Design Patterns

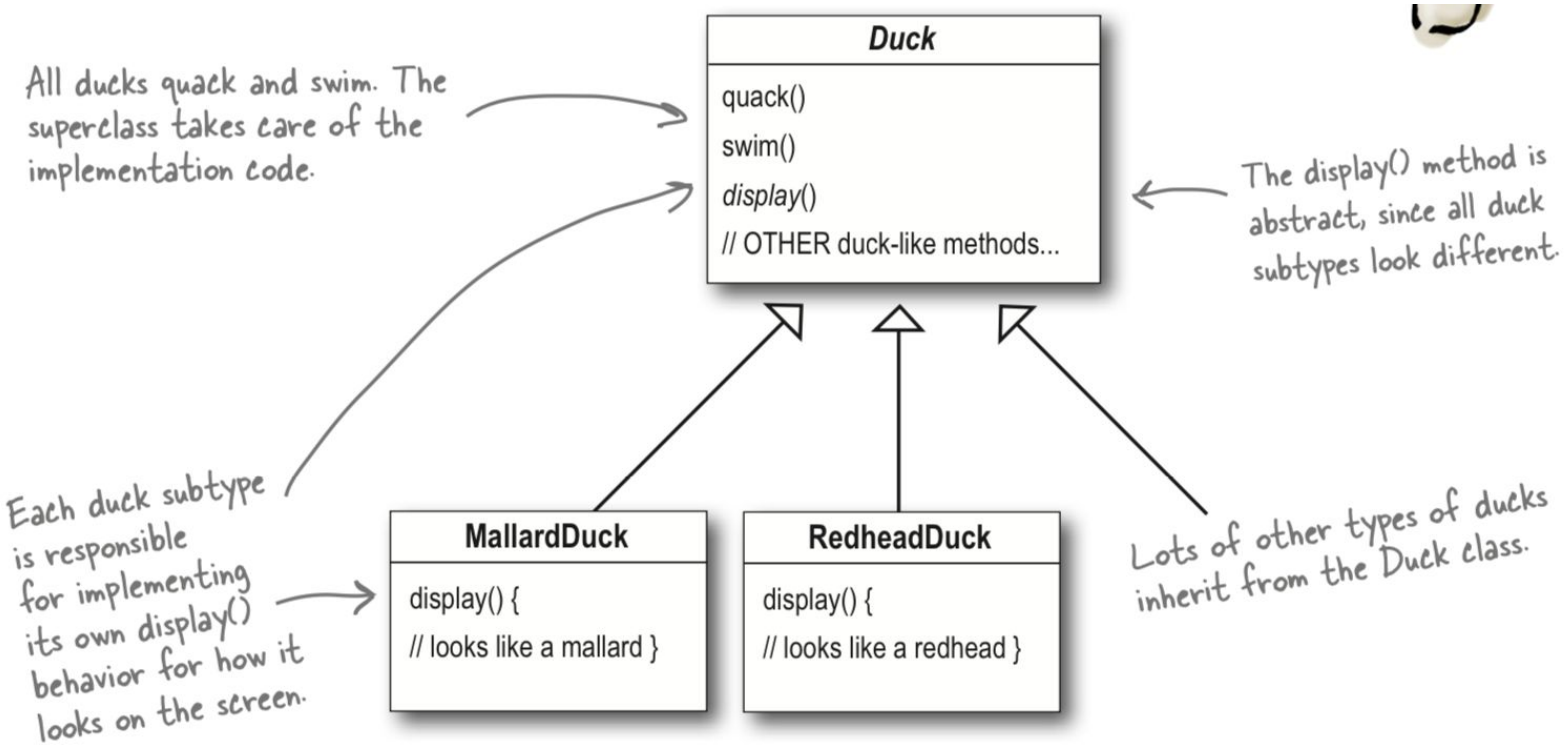
Design Patterns

Someone has already solved your problems !!

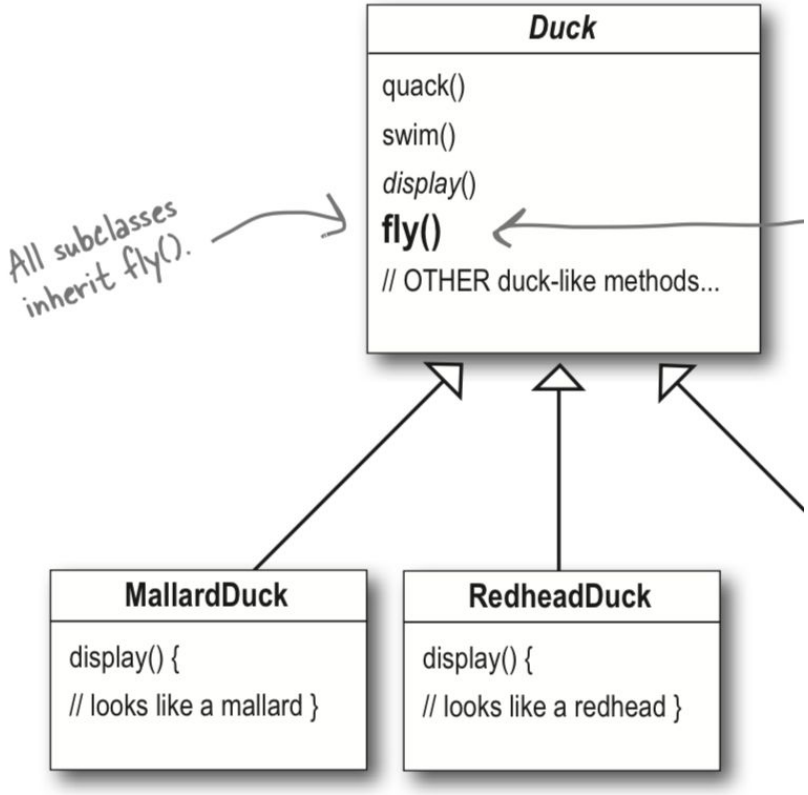
Duck Simulator Example (source: Head First Design Patterns, 2nd ed.)

- This simulator is for showing large variety of ducks quacking, flying etc.
- This is not for one type of duck. Many different variety of ducks
  - eg. Mallard duck, Redhead duck

Class Diagram: Design 1:

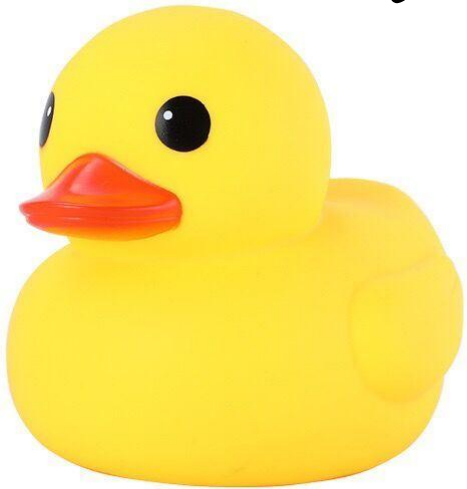


Design 1: Extensibility



Design 1: Flaw

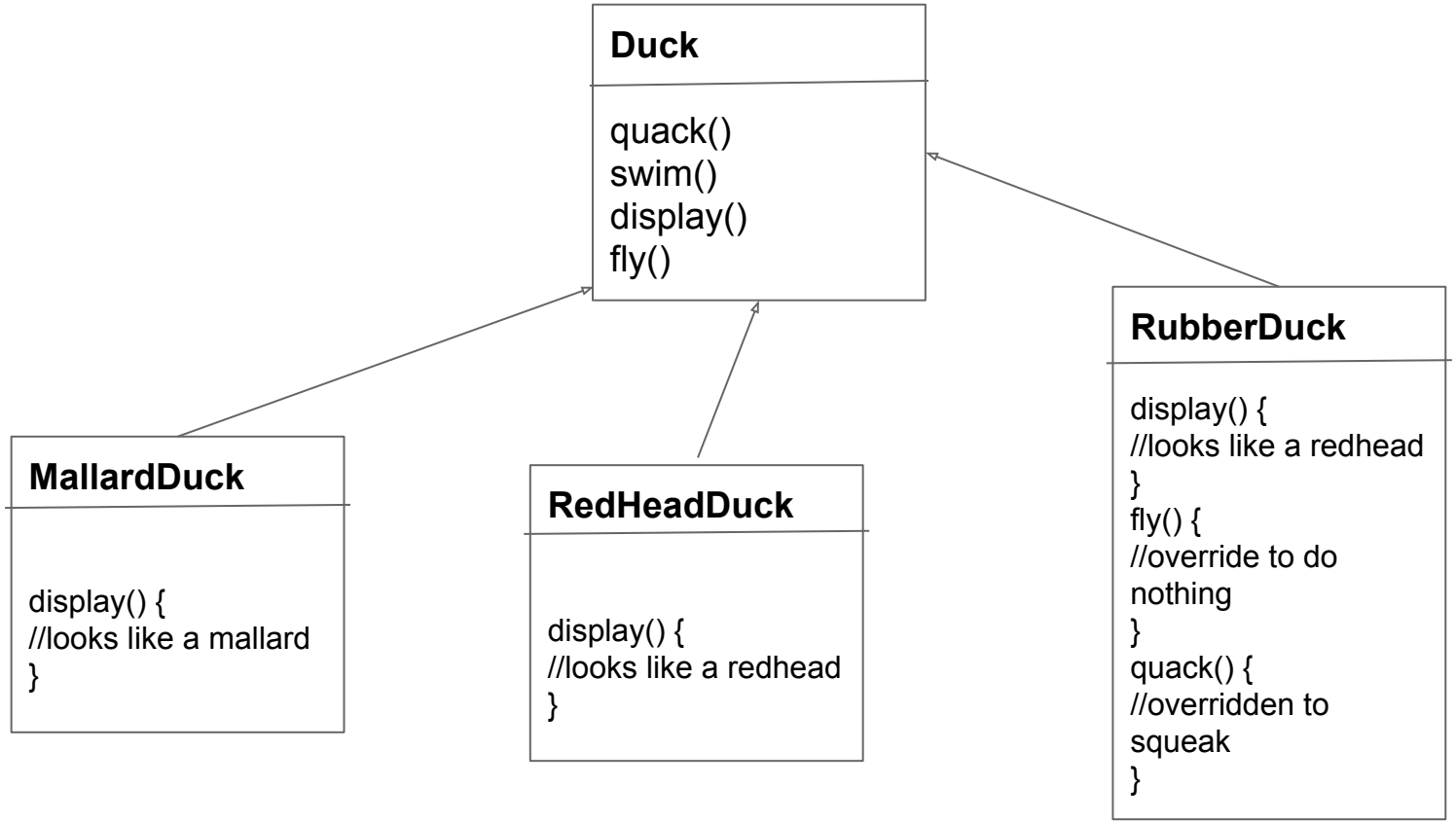
Can't fly



Rubber duck

Can Quack

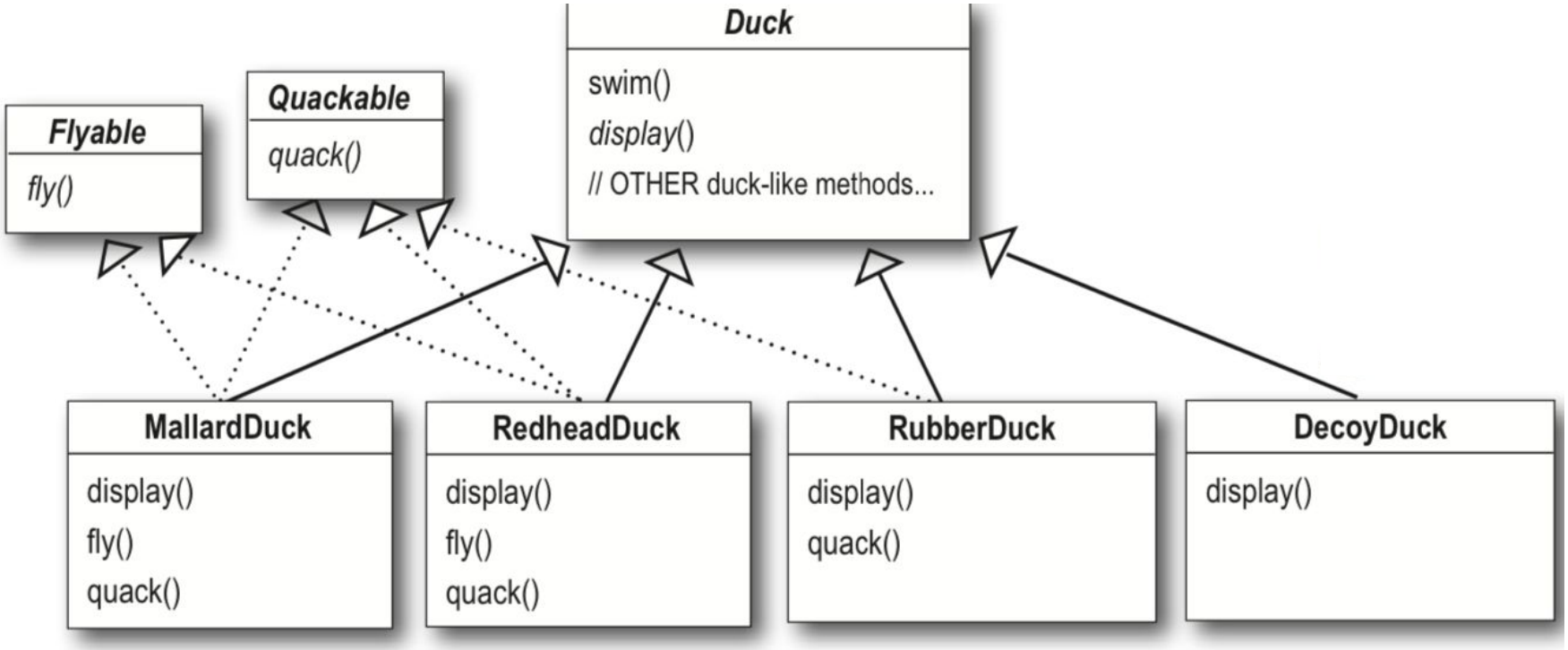
Design 2: Solution to Design 1 Flaw



Design 2: Flaw

- Overriding solution is not a cleaner way
- There will be issues with maintenance when there are modifications in the future.
- Whenever a new Duck subclass is added all the required overriding have to be done.

Design 3: Interfaces of Changing Behaviours



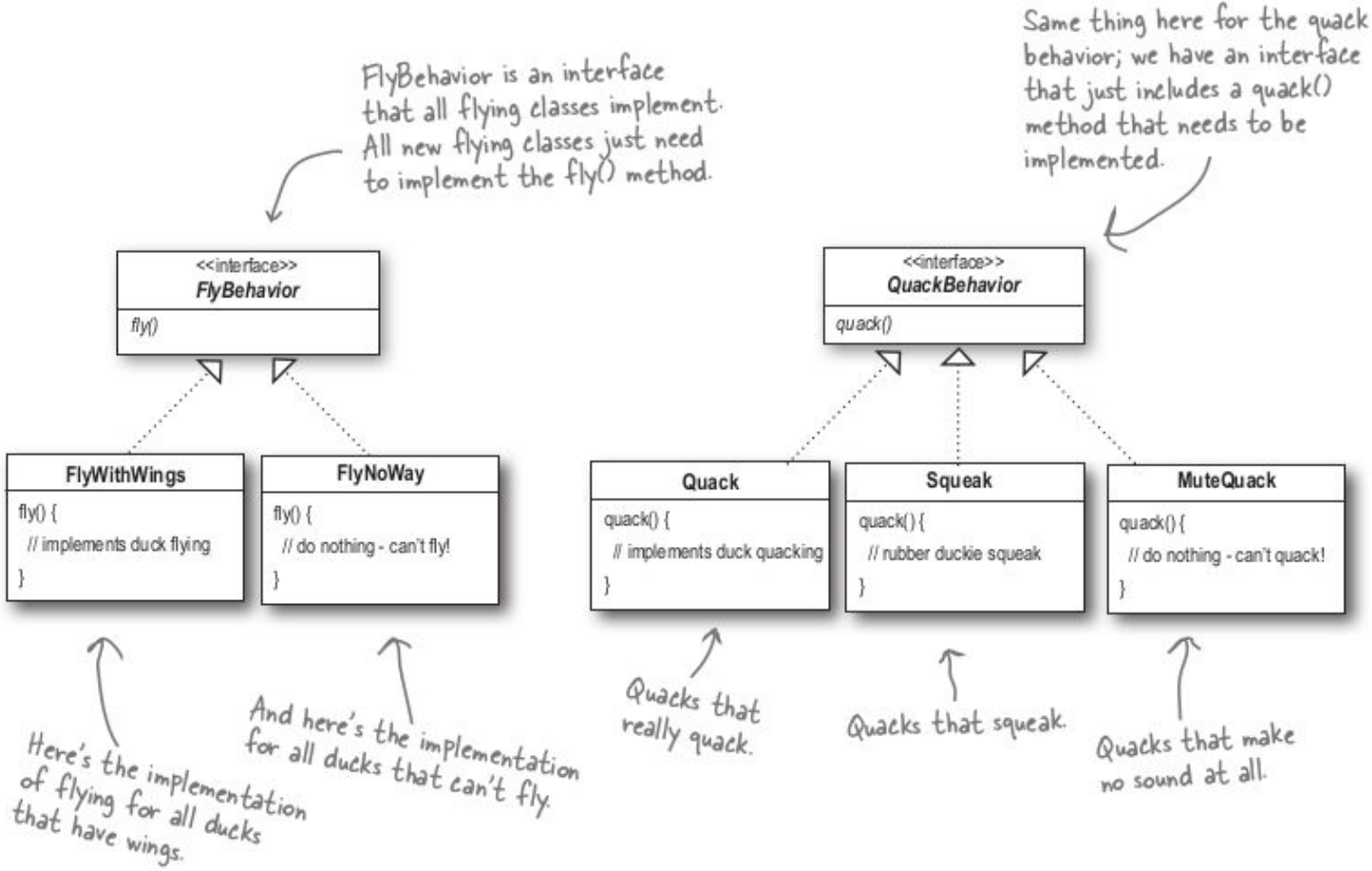
Design 3: Flaw

- No code reuse
- The fly method has to be implemented in all the subclasses even for the Ducks with the same fly method.
- For most of the modifications, we will have to make the modifications in multiple subclasses

The Final Solution

- The Duck subclasses will use interface for the behaviors (FlyBehavior, QuackBehavior)
- The concrete behavior is coded in the class that implements the above interfaces
- The actual behavior is not locked in the Duck subclasses

The Final Solution



Final Solution: Duck class

```
public abstract class Duck {
  FlyBehavior flyBehavior;
  QuackBehavior quackBehavior;

public Duck() { }
public abstract void display();

public void performFly() {
    flyBehavior.fly();
}

public void performQuack() {
    quackBehavior.quack();
}
}

public void swim() {
    System.out.println("All ducks
float, even decoys!");
}
}
```

Final Solution: Mallard Duck

```
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

Main Code in Simulator

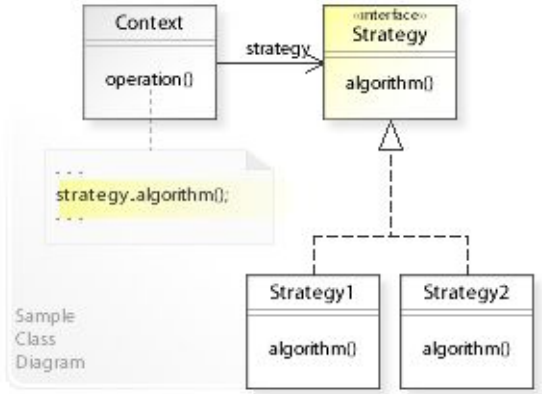
```
public class MiniDuckSimulator {

public static void main(String[] args) {
    Duck mallard = new MallardDuck();
    mallard.performQuack();
    mallard.performFly();
}
```

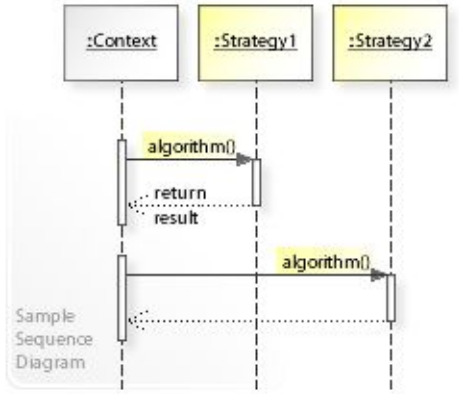
https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/strategy

What we were discussing till now?

Strategy Design Pattern

Is a Behavioral design pattern in which enables selecting the algorithm at runtime.





Source: https://en.wikipedia.org/wiki/Strategy_pattern

Strategy Design Pattern

The behaviors of a class should not be inherited. Instead, they should be encapsulated using interfaces.

Types of Design Patterns

- **Creational**

  Provides object or class creation mechanism
- **Structural**

  assemble object and classes into a structure ensuring that the structure should be flexible and efficient
- **Behavioural**

  Manages how one class communicates with another

Creational Design Pattern

Factory Method Pattern

Separates out the creation of objects/instances.

Making Pizza Example

```
Pizza orderPizza() {
  Pizza pizza = new Pizza();

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

Instantiation of different classes for different types of pizzas

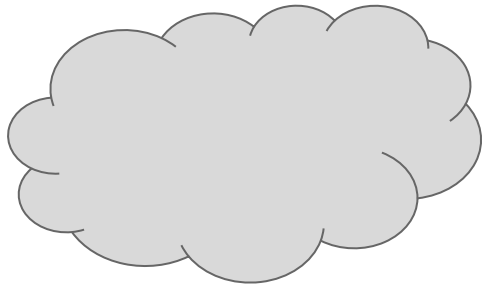
There is pressure to add more pizza types !!

```
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();
}
```

Encapsulating Object Creation

```
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
}
```

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



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

Simple Pizza Factory

```
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;
}
```

How does PizzaStore looks like?

First we give PizzaStore a reference to a SimplePizzaFactory.

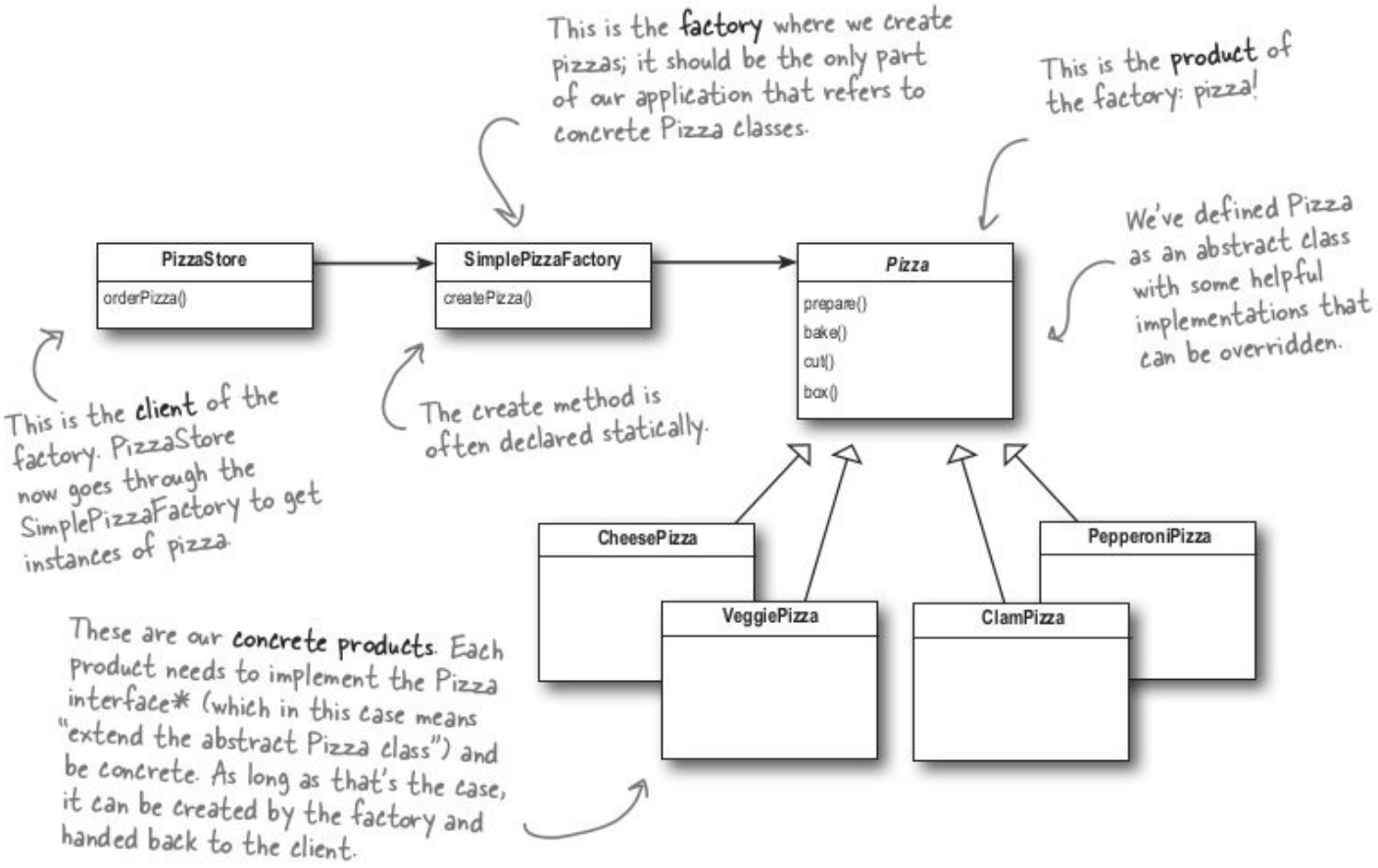
```
public class PizzaStore {
    SimplePizzaFactory factory;

    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.

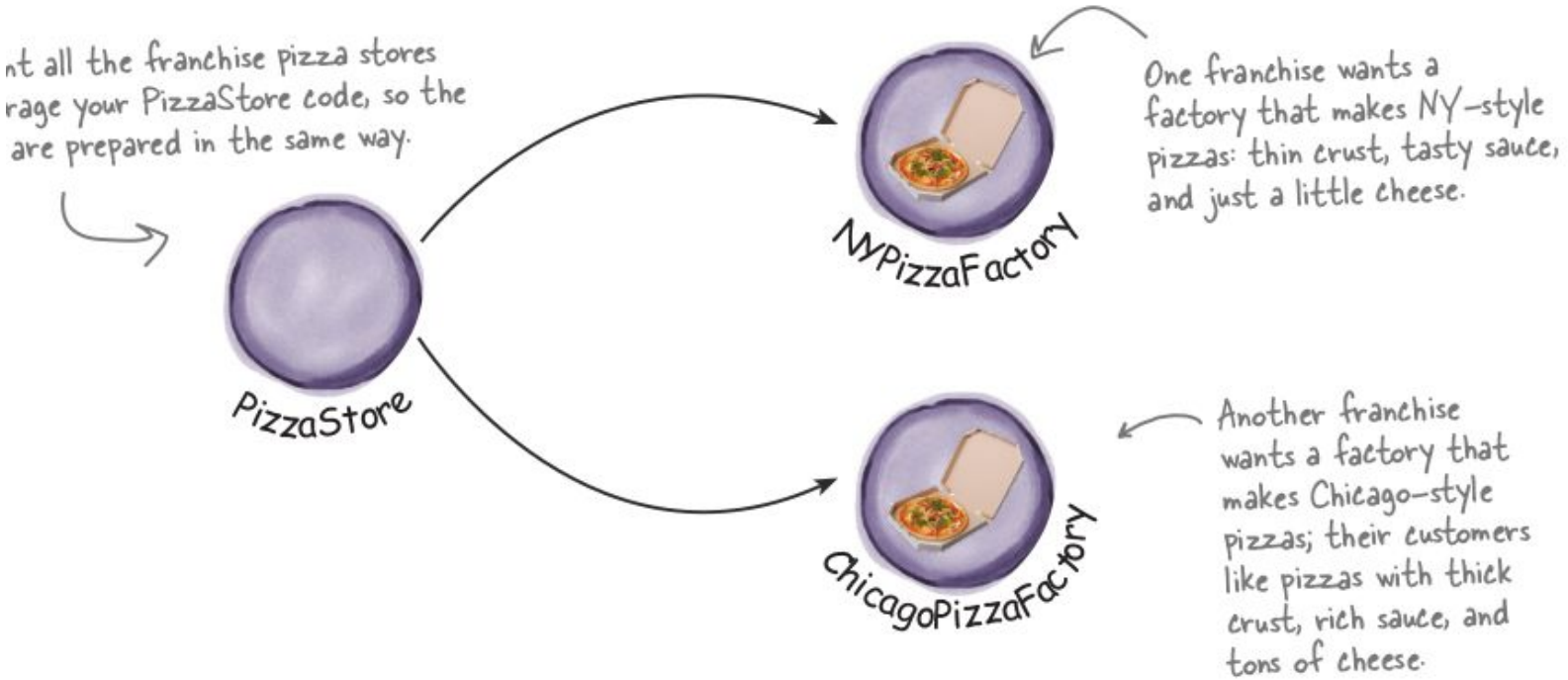
And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

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

Class Diagram of Pizza Store

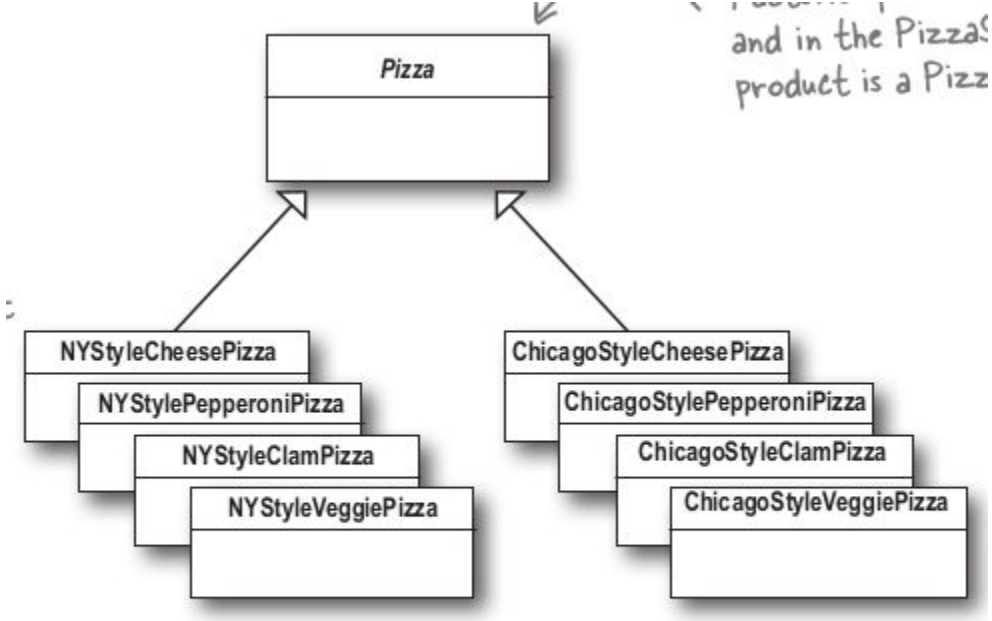


Different Style Pizza



a seen one approach

Pizza Class Diagram



New York Style

Cheese

Factory Classes for Different Styles

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.orderPizza("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.orderPizza("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-style ones.

A Different Design to Manage Styles

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);
}
```

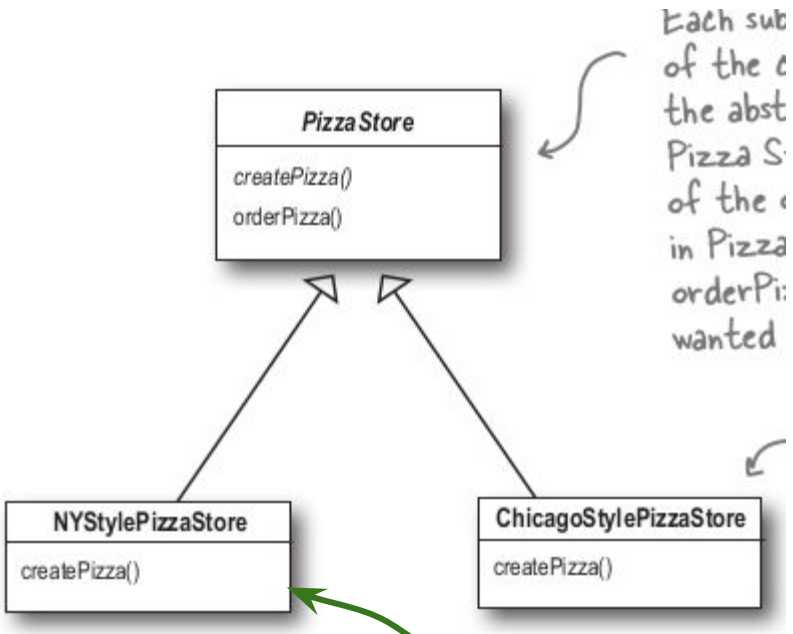
Now createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.

All this looks just the same...

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

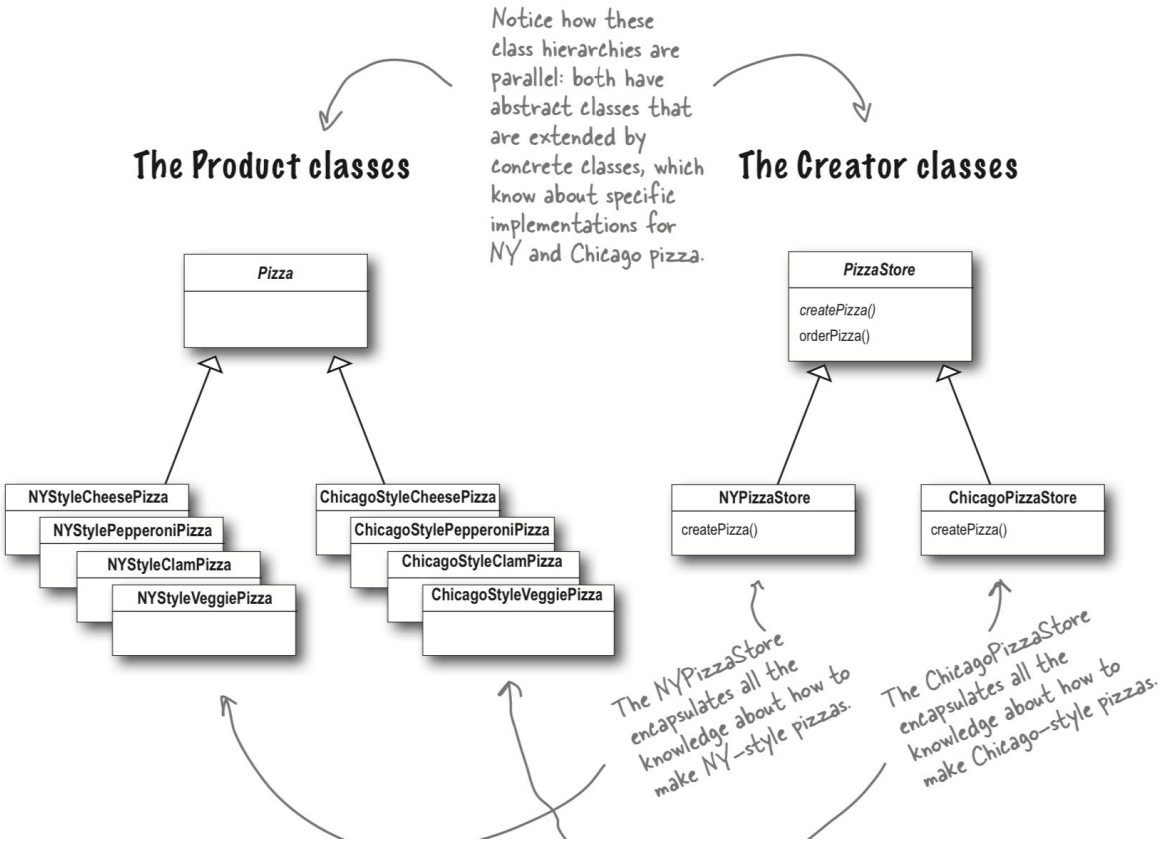
Our "factory method" is now abstract in PizzaStore.

Allow Subclasses to Decide

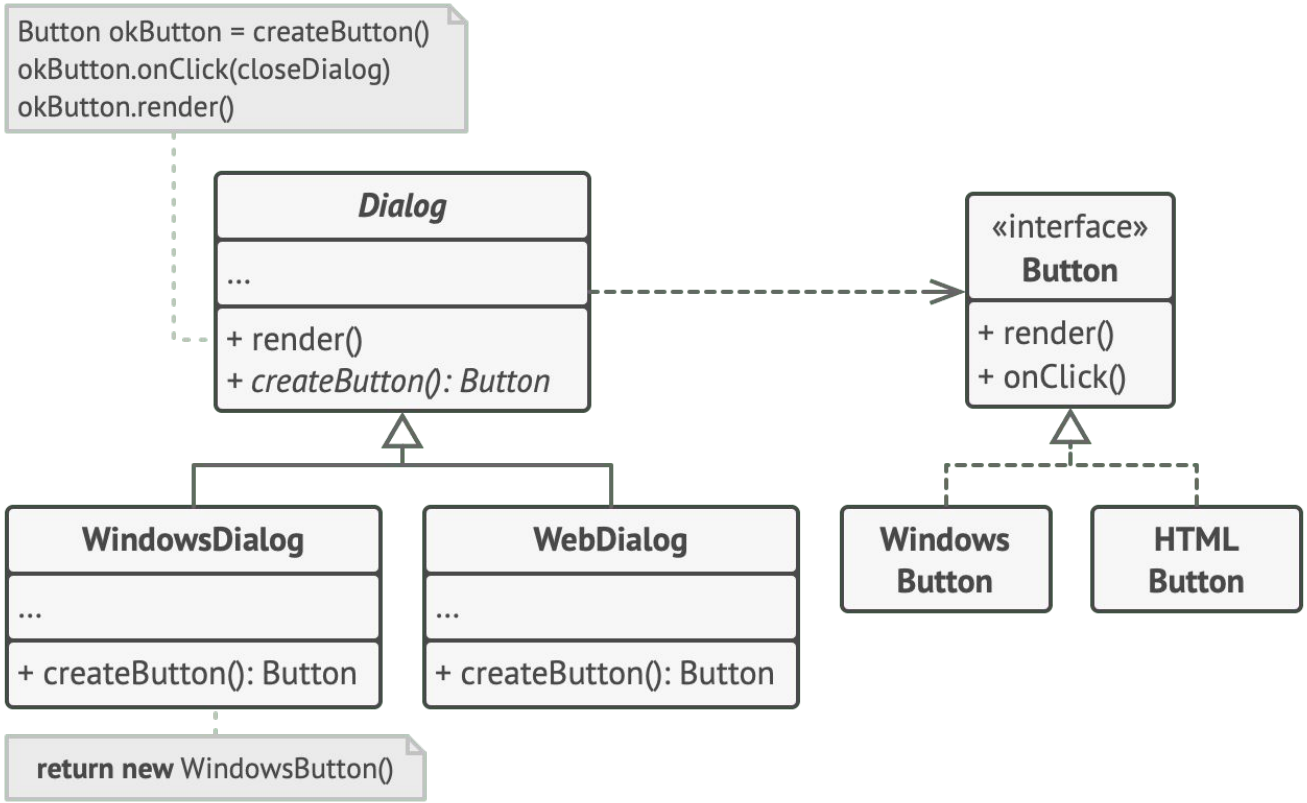



```
public class NYPizzaStore extends PizzaStore {

    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;
    }
}
```

Product and Creator Classes

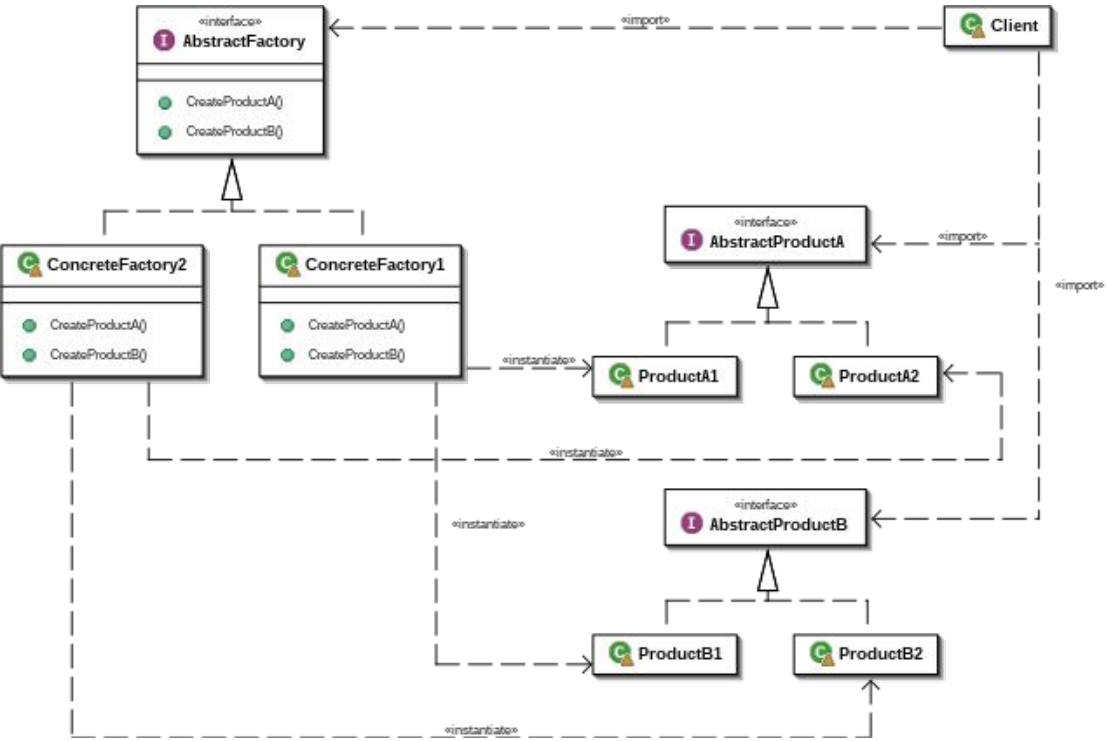


Another Example: Shapes



Source: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Abstract Factory Pattern



Factory Method pattern is responsible for creating products that belong to one family, while Abstract Factory pattern deals with multiple families of products.