Q1

Which of the following are disadvantages of using *inheritance* to provide Duck behavior? (Choose all that apply.)

- ❑ A. Code is duplicated across subclasses.
- ❑ B. Runtime behavior changes are difficult.
- ❑ C. We can't make ducks dance.
- ❑ D. Hard to gain knowledge of all duck behaviors.
- ❑ E. Ducks can't fly and quack at the same time.
- ☑ F. Changes can unintentionally affect other ducks.

Q2

Lots of things can drive change. List some reasons you've had to change code in your applications (we put in a couple of our own to get you started).

My customers or users decide they want something else, or they want new functionality.

My company decided it is going with another database vendor and it is also purchasing its data from another supplier that uses a different data format. Argh!

improvements of previous version

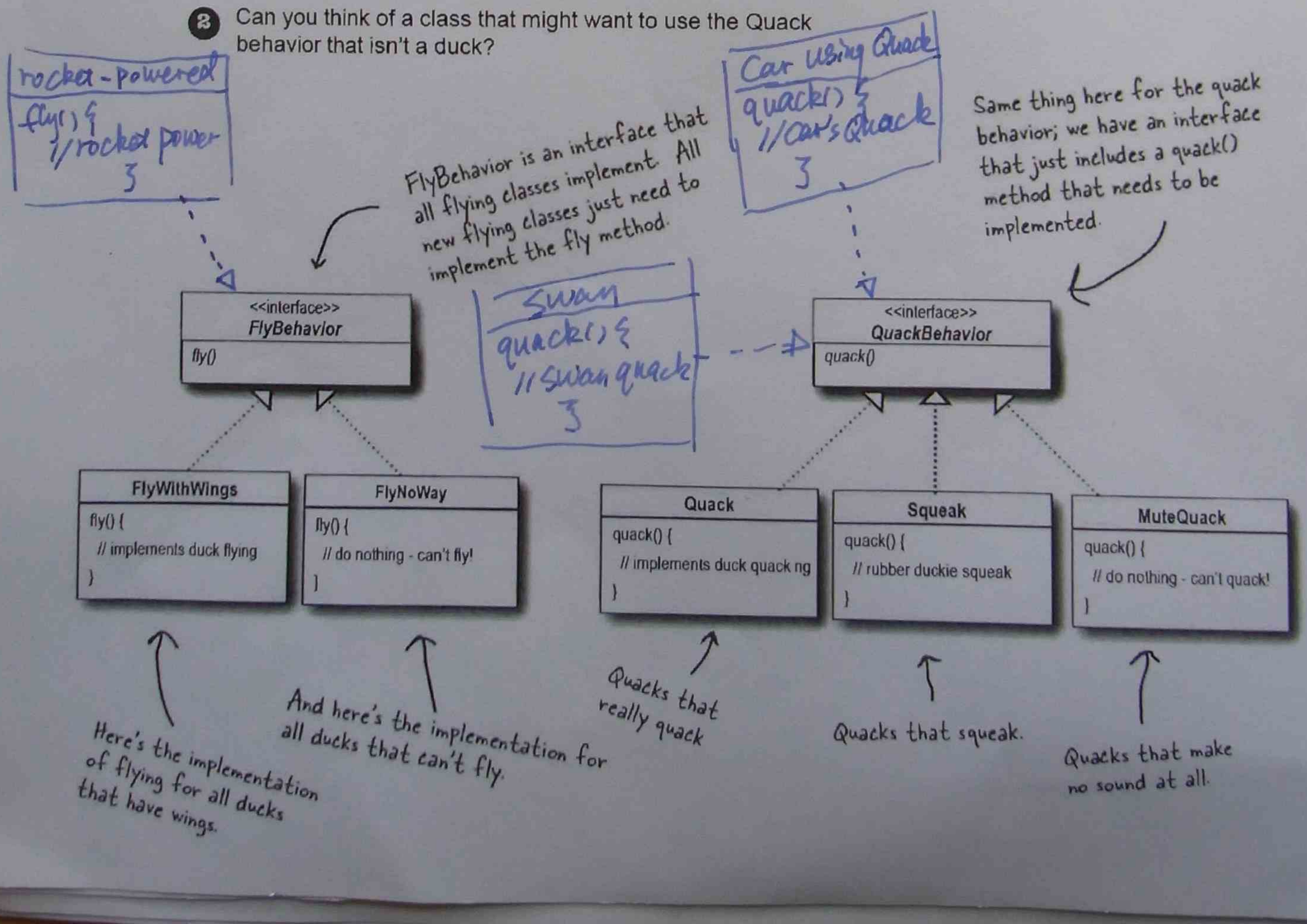
different users require different client's functions

A better or just want to change to new algorithm

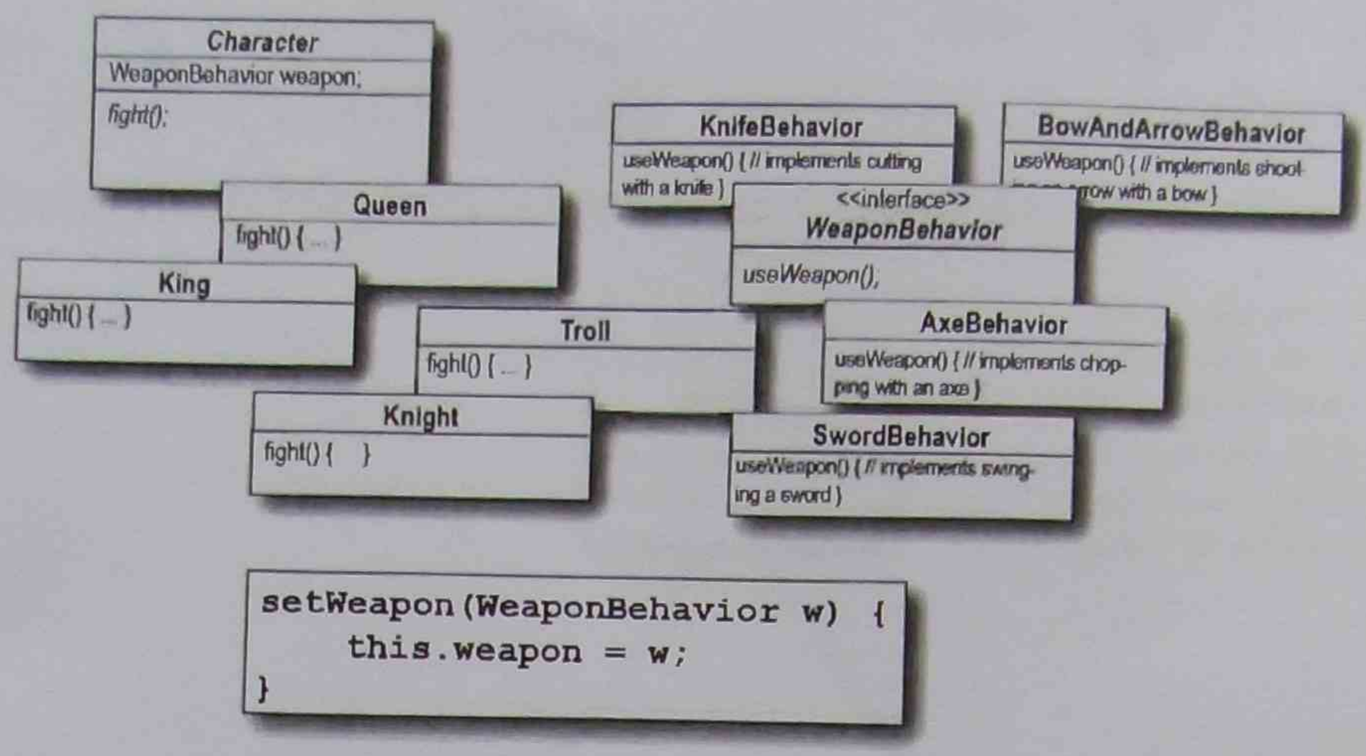
Q3

❶ Using our new design, what would you do if you needed to add rocket-powered flying to the SimUDuck app?

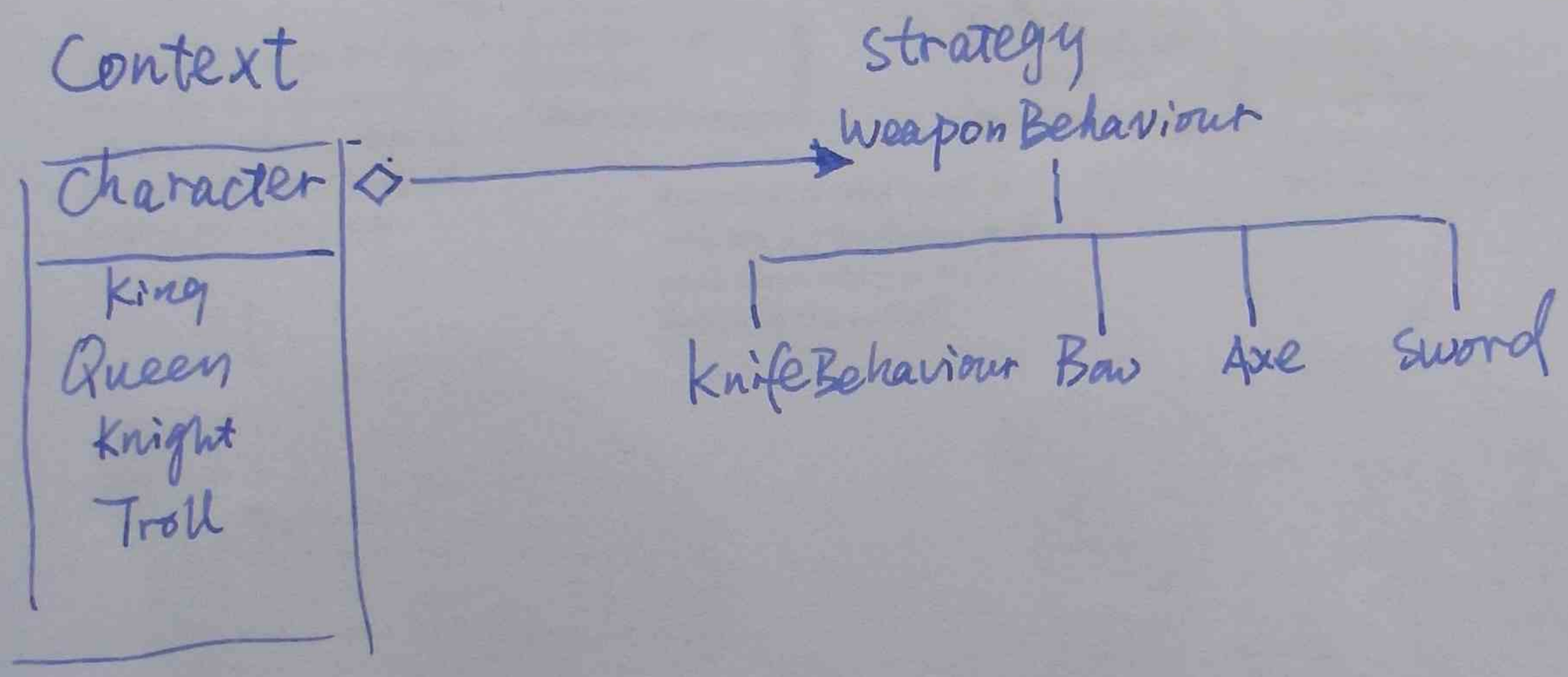
❷ Can you think of a class that might want to use the Quack behavior that isn't a duck?



Q4 Design Puzzle



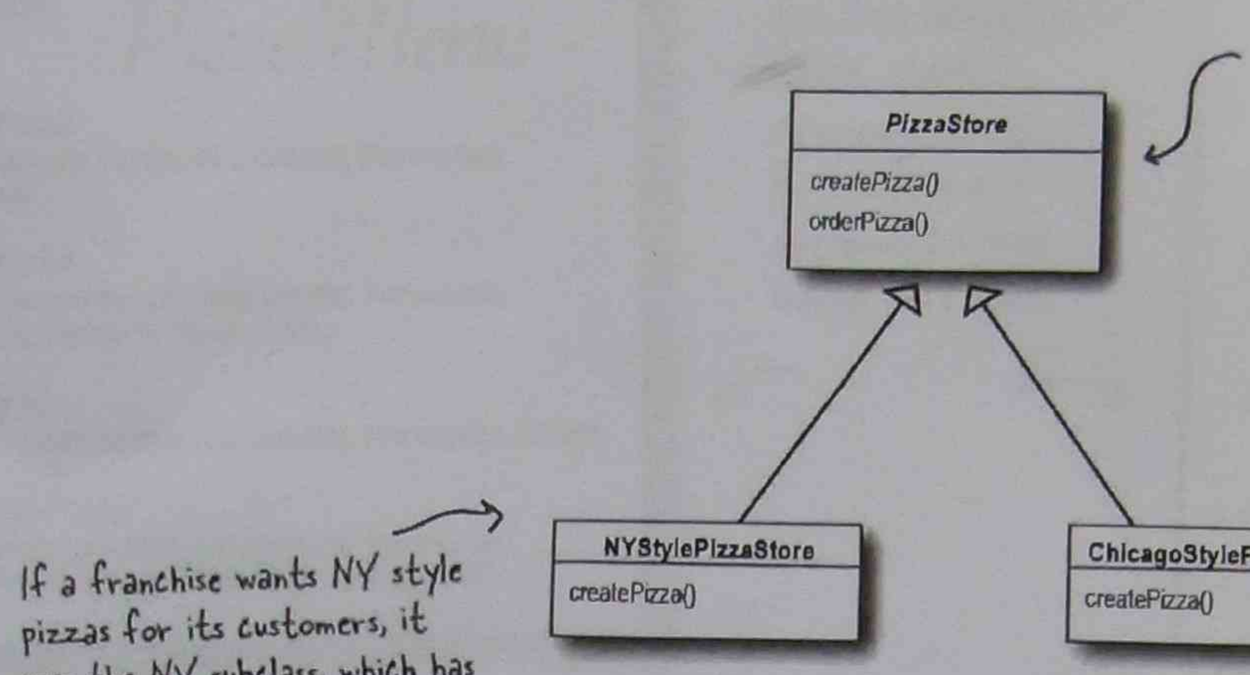

```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```



H1 What are the advantages of *SimplePizzaFactory* over *PizzaStore*?

It gives more flexibility and variety for Pizza types. The Customer can easily choose differe types of Pizza and the manufacturer can also easily customise recipe.

H2 Write the *NYStylePizzaStore* and *ChicagoStylePizzaStore* implementations



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.

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

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

```
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();
    }
}
```

It's already done. What do you want us to write or answer?

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 Pizza 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.

H3 Ensure consistency in your ingredients

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

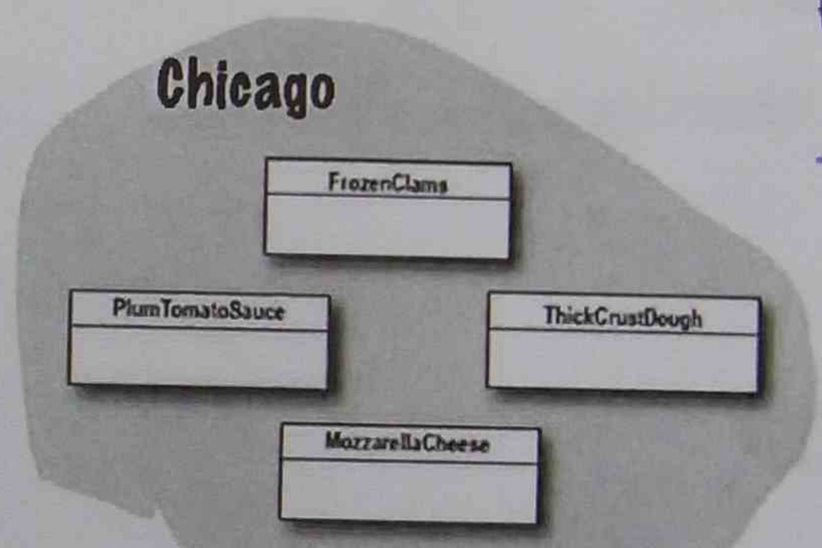
Clam Pizza
Marinara Sauce, Reggiano, Fresh Clams

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

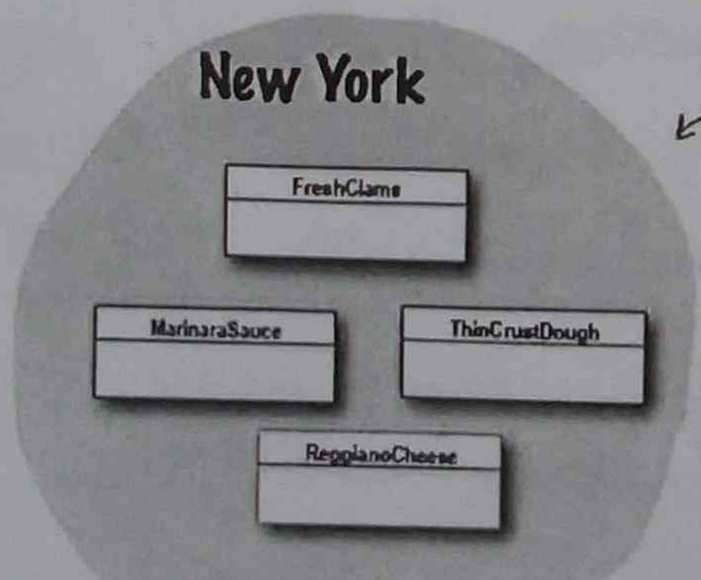
14 AUG 2013 16:24

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.



FrozenClams
PlumTomatoSauce
ThickCrustDough
MozzarellaCheese



FreshClams
MarinaraSauce
ThinCrustDough
ReggianoCheese

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

Calamari
BruschettaSauce
VeryThinCrust
GoatCheese

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

```
interface ingredients ( ) {
    Sauce Region;
    Cheese Region
    Dough Region
    Seafood Region }

createPizza extends ingredients ( ) {

}
```

Hint: use the *Abstract factory* design pattern

H2 H3

```
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 Pizza createPizza(String type);
}
```

```
Public interface ingredients(){
   sauce (String region);
   Cheese (String region);
   Dough (String region);
   Seafood (String region);
}
```

14 AUG 2013 16:24

```
Public NYStylePizzaStore extends Pizzastore{
    inherit orderPizza ...
    public Pizza createPizza(type) extends ingredients {
        if (type.equals("cheese")){
            pizza = new NYStyleCheesePizza();
        } else if (type.equals(      ){
        ... ...
}
```

```
NYStyleCheesePizza extends ingredients {
    sauce ("New York") {
      return MarinaraSauce;
    }
    .... other three inherited methods
    ..... Other methods for CheesePizza.
}
```

ChicagoStylePizzaStore do the similar things as NYStyle.

H4 Transform *ChocolateBoiler* according to the Singleton design pattern

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            System.out.println("Creating unique instance of Chocolate Boiler");
            uniqueInstance = new ChocolateBoiler();
        }
        System.out.println("Returning instance of Chocolate Boiler");
        return uniqueInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}
```