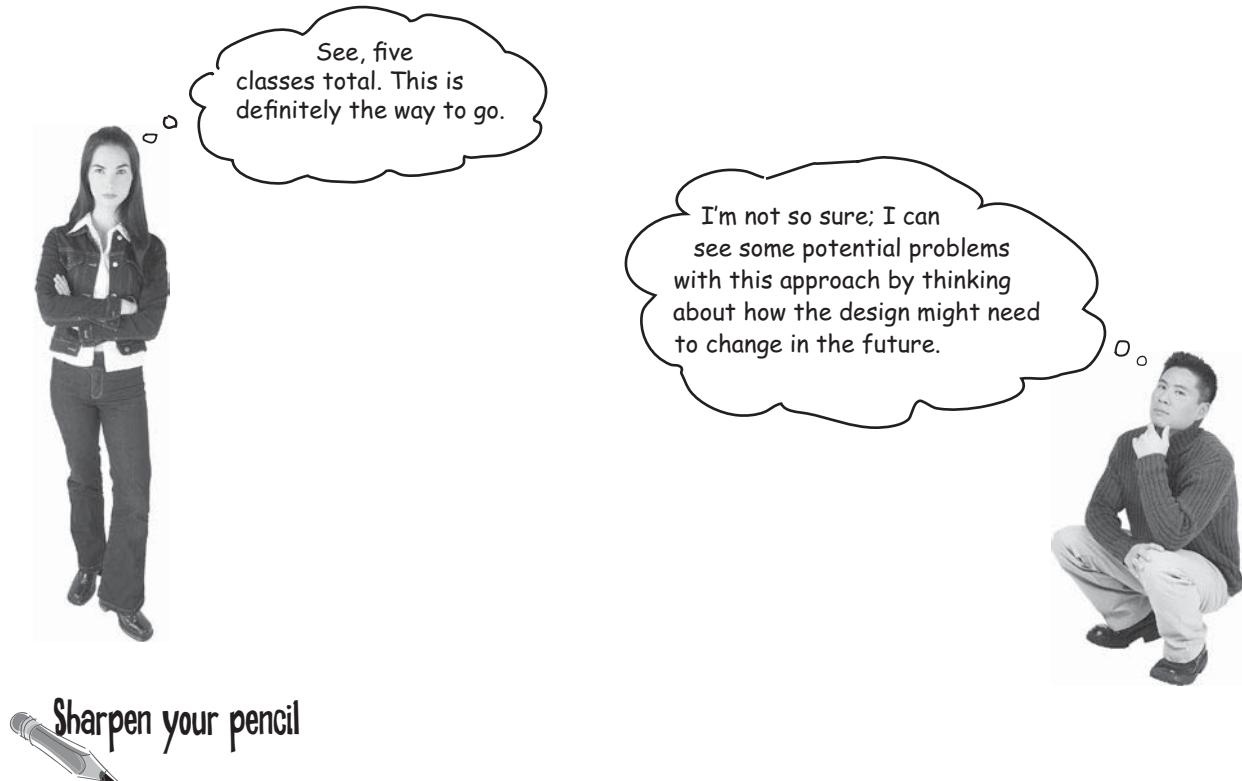
```

public class Beverage {
    public double cost() {
    }
}
  
```

```

public class DarkRoast extends Beverage {
    public DarkRoast() {
        description = "Most Excellent Dark Roast";
    }
    public double cost() {
    }
}
  
```

impact of change



Sharpen your pencil

What requirements or other factors might change that will impact this design?

Price changes for condiments will force us to alter existing code.

New condiments will force us to add new methods and alter the cost method in the superclass.

We may have new beverages. For some of these beverages (iced tea?), the condiments may not be appropriate, yet the Tea subclass will still inherit methods like hasWhip().

As we saw in
Chapter 1, this is
a very bad idea!

What if a customer wants a double mocha?

Your turn:



Master and Student...

Master: Grasshopper, it has been some time since our last meeting. Have you been deep in meditation on inheritance?

Student: Yes, Master. While inheritance is powerful, I have learned that it doesn't always lead to the most flexible or maintainable designs.

Master: Ah yes, you have made some progress. So, tell me my student, how then will you achieve reuse if not through inheritance?

Student: Master, I have learned there are ways of “inheriting” behavior at runtime through composition and delegation.

Master: Please, go on...

Student: When I inherit behavior by subclassing, that behavior is set statically at compile time. In addition, all subclasses must inherit the same behavior. If however, I can extend an object's behavior through composition, then I can do this dynamically at runtime.

Master: Very good, Grasshopper, you are beginning to see the power of composition.

Student: Yes, it is possible for me to add multiple new responsibilities to objects through this technique, including responsibilities that were not even thought of by the designer of the superclass. And, I don't have to touch their code!

Master: What have you learned about the effect of composition on maintaining your code?

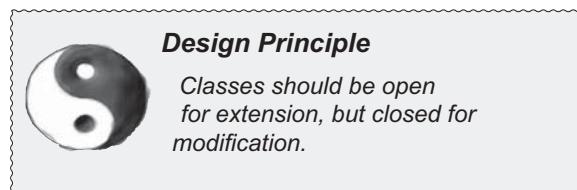
Student: Well, that is what I was getting at. By dynamically composing objects, I can add new functionality by writing new code rather than altering existing code. Because I'm not changing existing code, the chances of introducing bugs or causing unintended side effects in pre-existing code are much reduced.

Master: Very good. Enough for today, Grasshopper. I would like for you to go and meditate further on this topic... Remember, code should be closed (to change) like the lotus flower in the evening, yet open (to extension) like the lotus flower in the morning.

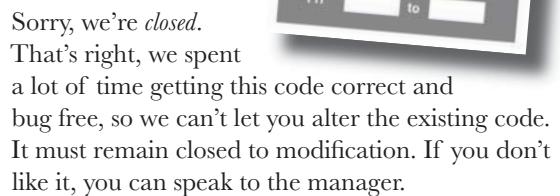
the open-closed principle

The Open-Closed Principle

Grasshopper is on to one of the most important design principles:



Come on in; we're *open*. Feel free to extend our classes with any new behavior you like. If your needs or requirements change (and we know they will), just go ahead and make your own extensions.



Sorry, we're *closed*. That's right, we spent a lot of time getting this code correct and bug free, so we can't let you alter the existing code. It must remain closed to modification. If you don't like it, you can speak to the manager.

Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. What do we get if we accomplish this? Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

there are no Dumb Questions

Q: Open for extension and closed for modification? That sounds very contradictory. How can a design be both?

A: That's a very good question. It certainly sounds contradictory at first. After all, the less modifiable something is, the harder it is to extend, right?

As it turns out, though, there are some clever OO techniques for allowing systems to be extended, even if we can't change the underlying code. Think about the Observer Pattern (in Chapter 2)... by adding new Observers, we can extend the Subject at any time, without adding code to the Subject. You'll see quite a few more ways of extending behavior with other OO design techniques.

Q: Okay, I understand Observable, but how do I generally design something to be extensible, yet closed for modification?

A: Many of the patterns give us time tested designs that protect your code from being modified by supplying a means of extension. In this chapter you'll see a good example of using the Decorator pattern to follow the Open-Closed principle.

Q: How can I make every part of my design follow the Open-Closed Principle?

A: Usually, you can't. Making OO design flexible and open to extension without the modification of existing code takes time and effort. In general, we don't have the luxury of tying down every part of our designs (and it would probably be wasteful). Following the Open-Closed Principle usually introduces new levels of abstraction, which adds complexity to our code. You want to concentrate on those areas that are most likely to change in your designs and apply the principles there.

Q: How do I know which areas of change are more important?

A: That is partly a matter of experience in designing OO systems and also a matter of the knowing the domain you are working in. Looking at other examples will help you learn to identify areas of change in your own designs.

While it may seem like a contradiction, there are techniques for allowing code to be extended without direct modification.

Be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful, unnecessary, and can lead to complex, hard to understand code.

Meet the Decorator Pattern

Okay, we've seen that representing our beverage plus condiment pricing scheme with inheritance has not worked out very well – we get class explosions, rigid designs, or we add functionality to the base class that isn't appropriate for some of the subclasses.

So, here's what we'll do instead: we'll start with a beverage and "decorate" it with the condiments at runtime. For example, if the customer wants a Dark Roast with Mocha and Whip, then we'll:

- ➊ Take a `DarkRoast` object**
- ➋ Decorate it with a `Mocha` object**
- ➌ Decorate it with a `Whip` object**
- ➍ Call the `cost()` method and rely on delegation to add on the condiment costs**

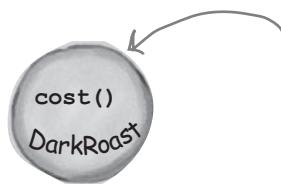
Okay, enough of the "Object Oriented Design Club." We have real problems here! Remember us? Starbuzz Coffee? Do you think you could use some of those design principles to actually help us?



Okay, but how do you "decorate" an object, and how does delegation come into this? A hint: think of decorator objects as "wrappers." Let's see how this works...

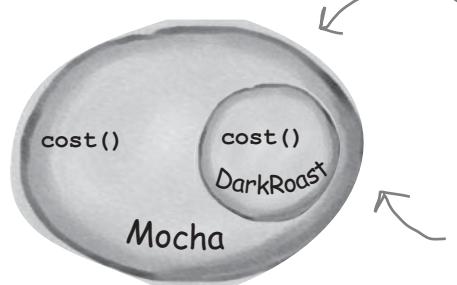
Constructing a drink order with Decorators

- ① We start with our **DarkRoast** object.



Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

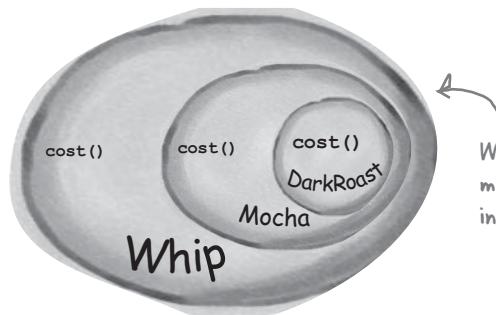
- ② The customer wants Mocha, so we create a **Mocha** object and wrap it around the **DarkRoast**.



The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type...)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

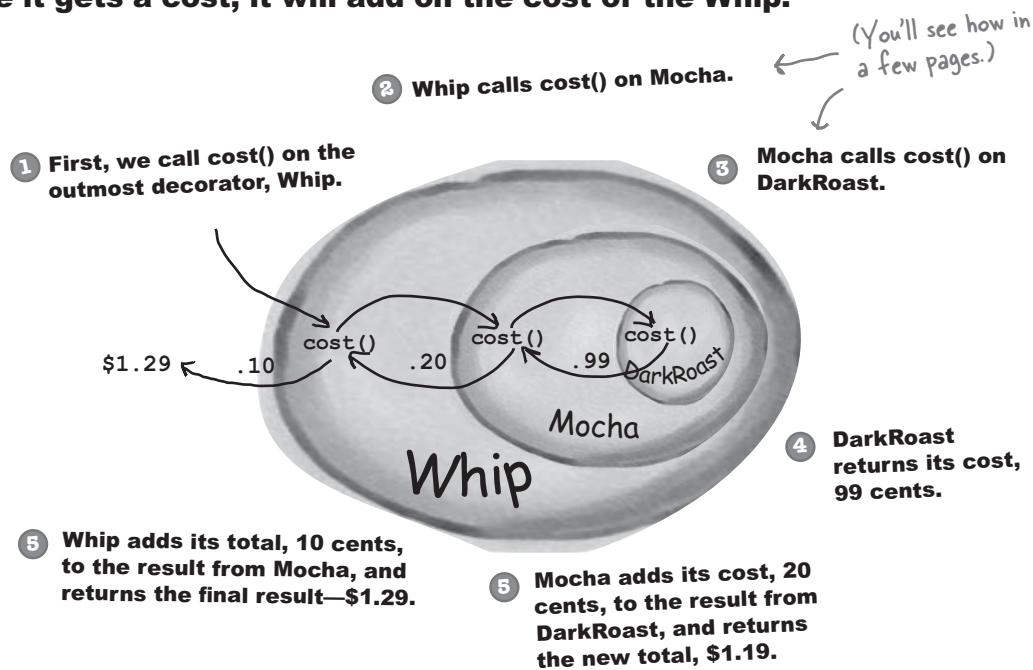
- ③ The customer also wants Whip, so we create a **Whip** decorator and wrap Mocha with it.



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

- ④ Now it's time to compute the cost for the customer. We do this by calling `cost()` on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.



Okay, here's what we know so far...

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Key Point!

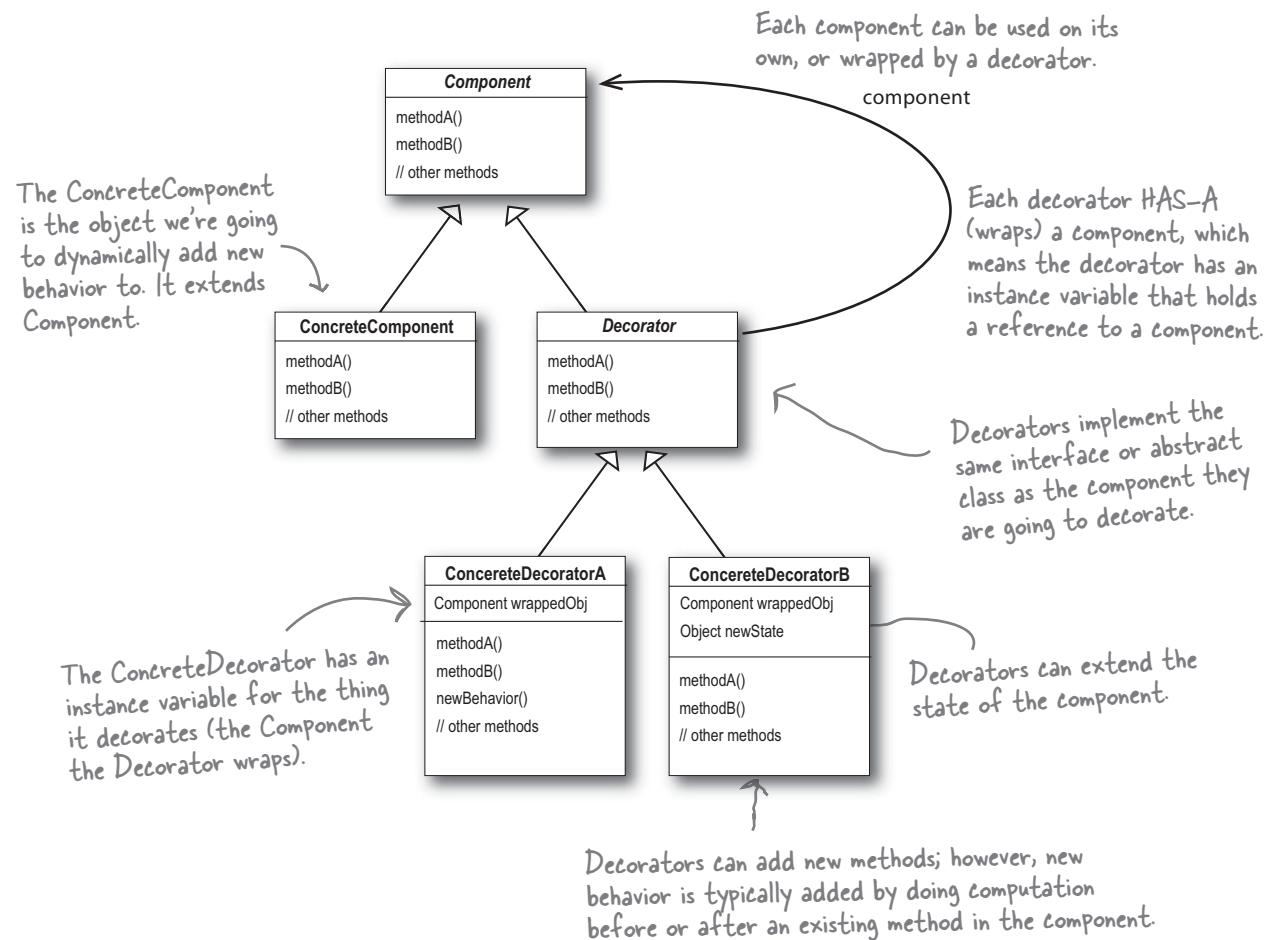
Now let's see how this all really works by looking at the Decorator Pattern definition and writing some code.

The Decorator Pattern defined

Let's first take a look at the Decorator Pattern description:

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

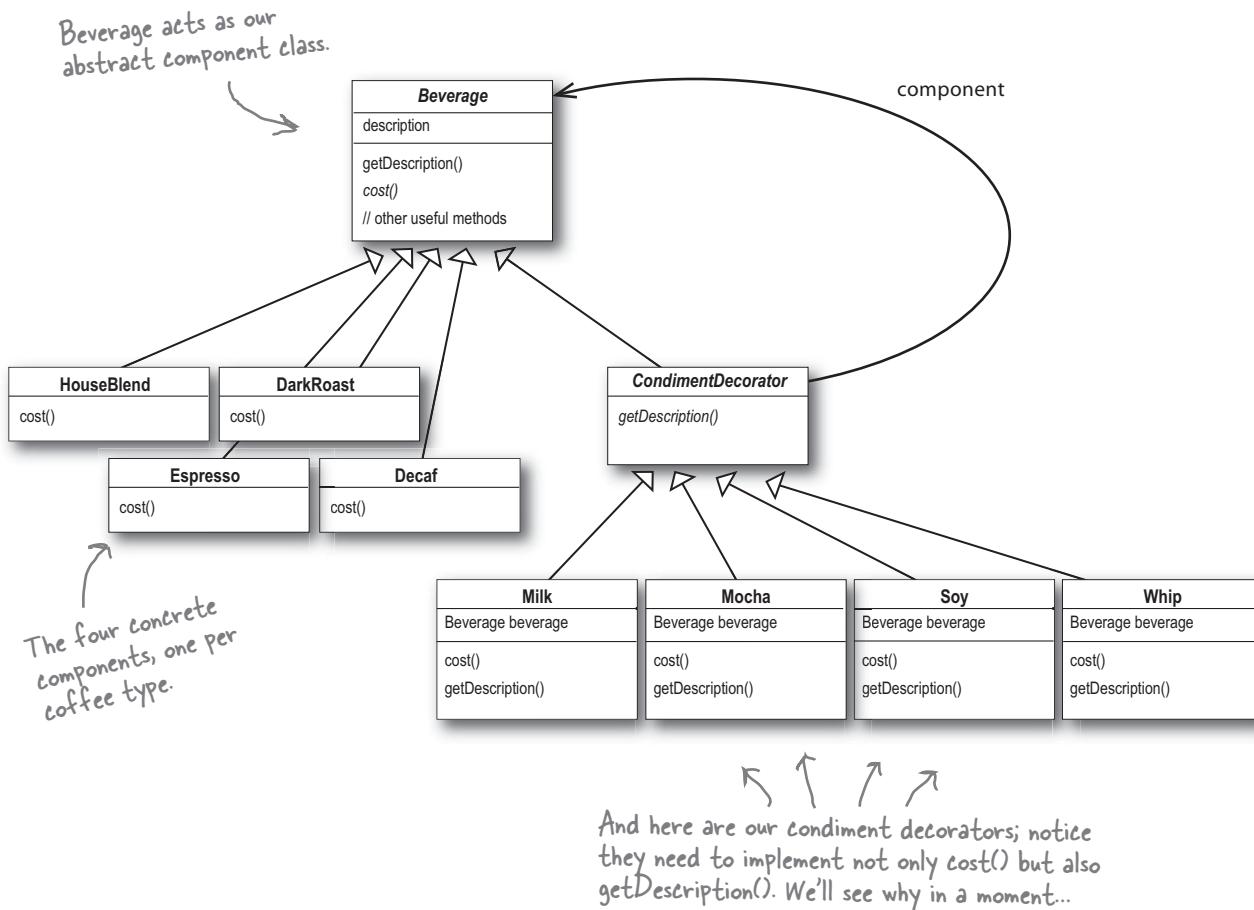
While that describes the *role* of the Decorator Pattern, it doesn't give us a lot of insight into how we'd *apply* the pattern to our own implementation. Let's take a look at the class diagram, which is a little more revealing (on the next page we'll look at the same structure applied to the beverage problem).



decorating beverages

Decorating our Beverages

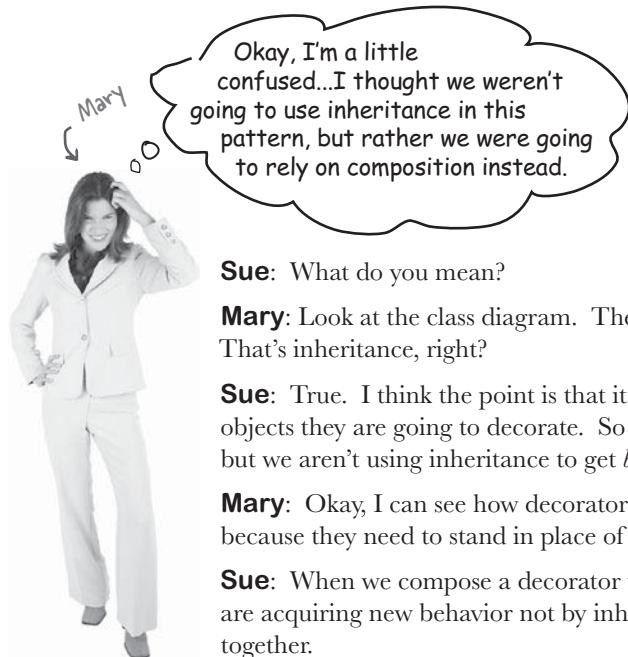
Okay, let's work our Starbuzz beverages into this framework...



Before going further, think about how you'd implement the `cost()` method of the coffees and the condiments. Also think about how you'd implement the `getDescription()` method of the condiments.

Cubicle Conversation

Some confusion over Inheritance versus Composition



Sue: What do you mean?

Mary: Look at the class diagram. The CondimentDecorator is extending the Beverage class. That's inheritance, right?

Sue: True. I think the point is that it's vital that the decorators have the same type as the objects they are going to decorate. So here we're using inheritance to achieve the *type matching*, but we aren't using inheritance to get *behavior*.

Mary: Okay, I can see how decorators need the same "interface" as the components they wrap because they need to stand in place of the component. But where does the behavior come in?

Sue: When we compose a decorator with a component, we are adding new behavior. We are acquiring new behavior not by inheriting it from a superclass, but by composing objects together.

Mary: Okay, so we're subclassing the abstract class Beverage in order to have the correct type, not to inherit its behavior. The behavior comes in through the composition of decorators with the base components as well as other decorators.

Sue: That's right.

Mary: Ooooh, I see. And because we are using object composition, we get a whole lot more flexibility about how to mix and match condiments and beverages. Very smooth.

Sue: Yes, if we rely on inheritance, then our behavior can only be determined statically at compile time. In other words, we get only whatever behavior the superclass gives us or that we override. With composition, we can mix and match decorators any way we like... *at runtime*.

Mary: And as I understand it, we can implement new decorators at any time to add new behavior. If we relied on inheritance, we'd have to go in and change existing code any time we wanted new behavior.

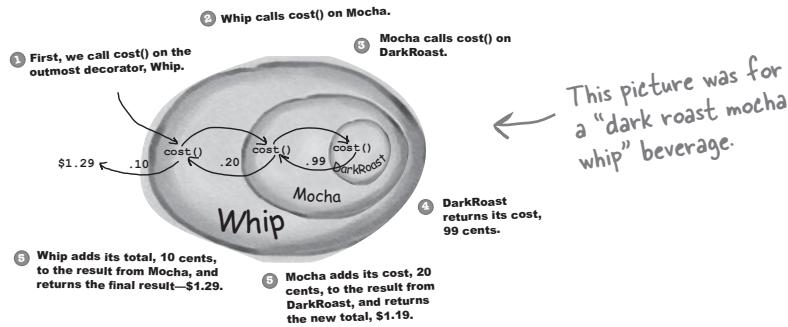
Sue: Exactly.

Mary: I just have one more question. If all we need to inherit is the type of the component, how come we didn't use an interface instead of an abstract class for the Beverage class?

Sue: Well, remember, when we got this code, Starbuzz already *had* an abstract Beverage class. Traditionally the Decorator Pattern does specify an abstract component, but in Java, obviously, we could use an interface. But we always try to avoid altering existing code, so don't "fix" it if the abstract class will work just fine.

New barista training

Make a picture for what happens when the order is for a "double mocha soy latte with whip" beverage. Use the menu to get the correct prices, and draw your picture using the same format we used earlier (from a few pages back):



Okay, I need for you to make me a double mocha, soy latte with whip.



Sharpen your pencil

Draw your picture here.

Starbuzz Coffee

Coffees

House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99

Condiments

Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10



Writing the Starbuzz code

It's time to whip this design into some real code.



Let's start with the Beverage class, which doesn't need to change from Starbuzz's original design. Let's take a look:

```
public abstract class Beverage {
    String description = "Unknown Beverage";
    public String getDescription() {
        return description;
    }
    public abstract double cost();
}
```

Beverage is an abstract class with the two methods `getDescription()` and `cost()`.

`getDescription` is already implemented for us, but we need to implement `cost()` in the subclasses.

Beverage is simple enough. Let's implement the abstract class for the Condiments (Decorator) as well:

```
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```

First, we need to be interchangeable with a Beverage, so we extend the Beverage class.

We're also going to require that the condiment decorators all reimplement the `getDescription()` method. Again, we'll see why in a sec...

implementing the beverages

Coding beverages

Now that we've got our base classes out of the way, let's implement some beverages. We'll start with Espresso. Remember, we need to set a description for the specific beverage and also implement the cost() method.

```
public class Espresso extends Beverage {  
  
    public Espresso() {  
        description = "Espresso";  
    }  
  
    public double cost() {  
        return 1.99;  
    }  
}
```

First we extend the Beverage class, since this is a beverage.

To take care of the description, we set this in the constructor for the class. Remember the description instance variable is inherited from Beverage.

Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: \$1.99.

```
public class HouseBlend extends Beverage {  
    public HouseBlend() {  
        description = "House Blend Coffee";  
    }  
  
    public double cost() {  
        return .89;  
    }  
}
```

Okay, here's another Beverage. All we do is set the appropriate description, "House Blend Coffee," and then return the correct cost: 89¢.

Starbuzz Coffee		
<u>Coffees</u>		
House Blend		.89
Dark Roast		.99
Decaf		1.05
Espresso		1.99
<u>Condiments</u>		
Steamed Milk		.10
Mocha		.20
Soy		.15
Whip		.10

You can create the other two Beverage classes (DarkRoast and Decaf) in exactly the same way.

Coding condiments

If you look back at the Decorator Pattern class diagram, you'll see we've now written our abstract component (Beverage), we have our concrete components (HouseBlend), and we have our abstract decorator (CondimentDecorator). Now it's time to implement the concrete decorators. Here's Mocha:

```

Mocha is a decorator, so we
extend CondimentDecorator.
↓
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}

Now we need to compute the cost of our beverage
with Mocha. First, we delegate the call to the
object we're decorating, so that it can compute the
cost; then, we add the cost of Mocha to the result.
    ↑
Remember, CondimentDecorator
extends Beverage.
We're going to instantiate Mocha with
a reference to a Beverage using:
(1) An instance variable to hold the
beverage we are wrapping.
(2) A way to set this instance
variable to the object we are wrapping.
Here, we're going to pass the beverage
we're wrapping to the decorator's
constructor.

We want our description to not only
include the beverage – say "Dark
Roast" – but also to include each
item decorating the beverage, for
instance, "Dark Roast, Mocha". So
we first delegate to the object we are
decorating to get its description, then
append ", Mocha" to that description.

```

On the next page we'll actually instantiate the beverage and wrap it with all its condiments (decorators), but first...



Sharpen your pencil

Write and compile the code for the other Soy and Whip condiments. You'll need them to finish and test the application.

testing the beverages

Serving some coffees

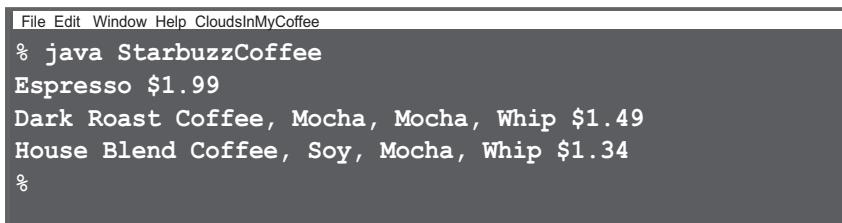
Congratulations. It's time to sit back, order a few coffees and marvel at the flexible design you created with the Decorator Pattern.

Here's some test code* to make orders:

```
public class StarbuzzCoffee {  
    public static void main(String args[]) {  
        Beverage beverage = new Espresso(); ← Order up an espresso, no condiments  
        System.out.println(beverage.getDescription() and print its description and cost.  
            + " $" + beverage.cost());  
  
        Beverage beverage2 = new DarkRoast(); ← Make a DarkRoast object.  
        beverage2 = new Mocha(beverage2); ← Wrap it with a Mocha.  
        beverage2 = new Mocha(beverage2); ← Wrap it in a second Mocha.  
        beverage2 = new Whip(beverage2); ← Wrap it in a Whip.  
        System.out.println(beverage2.getDescription()  
            + " $" + beverage2.cost());  
  
        Beverage beverage3 = new HouseBlend(); ← Finally, give us a HouseBlend  
        beverage3 = new Soy(beverage3); with Soy, Mocha, and Whip.  
        beverage3 = new Mocha(beverage3);  
        beverage3 = new Whip(beverage3);  
        System.out.println(beverage3.getDescription()  
            + " $" + beverage3.cost());  
    }  
}
```

*We're going to see a much better way of creating decorated objects when we cover the Factory and Builder Design Patterns.

Now, let's get those orders in:



```
File Edit Window Help CloudsInMyCoffee  
% java StarbuzzCoffee  
Espresso $1.99  
Dark Roast Coffee, Mocha, Mocha, Whip $1.49  
House Blend Coffee, Soy, Mocha, Whip $1.34  
%
```

there are no Dumb Questions

Q: I'm a little worried about code that might test for a specific concrete component – say, HouseBlend – and do something, like issue a discount. Once I've wrapped the HouseBlend with decorators, this isn't going to work anymore.

A: That is exactly right. If you have code that relies on the concrete component's type, decorators will break that code. As long as you only write code against the abstract component type, the use of decorators will remain transparent to your code. However, once you start writing code against concrete components, you'll want to rethink your application design and your use of decorators.

Q: Wouldn't it be easy for some client of a beverage to end up with a decorator that isn't the outermost decorator? Like if I had a DarkRoast with Mocha, Soy, and Whip, it would be easy to write code that somehow ended up with a reference to Soy instead of Whip, which means it would not include Whip in the order.

A: You could certainly argue that you have to manage more objects with the Decorator Pattern and so there is an increased chance that coding errors will introduce the kinds of problems you suggest. However, decorators are typically created by using other patterns like Factory and Builder. Once we've covered these patterns, you'll see that the creation of the concrete component with its decorator is "well encapsulated" and doesn't lead to these kinds of problems.

Q: Can decorators know about the other decorations in the chain? Say, I wanted my `getDescription()` method to print "Whip, Double Mocha" instead of "Mocha, Whip, Mocha"? That would require that my outermost decorator know all the decorators it is wrapping.

A: Decorators are meant to add behavior to the object they wrap. When you need to peek at multiple layers into the decorator chain, you are starting to push the decorator beyond its true intent. Nevertheless, such things are possible. Imagine a CondimentPrettyPrint decorator that parses the final description and can print "Mocha, Whip, Mocha" as "Whip, Double Mocha." Note that `getDescription()` could return an `ArrayList` of descriptions to make this easier.



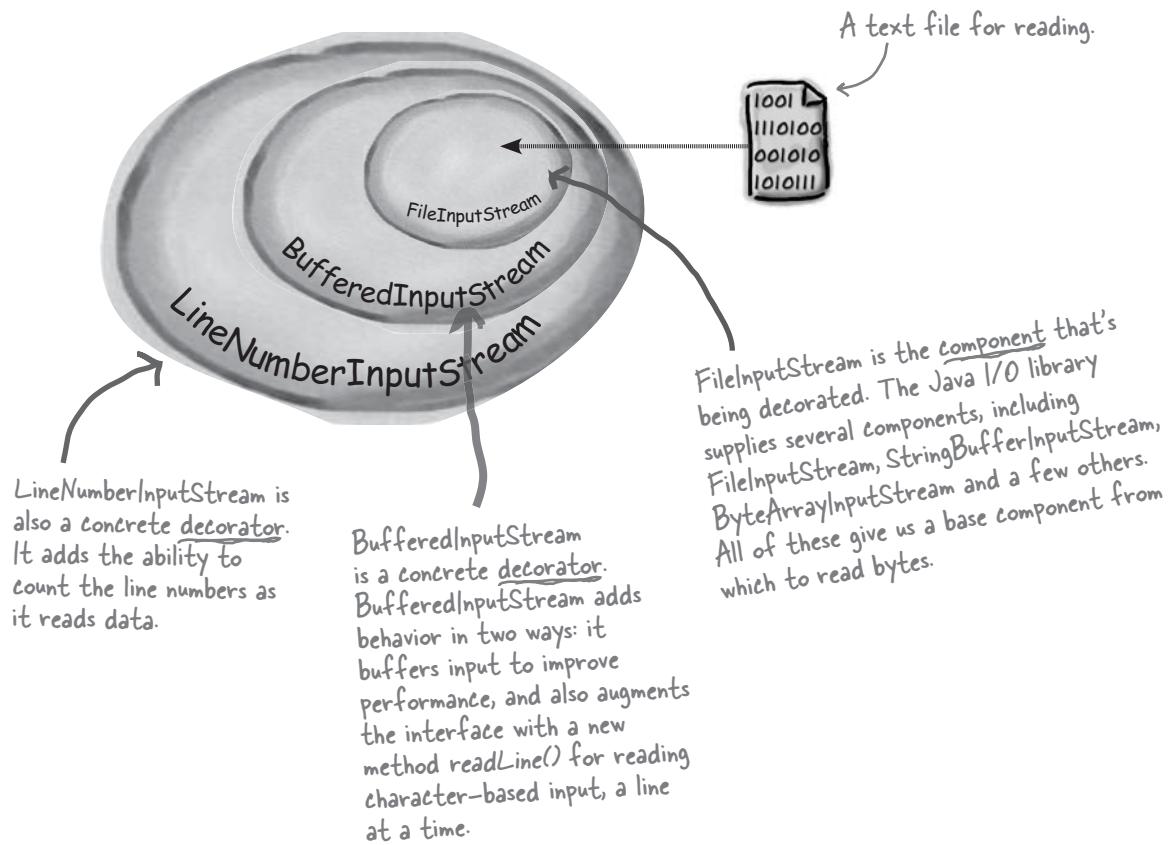
Sharpen your pencil

Our friends at Starbuzz have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (translation: small, medium, and large). Starbuzz saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class: `setSize()` and `getSize()`. They'd also like for the condiments to be charged according to size, so for instance, Soy costs 10¢, 15¢ and 20¢ respectively for tall, grande, and venti coffees.

How would you alter the decorator classes to handle this change in requirements?

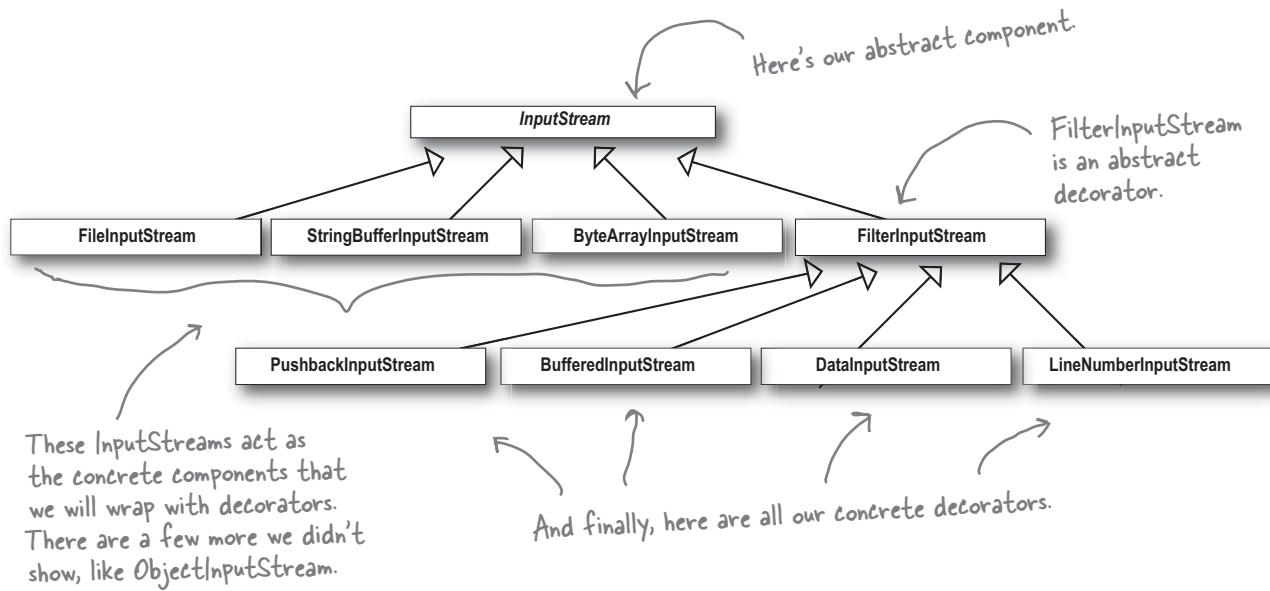
Real World Decorators: Java I/O

The large number of classes in the `java.io` package is... *overwhelming*. Don't feel alone if you said "whoa" the first (and second and third) time you looked at this API. But now that you know the Decorator Pattern, the I/O classes should make more sense since the `java.io` package is largely based on Decorator. Here's a typical set of objects that use decorators to add functionality to reading data from a file:



`BufferedInputStream` and `LineNumberInputStream` both extend `FilterInputStream`, which acts as the abstract decorator class.

Decorating the java.io classes



You can see that this isn't so different from the Starbuzz design. You should now be in a good position to look over the `java.io` API docs and compose decorators on the various *input* streams.

You'll see that the *output* streams have the same design. And you've probably already found that the Reader/Writer streams (for character-based data) closely mirror the design of the streams classes (with a few differences and inconsistencies, but close enough to figure out what's going on).

Java I/O also points out one of the *downsides* of the Decorator Pattern: designs using this pattern often result in a large number of small classes that can be overwhelming to a developer trying to use the Decorator-based API. But now that you know how Decorator works, you can keep things in perspective and when you're using someone else's Decorator-heavy API, you can work through how their classes are organized so that you can easily use wrapping to get the behavior you're after.

Writing your own Java I/O Decorator

Okay, you know the Decorator Pattern, you've seen the I/O class diagram. You should be ready to write your own input decorator.

How about this: write a decorator that converts all uppercase characters to lowercase in the input stream. In other words, if we read in "I know the Decorator Pattern therefore I RULE!" then your decorator converts this to "i know the decorator pattern therefore i rule!"

Don't forget to import
java.io... (not shown)

First, extend the FilterInputStream, the abstract decorator for all InputStreams.

```
public class LowerCaseInputStream extends FilterInputStream {  
    public LowerCaseInputStream(InputStream in) {  
        super(in);  
    }  
  
    public int read() throws IOException {  
        int c = super.read();  
        return (c == -1 ? c : Character.toLowerCase((char)c));  
    }  
  
    public int read(byte[] b, int offset, int len) throws IOException {  
        int result = super.read(b, offset, len);  
        for (int i = offset; i < offset+result; i++) {  
            b[i] = (byte)Character.toLowerCase((char)b[i]);  
        }  
        return result;  
    }  
}
```

REMEMBER: we don't provide import and package statements in the code listings. Get the complete source code from the wickedlysmart web site. You'll find the URL on page xxxiii in the Intro.

No problem. I just have to extend the FilterInputStream class and override the read() methods.



Now we need to implement two read methods. They take a byte (or an array of bytes) and convert each byte (that represents a character) to lowercase if it's an uppercase character.

Test out your new Java I/O Decorator

Write some quick code to test the I/O decorator:

```
public class InputTest {  
    public static void main(String[] args) throws IOException {  
        int c;  
        try {  
            InputStream in =  
                new LowerCaseInputStream(  
                    new BufferedInputStream(  
                        new FileInputStream("test.txt")));  
  
            while((c = in.read()) >= 0) {  
                System.out.print((char)c);  
            }  
  
            in.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Just use the stream to read characters until the end of file and print as we go.

Set up the FileInputStream and decorate it, first with a BufferedInputStream and then our brand new LowerCaseInputStream filter.

I know the Decorator Pattern therefore I RULE!

test.txt file

You need to make this file.

Give it a spin:

```
File Edit Window Help DecoratorsRule  
% java InputTest  
i know the decorator pattern therefore i rule!  
%
```



HeadFirst: Welcome Decorator Pattern. We've heard that you've been a bit down on yourself lately?

Decorator: Yes, I know the world sees me as the glamorous design pattern, but you know, I've got my share of problems just like everyone.

HeadFirst: Can you perhaps share some of your troubles with us?

Decorator: Sure. Well, you know I've got the power to add flexibility to designs, that much is for sure, but I also have a *dark side*. You see, I can sometimes add a lot of small classes to a design and this occasionally results in a design that's less than straightforward for others to understand.

HeadFirst: Can you give us an example?

Decorator: Take the Java I/O libraries. These are notoriously difficult for people to understand at first. But if they just saw the classes as a set of wrappers around an `InputStream`, life would be much easier.

HeadFirst: That doesn't sound so bad. You're still a great pattern, and improving this is just a matter of public education, right?

Decorator: There's more, I'm afraid. I've got typing problems: you see, people sometimes take a piece of client code that relies on specific types and introduce decorators without thinking through everything. Now, one great thing about me is that ***you can usually insert decorators transparently and the client never has to know it's dealing with a decorator.*** But like I said, some code is dependent on specific types and when you start introducing decorators, boom! Bad things happen.

HeadFirst: Well, I think everyone understands that you have to be careful when inserting decorators, I don't think this is a reason to be too down on yourself.

Decorator: I know, I try not to be. I also have the problem that introducing decorators can increase the complexity of the code needed to instantiate the component. Once you've got decorators, you've got to not only instantiate the component, but also wrap it with who knows how many decorators.

HeadFirst: I'll be interviewing the Factory and Builder patterns next week – I hear they can be very helpful with this?

Decorator: That's true; I should talk to those guys more often.

HeadFirst: Well, we all think you're a great pattern for creating flexible designs and staying true to the Open-Closed Principle, so keep your chin up and think positively!

Decorator: I'll do my best, thank you.



Tools for your Design Toolbox

You've got another chapter under your belt and a new principle and pattern in the toolbox.

OO Principles

- Encapsulate what varies.
- Favor composition over inheritance.
- Program to interfaces, not implementations.
- Strive for loosely coupled designs between objects that interact.
- Classes should be open for extension but closed for modification.

OO Basics

- Injection
- Isolation
- Morphism
- Balance

OO Patterns

- Strategy
- Encapsulation
- Observer
- Decorator
- Composite
- Facade
- Adapter
- Bridge
- Visitor
- Decorator - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

And here's our first pattern for creating designs that satisfy the Open-Closed Principle. Or was it really the first? Is there another pattern we've used that follows this principle as well?



BULLET POINTS

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- In our designs we should allow behavior to be extended without the need to modify existing code.
- Composition and delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)
- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
- You can wrap a component with any number of decorators.
- Decorators are typically transparent to the client of the component; that is, unless the client is relying on the component's concrete type.
- Decorators can result in many small objects in our design, and overuse can be complex.

Exercise solutions

```

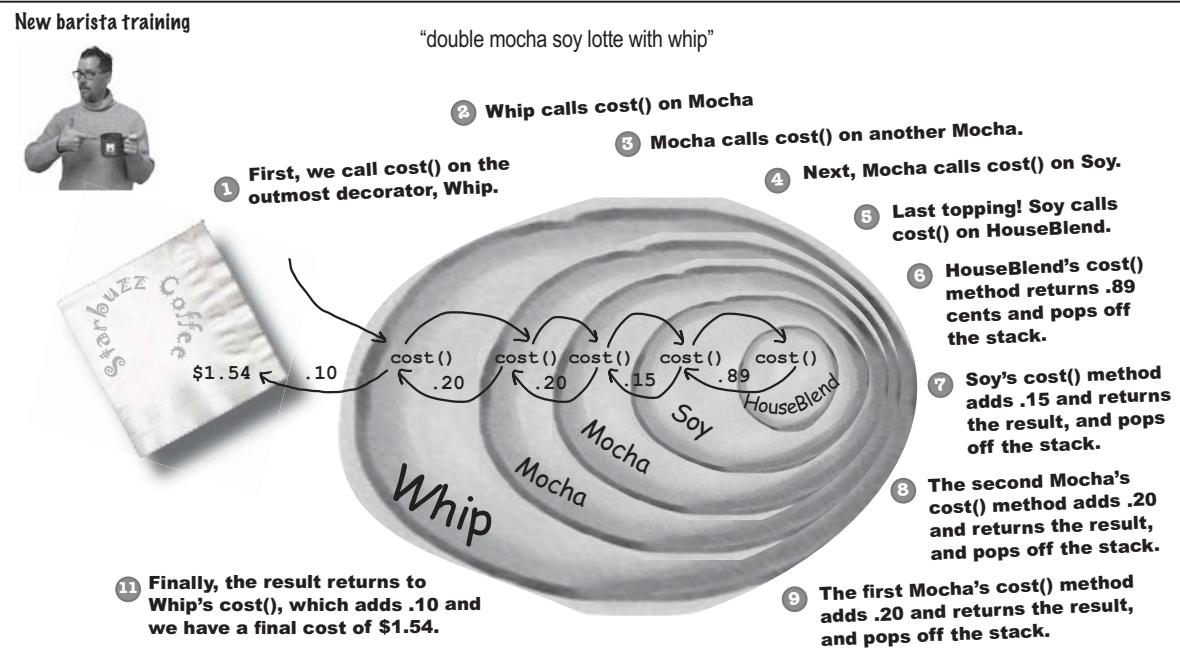
public class Beverage {
    // declare instance variables for milkCost,
    // soyCost, mochaCost, and whipCost, and
    // getters and setters for milk, soy, mocha
    // and whip.

    public float cost() {
        float condimentCost = 0.0;
        if (hasMilk()) {
            condimentCost += milkCost;
        }
        if (hasSoy()) {
            condimentCost += soyCost;
        }
        if (hasMocha()) {
            condimentCost += mochaCost;
        }
        if (hasWhip()) {
            condimentCost += whipCost;
        }
        return condimentCost;
    }
}

public class DarkRoast extends Beverage {
    public DarkRoast() {
        description = "Most Excellent Dark Roast";
    }

    public float cost() {
        return 1.99 + super.cost();
    }
}

```



Exercise solutions

Our friends at Starbuzz have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (for us normal folk: small, medium, and large). Starbuzz saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class: `setSize()` and `getSize()`. They'd also like for the condiments to be charged according to size, so for instance, Soy costs 10¢, 15¢, and 20¢ respectively for tall, grande, and venti coffees.

How would you alter the decorator classes to handle this change in requirements?

```
public class Soy extends CondimentDecorator {
    Beverage beverage;

    public Soy(Beverage beverage) {
        this.beverage = beverage;
    }

    public getSize() {
        return beverage.getSize();
    }

    public String getDescription() {
        return beverage.getDescription() + ", Soy";
    }

    public double cost() {
        double cost = beverage.cost();
        if (getSize() == Beverage.TALL) {
            cost += .10;
        } else if (getSize() == Beverage.GRANDE) {
            cost += .15;
        } else if (getSize() == Beverage.VENTI) {
            cost += .20;
        }
        return cost;
    }
}
```

Now we need to propagate the `getSize()` method to the wrapped beverage. We should also move this method to the abstract class since it's used in all condiment decorators.

Here we get the size (which propagates all the way to the concrete beverage) and then add the appropriate cost.

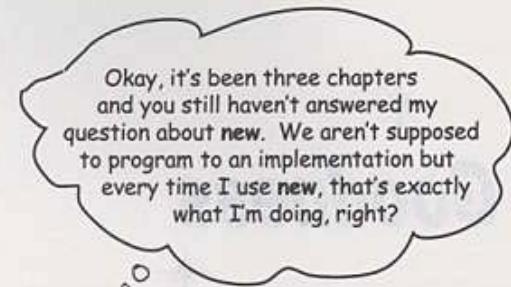
4 the Factory Pattern

Baking with OO Goodness



Get ready to bake some loosely coupled OO designs. There is more to making objects than just using the `new` operator. You'll learn that instantiation is an activity that shouldn't always be done in public and can often lead to *coupling problems*. And you don't want *that*, do you? Find out how Factory Patterns can help save you from embarrassing dependencies.

thinking about "new"



When you see “new”, think “concrete”.

Yes, when you use `new` you are certainly instantiating a concrete class, so that's definitely an implementation, not an interface. And it's a good question; you've learned that tying your code to a concrete class can make it more fragile and less flexible.

```
Duck duck = new MallardDuck();
```

We want to use interfaces
to keep code flexible.

But we have to create an
instance of a concrete class!

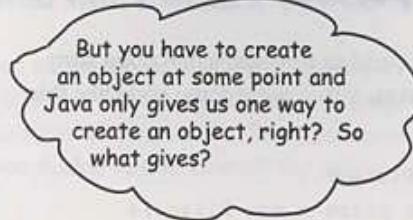
When you have a whole set of related concrete classes, often you're forced to write code like this:

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

We have a bunch of different
duck classes, and we don't know
until runtime which one we need
to instantiate.

Here we've got several concrete classes being instantiated, and the decision of which to instantiate is made at runtime depending on some set of conditions.

When you see code like this, you know that when it comes time for changes or extensions, you'll have to reopen this code and examine what needs to be added (or deleted). Often this kind of code ends up in several parts of the application making maintenance and updates more difficult and error-prone.



What's wrong with "new"?

Technically there's nothing wrong with `new`, after all, it's a fundamental part of Java. The real culprit is our old friend CHANGE and how change impacts our use of `new`.

By coding to an interface, you know you can insulate yourself from a lot of changes that might happen to a system down the road. Why? If your code is written to an interface, then it will work with any new classes implementing that interface through polymorphism. However, when you have code that makes use of lots of concrete classes, you're looking for trouble because that code may have to be changed as new concrete classes are added. So, in other words, your code will not be "closed for modification." To extend it with new concrete types, you'll have to reopen it.

So what can you do? It's times like these that you can fall back on OO Design Principles to look for clues. Remember, our first principle deals with change and guides us to *identify the aspects that vary and separate them from what stays the same*.

Remember that designs should be "open for extension but closed for modification" – see Chapter 3 for a review.



How might you take all the parts of your application that instantiate concrete classes and separate or encapsulate them from the rest of your application?

Identify what varies

Identifying the aspects that vary

Let's say you have a pizza shop, and as a cutting-edge pizza store owner in Objectville you might end up writing some code like this:

```
Pizza orderPizza() {  
    Pizza pizza = new Pizza();  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

For flexibility, we really want this to be an abstract class or interface, but we can't directly instantiate either of those.

But you need more than one type of pizza...

So then you'd add some code that *determines* the appropriate type of pizza and then goes about *making* the 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();  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

We're now passing in the type of pizza to orderPizza.

Based on the type of pizza, we instantiate the correct concrete class and assign it to the pizza instance variable. Note that each pizza here has to implement the Pizza interface.

Once we have a Pizza, we prepare it (you know, roll the dough, put on the sauce and add the toppings f cheese), then we bake it, cut it and box it!

Each Pizza subtype (CheesePizza, VeggiePizza, etc.) knows how to prepare itself.

But the pressure is on to add more pizza types

You realize that all of your competitors have added a couple of trendy pizzas to their menus: the Clam Pizza and the Veggie Pizza. Obviously you need to keep up with the competition, so you'll add these items to your menu. And you haven't been selling many Greek Pizzas lately, so you decide to take that off the menu:

```
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("clam")) {
        pizza = new ClamPizza();
    } else if (type.equals("veggie")) {
        pizza = new VeggiePizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

This code is NOT closed for modification. If the Pizza Shop changes its pizza offerings, we have to get into this code and modify it.

This is what varies. As the pizza selection changes over time, you'll have to modify this code over and over.

This is what we expect to stay the same. For the most part, preparing, cooking, and packaging a pizza has remained the same for years and years. So, we don't expect this code to change, just the pizzas it operates on.

Clearly, dealing with *which* concrete class is instantiated is really messing up our `orderPizza()` method and preventing it from being closed for modification. But now that we know what is varying and what isn't, it's probably time to encapsulate it.

encapsulate object creation

Encapsulating object creation

So now we know we'd be better off moving the object creation out of the `orderPizza()` method. But how? Well, what we're going to do is take the creation code and move it out into another object that is only going to be concerned with creating pizzas.

```
Pizza orderPizza(String type) {  
    Pizza pizza;  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

First we pull the object creation code out of the `orderPizza` Method

What's going to go here?

```
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();  
}
```

Then we place that code in an object that is only going to worry about how to create pizzas. If any other object needs a pizza created, this is the object to come to



We've got a name for this new object: we call it a **Factory**.

Factories handle the details of object creation. Once we have a `SimplePizzaFactory`, our `orderPizza()` method just becomes a client of that object. Any time it needs a pizza, it asks the pizza factory to make one. Gone are the days when the `orderPizza()` method needs to know about Greek versus Clam pizzas. Now the `orderPizza()` method just cares that it gets a pizza, which implements the `Pizza` interface so that it can call `prepare()`, `bake()`, `cut()`, and `box()`.

We've still got a few details to fill in here; for instance, what does the `orderPizza()` method replace its creation code with? Let's implement a simple factory for the pizza store and find out...

Building a simple pizza factory

We'll start with the factory itself. What we're going to do is define a class that encapsulates the object creation for all pizzas. Here it is...

```
Here's our new class, the SimplePizzaFactory. It has
one job in life: creating pizzas for its clients.
```

```
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;
    }
}
```

```
First we define a
createPizza() method in
the factory. This is the
method all clients will use to
instantiate new objects.
```

```
Here's the code we
plucked out of the
orderPizza() method.
```

```
This code is still parameterized by
the type of the pizza, just like our
original orderPizza() method was.
```

Q: What's the advantage of this?
It looks like we are just pushing the problem off to another object.

A: One thing to remember is that the SimplePizzaFactory may have many clients. We've only seen the orderPizza() method; however, there may be a PizzaShopMenu class that uses the factory to get pizzas for their current description and price. We might also have a HomeDelivery class that handles pizzas in a different way than our

there are no Dumb Questions

PizzaShop class but is also a client of the factory.

So, by encapsulating the pizza creating in one class, we now have only one place to make modifications when the implementation changes.

Don't forget, we are also just about to remove the concrete instantiations from our client code!

Q: I've seen a similar design where a factory like this is defined as a static method. What is the difference?

A: Defining a simple factory as a static method is a common technique and is often called a static factory. Why use a static method? Because you don't need to instantiate an object to make use of the create method. But remember it also has the disadvantage that you can't subclass and change the behavior of the create method.

Reworking the PizzaStore class

Now it's time to fix up our client code. What we want to do is rely on the factory to create the pizzas for us. Here are the changes:

Now we give PizzaStore a reference to a SimplePizzaFactory.

```
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.cut();  
        pizza.box();  
        return pizza;  
    }  
  
    // other methods here  
}
```

PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we've replaced the new operator with a create method on the factory object. No more concrete instantiations here!



BRAIN POWER

We know that object composition allows us to change behavior dynamically at runtime (among other things) because we can swap in and out implementations. How might we be able to use that in the PizzaStore? What factory implementations might we be able to swap in and out?

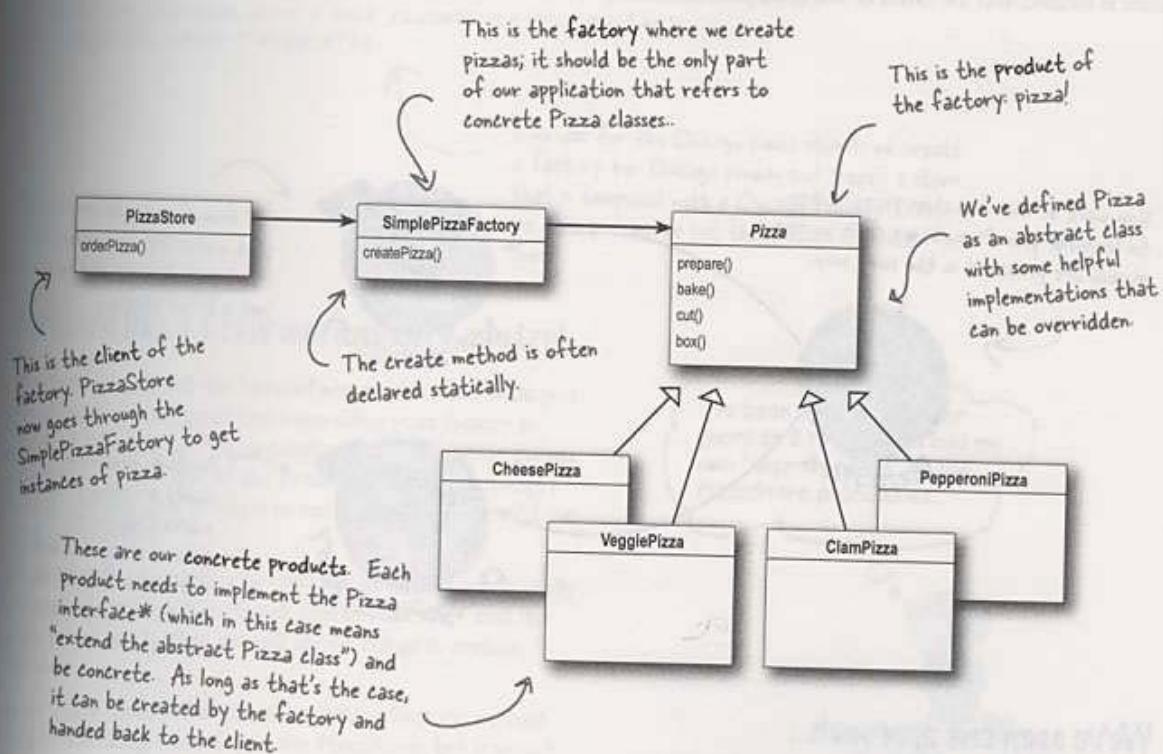
We don't know about you, but we're thinking New York, Chicago, and California style pizza factories (let's not forget New Haven, too!)

The Simple Factory defined

The Simple Factory isn't actually a Design Pattern; it's more of a programming idiom. But it is commonly used, so we'll give it a Head First Pattern Honorable Mention. Some developers do mistake this idiom for the "Factory Pattern," so the next time there is an awkward silence between you and another developer, you've got a nice topic to break the ice.



Just because Simple Factory isn't a REAL pattern doesn't mean we shouldn't check out how it's put together. Let's take a look at the class diagram of our new Pizza Store:



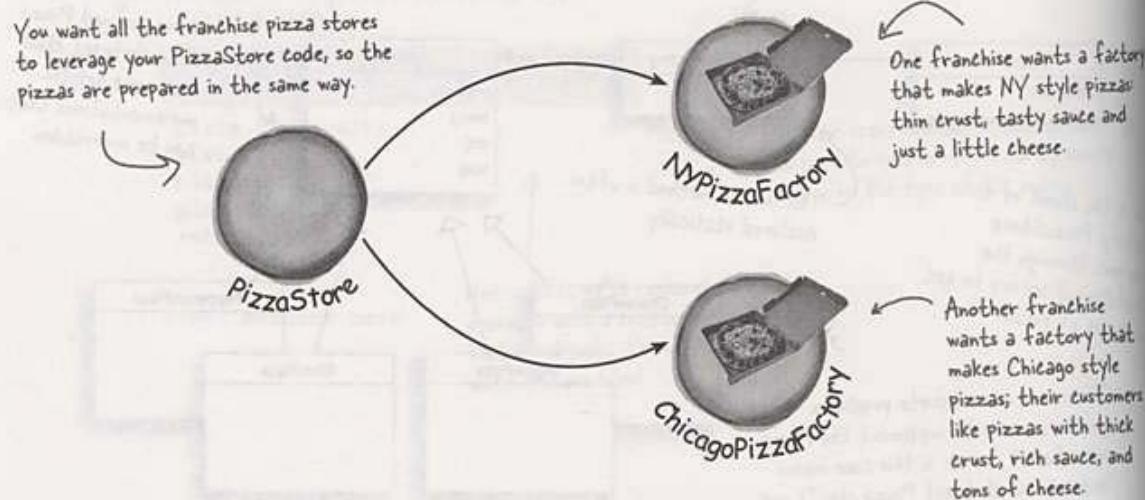
Think of Simple Factory as a warm up. Next, we'll explore two heavy duty patterns that are both factories. But don't worry, there's more pizza to come!

*Just another reminder: in design patterns, the phrase "implement an interface" does NOT always mean "write a class that implements a Java interface, by using the 'implements' keyword in the class declaration." In the general use of the phrase, a concrete class implementing a method from a supertype (which could be a class OR interface) is still considered to be "implementing the interface" of that supertype.

Franchising the pizza store

Your Objectville PizzaStore has done so well that you've trounced the competition and now everyone wants a PizzaStore in their own neighborhood. As the franchiser, you want to ensure the quality of the franchise operations and so you want them to use your time-tested code.

But what about regional differences? Each franchise might want to offer different styles of pizzas (New York, Chicago, and California, to name a few), depending on where the franchise store is located and the tastes of the local pizza connoisseurs.



We've seen one approach...

If we take out SimplePizzaFactory and create three different factories, NYPizzaFactory, ChicagoPizzaFactory and CaliforniaPizzaFactory, then we can just compose the PizzaStore with the appropriate factory and a franchise is good to go. That's one approach.

Let's see what that would look like...

```
NYPizzaFactory nyFactory = new NYPizzaFactory();  
PizzaStore nyStore = new PizzaStore(nyFactory);  
nyStore.order("Veggie");
```

Here we create a factory
for making NY style pizzas.

Then we create a PizzaStore and pass it
a reference to the NY factory.

...and when we make pizzas, we
get NY-style pizzas.

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();  
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);  
chicagoStore.order("Veggie");
```

Likewise for the Chicago pizza stores: we create
a factory for Chicago pizzas and create a store
that is composed with a Chicago factory. When
we make pizzas, we get the Chicago flavored
ones

But you'd like a little more quality control...

So you test marketed the SimpleFactory idea, and what you found was that the franchises were using your factory to create pizzas, but starting to employ their own home grown procedures for the rest of the process: they'd bake things a little differently, they'd forget to cut the pizza and they'd use third-party boxes.

Rethinking the problem a bit, you see that what you'd really like to do is create a framework that ties the store and the pizza creation together, yet still allows things to remain flexible.

In our early code, before the SimplePizzaFactory, we had the pizza-making code tied to the PizzaStore, but it wasn't flexible. So, how can we have our pizza and eat it too?

I've been making pizza for
years so I thought I'd add my
own "improvements" to the
PizzaStore procedures...

Not what you want in a good
franchise. You do NOT want to know
what he puts on his pizzas.



let the subclasses decide

A framework for the pizza store

There is a way to localize all the pizza making activities to the PizzaStore class, and yet give the franchises freedom to have their own regional style.

What we're going to do is put the `createPizza()` method back into PizzaStore, but this time as an **abstract method**, and then create a PizzaStore subclass for each regional style.

First, let's look at the changes to the PizzaStore:

```
PizzaStore is now abstract (see why below).  
↓  
public abstract class PizzaStore {  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza;  
        pizza = createPizza(type);  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
  
        return pizza;  
    }  
  
    abstract createPizza(String type);  
}
```

Now `createPizza` is back to being a call to a method in the `PizzaStore` rather than on a factory object.

All this looks just the same...

Now we've moved our factory object to this method.

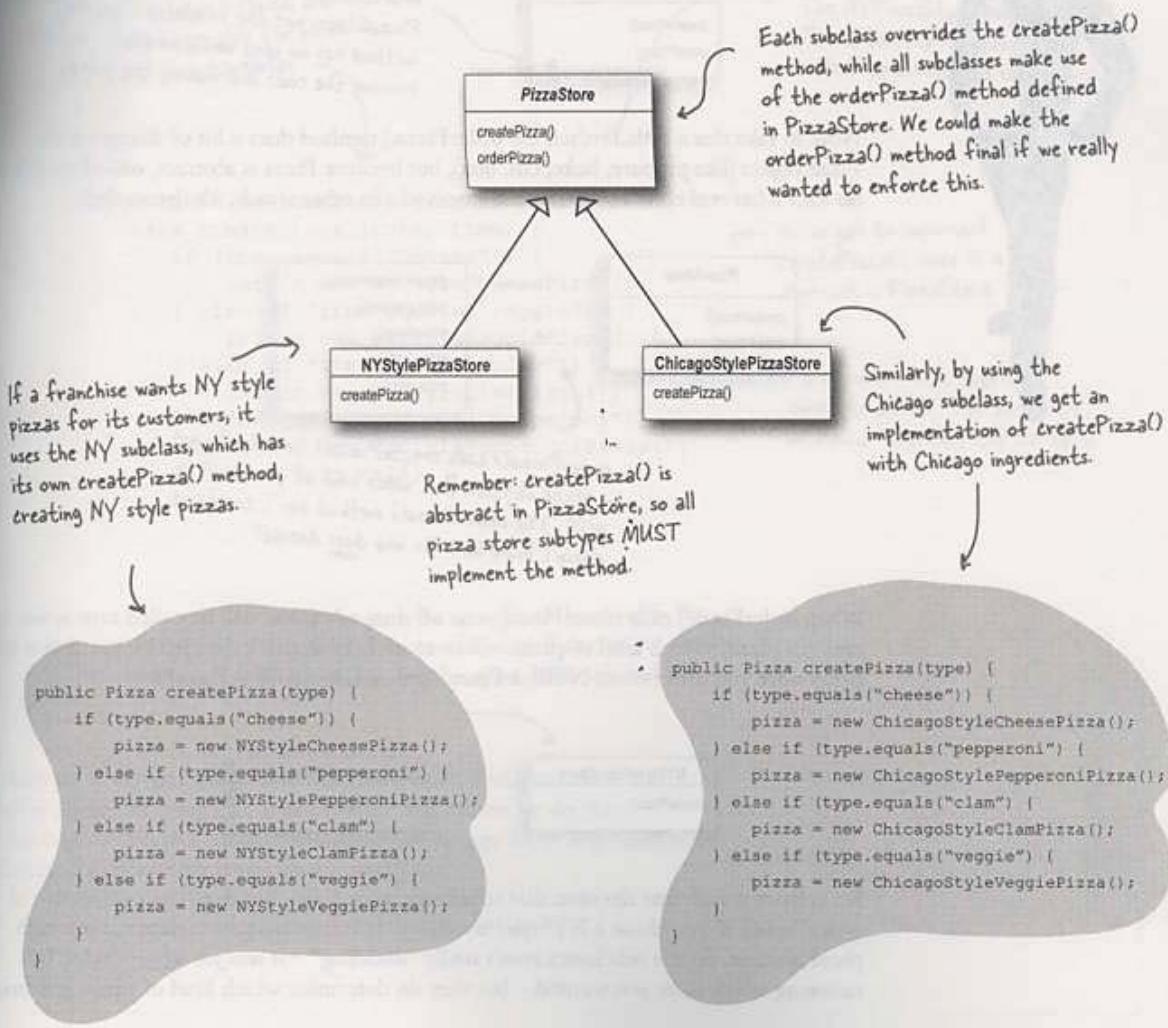
Our "factory method" is now abstract in `PizzaStore`.

Now we've got a store waiting for subclasses; we're going to have a subclass for each regional type (`NYPizzaStore`, `ChicagoPizzaStore`, `CaliforniaPizzaStore`) and each subclass is going to make the decision about what makes up a pizza. Let's take a look at how this is going to work.

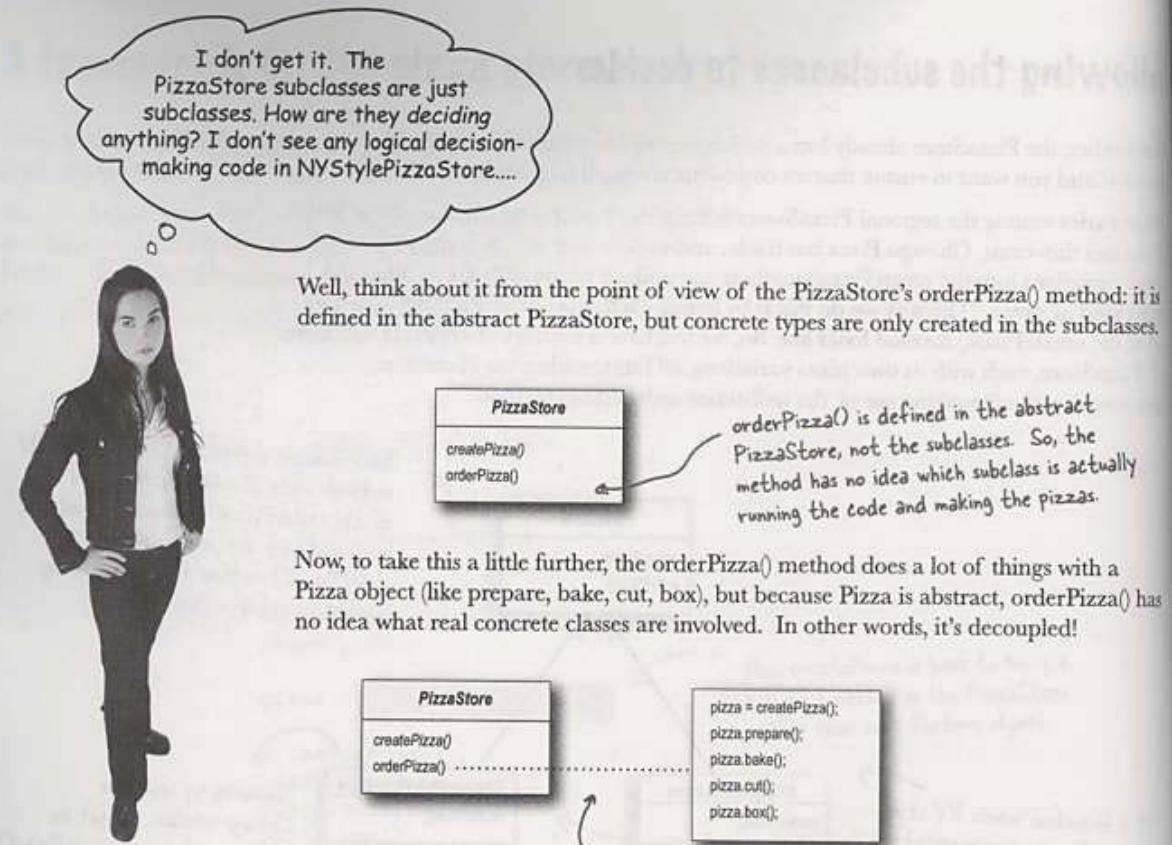
Allowing the subclasses to decide

Remember, the PizzaStore already has a well-honed order system in the `orderPizza()` method and you want to ensure that it's consistent across all franchises.

What varies among the regional PizzaStores is the style of pizzas they make – New York Pizza has thin crust, Chicago Pizza has thick, and so on – and we are going to push all these variations into the `createPizza()` method and make it responsible for creating the right kind of pizza. The way we do this is by letting each subclass of `PizzaStore` define what the `createPizza()` method looks like. So, we will have a number of concrete subclasses of `PizzaStore`, each with its own pizza variations, all fitting within the `PizzaStore` framework and still making use of the well-tuned `orderPizza()` method.



how do subclasses decide?



Now, to take this a little further, the orderPizza() method does a lot of things with a Pizza object (like prepare, bake, cut, box), but because Pizza is abstract, orderPizza() has no idea what real concrete classes are involved. In other words, it's decoupled!

When orderPizza() calls createPizza(), one of your subclasses will be called into action to create a pizza. Which kind of pizza will be made? Well, that's decided by the choice of pizza store you order from, NYStylePizzaStore or ChicagoStylePizzaStore.

So, is there a real-time decision that subclasses make? No, but from the perspective of orderPizza(), if you chose a NYStylePizzaStore, that subclass gets to determine which pizza is made. So the subclasses aren't really "deciding" – it was *you* who decided by choosing which store you wanted – but they do determine which kind of pizza gets made.

Let's make a PizzaStore

Being a franchise has its benefits. You get all the PizzaStore functionality for free. All the regional stores need to do is subclass PizzaStore and supply a createPizza() method that implements their style of Pizza. We'll take care of the big three pizza styles for the franchisees.

Here's the New York regional style:

createPizza() returns a Pizza, and the subclass is fully responsible for which concrete Pizza it instantiates

The NYPizzaStore extends PizzaStore, so it inherits the orderPizza() method (among others).

```
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;
    }
}
```

← We've got to implement createPizza(), since it is abstract in PizzaStore.

← Here's where we create our concrete classes. For each type of Pizza we create the NY style.

* Note that the orderPizza() method in the superclass has no clue which Pizza we are creating; it just knows it can prepare, bake, cut, and box it!

Once we've got our PizzaStore subclasses built, it will be time to see about ordering up a pizza or two. But before we do that, why don't you take a crack at building the Chicago Style and California Style pizza stores on the next page.

Sharpen your pencil

We've knocked out the NYPizzaStore, just two more to go and we'll be ready to franchise! Write the Chicago and California PizzaStore implementations here:

Declaring a factory method

With just a couple of transformations to the PizzaStore we've gone from having an object handle the instantiation of our concrete classes to a set of subclasses that are now taking on that responsibility. Let's take a closer look:

```
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    protected abstract Pizza createPizza(String type);
    // other methods here
}
```

The subclasses of PizzaStore handle object instantiation for us in the createPizza() method.

NYStylePizzaStore
createPizza()

ChicagoStylePizzaStore
createPizza()

All the responsibility for instantiating Pizzas has been moved into a method that acts as a factory.

Code Up Close

A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

A factory method may be parameterized (or not) to select among several variations of a product

abstract Product factoryMethod(String type)

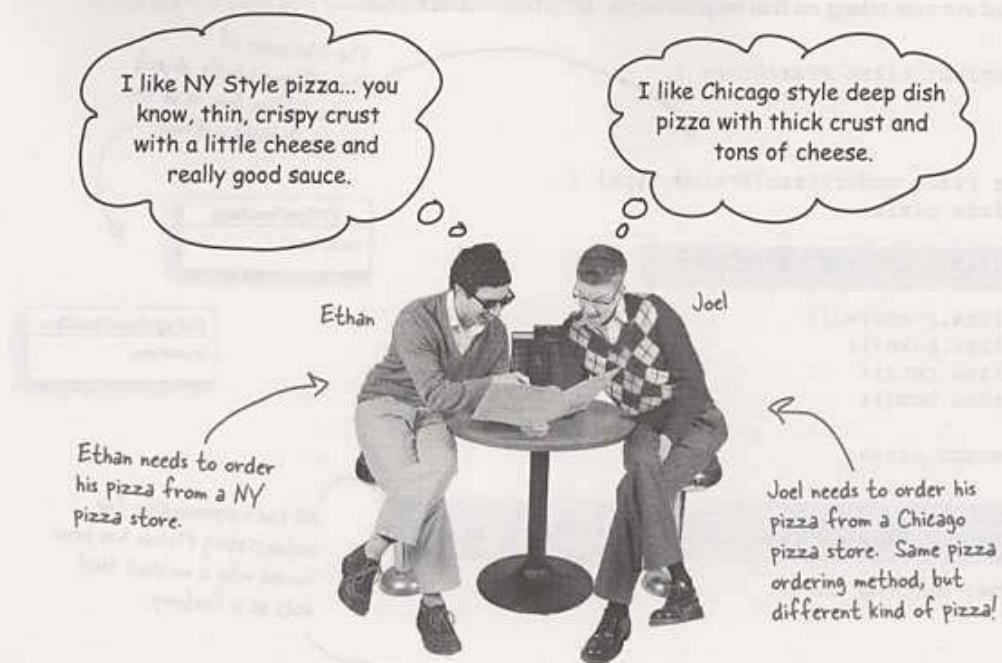
A factory method is abstract so the subclasses are counted on to handle object creation.

A factory method returns a Product that is typically used within methods defined in the superclass.

A factory method isolates the client (the code in the superclass, like orderPizza()) from knowing what kind of concrete Product is actually created.

ordering a pizza

Let's see how it works: ordering pizzas with the pizza factory method



So how do they order?

- 1 First, Joel and Ethan need an instance of a `PizzaStore`. Joel needs to instantiate a `ChicagoPizzaStore` and Ethan needs a `NYPizzaStore`.
- 2 With a `PizzaStore` in hand, both Ethan and Joel call the `orderPizza()` method and pass in the type of pizza they want (cheese, veggie, and so on).
- 3 To create the pizzas, the `createPizza()` method is called, which is defined in the two subclasses `NYPizzaStore` and `ChicagoPizzaStore`. As we defined them, the `NYPizzaStore` instantiates a NY style pizza, and the `ChicagoPizzaStore` instantiates Chicago style pizza. In either case, the `Pizza` is returned to the `orderPizza()` method.
- 4 The `orderPizza()` method has no idea what kind of pizza was created, but it knows it is a pizza and it prepares, bakes, cuts, and boxes it for Ethan and Joel.

Let's check out how these pizzas are really made to order...

Behind the Scenes



1 Let's follow Ethan's order: first we need a NY PizzaStore:

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

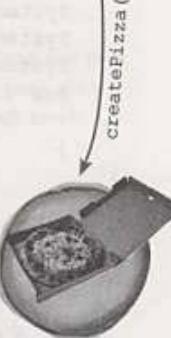
Creates a instance of NYPizzaStore.



2 Now that we have a store, we can take an order:

```
nyPizzaStore.orderPizza("cheese");
```

The orderPizza() method is called on the nyPizzaStore instance (the method defined inside PizzaStore runs).



3 The orderPizza() method then calls the createPizza() method:

```
Pizza pizza = createPizza("cheese");
```

Remember, createPizza(), the factory method, is implemented in the subclass. In this case it returns a NY Cheese Pizza.

Pizza

4 Finally we have the unprepared pizza in hand and the orderPizza() method finishes preparing it:

```
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
```

The orderPizza() method gets back a Pizza, without knowing exactly what concrete class it is.

All of these methods are defined in the specific pizza returned from the factory method createPizza(), defined in the NYPizzaStore.

We're just missing one thing: PIZZA!

Our PizzaStore isn't going to be very popular without some pizzas, so let's implement them:



We'll start with an abstract
Pizza class and all the concrete
pizzas will derive from this.

```
public abstract class Pizza {  
    String name;  
    String dough;  
    String sauce;  
    ArrayList toppings = new ArrayList();  
  
    void prepare() {  
        System.out.println("Preparing " + name);  
        System.out.println("Tossing dough...");  
        System.out.println("Adding sauce...");  
        System.out.println("Adding toppings: ");  
        for (int i = 0; i < toppings.size(); i++) {  
            System.out.println(" " + toppings.get(i));  
        }  
    }  
  
    void bake() {  
        System.out.println("Bake for 25 minutes at 350");  
    }  
  
    void cut() {  
        System.out.println("Cutting the pizza into diagonal slices");  
    }  
  
    void box() {  
        System.out.println("Place pizza in official PizzaStore box");  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

Each Pizza has a name, a type of dough, a
type of sauce, and a set of toppings.

The abstract class provides
some basic defaults for baking,
cutting and boxing.

Preparation follows a
number of steps in a
particular sequence.

REMEMBER: we don't provide import and package statements in the
code listings. Get the complete source code from the wickedlysmart
web site. You'll find the URL on page xxxiii in the Intro.

Now we just need some concrete subclasses... how about defining New York and Chicago style cheese pizzas?

```
public class NYStyleCheesePizza extends Pizza {  
    public NYStyleCheesePizza() {  
        name = "NY Style Sauce and Cheese Pizza";  
        dough = "Thin Crust Dough";  
        sauce = "Marinara Sauce";  
  
        toppings.add("Grated Reggiano Cheese");  
    }  
}
```

The NY Pizza has its own marinara style sauce and thin crust.

And one topping, reggiano cheese!

```
public class ChicagoStyleCheesePizza extends Pizza {  
    public ChicagoStyleCheesePizza() {  
        name = "Chicago Style Deep Dish Cheese Pizza";  
        dough = "Extra Thick Crust Dough";  
        sauce = "Plum Tomato Sauce";  
  
        toppings.add("Shredded Mozzarella Cheese");  
    }  
  
    void cut() {  
        System.out.println("Cutting the pizza into square slices");  
    }  
}
```

The Chicago Pizza uses plum tomatoes as a sauce along with extra thick crust.

The Chicago style deep dish pizza has lots of mozzarella cheese!

The Chicago style pizza also overrides the cut() method so that the pieces are cut into squares.

make some pizzas

You've waited long enough, time for some pizzas!

```
public class PizzaTestDrive {  
    public static void main(String[] args) {  
        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");  
    }  
}
```

First we create two different stores.

Then use one store to make Ethan's order.

And the other for Joel's.

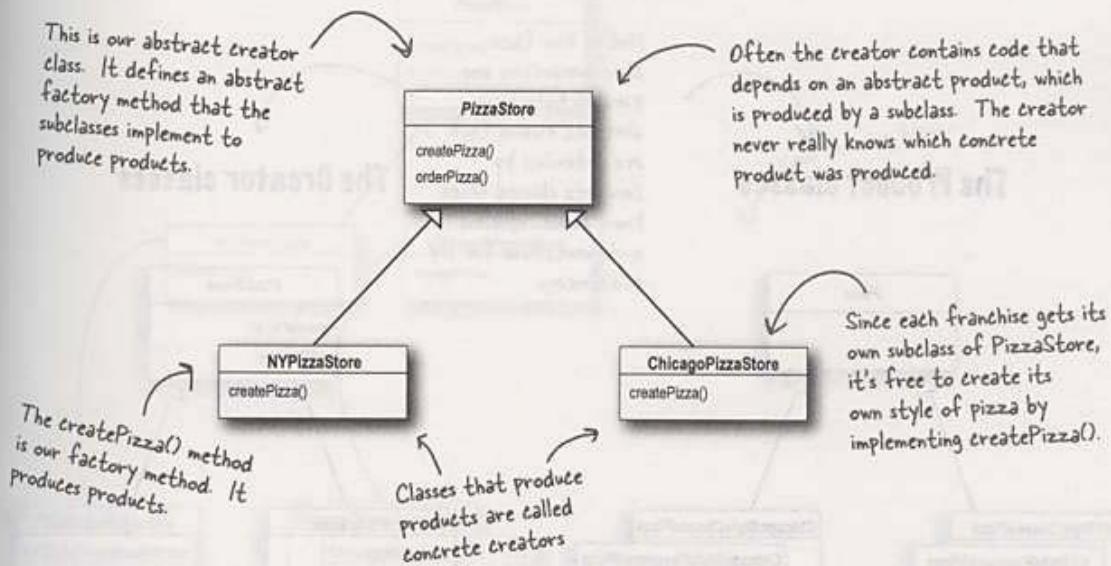
```
File Edit Window Help YouWantMootzOnThatPizza?  
%java PizzaTestDrive  
  
Preparing NY Style Sauce and Cheese Pizza  
Tossing dough...  
Adding sauce...  
Adding toppings:  
    Grated Regiano cheese  
Bake for 25 minutes at 350  
Cutting the pizza into diagonal slices  
Place pizza in official PizzaStore box  
Ethan ordered a NY Style Sauce and Cheese Pizza  
  
Preparing Chicago Style Deep Dish Cheese Pizza  
Tossing dough...  
Adding sauce...  
Adding toppings:  
    Shredded Mozzarella Cheese  
Bake for 25 minutes at 350  
Cutting the pizza into square slices  
Place pizza in official PizzaStore box  
Joel ordered a Chicago Style Deep Dish Cheese Pizza
```

Both pizzas get prepared, the toppings added, and the pizzas baked, cut and boxed. Our superclass never had to know the details; the subclass handled all that just by instantiating the right pizza.

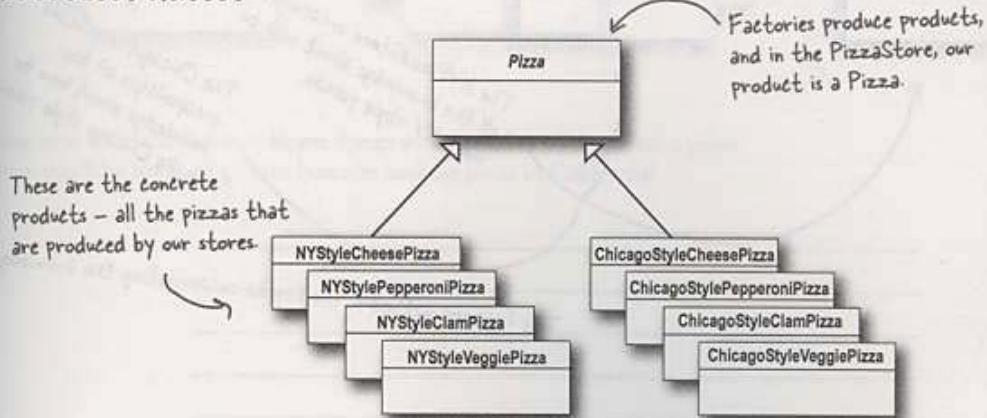
It's finally time to meet the Factory Method Pattern

All factory patterns encapsulate object creation. The Factory Method Pattern encapsulates object creation by letting subclasses decide what objects to create. Let's check out these class diagrams to see who the players are in this pattern:

The Creator classes



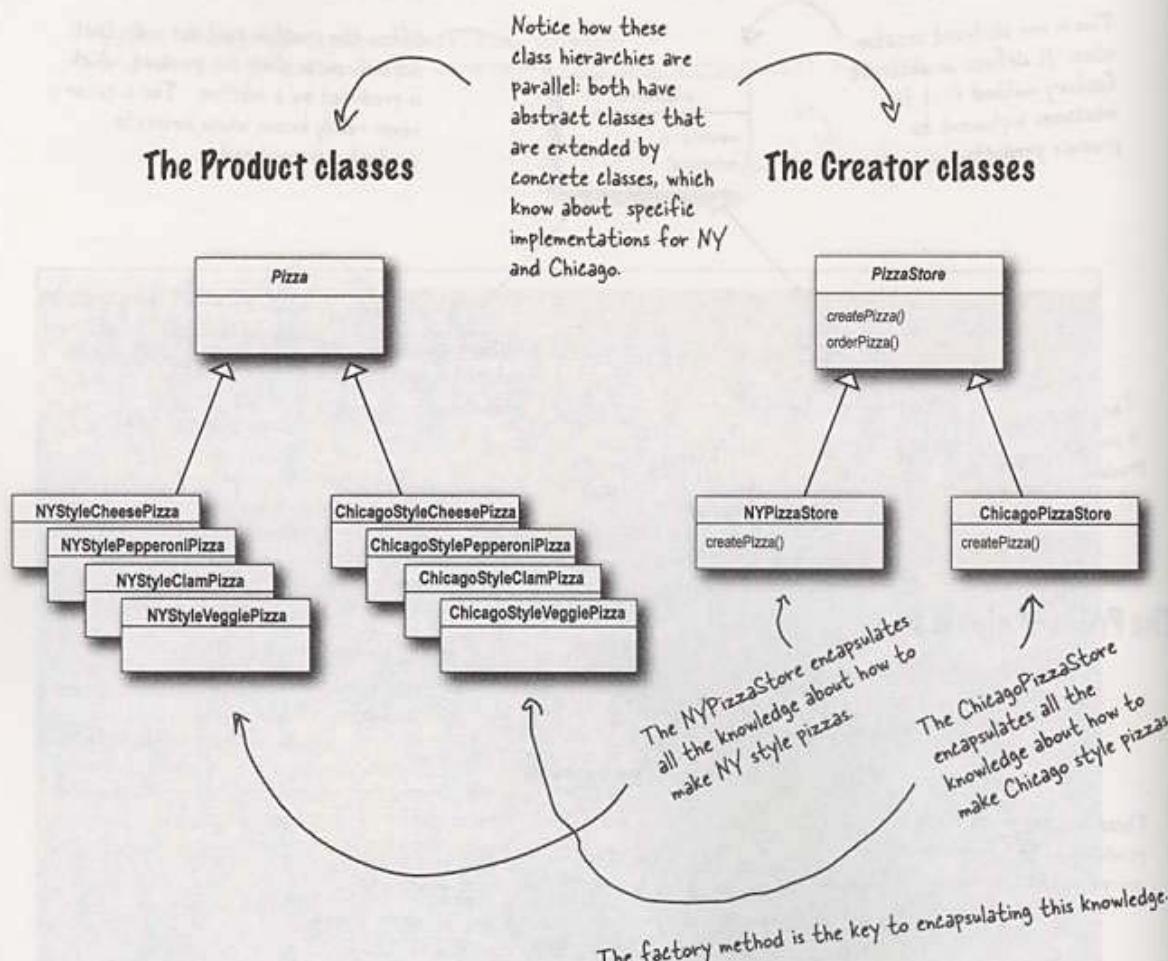
The Product classes



Another perspective: parallel class hierarchies

We've seen that the factory method provides a framework by supplying an `orderPizza()` method that is combined with a factory method. Another way to look at this pattern as a framework is in the way it encapsulates product knowledge into each creator.

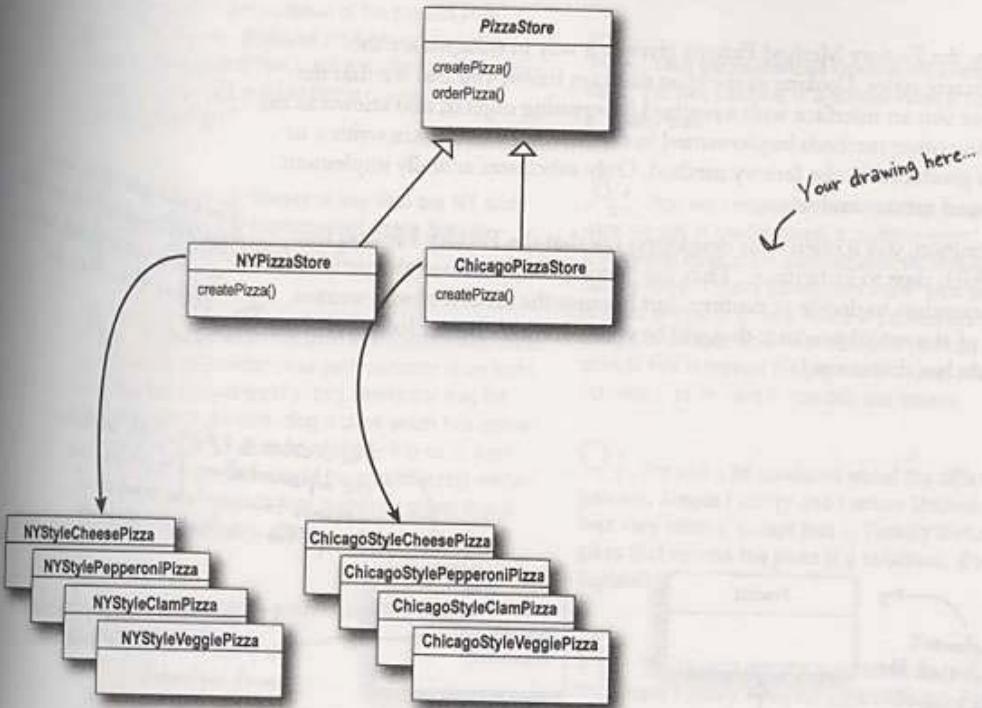
Let's look at the two parallel class hierarchies and see how they relate:





Design Puzzle

We need another kind of pizza for those crazy Californians (crazy in a *good* way of course). Draw another parallel set of classes that you'd need to add a new California region to our PizzaStore.



Okay, now write the five *most bizarre* things you can think of to put on a pizza. Then, you'll be ready to go into business making pizza in California!

Factory Method Pattern defined

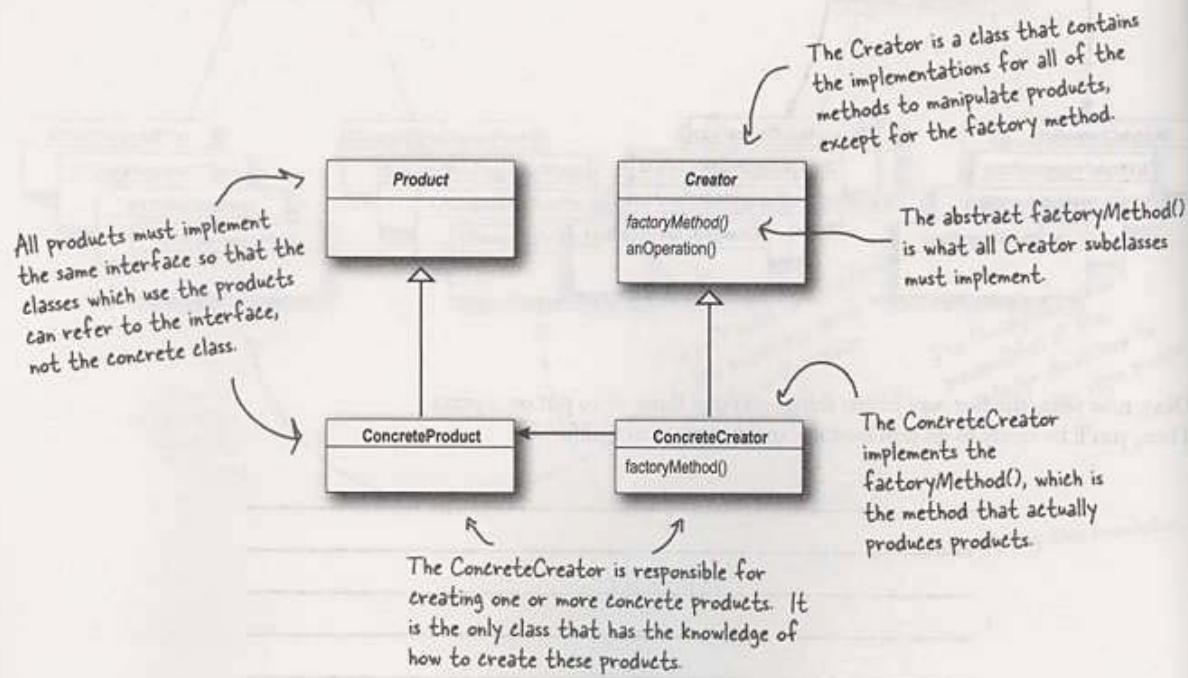
It's time to roll out the official definition of the Factory Method Pattern:

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 subclasses.

As with every factory, the Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types. Looking at the class diagram below, you can see that the abstract Creator gives you an interface with a method for creating objects, also known as the "factory method." Any other methods implemented in the abstract Creator are written to operate on products produced by the factory method. Only subclasses actually implement the factory method and create products.

As in the official definition, you'll often hear developers say that the Factory Method lets subclasses decide which class to instantiate. They say "decides" not because the pattern allows subclasses themselves to decide at runtime, but because the creator class is written without knowledge of the actual products that will be created, which is decided purely by the choice of the subclass that is used.

You could ask them what "decides" means, but we bet you now understand this better than they do!



there are no Dumb Questions

Q: What's the advantage of the Factory Method Pattern when you only have one `ConcreteCreator`?

A: The Factory Method Pattern is useful if you've only got one concrete creator because you are decoupling the implementation of the product from its use. If you add additional products or change a product's implementation, it will not affect your Creator (because the Creator is not tightly coupled to any `ConcreteProduct`).

Q: Would it be correct to say that our NY and Chicago stores are implemented using Simple Factory? They look just like it.

A: They're similar, but used in different ways. Even though the implementation of each concrete store looks a lot like the `SimplePizzaFactory`, remember that the concrete stores are extending a class which has defined `createPizza()` as an abstract method. It is up to each store to define the behavior of the `createPizza()` method. In Simple Factory, the factory is another object that is composed with the `PizzaStore`.

Q: Are the factory method and the Creator always abstract?

A: No, you can define a default factory method to produce some concrete product. Then you always have a means of creating products even if there are no subclasses of the Creator.

Q: Each store can make four different kinds of pizzas based on the type passed in. Do all concrete creators make multiple products, or do they sometimes just make one?

A: We implemented what is known as the parameterized factory method. It can make more than one object based on a parameter passed in, as you noticed. Often, however, a factory just produces one object and is not parameterized. Both are valid forms of the pattern.

Q: Your parameterized types don't seem "type-safe." I'm just passing in a `String`! What if I asked for a "CalmPizza"?

A: You are certainly correct and that would cause, what we call in the business, a "runtime error." There are several other more sophisticated techniques that can be used to make parameters more "type safe", or, in other words, to ensure errors in parameters can be caught at compile time. For instance, you can create objects that represent the parameter types, use static constants, or, in Java 5, you can use `enums`.

Q: I'm still a bit confused about the difference between Simple Factory and Factory Method. They look very similar, except that in Factory Method, the class that returns the pizza is a subclass. Can you explain?

A: You're right that the subclasses do look a lot like Simple Factory, however think of Simple Factory as a one shot deal, while with Factory Method you are creating a framework that let's the subclasses decide which implementation will be used. For example, the `orderPizza()` method in the Factory Method provides a general framework for creating pizzas that relies on a factory method to actually create the concrete classes that go into making a pizza. By subclassing the `PizzaStore` class, you decide what concrete products go into making the pizza that `orderPizza()` returns. Compare that with SimpleFactory, which gives you a way to encapsulate object creation, but doesn't give you the flexibility of the Factory Method because there is no way to vary the products you're creating.



Master and Student...

Master: Grasshopper, tell me how your training is going?

Student: Master, I have taken my study of "encapsulate what varies" further.

Master: Go on...

Student: I have learned that one can encapsulate the code that creates objects. When you have code that instantiates concrete classes, this is an area of frequent change. I've learned a technique called "factories" that allows you to encapsulate this behavior of instantiation.

Master: And these "factories," of what benefit are they?

Student: There are many. By placing all my creation code in one object or method, I avoid duplication in my code and provide one place to perform maintenance. That also means clients depend only upon interfaces rather than the concrete classes required to instantiate objects. As I have learned in my studies, this allows me to program to an interface, not an implementation, and that makes my code more flexible and extensible in the future.

Master: Yes Grasshopper, your OO instincts are growing. Do you have any questions for your master today?

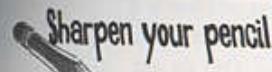
Student: Master, I know that by encapsulating object creation I am coding to abstractions and decoupling my client code from actual implementations. But my factory code must still use concrete classes to instantiate real objects. Am I not pulling the wool over my own eyes?

Master: Grasshopper, object creation is a reality of life; we must create objects or we will never create a single Java program. But, with knowledge of this reality, we can design our code so that we have corralled this creation code like the sheep whose wool you would pull over your eyes. Once corralled, we can protect and care for the creation code. If we let our creation code run wild, then we will never collect its "wool."

Student: Master, I see the truth in this.

Master: As I knew you would. Now, please go and meditate on object dependencies.

A very dependent PizzaStore



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:

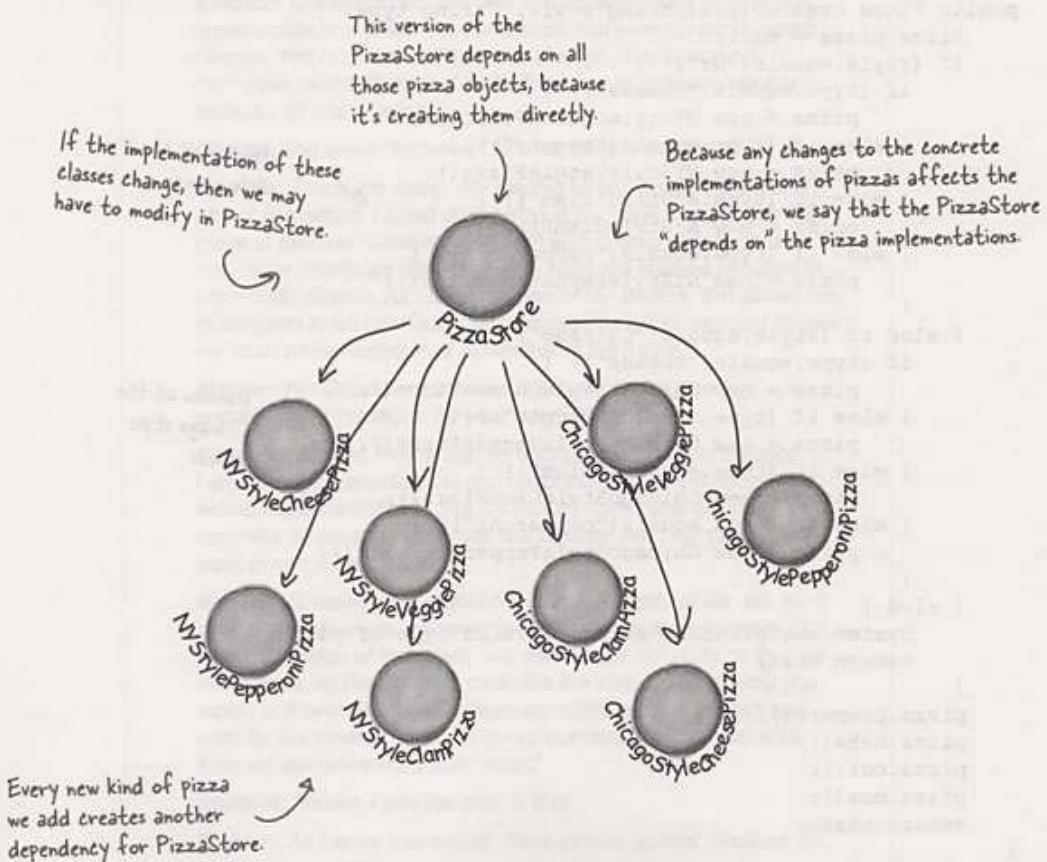
number

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*.

Here’s the general principle:



Design Principle

Depend upon abstractions. Do not depend upon concrete classes.

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.

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. 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...

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.

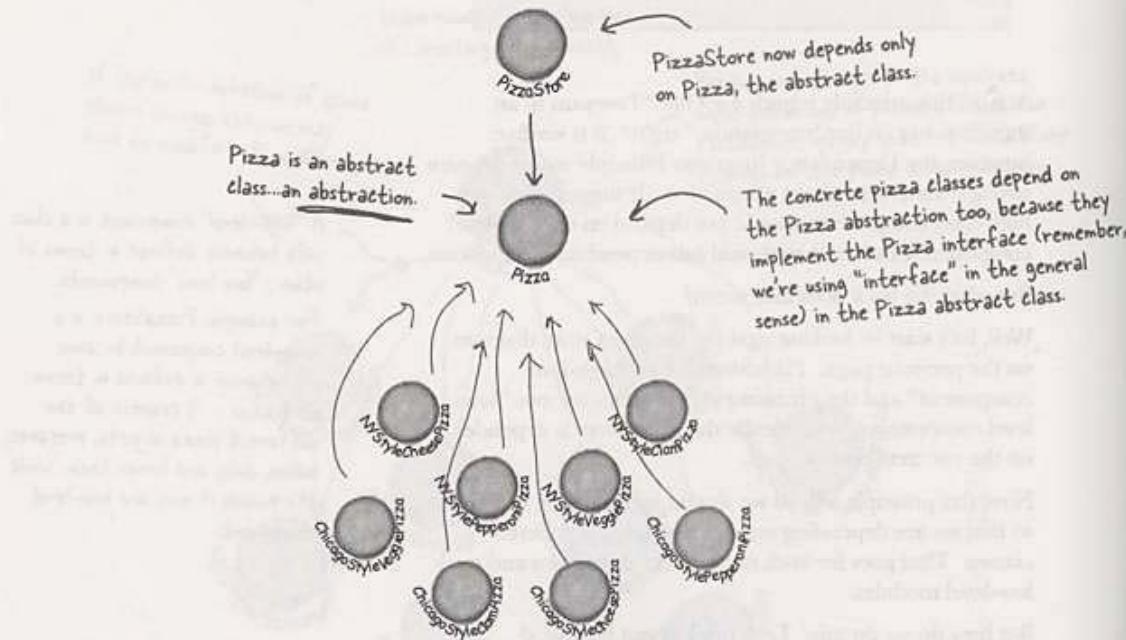
Applying the Principle

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.

Okay, I get the dependency part, but why is it called dependency **inversion**?

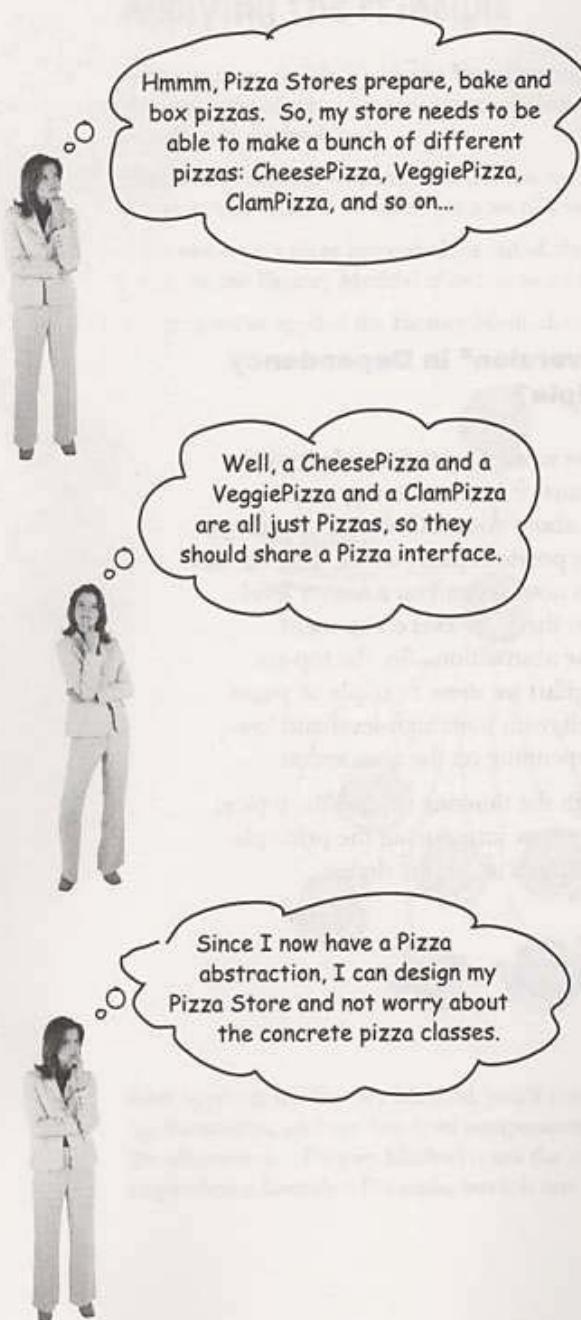
Where's the “inversion” in Dependency Inversion Principle?

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...

Invert your thinking

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.
- No class should derive from a concrete class.
- No method should override an implemented method of any of its base classes.

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

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.

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.

But wait, aren't these
guidelines impossible to follow?
If I follow these, I'll never be
able to write a single program!

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.



Meanwhile, back at the PizzaStore...

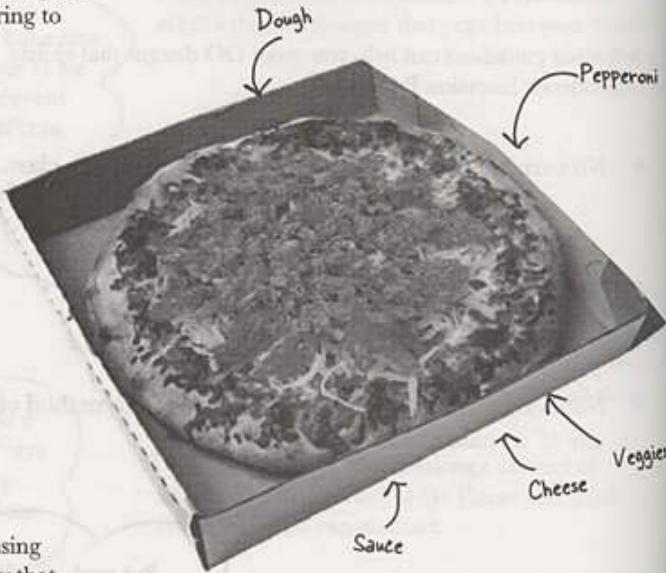
The design for the PizzaStore is really shaping up: it's got a flexible framework and it does a good job of adhering to design principles.

Now, the key to Objectville Pizza's success has always been fresh, quality ingredients, and what you've discovered is that with the new framework your franchises have been following your *procedures*, but a few franchises have been substituting inferior ingredients in their pies to lower costs and increase their margins. You know you've got to do something, because in the long term this is going to hurt the Objectville brand!

Ensuring consistency in your ingredients

So how are you going to ensure each franchise is using quality ingredients? You're going to build a factory that produces them and ships them to your franchises!

Now there is only one problem with this plan: the franchises are located in different regions and what is red sauce in New York is not red sauce in Chicago. So, you have one set of ingredients that need to be shipped to New York and a *different* set that needs to be shipped to Chicago. Let's take a closer look:



Chicago Pizza Menu

Cheese Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Oregano

Veggie Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives

Clam Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Clams

Pepperoni Pizza
Plum Tomato Sauce, Mozzarella, Parmesan, Eggplant, Spinach, Black Olives, Pepperoni

We've got the same product families (dough, sauce, cheese, veggies, meats) but different implementations based on region.

New York Pizza Menu

Cheese Pizza
Marinara Sauce, Reggiano, Garlic

Veggie Pizza
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers

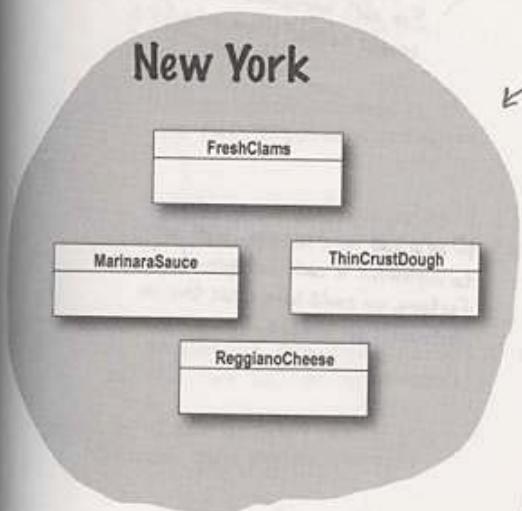
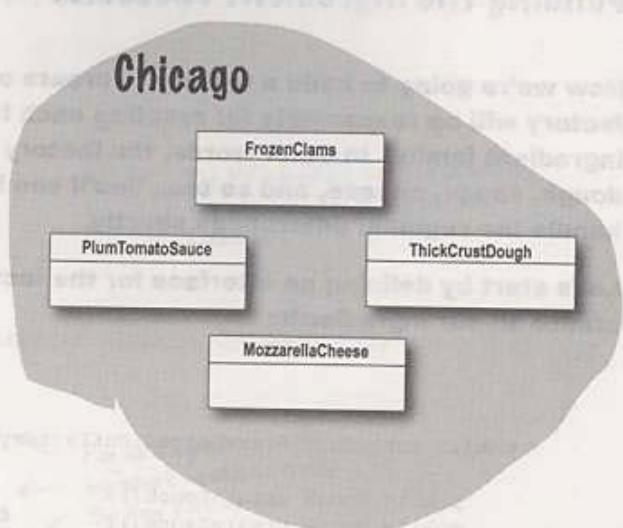
Clam Pizza
Marinara Sauce, Reggiano, Fresh Clams

Pepperoni Pizza
Marinara Sauce, Reggiano, Mushrooms, Onions, Red Peppers, Pepperoni

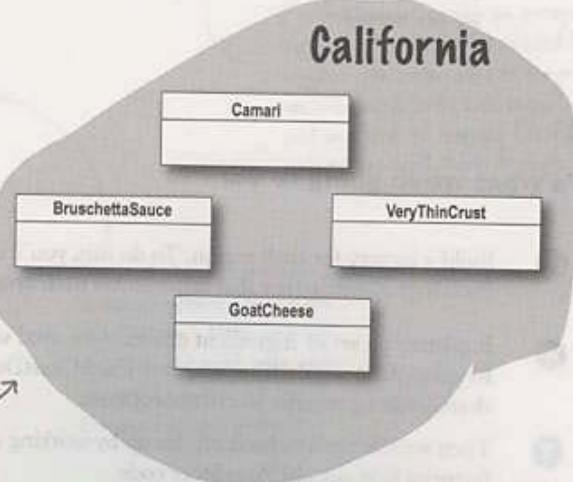
Families of ingredients...

New York uses one set of ingredients and Chicago another. Given the popularity of Objectville Pizza it won't be long before you also need to ship another set of regional ingredients to California, and what's next? Seattle?

For this to work, you are going to have to figure out how to handle families of ingredients.



All Objectville's Pizzas are made from the same components, but each region has a different implementation of those components.



Each family consists of a type of dough, a type of sauce, a type of cheese, and a seafood topping (along with a few more we haven't shown, like veggies and spices).

In total, these three regions make up ingredient families, with each region implementing a complete family of ingredients.

Building the ingredient factories

Now we're going to build a factory to create our ingredients; the factory will be responsible for creating each ingredient in the ingredient family. In other words, the factory will need to create dough, sauce, cheese, and so on... You'll see how we are going to handle the regional differences shortly.

Let's start by defining an interface for the factory that is going to create all our ingredients:

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

Lots of new classes here,
one per ingredient

For each ingredient we define a
create method in our interface.

If we'd had some common "machinery"
to implement in each instance of
factory, we could have made this an
abstract class instead...

Here's what we're going to do:

- ➊ Build a factory for each region. To do this, you'll create a subclass of `PizzaIngredientFactory` that implements each create method
- ➋ Implement a set of ingredient classes to be used with the factory, like `ReggianoCheese`, `RedPeppers`, and `ThickCrustDough`. These classes can be shared among regions where appropriate.
- ➌ Then we still need to hook all this up by working our new ingredient factories into our old `PizzaStore` code.

Building the New York ingredient factory

Okay, here's the implementation for the New York ingredient factory. This factory specializes in Marinara sauce, Reggiano Cheese, Fresh Clams...

The NY ingredient factory implements the interface for all ingredient factories

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {  
    public Dough createDough() {  
        return new ThinCrustDough();  
    }  
  
    public Sauce createSauce() {  
        return new MarinaraSauce();  
    }  
  
    public Cheese createCheese() {  
        return new ReggianoCheese();  
    }  
  
    public Veggies[] createVeggies() {  
        Veggies veggies[] = { new Garlic(), new Onion(), new Mushroom(), new RedPepper() };  
        return veggies;  
    }  
  
    public Pepperoni createPepperoni() {  
        return new SlicedPepperoni();  
    }  
    public Clams createClam() {  
        return new FreshClams();  
    }  
}
```

For each ingredient in the ingredient family, we create the New York version.

For veggies, we return an array of Veggies. Here we've hardcoded the veggies. We could make this more sophisticated, but that doesn't really add anything to learning the factory pattern, so we'll keep it simple.

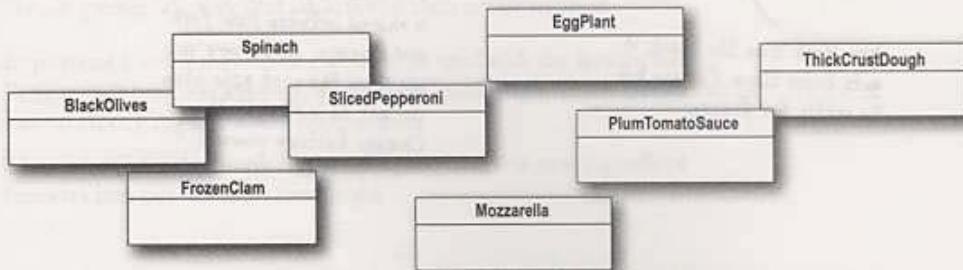
The best sliced pepperoni. This is shared between New York and Chicago. Make sure you use it on the next page when you get to implement the Chicago factory yourself

New York is on the coast; it gets fresh clams. Chicago has to settle for frozen.



Sharpen your pencil

Write the `ChicagoPizzaIngredientFactory`. You can reference the classes below in your implementation:



Reworking the pizzas...

We've got our factories all fired up and ready to produce quality ingredients; now we just need to rework our Pizzas so they only use factory-produced ingredients. We'll start with our abstract Pizza class:

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

    abstract void prepare();
    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

    void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }

    void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
        // code to print pizza here
    }
}
```

Each pizza holds a set of ingredients that are used in its preparation.

We've now made the `prepare` method abstract. This is where we are going to collect the ingredients needed for the pizza, which of course will come from the ingredient factory.

Our other methods remain the same, with the exception of the `prepare` method.

Reworking the pizzas, continued...

Now that you've got an abstract Pizza to work from, it's time to create the New York and Chicago style Pizzas – only this time around they will get their ingredients straight from the factory. The franchisees' days of skimping on ingredients are over!

When we wrote the Factory Method code, we had a NYCheesePizza and a ChicagoCheesePizza class. If you look at the two classes, the only thing that differs is the use of regional ingredients. The pizzas are made just the same (dough + sauce + cheese). The same goes for the other pizzas: Veggie, Clam, and so on. They all follow the same preparation steps; they just have different ingredients.

So, what you'll see is that we really don't need two classes for each pizza; the ingredient factory is going to handle the regional differences for us. Here's the Cheese Pizza:

```
public class CheesePizza extends Pizza {  
    PizzaIngredientFactory ingredientFactory;  
  
    public CheesePizza(PizzaIngredientFactory ingredientFactory)  
    {  
        this.ingredientFactory = ingredientFactory;  
    }  
  
    void prepare() {  
        System.out.println("Preparing " + name);  
        dough = ingredientFactory.createDough();  
        sauce = ingredientFactory.createSauce();  
        cheese = ingredientFactory.createCheese();  
    }  
}
```

To make a pizza now, we need a factory to provide the ingredients. So each Pizza class gets a factory passed into its constructor, and it's stored in an instance variable.

← Here's where the magic happens!

The prepare() method steps through creating a cheese pizza, and each time it needs an ingredient, it asks the factory to produce it.



Code Up Close

The Pizza code uses the factory it has been composed with to produce the ingredients used in the pizza. The ingredients produced depend on which factory we're using. The Pizza class doesn't care; it knows how to make pizzas. Now, it's decoupled from the differences in regional ingredients and can be easily reused when there are factories for the Rockies, the Pacific Northwest, and beyond.

```
sauce = ingredientFactory.createSauce();
```

We're setting the Pizza instance variable to refer to the specific sauce used in this pizza.

This is our ingredient factory. The Pizza doesn't care which factory is used, as long as it is an ingredient factory.

The createSauce() method returns the sauce that is used in its region. If this is a NY ingredient factory, then we get marinara sauce.

Let's check out the ClamPizza as well:

```
public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public ClamPizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
    }
}
```

ClamPizza also stashes an ingredient factory.

To make a clam pizza, the prepare method collects the right ingredients from its local factory.

If it's a New York factory, the clams will be fresh; if it's Chicago, they'll be frozen.

use the right ingredient factory

Revisiting our pizza stores

We're almost there; we just need to make a quick trip to our franchise stores to make sure they are using the correct Pizzas. We also need to give them a reference to their local ingredient factories:

```
public class NYPizzaStore extends PizzaStore {  
  
    protected Pizza createPizza(String item) {  
        Pizza pizza = null;  
        PizzaIngredientFactory ingredientFactory =  
            new NYPizzaIngredientFactory();  
  
        if (item.equals("cheese")) {  
            pizza = new CheesePizza(ingredientFactory);  
            pizza.setName("New York Style Cheese Pizza");  
        } else if (item.equals("veggie")) {  
            pizza = new VeggiePizza(ingredientFactory);  
            pizza.setName("New York Style Veggie Pizza");  
        } else if (item.equals("clam")) {  
            pizza = new ClamPizza(ingredientFactory);  
            pizza.setName("New York Style Clam Pizza");  
        } else if (item.equals("pepperoni")) {  
            pizza = new PepperoniPizza(ingredientFactory);  
            pizza.setName("New York Style Pepperoni Pizza");  
        }  
        return pizza;  
    }  
}
```

The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY style pizzas.

We now pass each pizza the factory that should be used to produce its ingredients.

Look back one page and make sure you understand how the pizza and the factory work together!

For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.



Compare this version of the `createPizza()` method to the one in the Factory Method implementation earlier in the chapter.

What have we done?

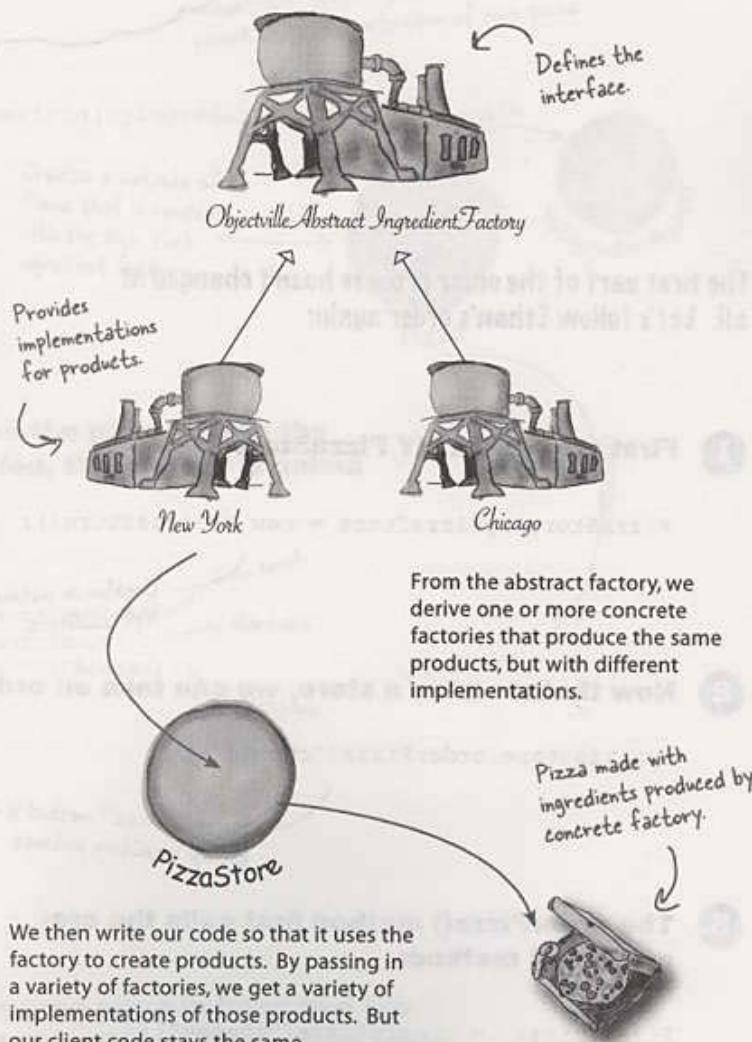
That was quite a series of code changes; what exactly did we do?

We provided a means of creating a family of ingredients for pizzas by introducing a new type of factory called an Abstract Factory.

An Abstract Factory gives us an interface for creating a family of products. By writing code that uses this interface, we decouple our code from the actual factory that creates the products. That allows us to implement a variety of factories that produce products meant for different contexts – such as different regions, different operating systems, or different look and feels.

Because our code is decoupled from the actual products, we can substitute different factories to get different behaviors (like getting marinara instead of plum tomatoes).

An Abstract Factory provides an interface for a family of products. What's a family? In our case it's all the things we need to make a pizza: dough, sauce, cheese, meats and veggies.



order some more pizza

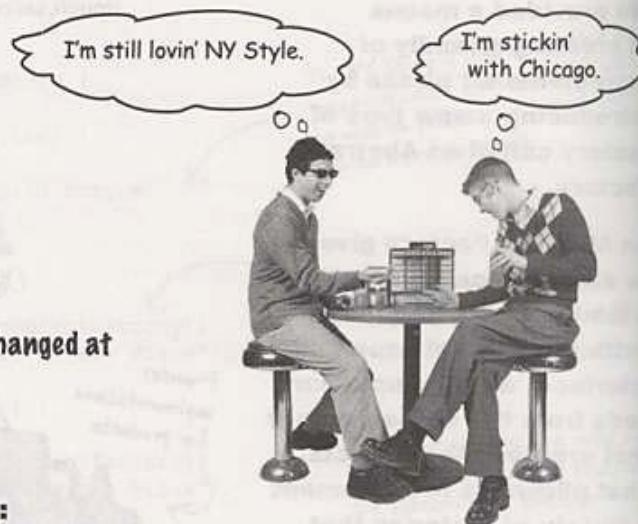
More pizza for Ethan and Joel...

Ethan and Joel can't get enough Objectville Pizza! What they don't know is that now their orders are making use of the new ingredient factories. So now when they order...

Behind
the Scenes



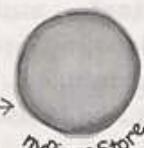
The first part of the order process hasn't changed at all. Let's follow Ethan's order again:



1 First we need a NY PizzaStore:

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

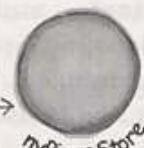
Creates an instance of
NYPizzaStore.



2 Now that we have a store, we can take an order:

```
nyPizzaStore.orderPizza("cheese");
```

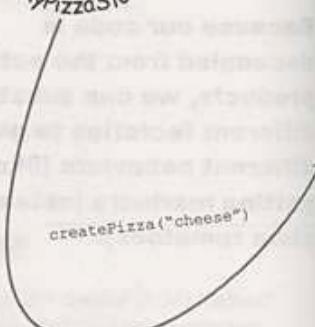
the orderPizza() method is called on
the nyPizzaStore instance.



3 The orderPizza() method first calls the createPizza() method:

```
Pizza pizza = createPizza("cheese");
```

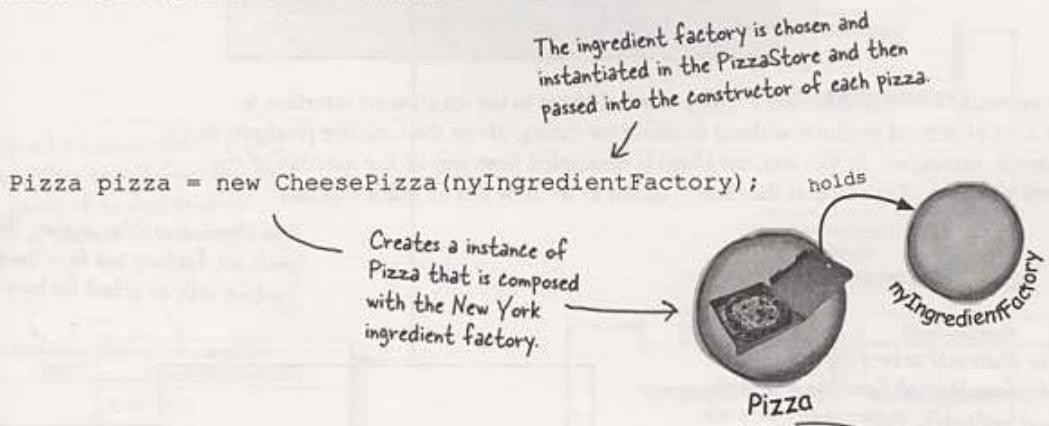
createPizza("cheese")



From here things change, because we are using an ingredient factory



- ④ When the `createPizza()` method is called, that's when our ingredient factory gets involved:



- ⑤ Next we need to prepare the pizza. Once the `prepare()` method is called, the factory is asked to prepare ingredients:

```

void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
  
```

For Ethan's pizza the New York ingredient factory is used, and so we get the NY ingredients.

- ⑥ Finally we have the prepared pizza in hand and the `orderPizza()` method bakes, cuts, and boxes the pizza.

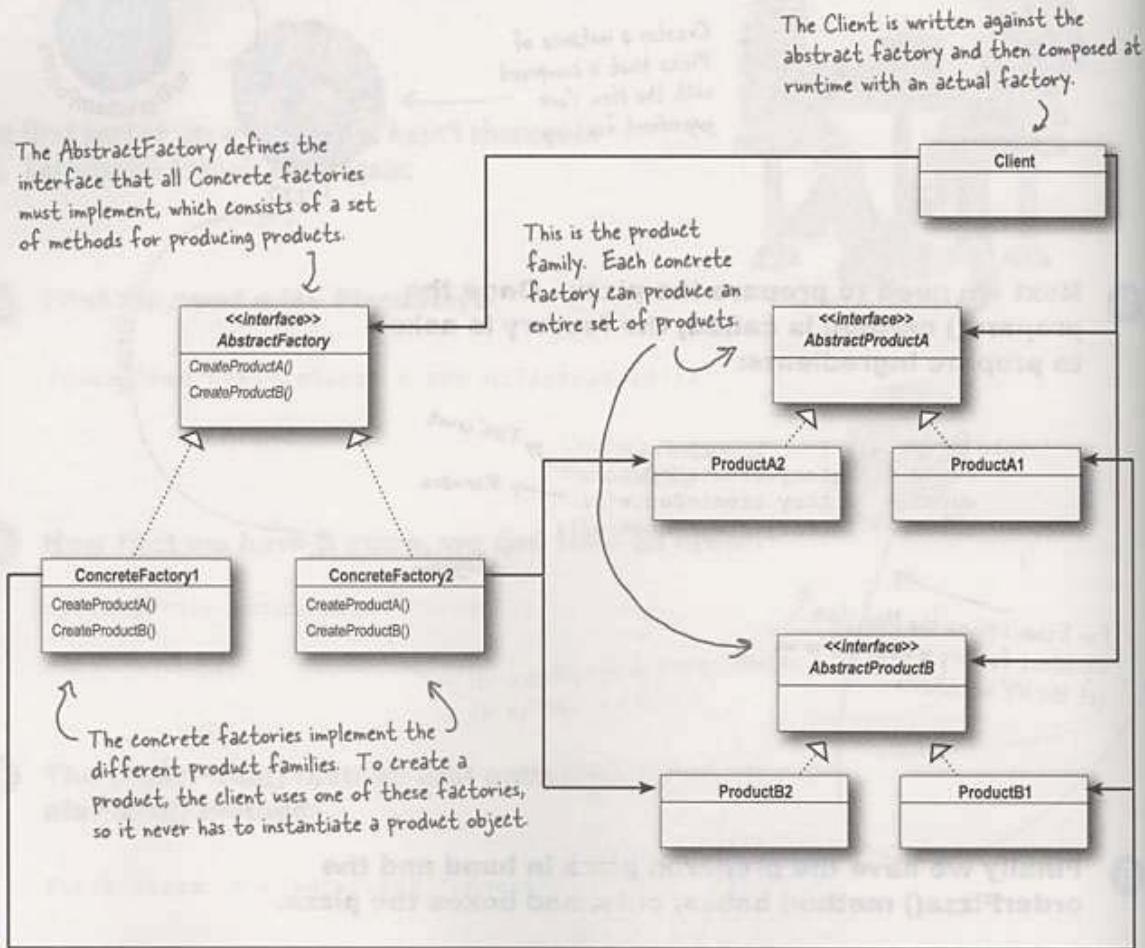
abstract factory defined

Abstract Factory Pattern defined

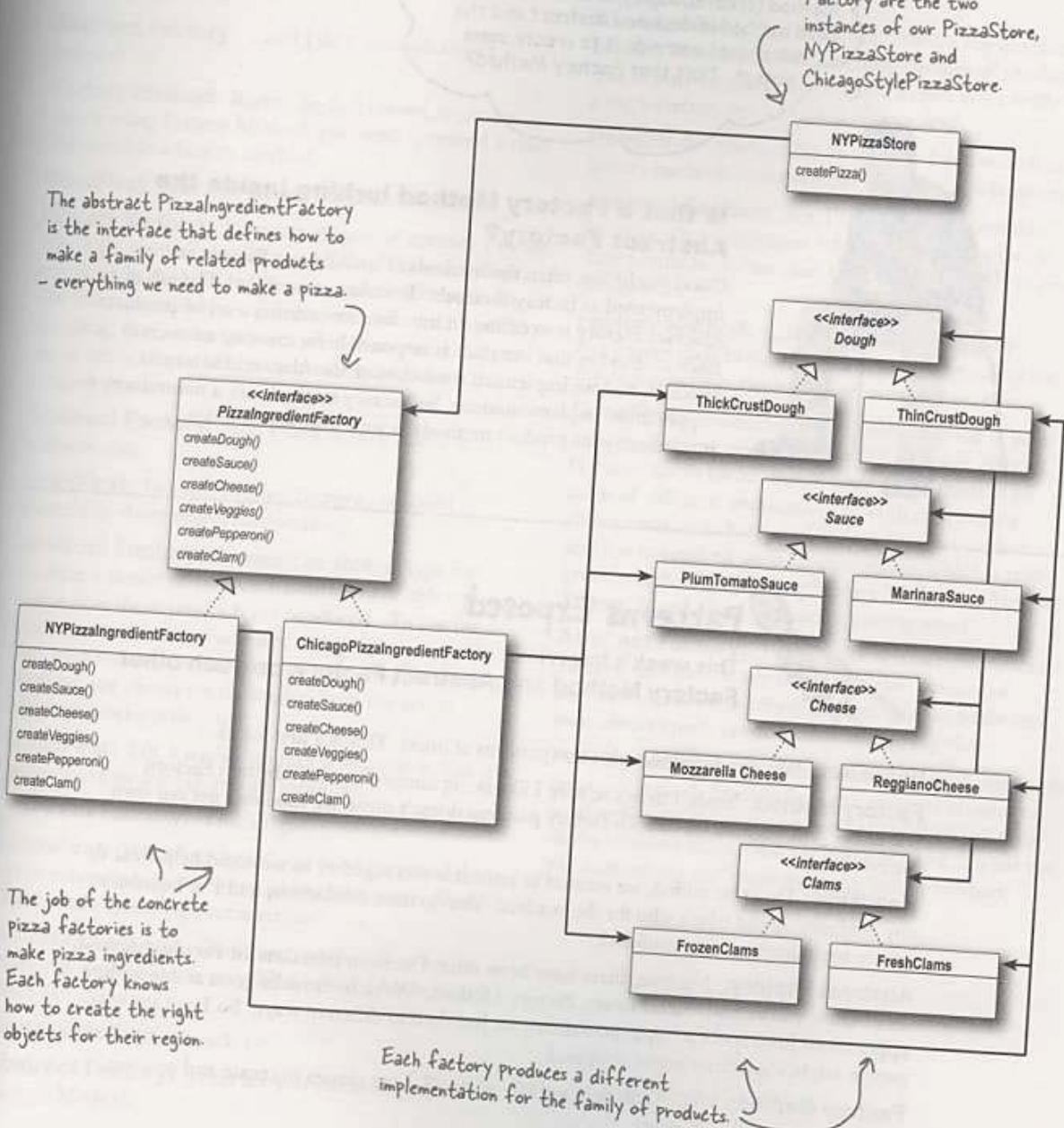
We're adding yet another factory pattern to our pattern family, one that lets us create families of products. Let's check out the official definition for this pattern:

The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

We've certainly seen that Abstract Factory allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced. In this way, the client is decoupled from any of the specifics of the concrete products. Let's look at the class diagram to see how this all holds together:



That's a fairly complicated class diagram; let's look at it all in terms of our PizzaStore:



interview with factory patterns



I noticed that each method in the Abstract Factory actually looks like a Factory Method (`createDough()`, `createSauce()`, etc.). Each method is declared abstract and the subclasses override it to create some object. Isn't that Factory Method?

Is that a Factory Method lurking inside the Abstract Factory?

Good catch! Yes, often the methods of an Abstract Factory are implemented as factory methods. It makes sense, right? The job of an Abstract Factory is to define an interface for creating a set of products. Each method in that interface is responsible for creating a concrete product, and we implement a subclass of the Abstract Factory to supply those implementations. So, factory methods are a natural way to implement your product methods in your abstract factories.



Patterns Exposed

This week's interview:

Factory Method and Abstract Factory, on each other

HeadFirst: Wow, an interview with two patterns at once! This is a first for us.

Factory Method: Yeah, I'm not so sure I like being lumped in with Abstract Factory, you know. Just because we're both factory patterns doesn't mean we shouldn't get our own interviews.

HeadFirst: Don't be miffed, we wanted to interview you together so we could help clear up any confusion about who's who for the readers. You do have similarities, and I've heard that people sometimes get you confused.

Abstract Factory: It is true, there have been times I've been mistaken for Factory Method, and I know you've had similar issues, Factory Method. We're both really good at decoupling applications from specific implementations; we just do it in different ways. So I can see why people might sometimes get us confused.

Factory Method: Well, it still ticks me off. After all, I use classes to create and you use objects; that's totally different!

HeadFirst: Can you explain more about that, Factory Method?

Factory Method: Sure. Both Abstract Factory and I create objects – that's our jobs. But I do it through inheritance...

Abstract Factory: ...and I do it through object composition.

Factory Method: Right. So that means, to create objects using Factory Method, you need to extend a class and override a factory method.

HeadFirst: And that factory method does what?

Factory Method: It creates objects, of course! I mean, the whole point of the Factory Method Pattern is that you're using a subclass to do your creation for you. In that way, clients only need to know the abstract type they are using, the subclass worries about the concrete type. So, in other words, I keep clients decoupled from the concrete types.

Abstract Factory: And I do too, only I do it in a different way.

HeadFirst: Go on, Abstract Factory... you said something about object composition?

Abstract Factory: I provide an abstract type for creating a family of products. Subclasses of this type define how those products are produced. To use the factory, you instantiate one and pass it into some code that is written against the abstract type. So, like Factory Method, my clients are decoupled from the actual concrete products they use.

HeadFirst: Oh, I see, so another advantage is that you group together a set of related products.

Abstract Factory: That's right.

HeadFirst: What happens if you need to extend that set of related products, to say add another one? Doesn't that require changing your interface?

Abstract Factory: That's true; my interface has to change if new products are added, which I know people don't like to do....

Factory Method: <snicker>

Abstract Factory: What are you snickering at, Factory Method?

Factory Method: Oh, come on, that's a big deal! Changing your interface means you have to go in and change the interface of every subclass! That sounds like a lot of work.

Abstract Factory: Yeah, but I need a big interface because I am used to create entire families of products. You're only creating one product, so you don't really need a big interface, you just need one method.

HeadFirst: Abstract Factory, I heard that you often use factory methods to implement your concrete factories?

Abstract Factory: Yes, I'll admit it, my concrete factories often implement a factory method to create their products. In my case, they are used purely to create products...

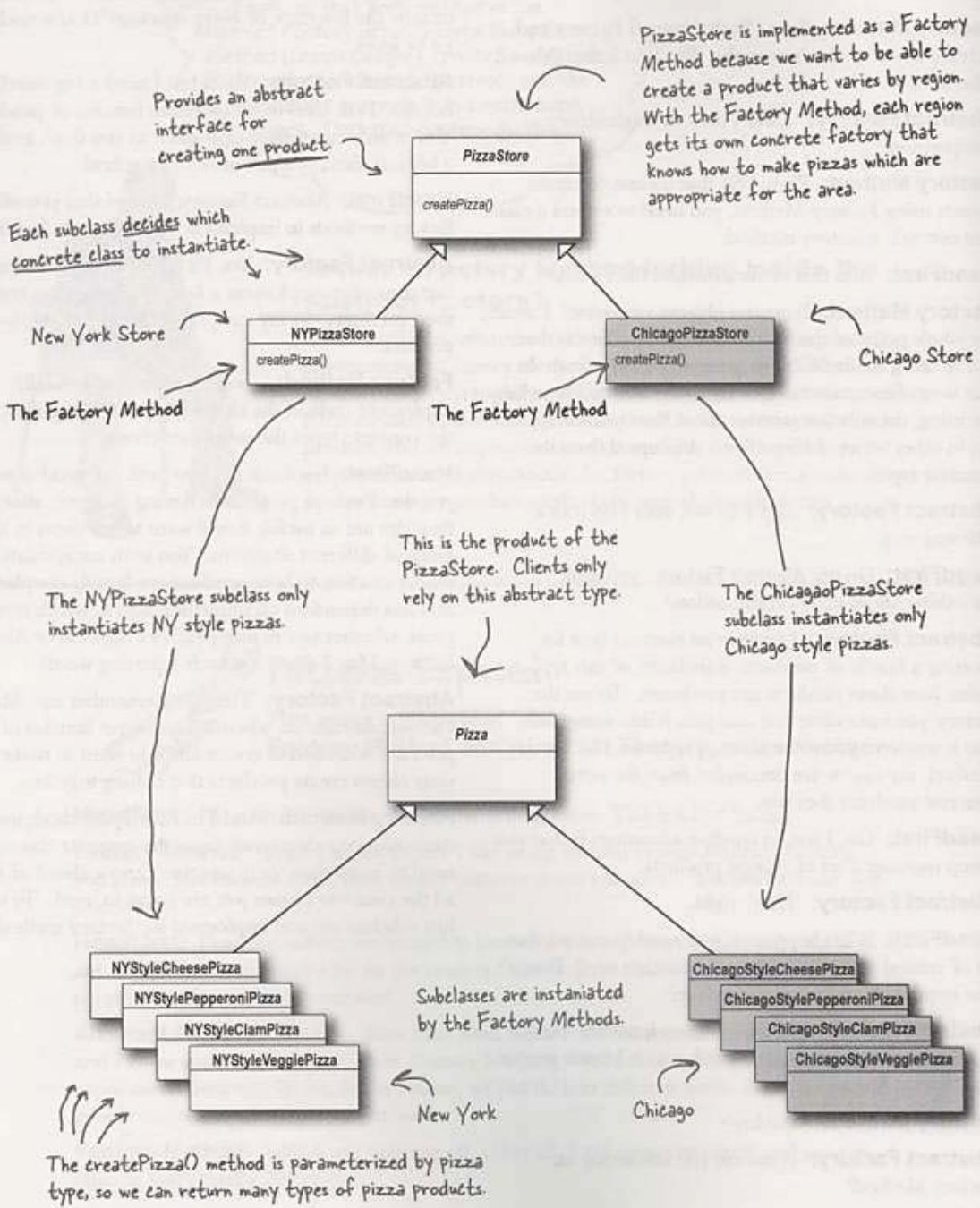
Factory Method: ...while in my case I usually implement code in the abstract creator that makes use of the concrete types the subclasses create.

HeadFirst: It sounds like you both are good at what you do. I'm sure people like having a choice; after all, factories are so useful, they'll want to use them in all kinds of different situations. You both encapsulate object creation to keep applications loosely coupled and less dependent on implementations, which is really great, whether you're using Factory Method or Abstract Factory. May I allow you each a parting word?

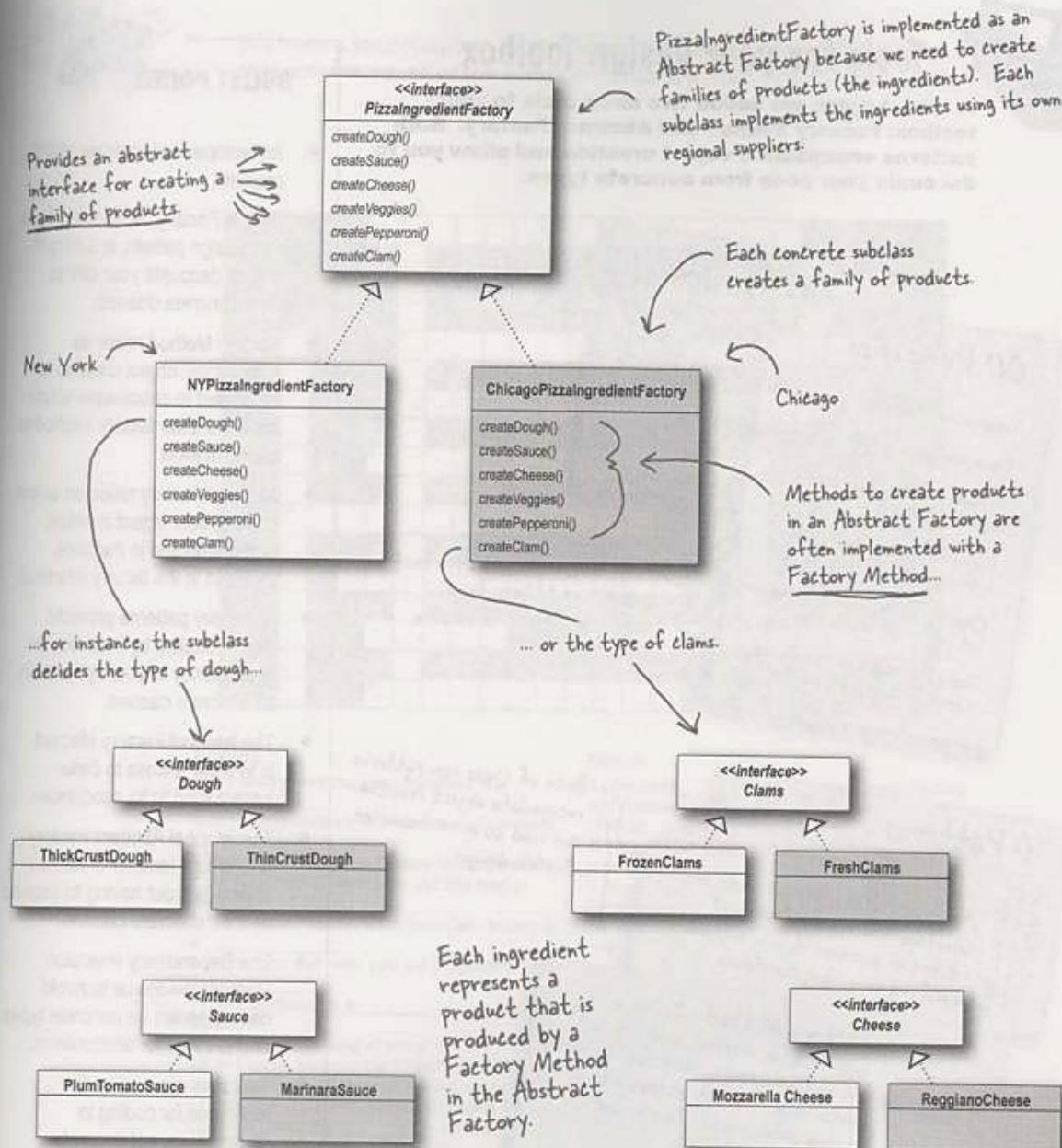
Abstract Factory: Thanks. Remember me, Abstract Factory, and use me whenever you have families of products you need to create and you want to make sure your clients create products that belong together.

Factory Method: And I'm Factory Method; use me to decouple your client code from the concrete classes you need to instantiate, or if you don't know ahead of time all the concrete classes you are going to need. To use me, just subclass me and implement my factory method!

Factory Method and Abstract Factory compared



the factory pattern

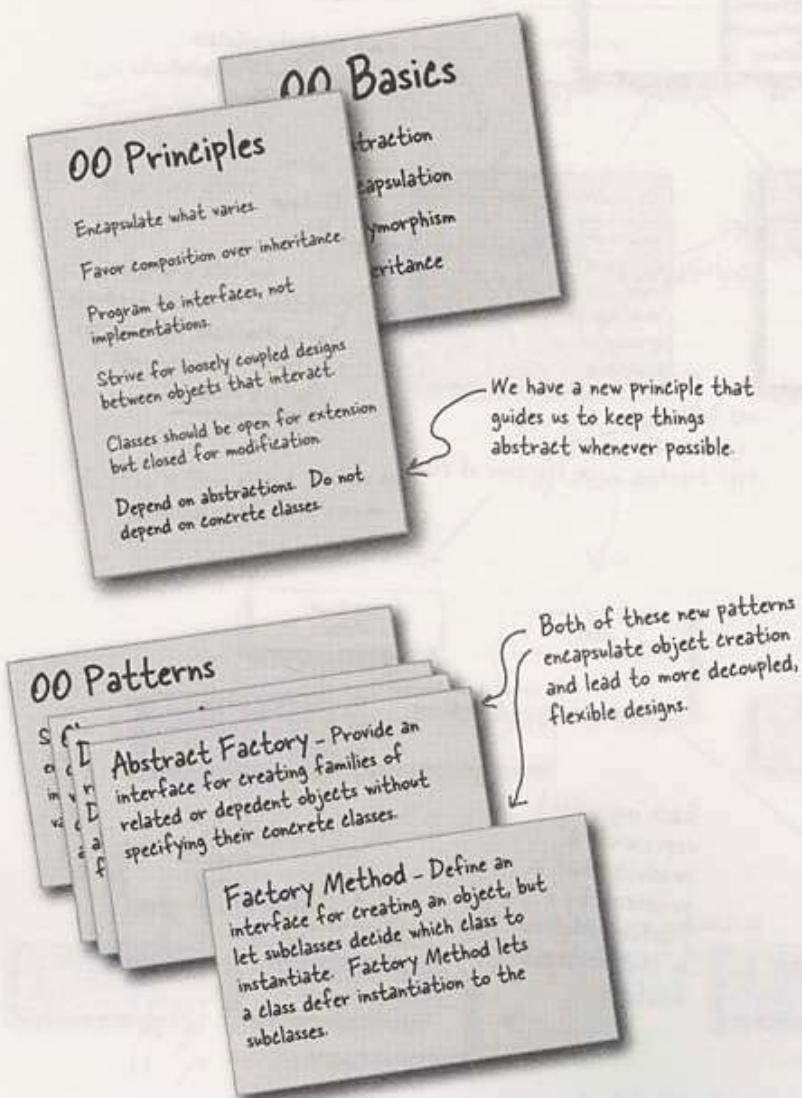


The product subclasses create parallel sets of product families. Here we have a New York ingredient family and a Chicago family.



Tools for your Design Toolbox

In this chapter, we added two more tools to your toolbox: **Factory Method** and **Abstract Factory**. Both patterns encapsulate object creation and allow you to decouple your code from concrete types.

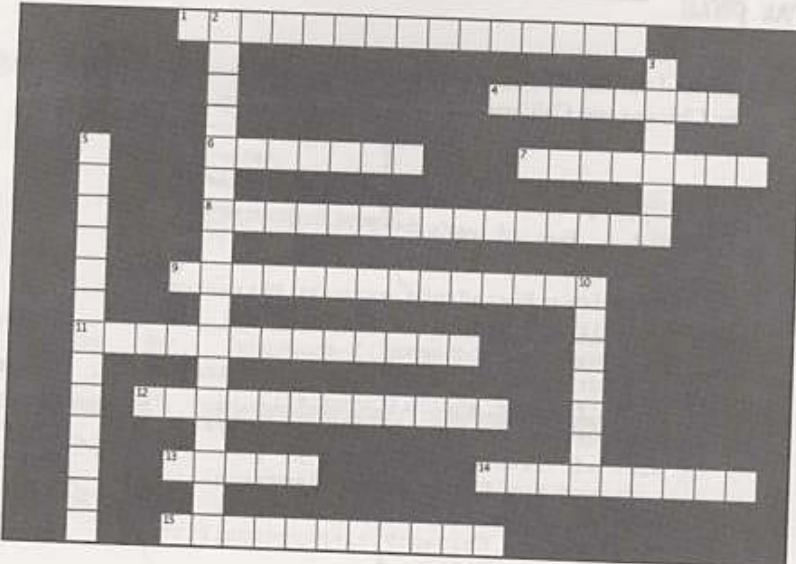


BULLET POINTS

- All factories encapsulate object creation.
- Simple Factory, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes.
- Factory Method relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects.
- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.
- All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes.
- The intent of Factory Method is to allow a class to defer instantiation to its subclasses.
- The intent of Abstract Factory is to create families of related objects without having to depend on their concrete classes.
- The Dependency Inversion Principle guides us to avoid dependencies on concrete types and to strive for abstractions.
- Factories are a powerful technique for coding to abstractions, not concrete classes



It's been a long chapter. Grab a slice of Pizza and relax while doing this crossword; all of the solution words are from this chapter.



Across

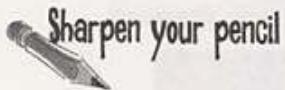
1. In Factory Method, each franchise is a _____
4. In Factory Method, who decides which class to instantiate?
6. Role of PizzaStore in Factory Method Pattern
7. All New York Style Pizzas use this kind of cheese
8. In Abstract Factory, each ingredient factory is a _____
9. When you use new, you are programming to an _____
11. createPizza() is a _____ (two words)
12. Joel likes this kind of pizza
13. In Factory Method, the PizzaStore and the concrete Pizzas all depend on this abstraction
14. When a class instantiates an object from a concrete class, it's _____ on that object
15. All factory patterns allow us to _____ object creation

Down

2. We used _____ in Simple Factory and Abstract Factory and inheritance in Factory Method
3. Abstract Factory creates a _____ of products
5. Not a REAL factory pattern, but handy nonetheless
10. Ethan likes this kind of pizza



Exercise solutions



We've knocked out the NYPizzaStore; just two more to go and we'll be ready to franchise! Write the Chicago and California PizzaStore implementations here:

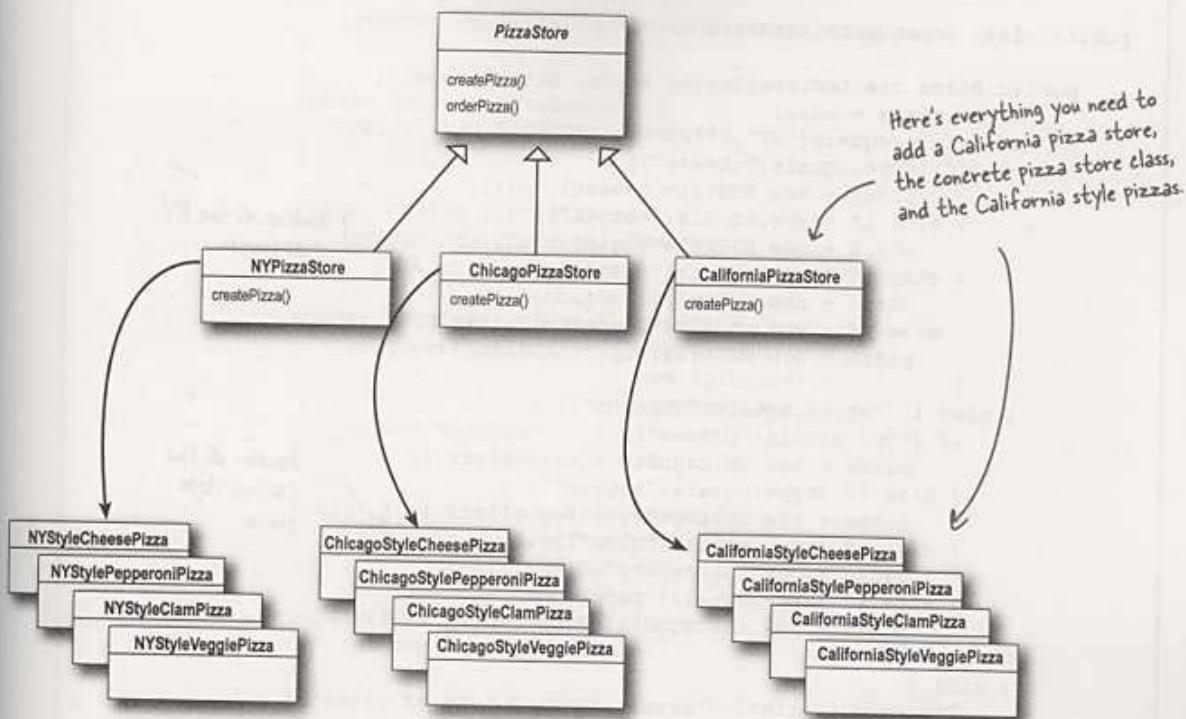
Both of these stores are almost exactly like the New York store... they just create different kinds of pizzas

```
public class ChicagoPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new ChicagoStyleCheesePizza(); ← For the Chicago pizza
        } else if (item.equals("veggie")) { ← store, we just have to
            return new ChicagoStyleVeggiePizza(); ← make sure we create
        } else if (item.equals("clam")) { ← Chicago style pizzas...
            return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) { ←
            return new ChicagoStylePepperoniPizza();
        } else return null;
    }
}
```

```
public class CaliforniaPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new CaliforniaStyleCheesePizza(); ← and for the California
        } else if (item.equals("veggie")) { ← pizza store, we create
            return new CaliforniaStyleVeggiePizza(); ← California style pizzas.
        } else if (item.equals("clam")) {
            return new CaliforniaStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new CaliforniaStylePepperoniPizza();
        } else return null;
    }
}
```

Design Puzzle Solution

We need another kind of pizza for those crazy Californians (crazy in a GOOD way of course). Draw another parallel set of classes that you'd need to add a new California region to our PizzaStore.

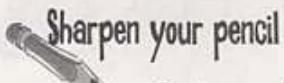


Okay, now write the five silliest things you can think of to put on a pizza. Then, you'll be ready to go into business making pizza in California!

Here are our suggestions...

- Mashed Potatoes with Roasted Garlic
- BBQ Sauce
- Artichoke Hearts
- M&M's
- Peanuts

A very dependent PizzaStore



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:

8 number

12 number with California too



Sharpen your pencil

Go ahead and write the ChicagoPizzaIngredientFactory; you can reference the classes below in your implementation:

```
public class ChicagoPizzaIngredientFactory
    implements PizzaIngredientFactory
{
    public Dough createDough() {
        return new ThickCrustDough();
    }

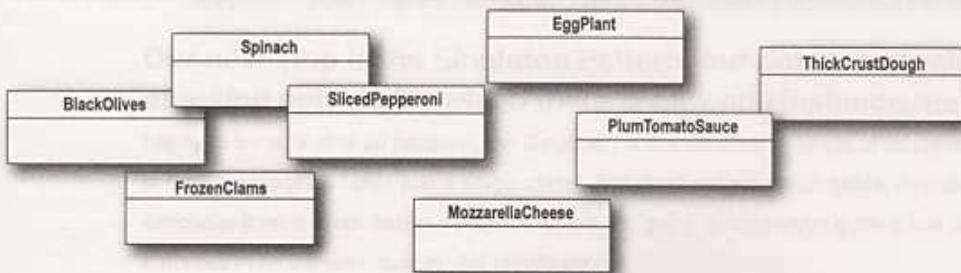
    public Sauce createSauce() {
        return new PlumTomatoSauce();
    }

    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(),
            new Spinach(),
            new Eggplant() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FrozenClams();
    }
}
```



crossword puzzle solution



Puzzle Solution

	¹ C	² O	³ N	⁴ C	⁵ R	⁶ E	⁷ T	⁸ O	⁹ R	¹⁰ A	¹¹ T	¹² M	¹³ E	¹⁴ D	¹⁵ E
	B									F					
	J								S	U	B	C	L	A	S
	E								U	S	C	L	A	S	
⁵ S		⁶ C	⁷ R	⁸ E	⁹ A	¹⁰ M	¹¹ T	¹² O	¹³ P	¹⁴ I	¹⁵ G	¹ R	² E	³ G	⁴ I
⁶ I		⁷ R	⁸ E	⁹ A	¹⁰ M	¹¹ T	¹² O	¹³ P	¹⁴ I	¹⁵ G	¹ R	² E	³ G	⁴ I	
⁷ M		⁸ C	⁹ O	¹⁰ N	¹¹ E	¹² T	¹³ O	¹⁴ D	¹⁵ E	¹ C	² H	³ I	⁴ C	⁵ A	⁶ S
⁸ P		⁹ O	¹⁰ N	¹¹ E	¹² T	¹³ O	¹⁴ D	¹⁵ E	¹ C	² H	³ I	⁴ C	⁵ A	⁶ S	
⁹ L		¹⁰ I	¹¹ M	¹² E	¹³ T	¹⁴ D	¹⁵ E	¹ C	² H	³ I	⁴ C	⁵ A	⁶ S	⁷ E	⁸ T
¹⁰ E		¹¹ M	¹² E	¹³ T	¹⁴ D	¹⁵ E	¹ C	² H	³ I	⁴ C	⁵ A	⁶ S	⁷ E	⁸ T	
¹¹ F		¹² C	¹³ H	¹⁴ I	¹⁵ C	¹ E	² H	³ I	⁴ C	⁵ A	⁶ S	⁷ E	⁸ T		
¹² A		¹³ H	¹⁴ I	¹⁵ C	¹ E	² H	³ I	⁴ C	⁵ A	⁶ S	⁷ E	⁸ T			
¹³ C		¹⁴ I	¹⁵ C	¹ E	² H	³ I	⁴ C	⁵ A	⁶ S	⁷ E	⁸ T				
¹⁴ T		¹⁵ C	¹ E	² H	³ I	⁴ C	⁵ A	⁶ S	⁷ E	⁸ T					
¹⁵ O		¹ E	² H	³ I	⁴ C	⁵ A	⁶ S	⁷ E	⁸ T						