Questions

- What is the key design principle for the past two lectures?
- What is the key design principle for design patterns in general?

Creational Patterns

How the world is created

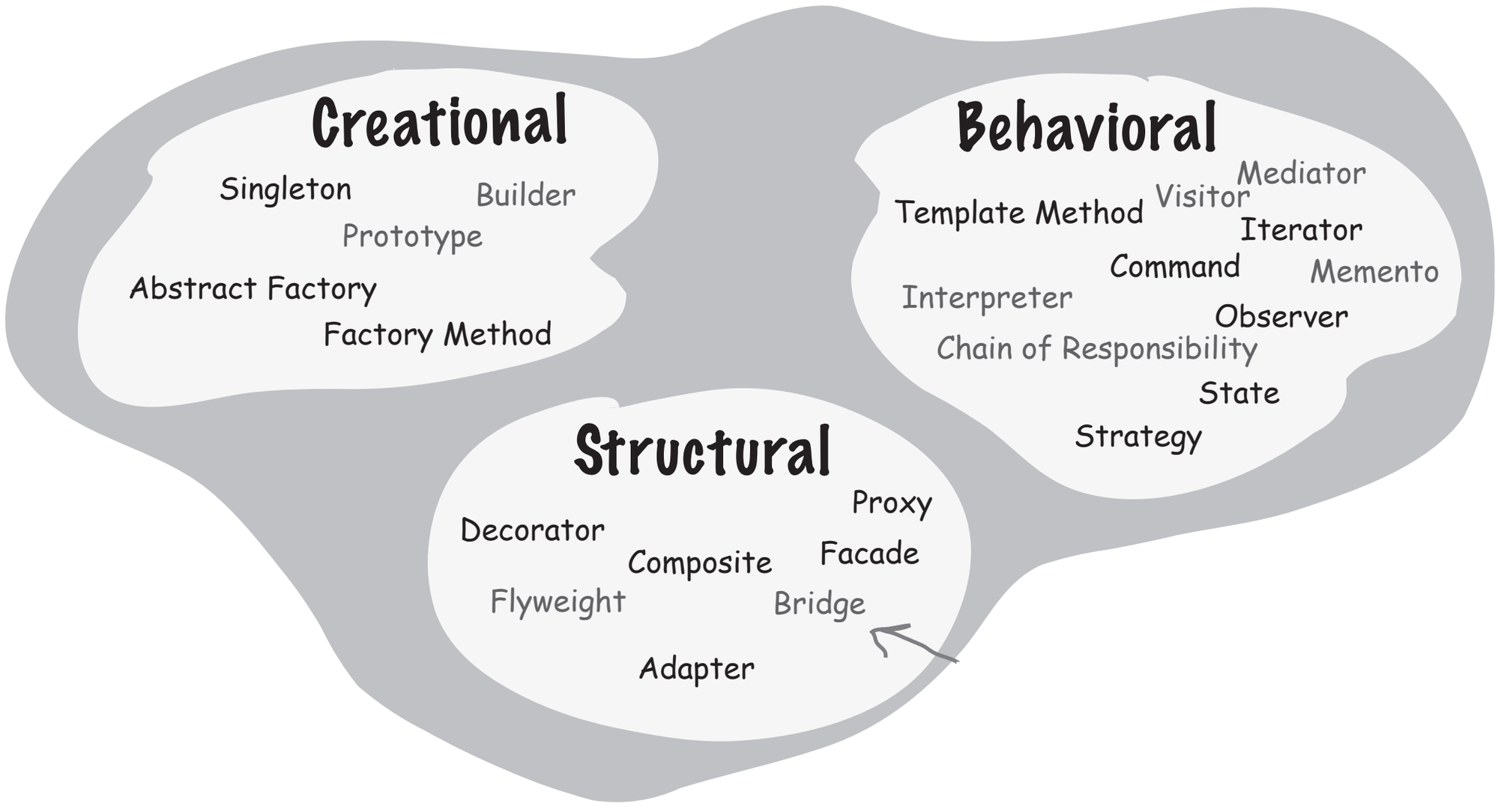
Today's lecture is based on

Slides from:

Tom Zimmermann
Microsoft Research

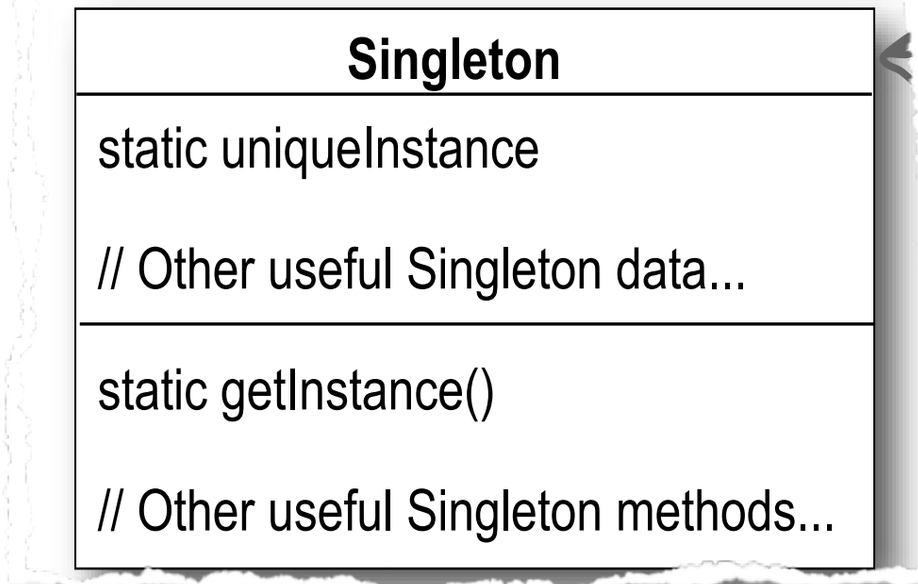
Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.

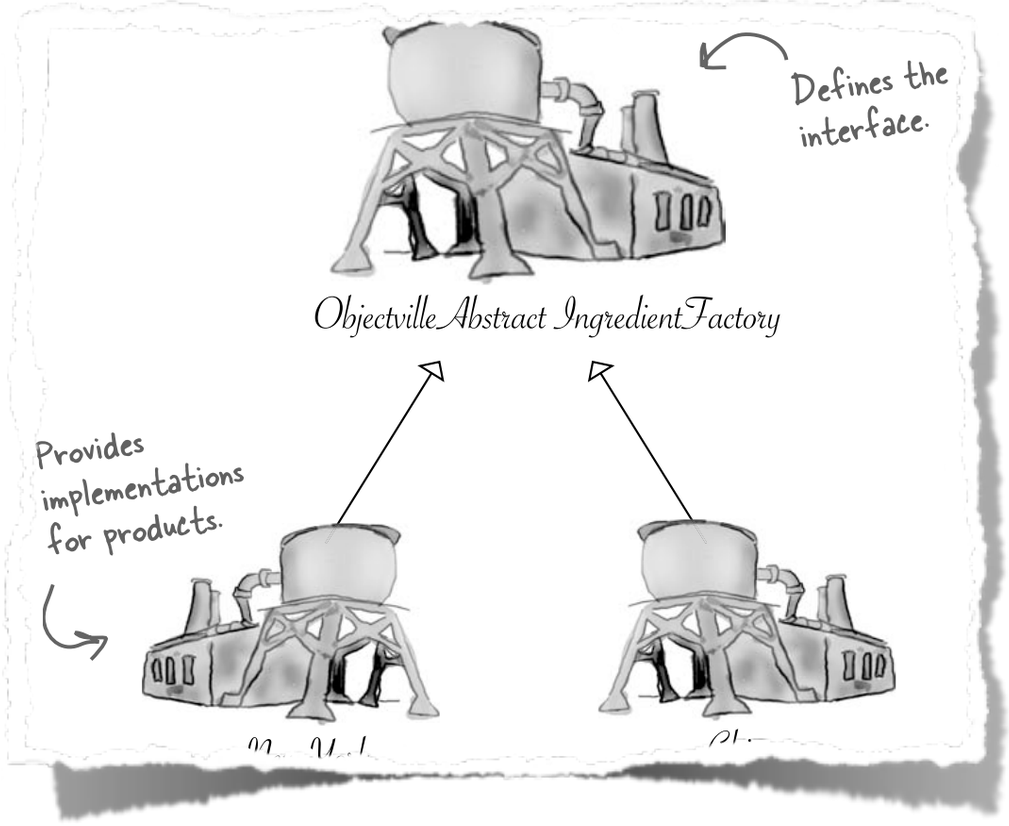


Structural patterns let you compose classes or objects into larger structures.

Today



Singleton



Factory

Only one-of-a-kind

Our next stop is the Singleton Pattern, our ticket to creating one-of-a-kind objects for which there is only one instance. You might be happy to know that of all patterns, the Singleton is the simplest in terms of its class diagram; in fact, the diagram holds just a single class! But don't get too comfortable; despite its simplicity from a class design perspective, we are going to encounter quite a few bumps and potholes in its implementation. So buckle up.

What is this? An entire chapter about how to instantiate just ONE OBJECT!

That's one and ONLY ONE object.

Why only one-of-a-kind?

- Thread pools
- Printers
- Dialog boxes
- Objects used for logging
- Objects handles graphics cards?
- System.out?

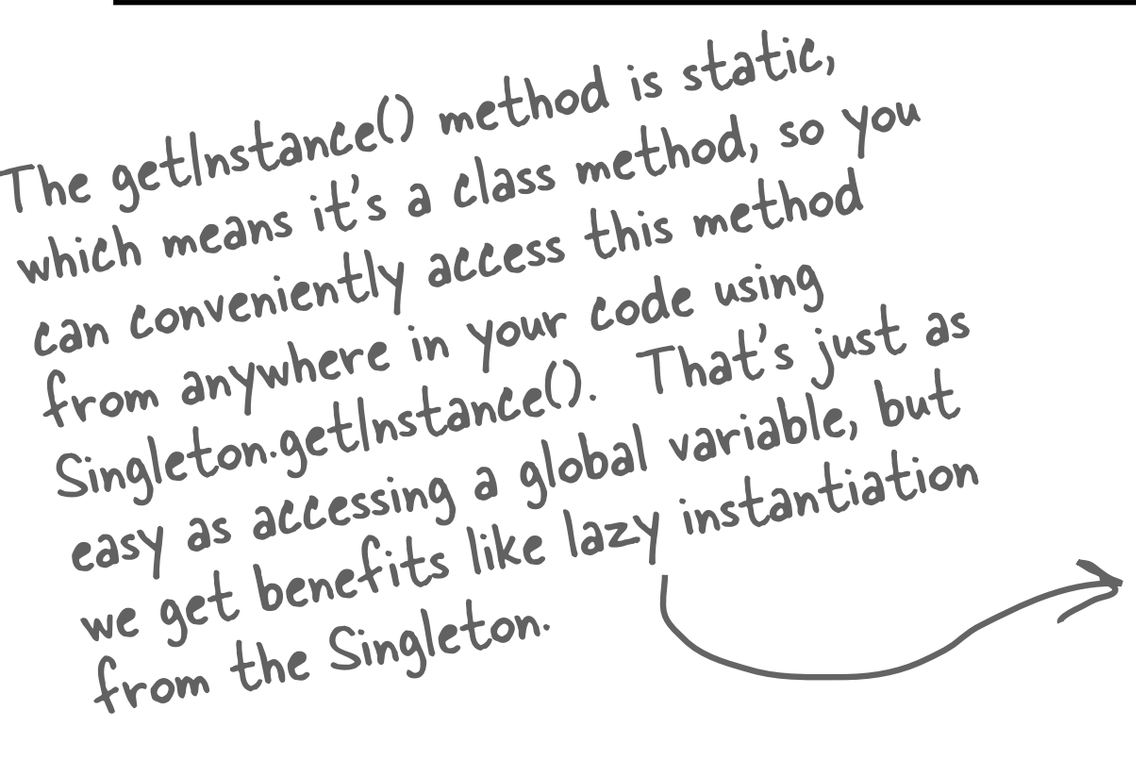
Design Requirements

- Cannot allow programmers to create the object.
  - What should we do?
- A single variable that is visible to the whole program.
  - What is this type of variable?

The Singleton pattern

The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.

The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.



The uniqueInstance class variable holds our one and only instance of Singleton.

A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

The Singleton pattern

Let's rename MyClass to Singleton.

We have a static variable to hold our one instance of the class Singleton.

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

Of course, Singleton is a normal class; it has other useful instance variables and methods.

The Singleton pattern

Code Up Close

uniqueInstance holds our ONE instance; remember, it is a static variable.

If uniqueInstance is null, then we haven't created the instance yet...

...and, if it doesn't exist, we instantiate Singleton through its private constructor and assign it to uniqueInstance. Note that if we never need the instance, it never gets created; this is lazy instantiation.

```
if (uniqueInstance == null) {
    uniqueInstance = new MyClass();
}
return uniqueInstance;
```

If uniqueInstance wasn't null, then it was previously created. We just fall through to the return statement.

By the time we hit this code, we have an instance and we return it.

Thread-safe Singleton

Move to an eagerly created instance rather than a lazily created one

If your application always creates and uses an instance of the Singleton or the overhead of creation and runtime aspects of the Singleton are not onerous, you may want to create your Singleton eagerly, like this:

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it.

Allocating space for strings

- Address book
  - Charles Zhang, 39 Mass Avenue, 91210, CA, USA
  - Leonardo Dicaprio, 38 Mass Avenue, 91210, CA, USA
- Storage
  - We need to allocate 91 characters if we use "new"
  - Sometimes the better way is not to create

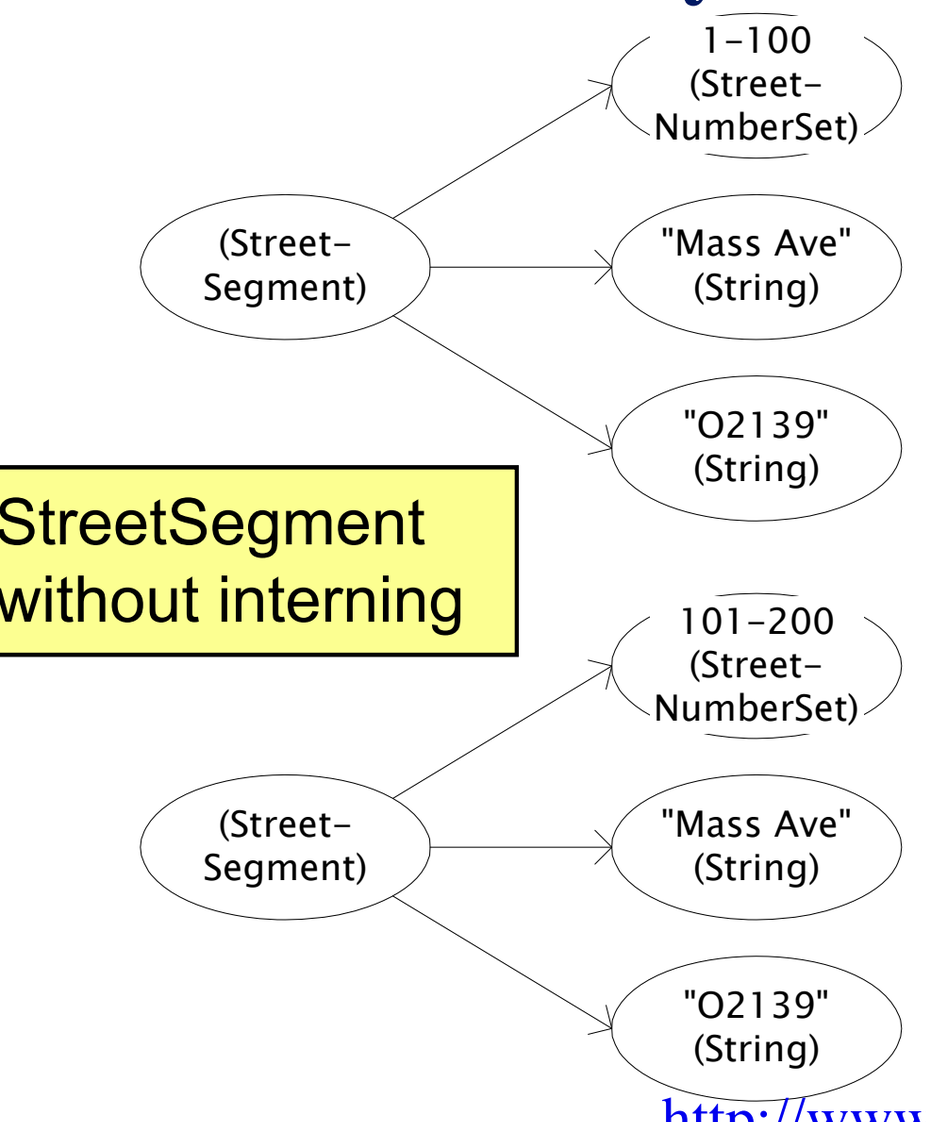
Interning pattern

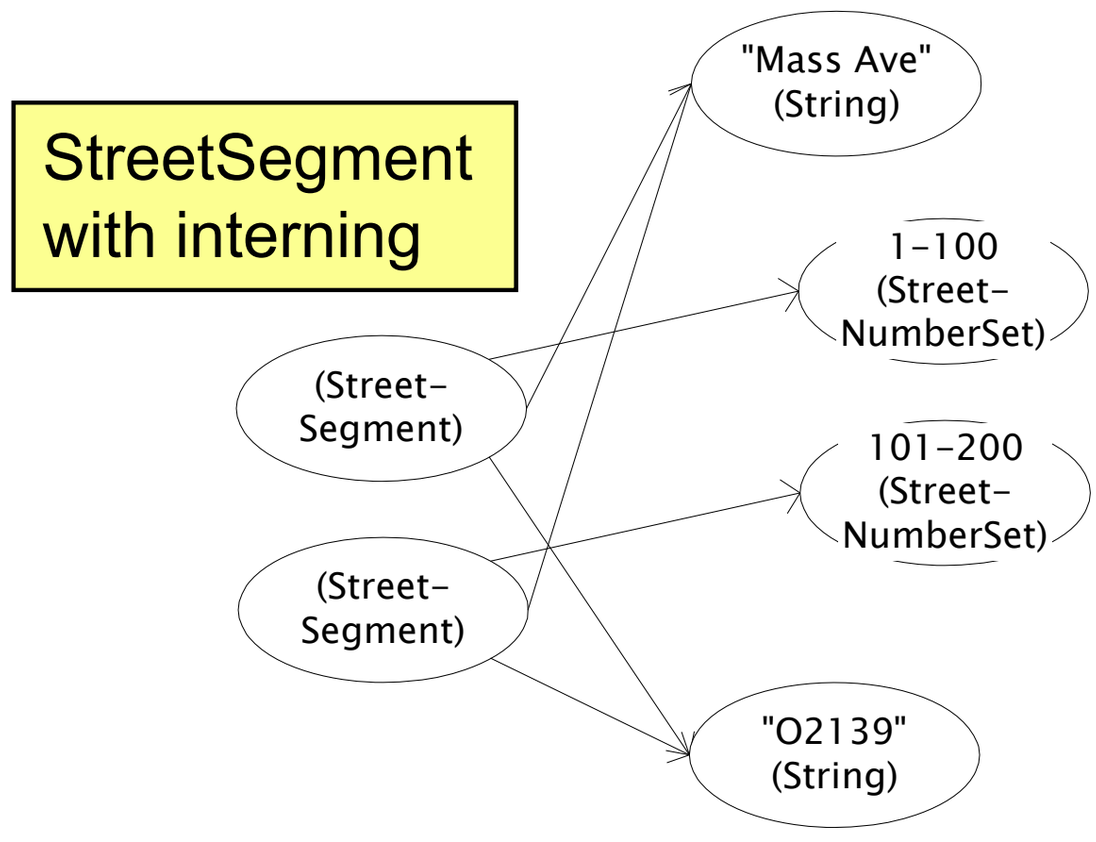
Reuse existing objects instead of creating new ones

Less space

May compare with `==` instead of `equals()`

Permitted only for immutable objects





http://www.cs.washington.edu/education/courses/403/09sp/calendar403.html.

Interning mechanism

Maintain a collection of all objects

If an object already appears, return that instead

```
HashMap<String, String> segnames;
String canonicalName(String n) {
  if (segnames.containsKey(n)) {
    return segnames.get(n);
  } else {
    segnames.put(n, n);
    return n;
  }
}
```

Java builds this in for strings: `String.intern()`

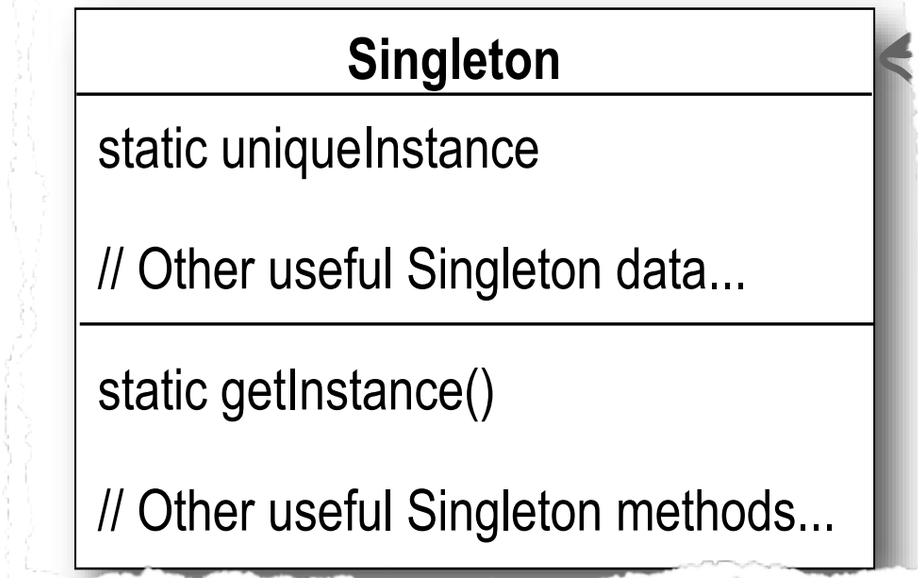
Two approaches:

- create the object, but perhaps discard it and return another
- check against the arguments before creating the new object

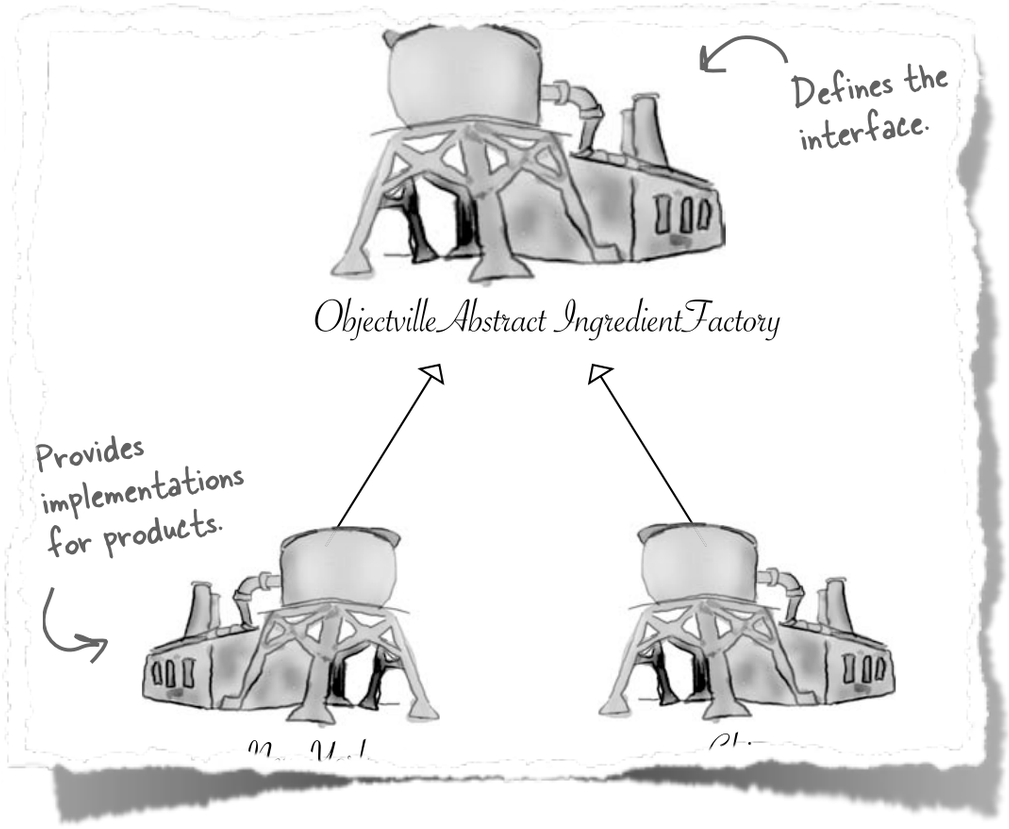
Design Improvement

- Using intern, only 63 character spaces needed
- We can go a step further, there are only 26 characters in English
  - We can do this if we are really limited for space
  - We need to address the "word" relationship of characters else.
  - Bonus pattern: Flyweight. (For your reference only)

Today



Singleton



Factory

Making pizza

```
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.

Making 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 & cheese), then we bake it, cut it and box it!

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

Adding more pizza

```
Pizza orderPizza(String type) {
        Pizza pizza;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("greek") {
            pizza = new GreekPizza();
        } else if (type.equals("pepperoni") {
            pizza = new PepperoniPizza();
        } else if (type.equals("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.

Encapsulating object creation

```
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();
}
```

```
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?

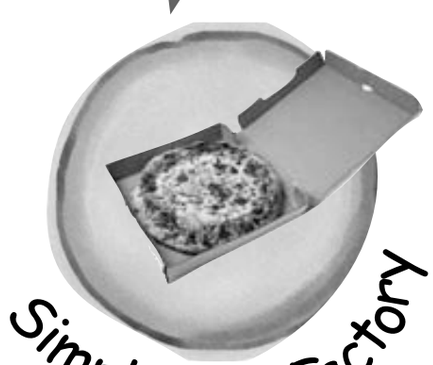
Encapsulating object creation

```
Pizza orderPizza(String type) {
    Pizza pizza;

    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();
    }

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

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

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.

What's going to go here?



SimplePizzaFactory

A simple pizza factory

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

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

```
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;
    }
}
```

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.

Reworking the pizza store

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.

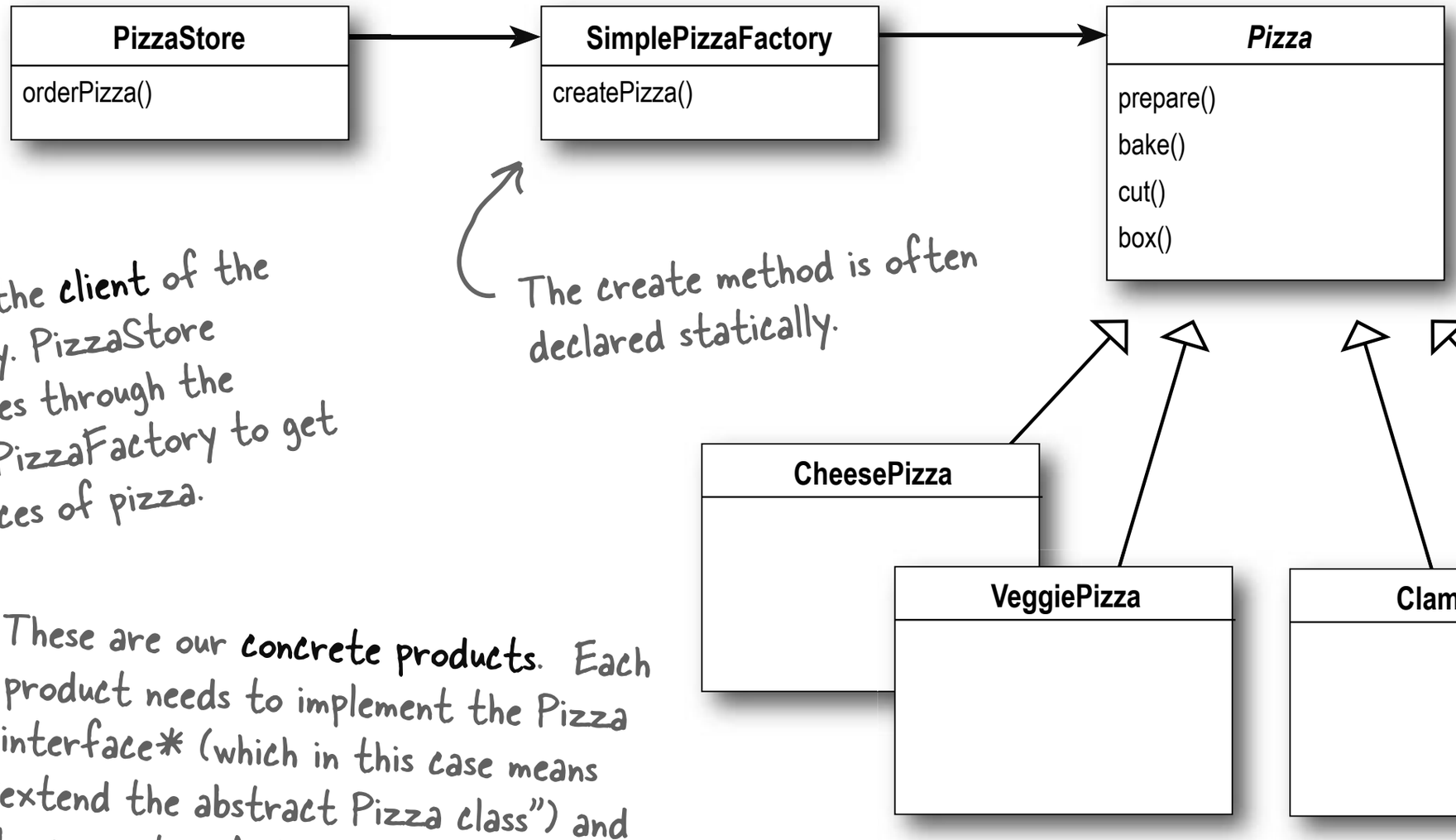
And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

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

The Simple Factory defined

This is the **factory** where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes..

This is the **product** of the factory: pizza!



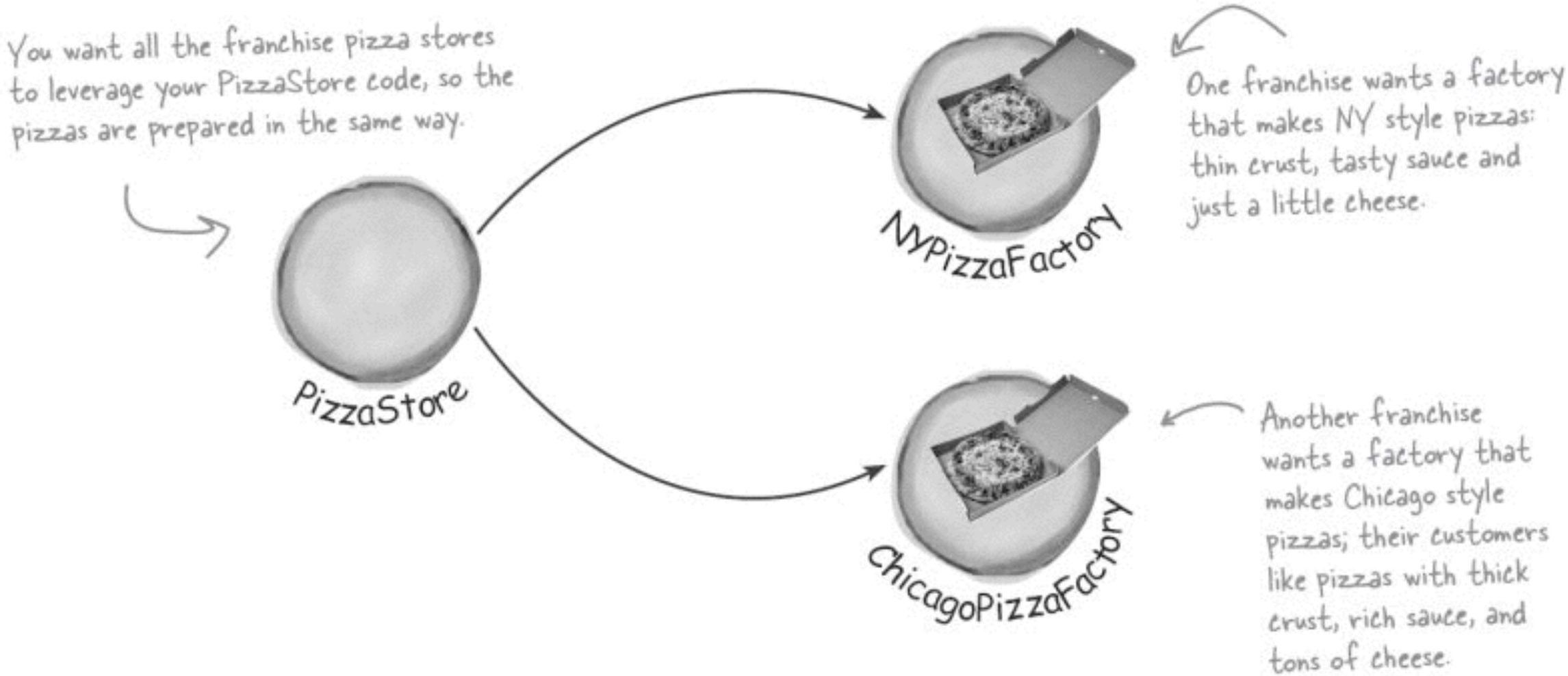
We've defined Pizza as an abstract class with some helpful implementations that can be overridden.

This is the **client** of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.

The create method is often declared statically.

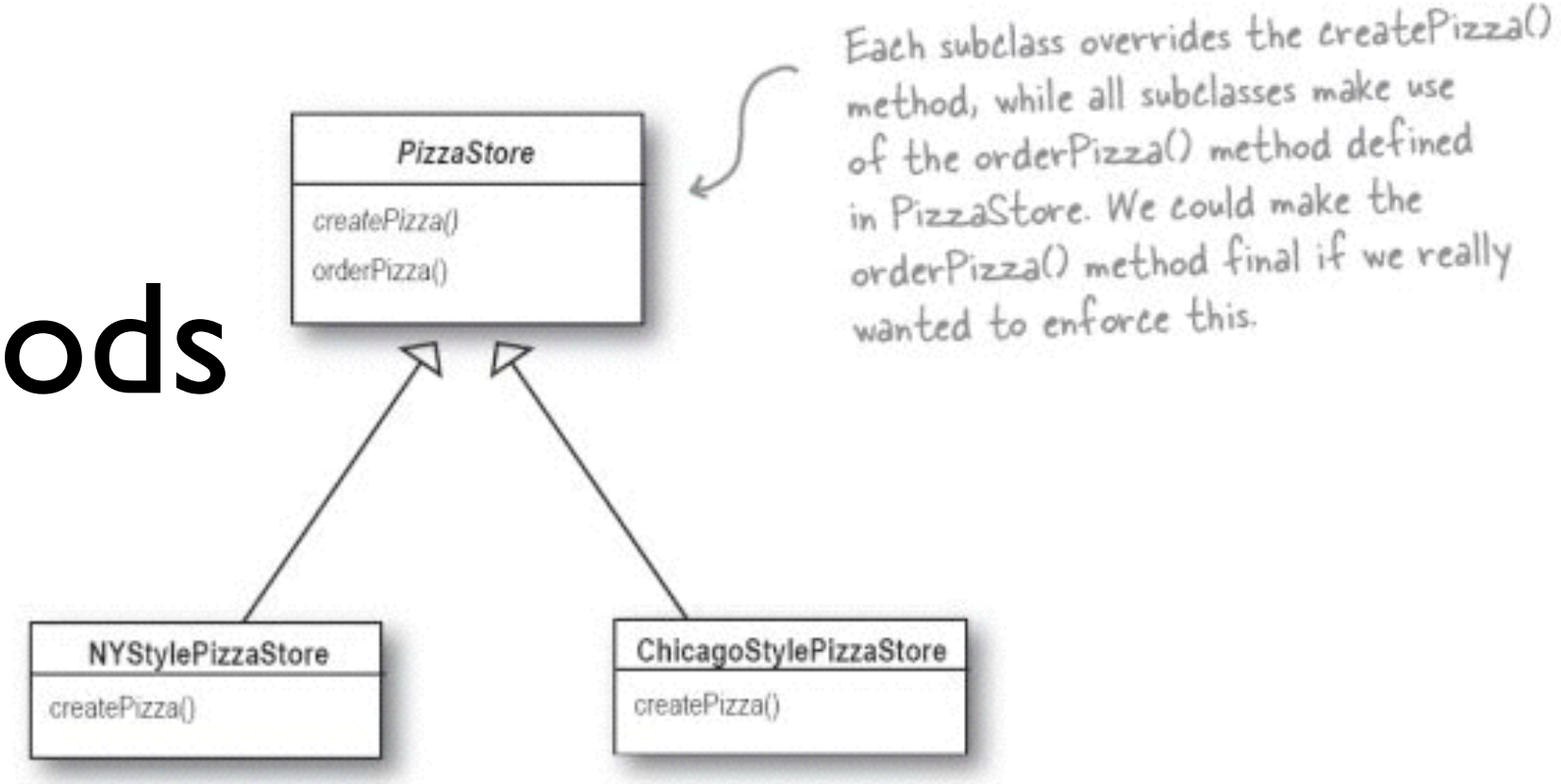
These are our **concrete products**. Each product needs to implement the Pizza interface* (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

Franchising pizza stores

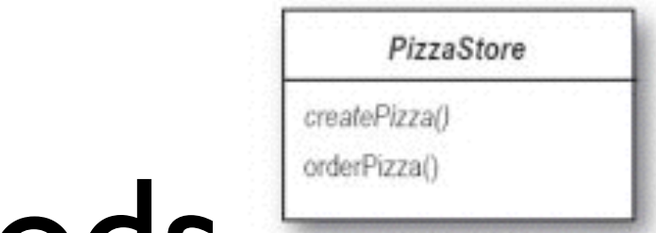



```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("Veggie");
```

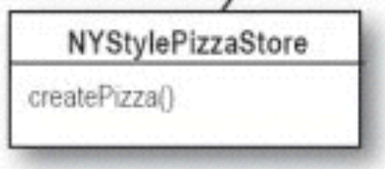
Alternative: abstract methods



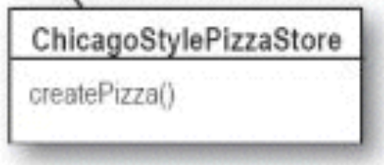
Alternative: abstract methods



Each subclass overrides the createPizza() method, while all subclasses make use of the orderPizza() method defined in PizzaStore. We could make the orderPizza() method final if we really wanted to enforce this.



If a franchise wants NY style pizzas for its customers, it uses the NY subclass, which has its own createPizza() method, creating NY style pizzas.



Similarly, by using the Chicago subclass, we get an implementation of createPizza() with Chicago ingredients.

Remember: createPizza() is abstract in PizzaStore, so all pizza store subtypes MUST implement the method.

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new NYStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new NYStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new NYStyleVeggiePizza();
    }
}
```

```
public Pizza createPizza(type) {
    if (type.equals("cheese")) {
        pizza = new ChicagoStyleCheesePizza();
    } else if (type.equals("pepperoni") {
        pizza = new ChicagoStylePepperoniPizza();
    } else if (type.equals("clam") {
        pizza = new ChicagoStyleClamPizza();
    } else if (type.equals("veggie") {
        pizza = new ChicagoStyleVeggiePizza();
    }
}
```

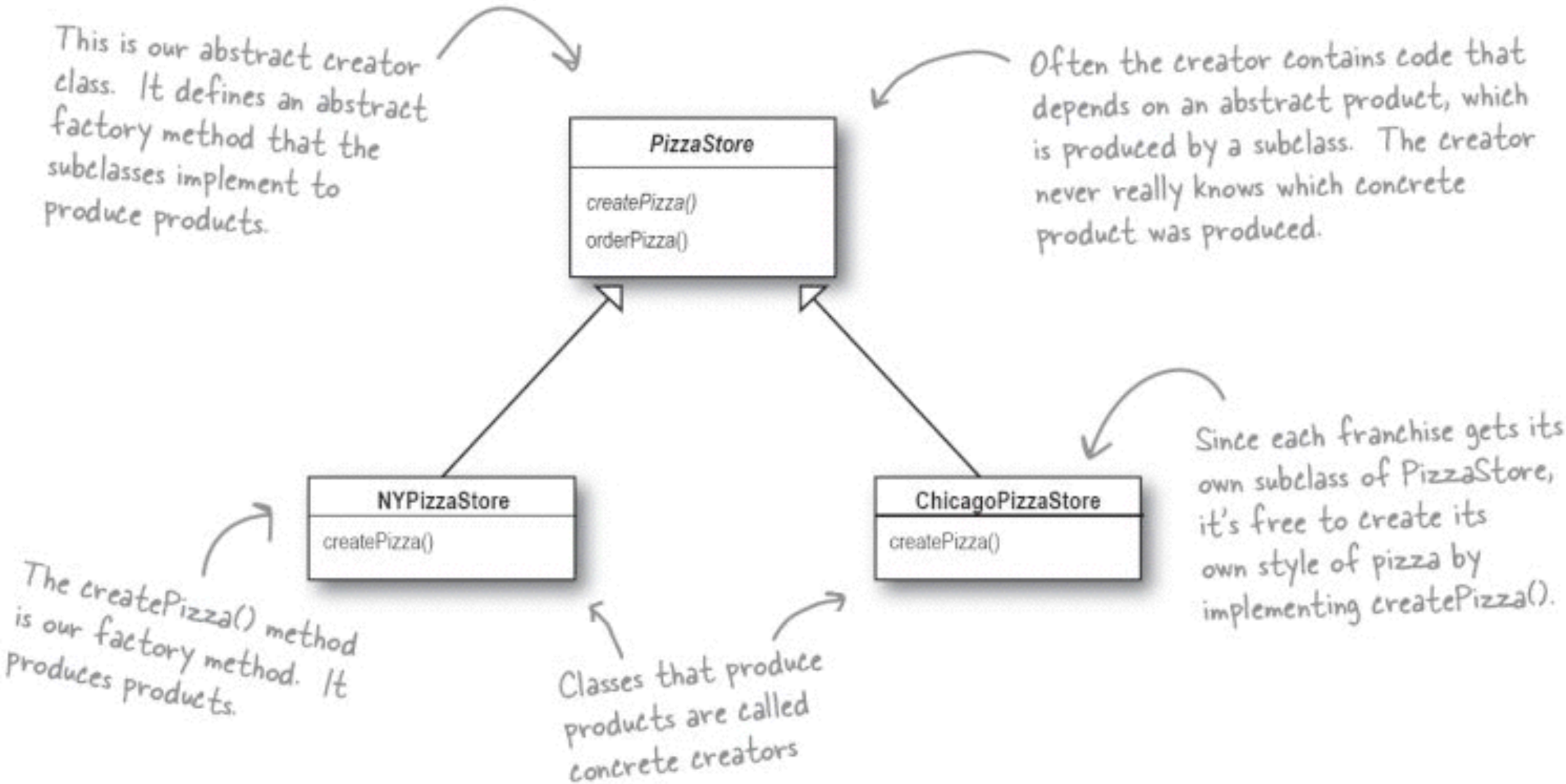
Declaring a factory method

```
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
}
```

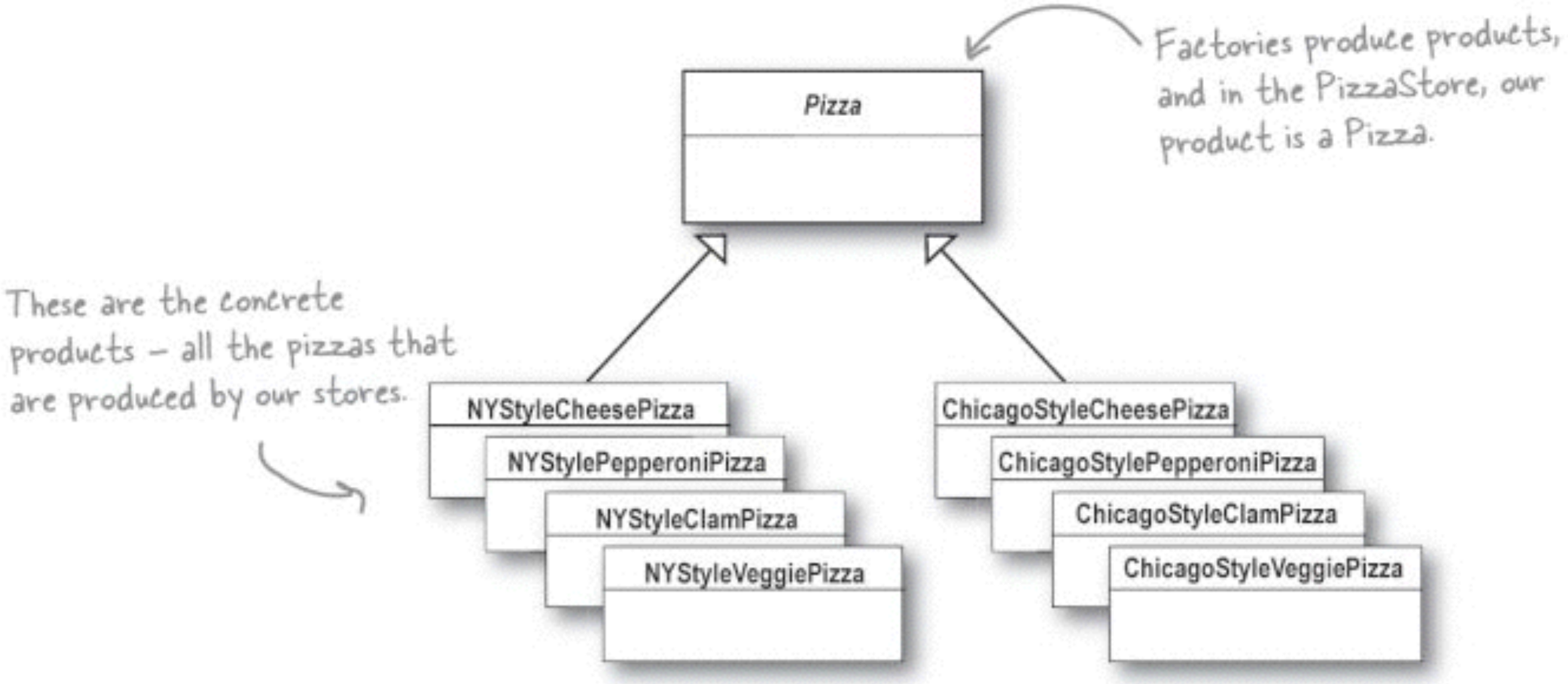
The subclasses of PizzaStore handle object instantiation for us in the createPizza() method.

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

The Creator classes

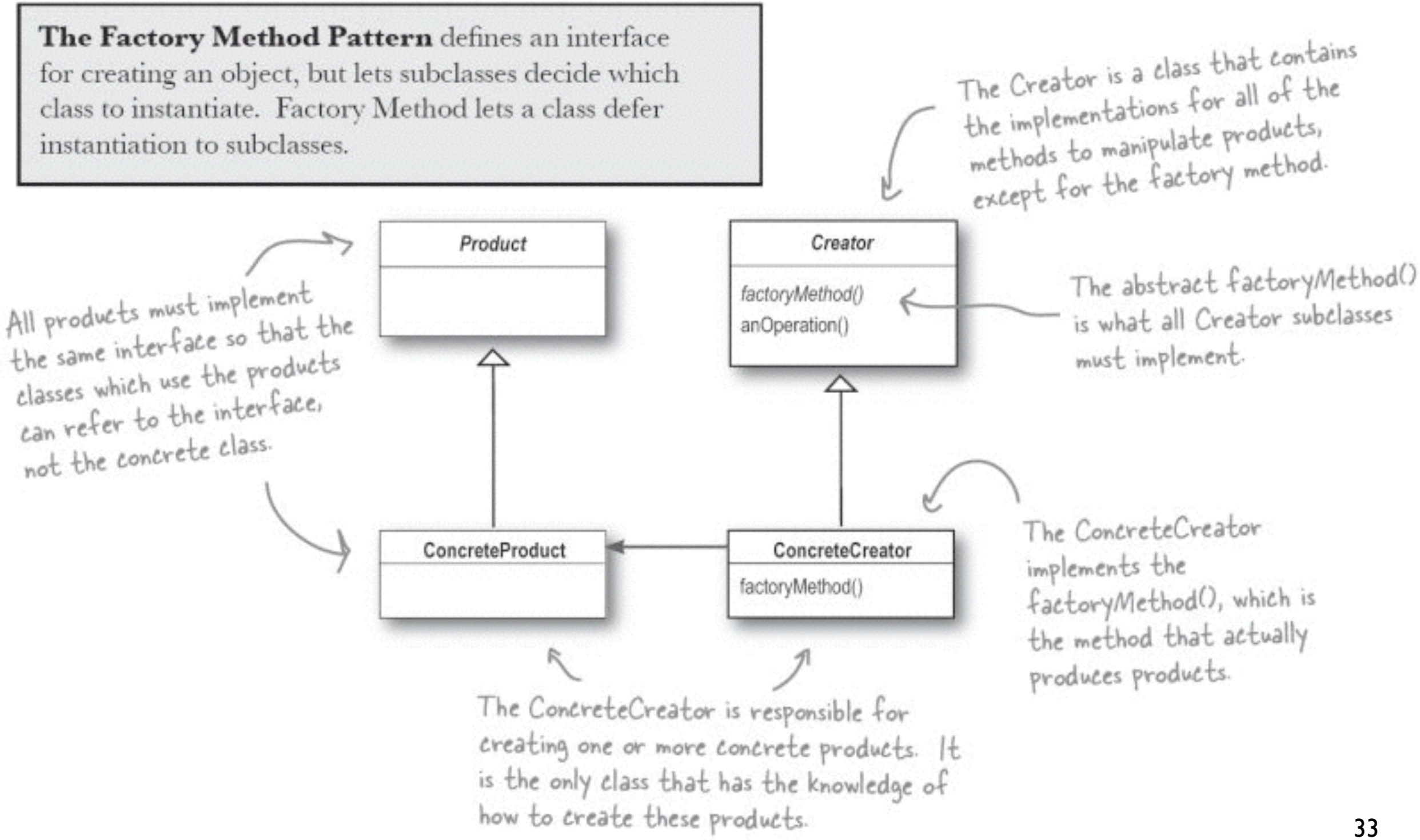


The Product classes



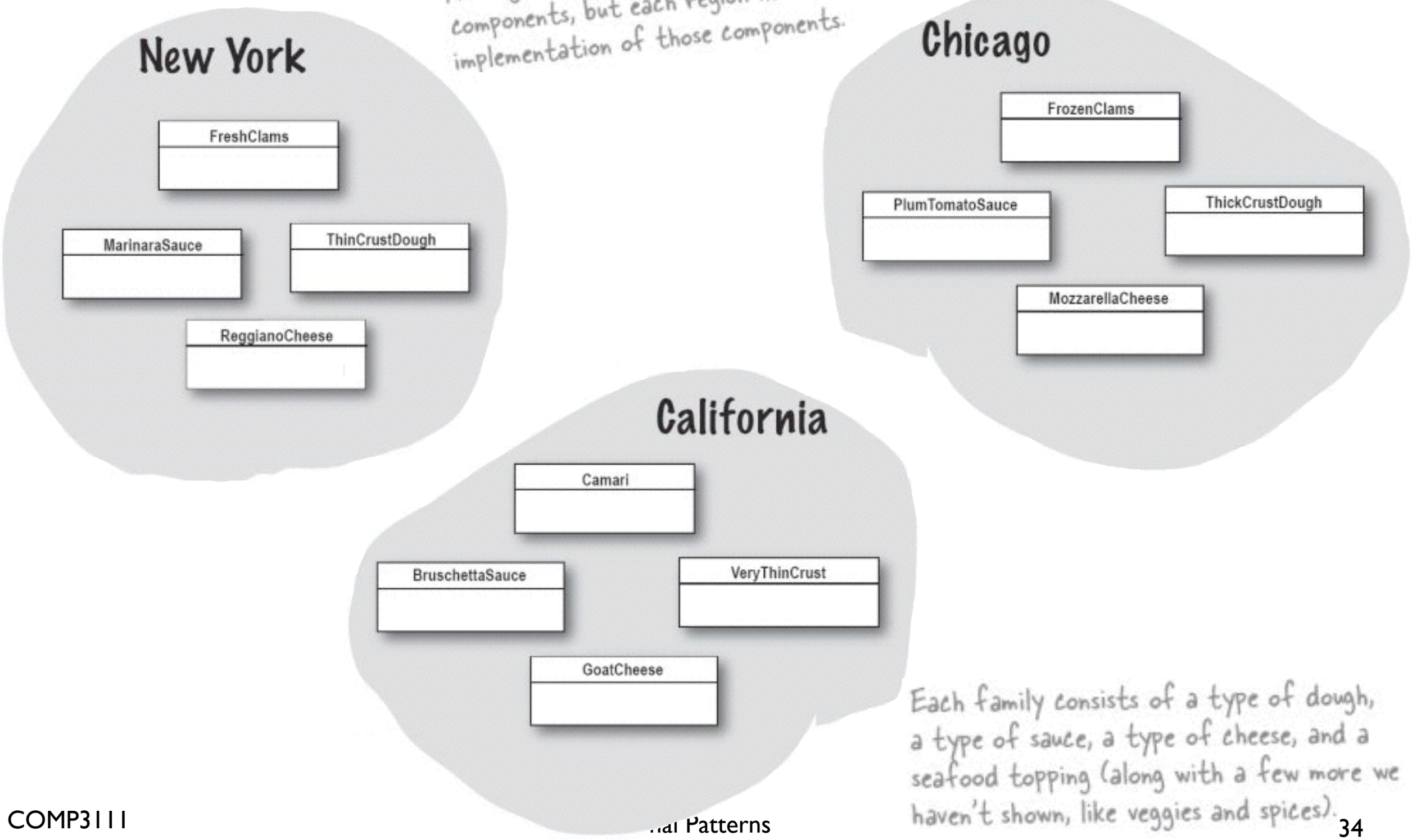
The Factory Method pattern

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.



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).

Revisiting pizza stores

```
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!

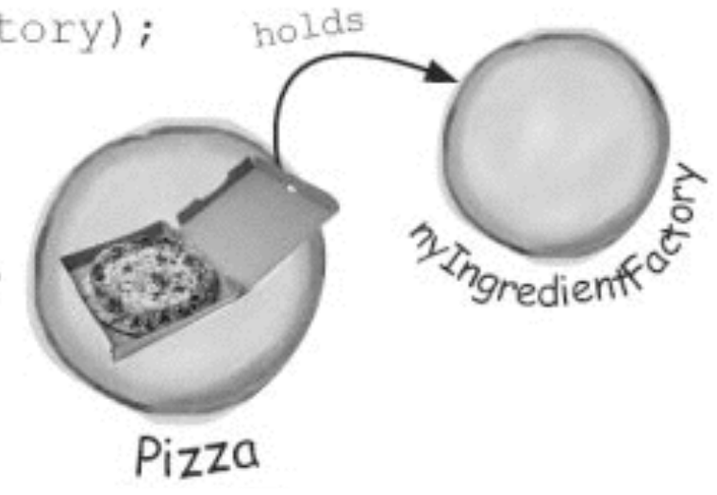
For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.

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

The ingredient factory is chosen and instantiated in the PizzaStore and then passed into the constructor of each pizza.

```
Pizza pizza = new CheesePizza(nyIngredientFactory);
```

Creates a instance of Pizza that is composed with the New York ingredient factory.

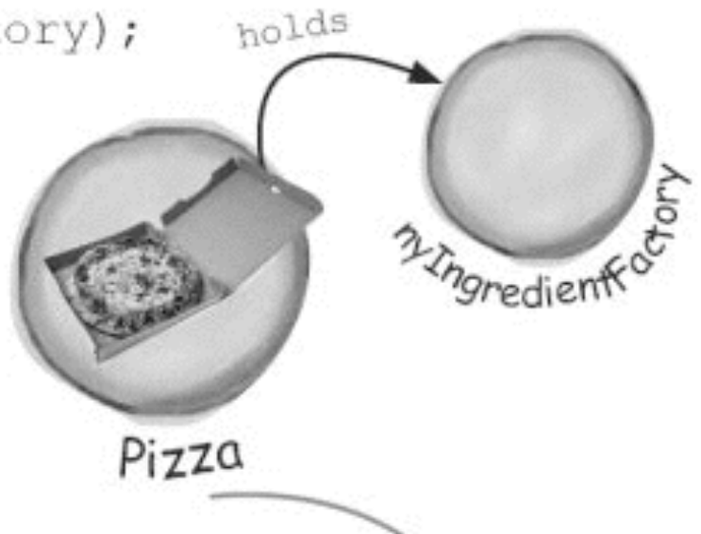


4 **When the createPizza() method is called, that's when our ingredient factory gets involved:**

The ingredient factory is chosen and instantiated in the PizzaStore and then passed into the constructor of each pizza.

```
Pizza pizza = new CheesePizza(nyIngredientFactory);
```

Creates a instance of Pizza that is composed with the New York ingredient factory.



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();
}
```

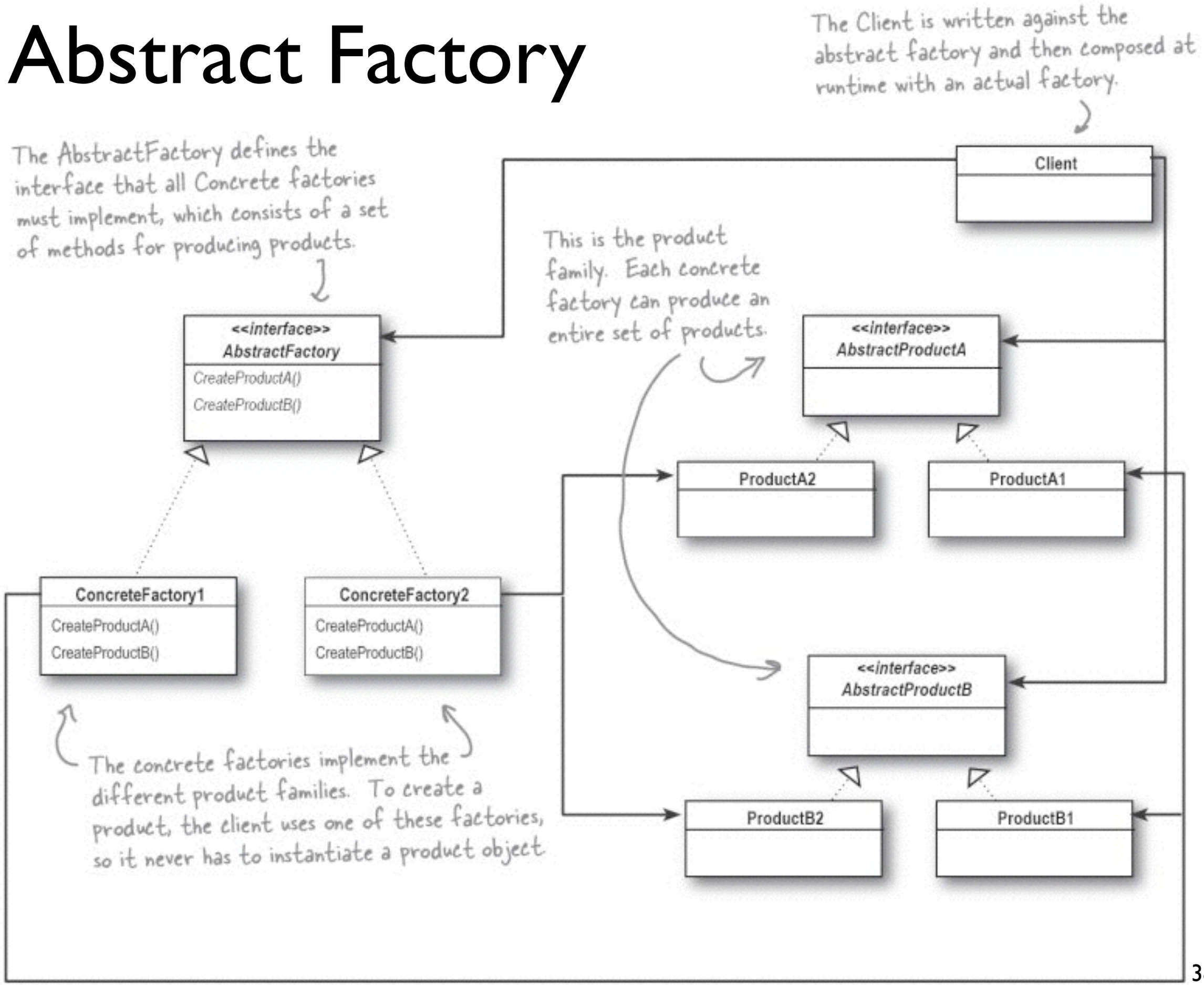
Thin crust

Marinara

Reggiano

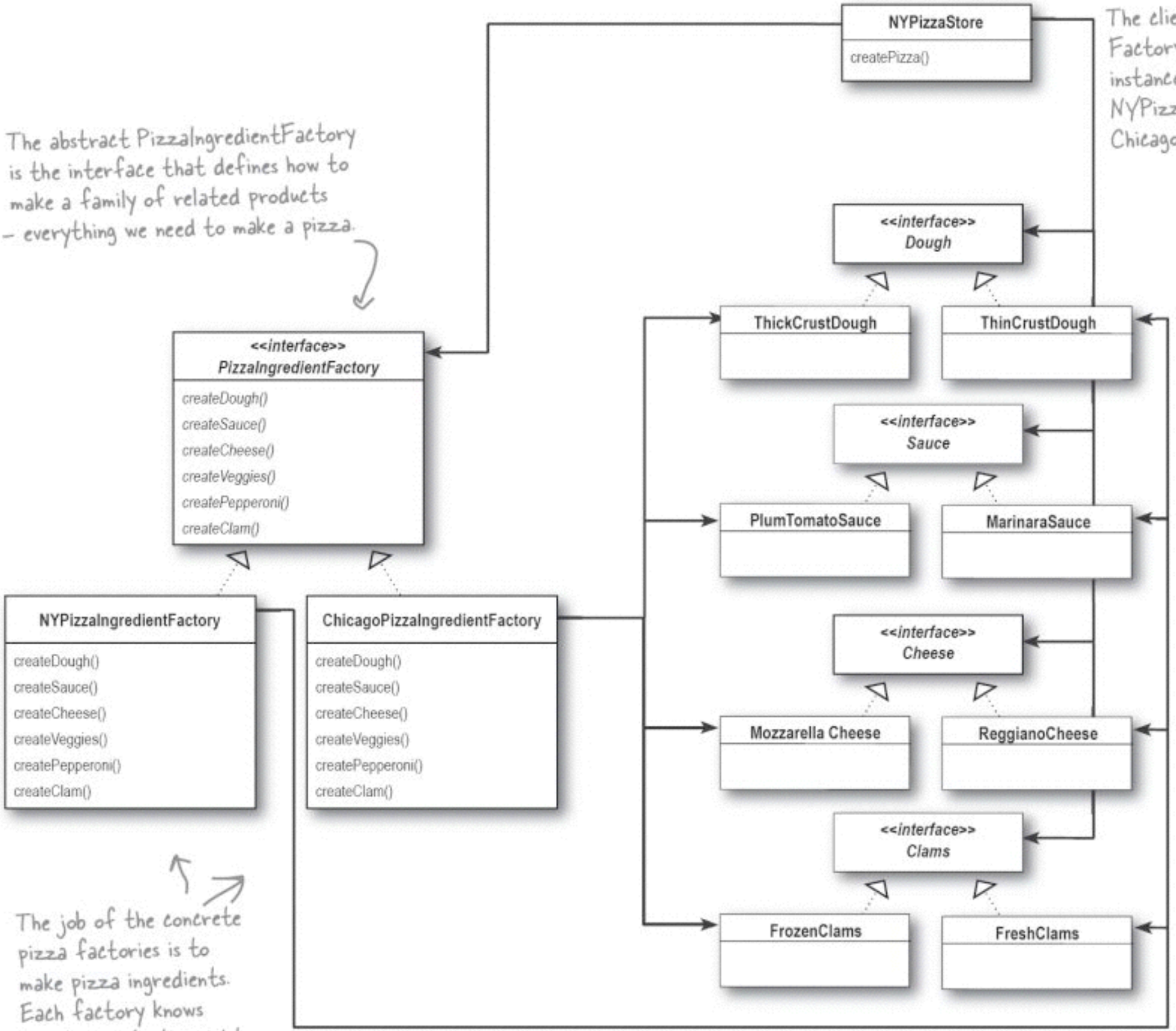
For Ethan's pizza the New York ingredient factory is used, and so we get the NY ingredients.

Abstract Factory



The abstract PizzaIngredientFactory is the interface that defines how to make a family of related products – everything we need to make a pizza.

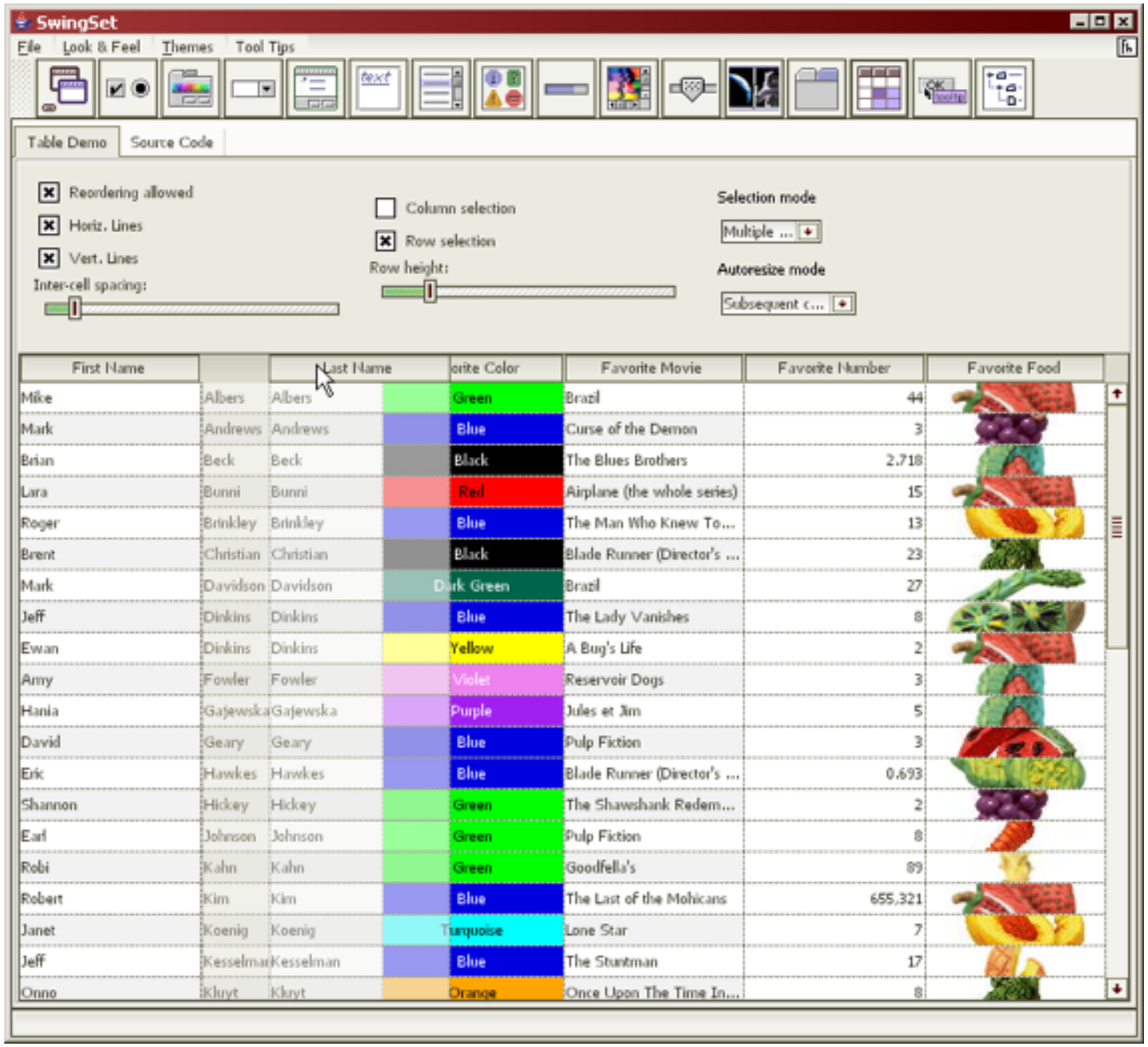
The clients of the Abstract Factory are the two instances of our PizzaStore, NYPizzaStore and ChicagoStylePizzaStore.



The job of the concrete pizza factories is to make pizza ingredients. Each factory knows how to create the right objects for their region.

Each factory produces a different implementation for the family of products.

Abstract Factory: Java Swing



There is a problem

- Is the use of factory we have seen so far really closed for modification?

Prototype

```
interface Pizza{
    Pizza clone();
}

class CheesePizza implements Pizza{
    public Pizza clone(){
        return this;
    }
}
```

In main:

```
Class c = Class.forName(classname);
Pizza p = (Pizza) c.newInstance();
simplefactory.initialize("cheesepizza", p);
```

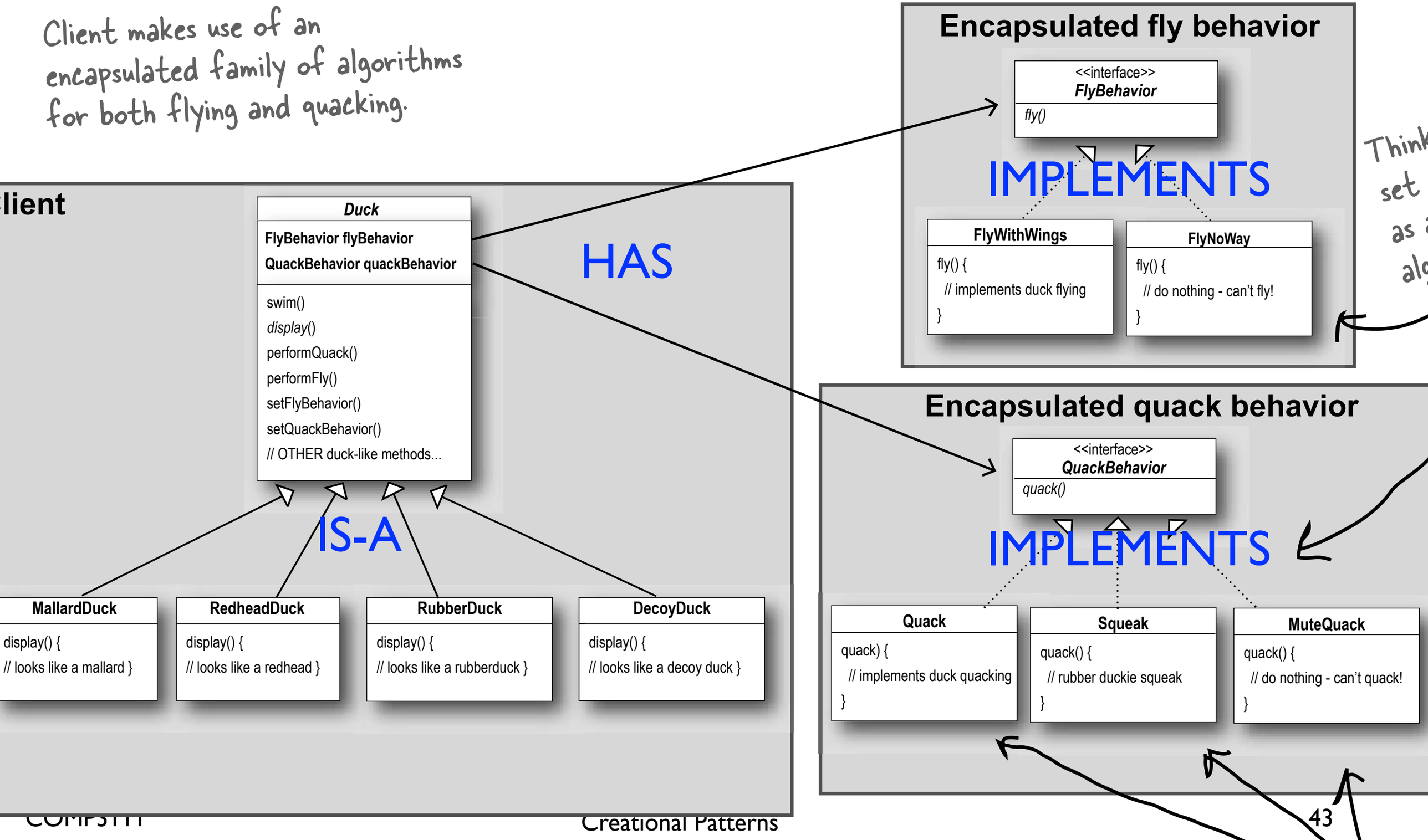
In a file:

```
cheesepizza, ust.comp3111.CheesePizza;
veggiepizza,use.comp3111.VeggiePizza;
```

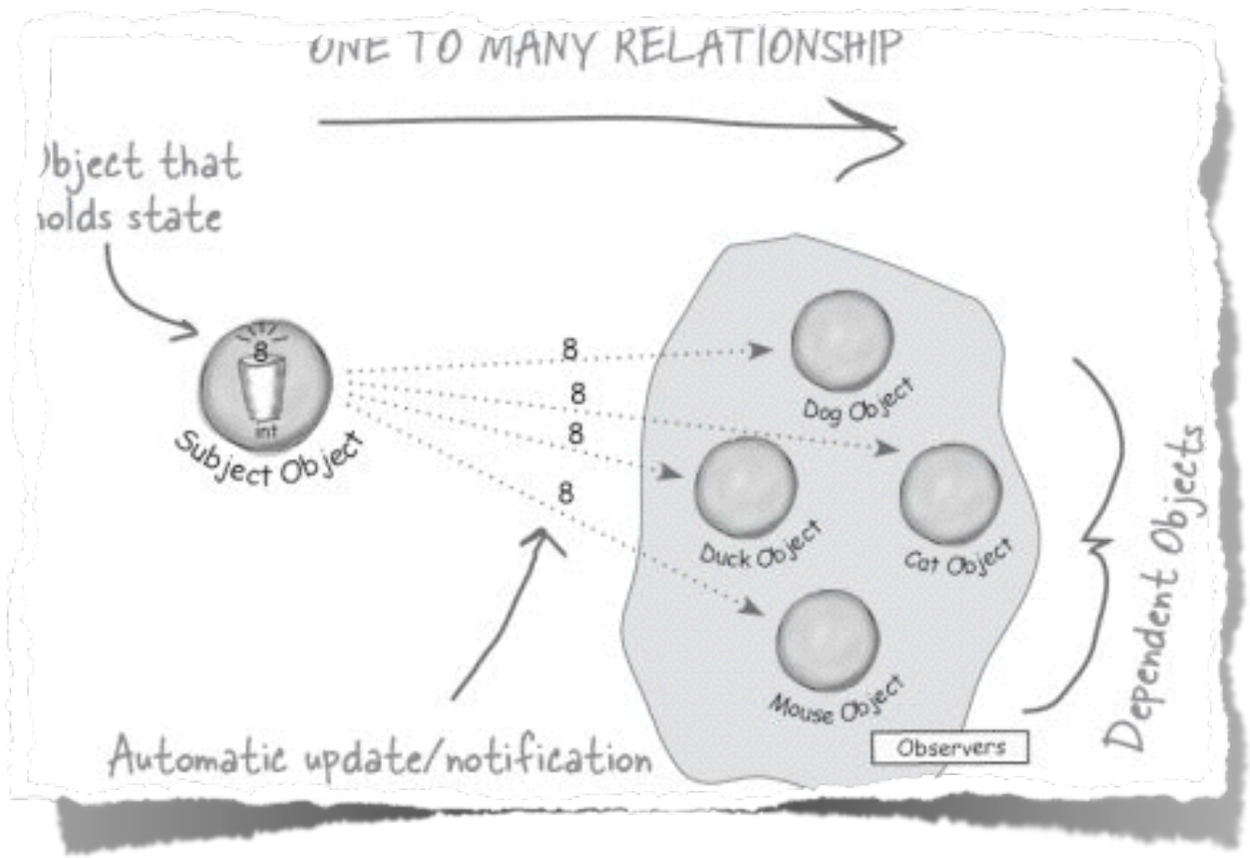
```
class SimpleFactory {
    Hashtable t;
    void initialize(String s, Pizza p){
        t.put(s, p);
    }
    Pizza create(String s){
        t.get(s).clone();
    }
}
```

Strategy pattern

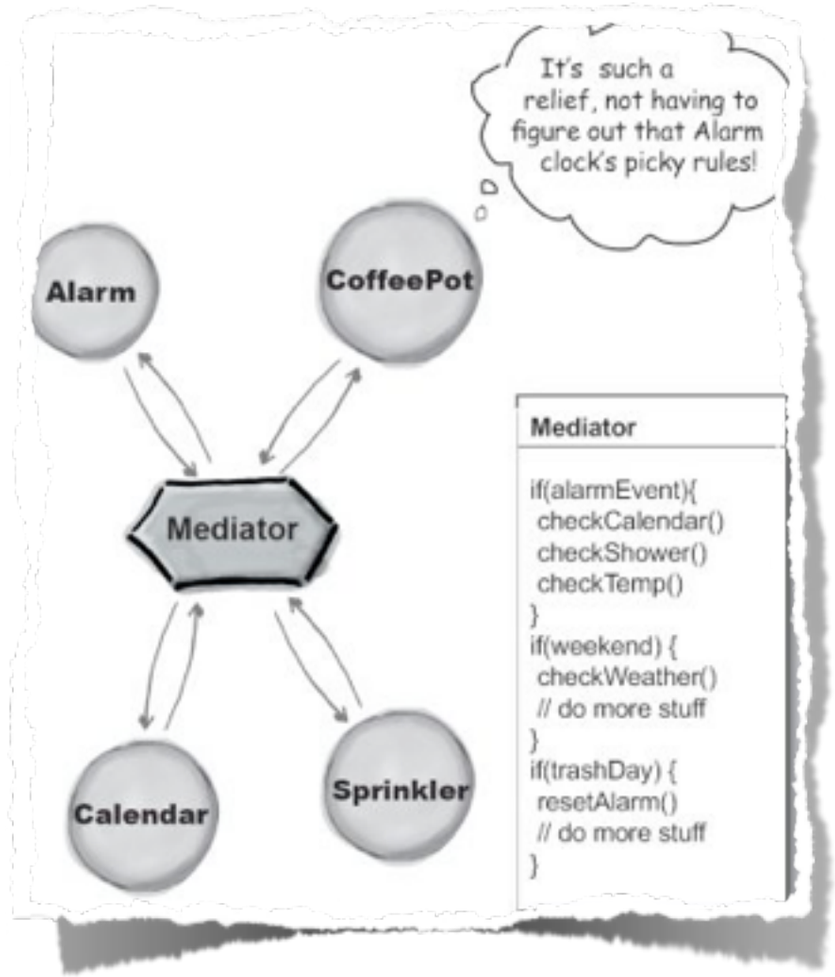
Client makes use of an encapsulated family of algorithms for both flying and quacking.



Behavioral Patterns

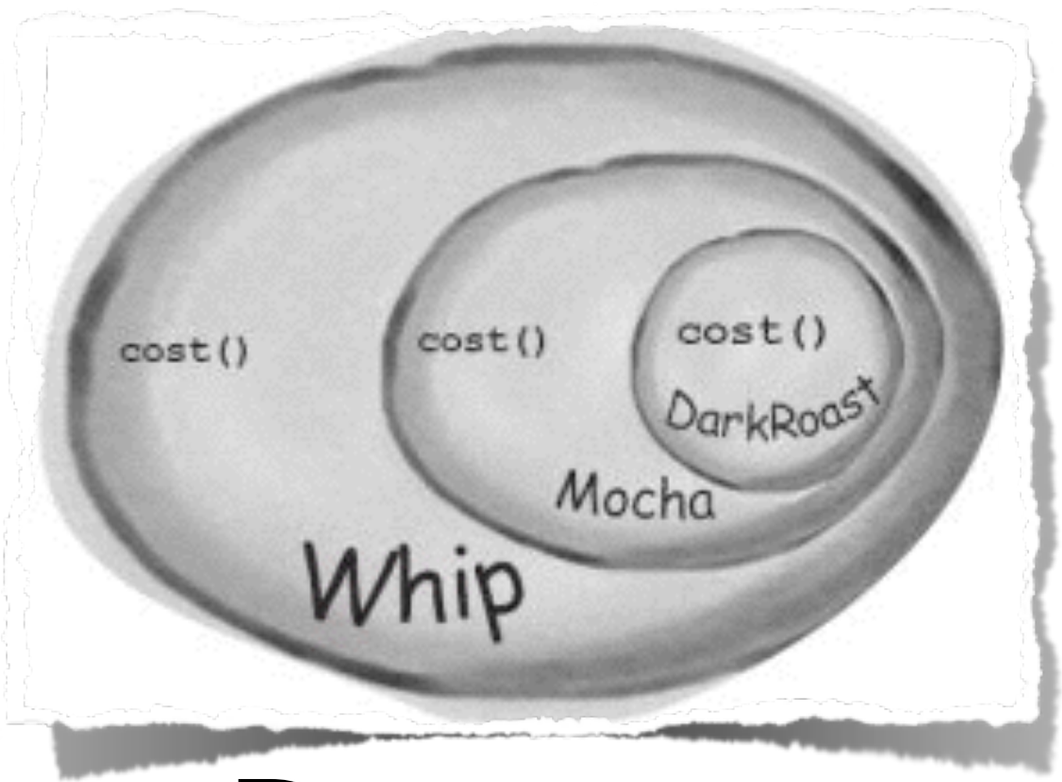


Observer

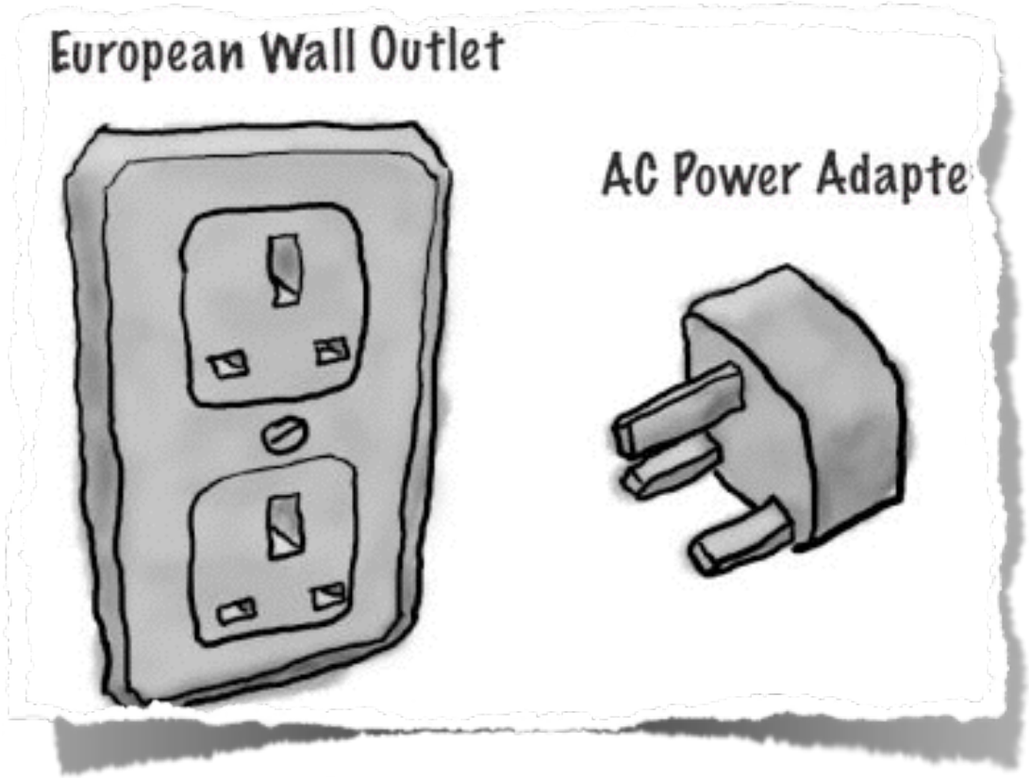


Mediator

Structural Patterns

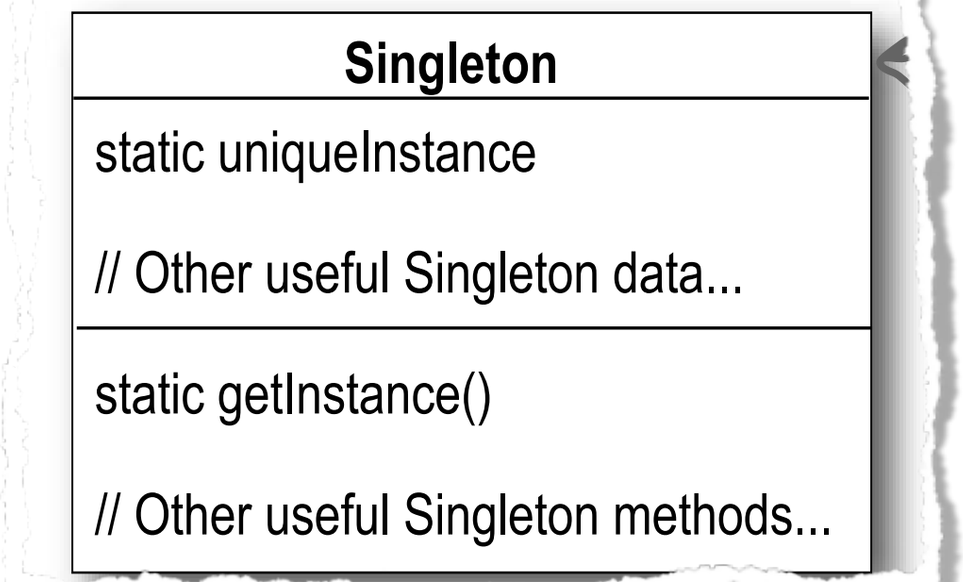


Decorator

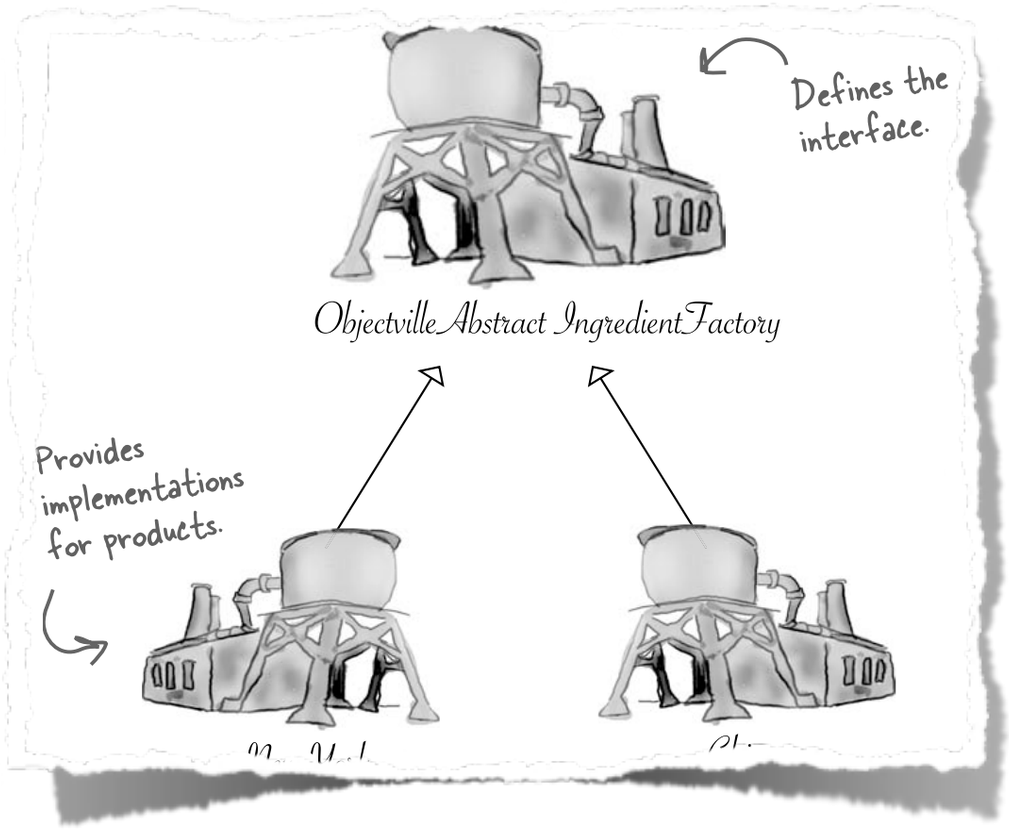


Adapter

Summary



Singleton



Factory

Design Principles

Design Principle

Identify the aspects of your application that vary and separate them from what stays the same.

Design Principle

Favor composition over inheritance.

Design Principle

Program to an interface, not an implementation.

Design Principle

Strive for loosely coupled designs between objects that interact.

Design Principle

Classes should be open for extension, but closed for modification.

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.

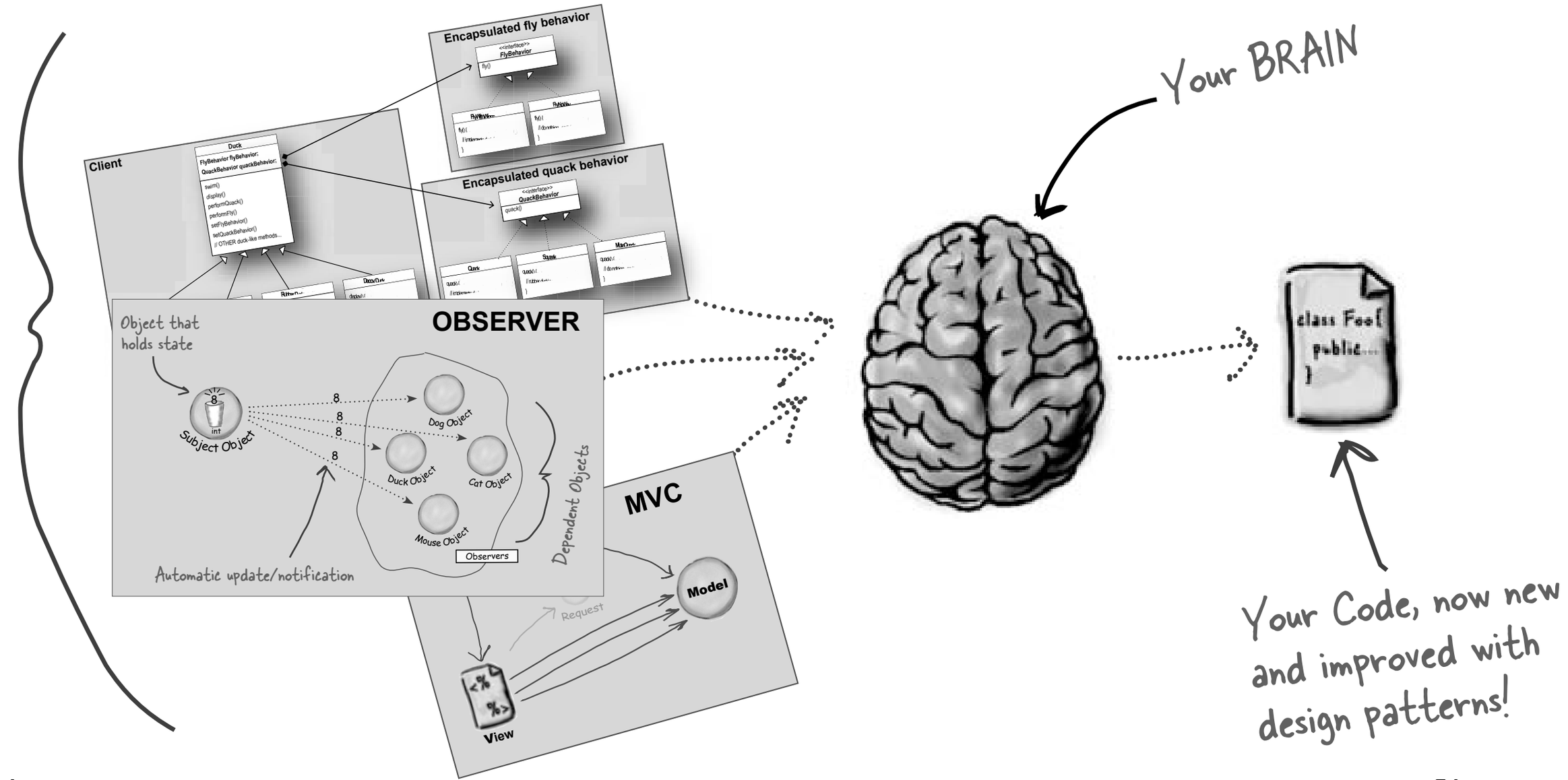
Thinking in Patterns

- Keep it Simple (KISS)
- Design patterns aren't a silver bullet
  - In fact, they are not even a bullet!
- Take out what you don't really need
  - Don't be afraid to remove a design pattern
- If you don't need it now, don't do it now

WARNING: Overuse of design patterns can lead to code that is downright over-engineered. Always go with the simplest solution that does the job and introduce patterns where the need emerges.

Of course we want you to use Design Patterns!

How to use Design Patterns?



Shared vocabulary

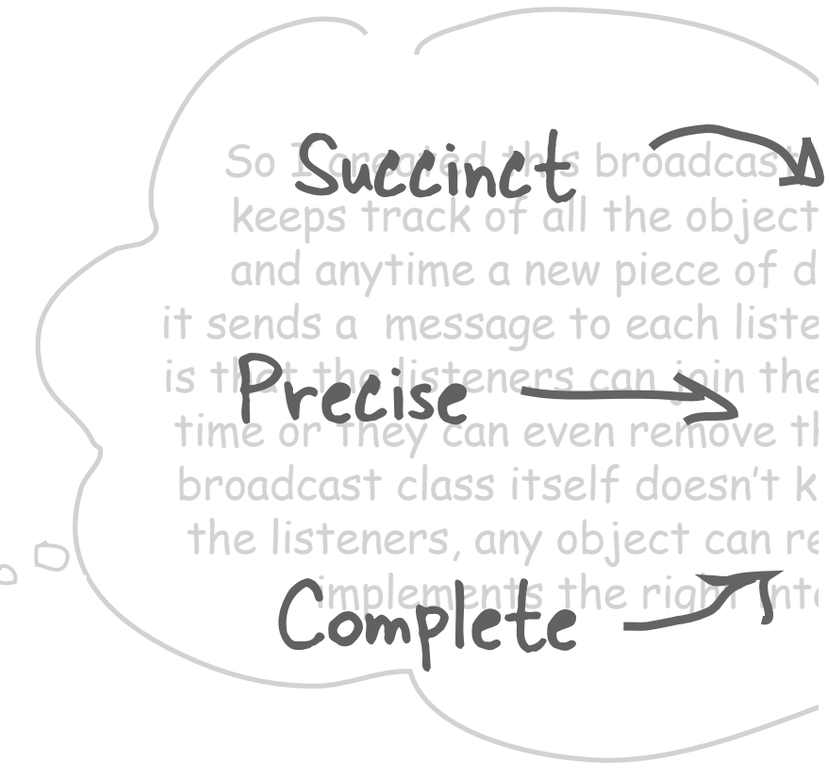
So I created this broadcast class. It keeps track of all the objects listening to it and anytime a new piece of data comes along it sends a message to each listener. What's cool is that the listeners can join the broadcast at any time or they can even remove themselves. And the broadcast class itself doesn't know anything about the listeners, any object can register that implements the right interface.

Incomplete

Confusing

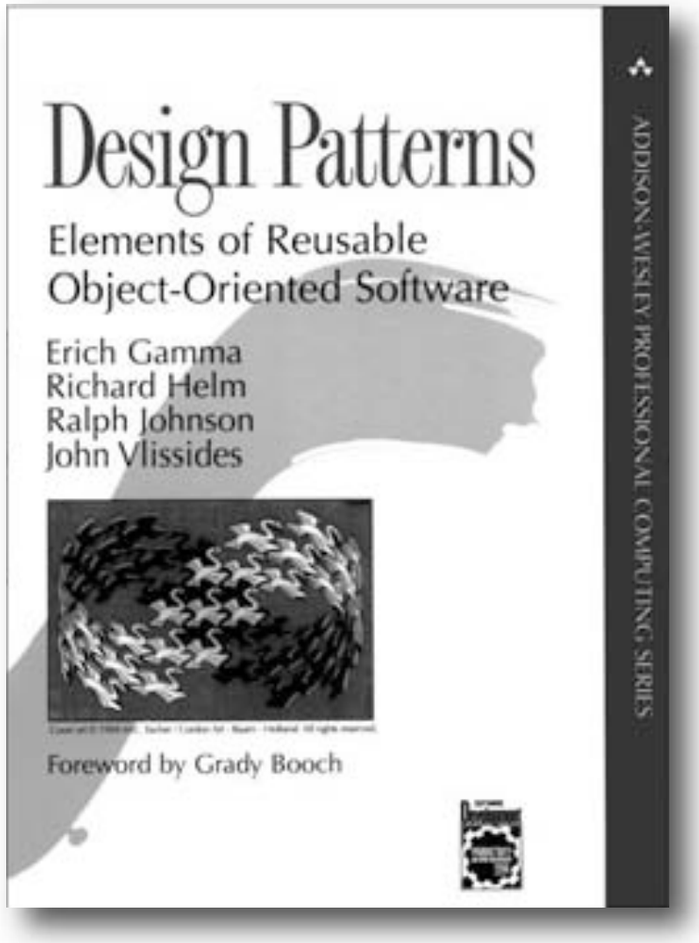
Time-consuming

Shared vocabulary



Your journey has just begun...

Now that you're on top of Design Patterns and ready to dig deeper, we've got three definitive texts that you need to add to your bookshelf...



The definitive Design Patterns text

This is the book that kicked off the entire field of Design Patterns when it was released in 1995. You'll find all the fundamental patterns here. In fact, this book is the basis for the set of patterns we used in *Head First Design Patterns*.

You won't find this book to be the last word on Design Patterns – the field has grown substantially since its publication – but it is the first and most definitive.

Picking up a copy of *Design Patterns* is a great way to start exploring patterns after Head First.

The authors of Design Patterns are affectionately known as the "Gang of Four" or GoF for short.