Welcome to Design Pattern

What is a good design

Key Questions

- What is the difference between object-oriented code and *good* object-oriented code?
- Why is inheritance bad?
- What is programming to interface?
- What is design pattern and why is design pattern important, especially for young grasshoppers like you?

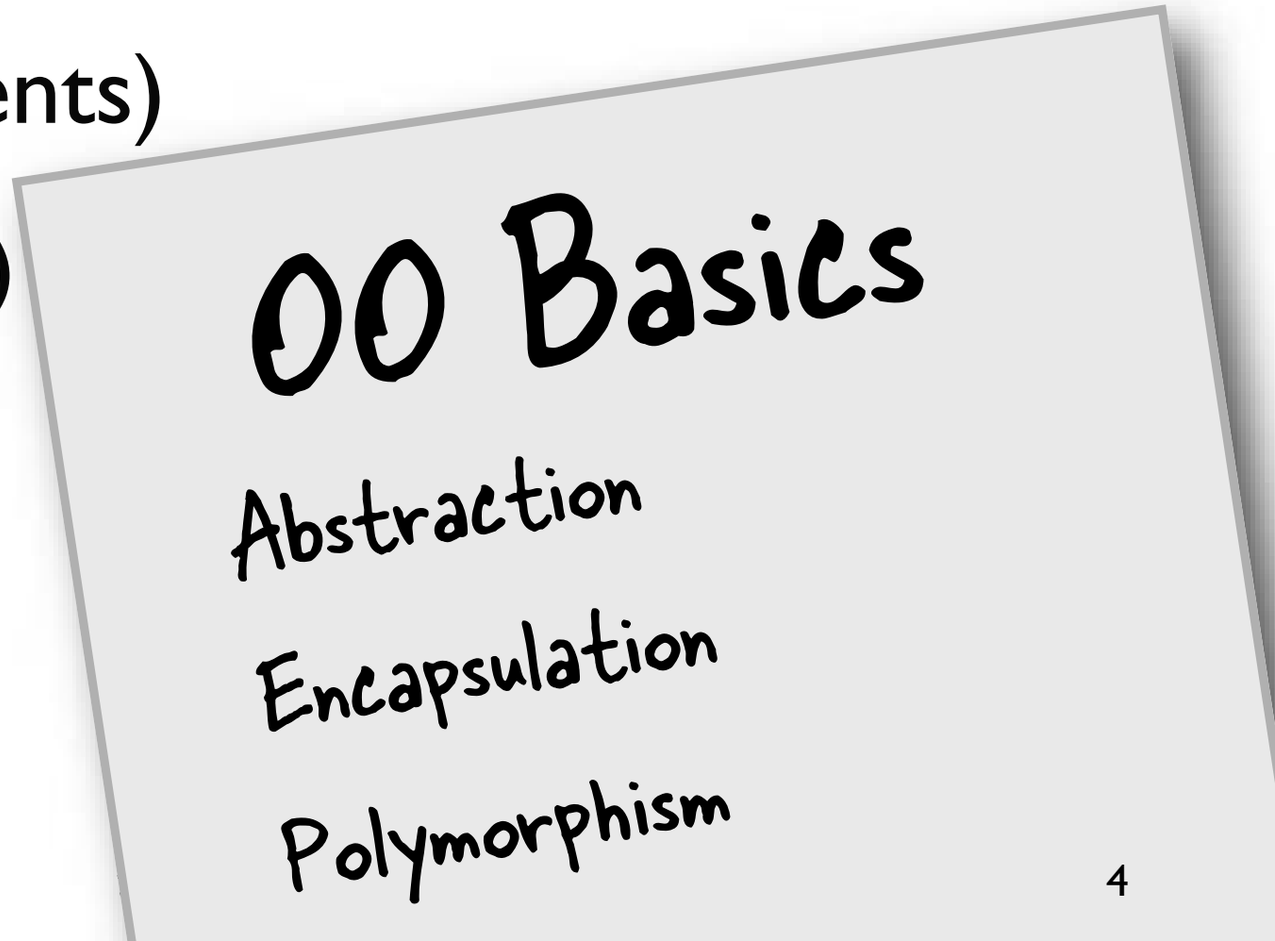
Today's lecture is based on

Slides from:

Tom Zimmermann
Microsoft Research

Recapturing OO concepts

- Inheritance (java: extend)
- Interface (java: implements)
- Abstract (java: abstract)

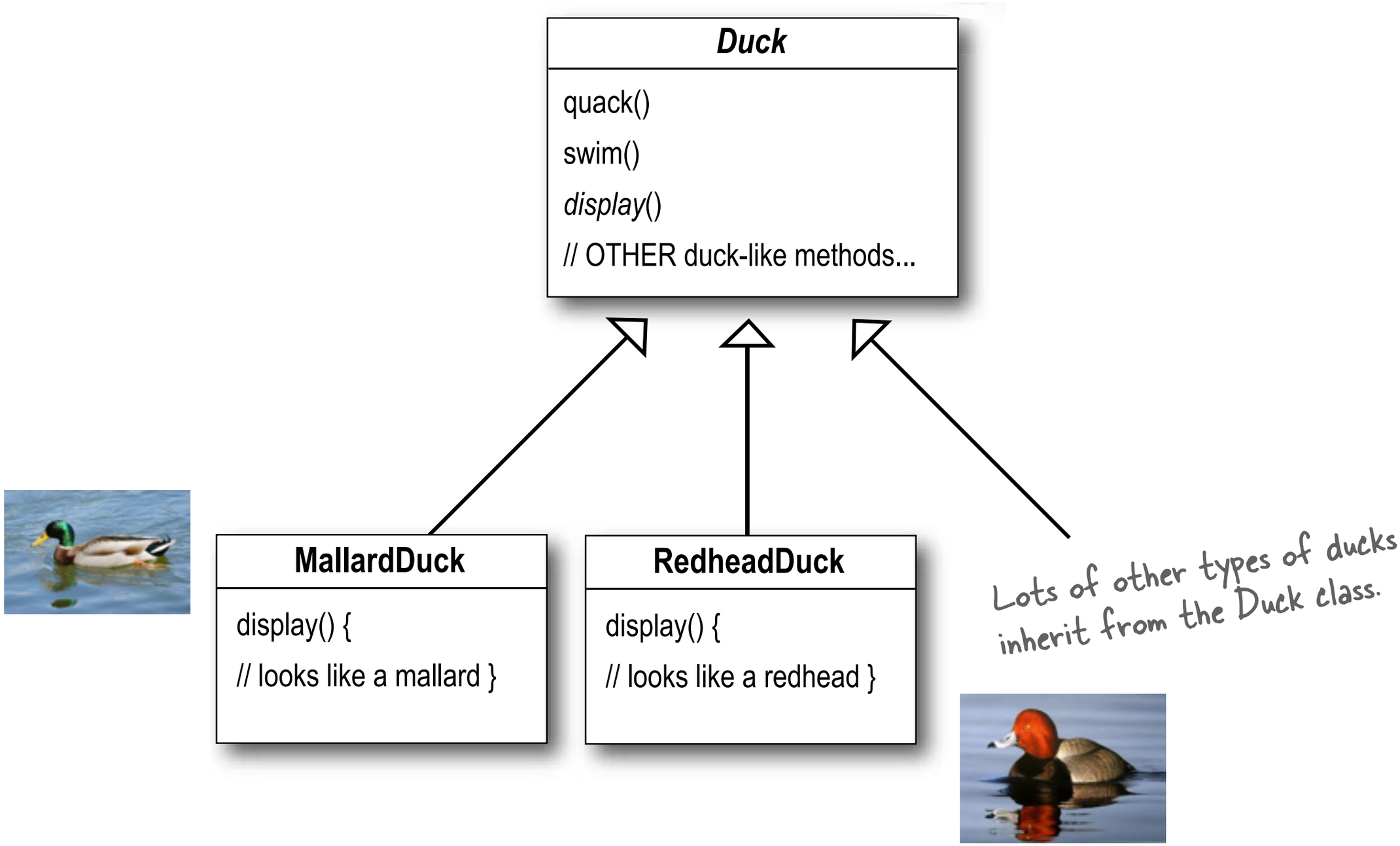


Meet Joe

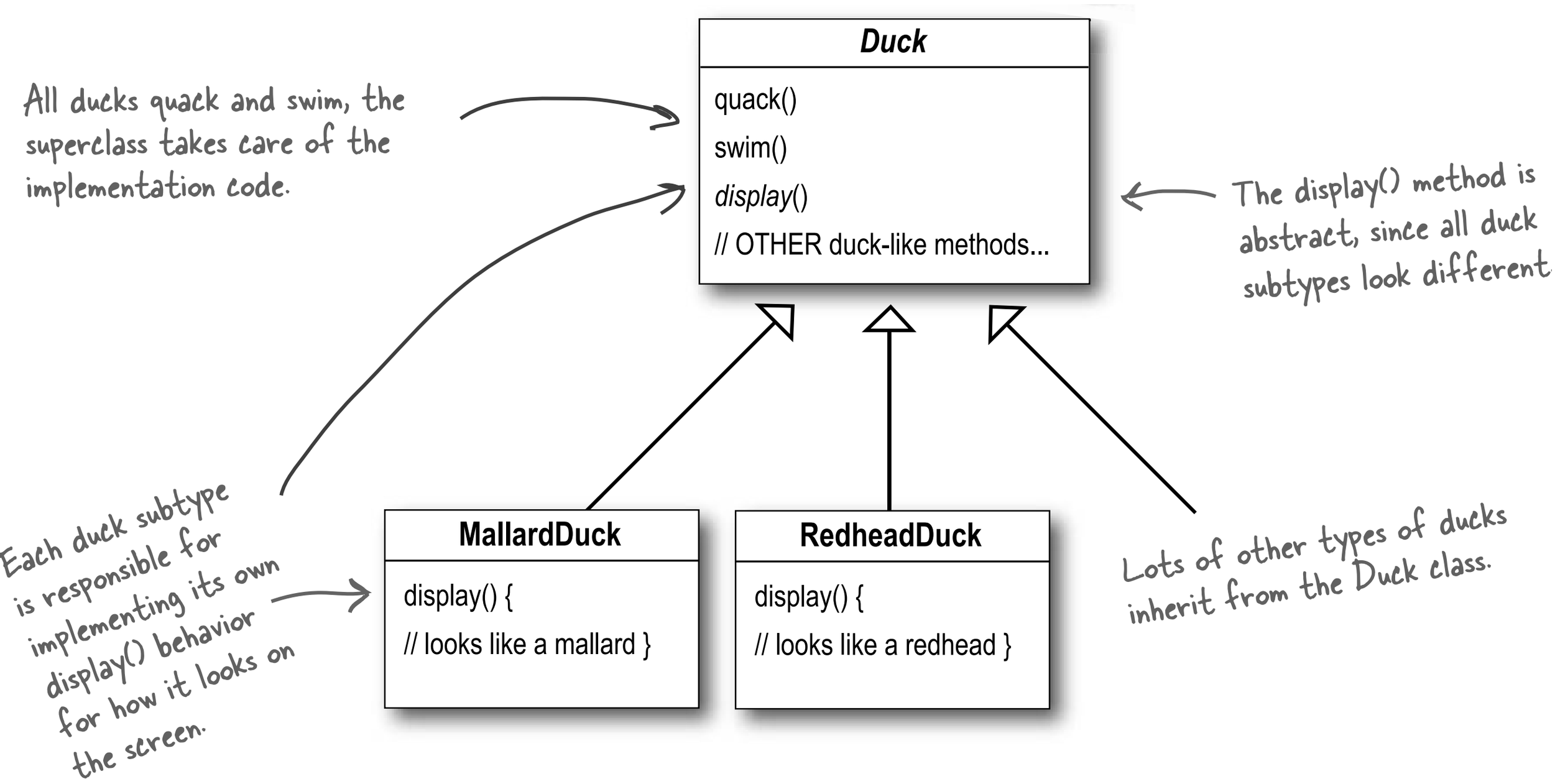
Joe works on SimUDuck.app

... a successful duck pond simulation game

SimUDuck is OO



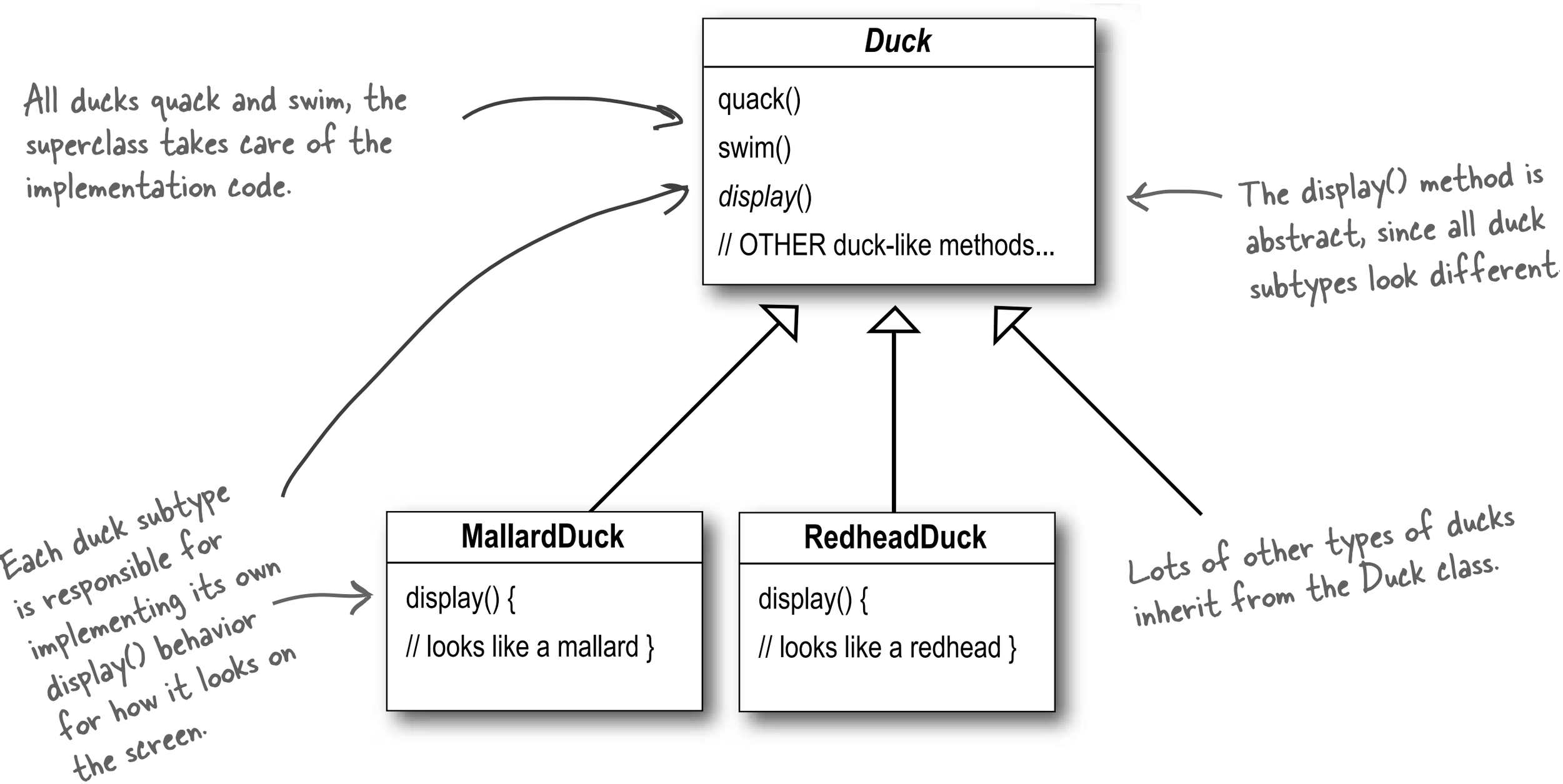
SimUDuck is OO



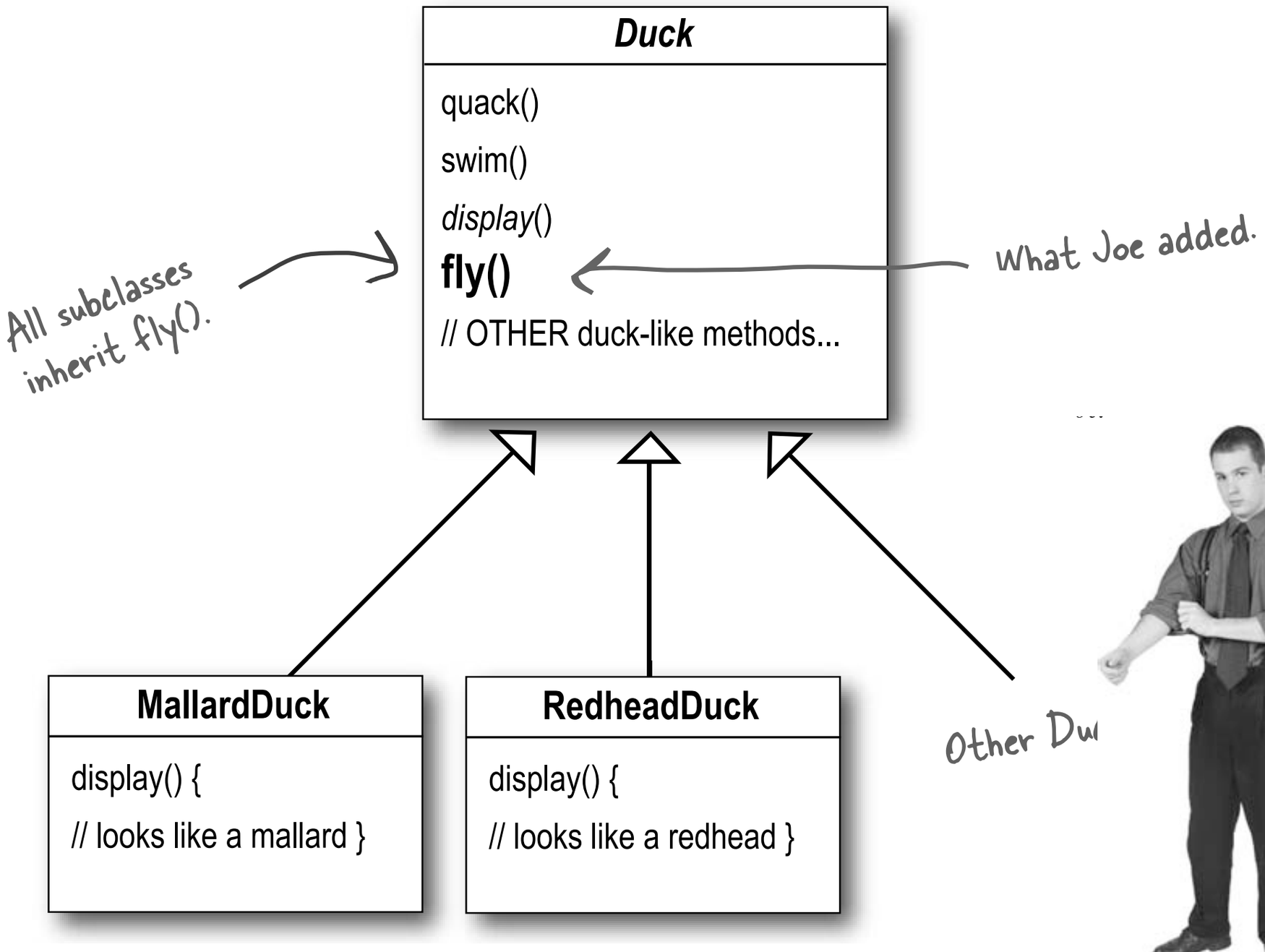
Prepare for the big innovation

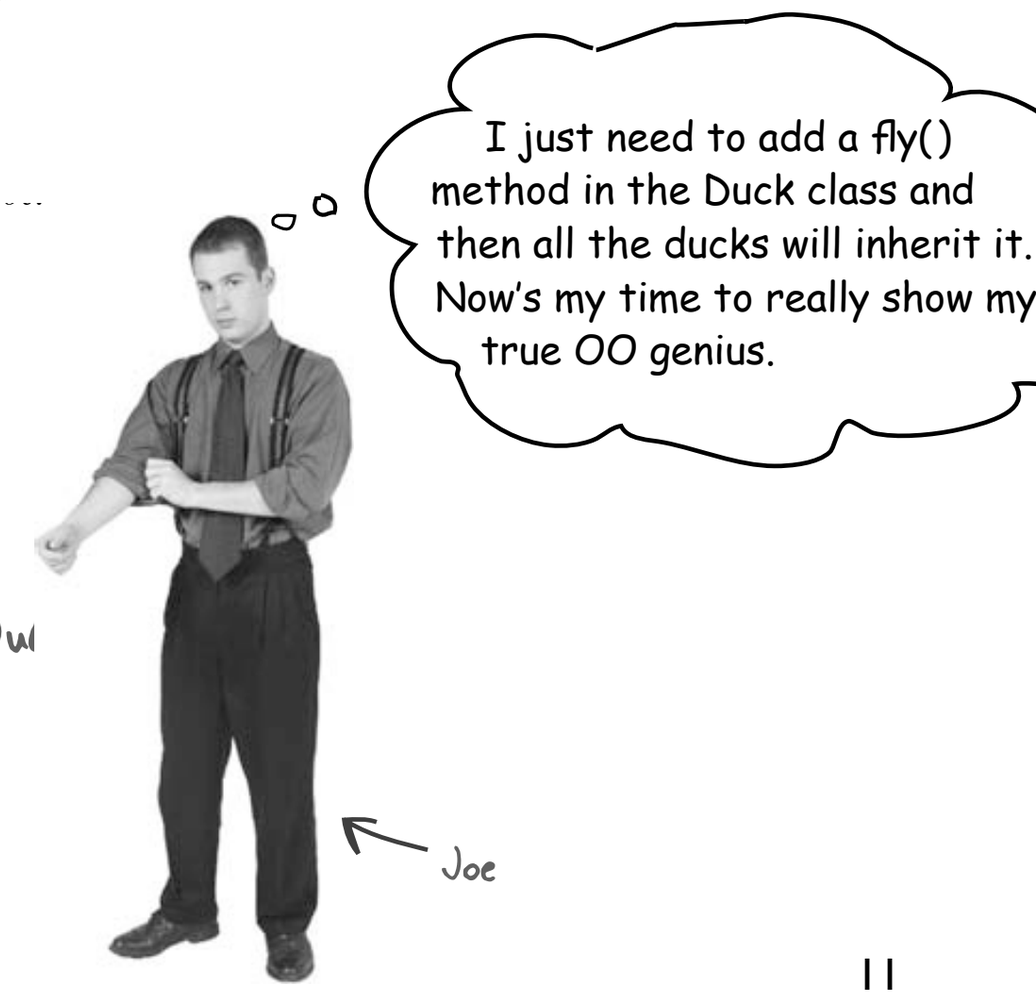
... now the ducks need to fly

How would you do that?



Add fly() to Duck



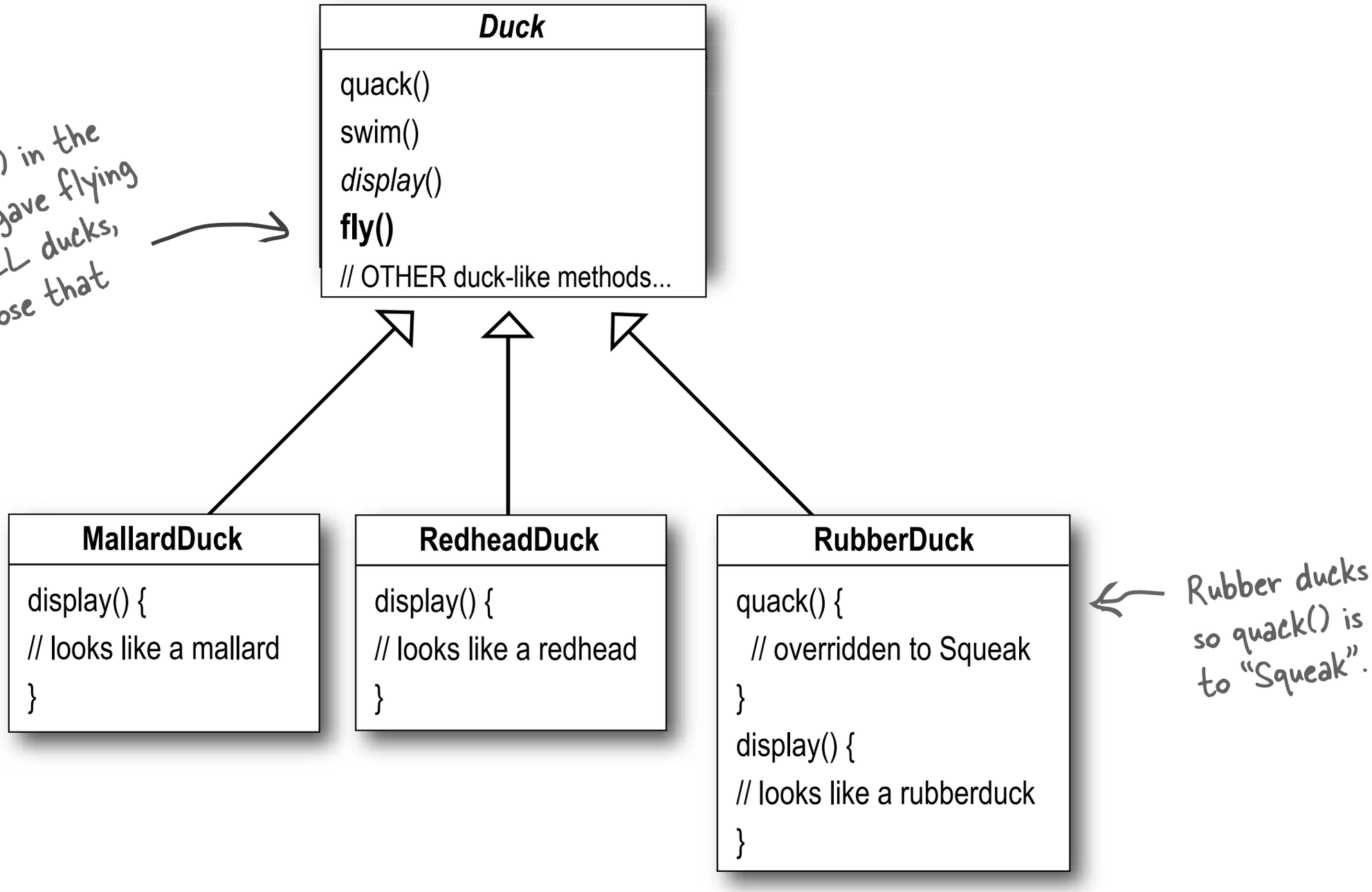


But something went wrong

... flying rubber duckies

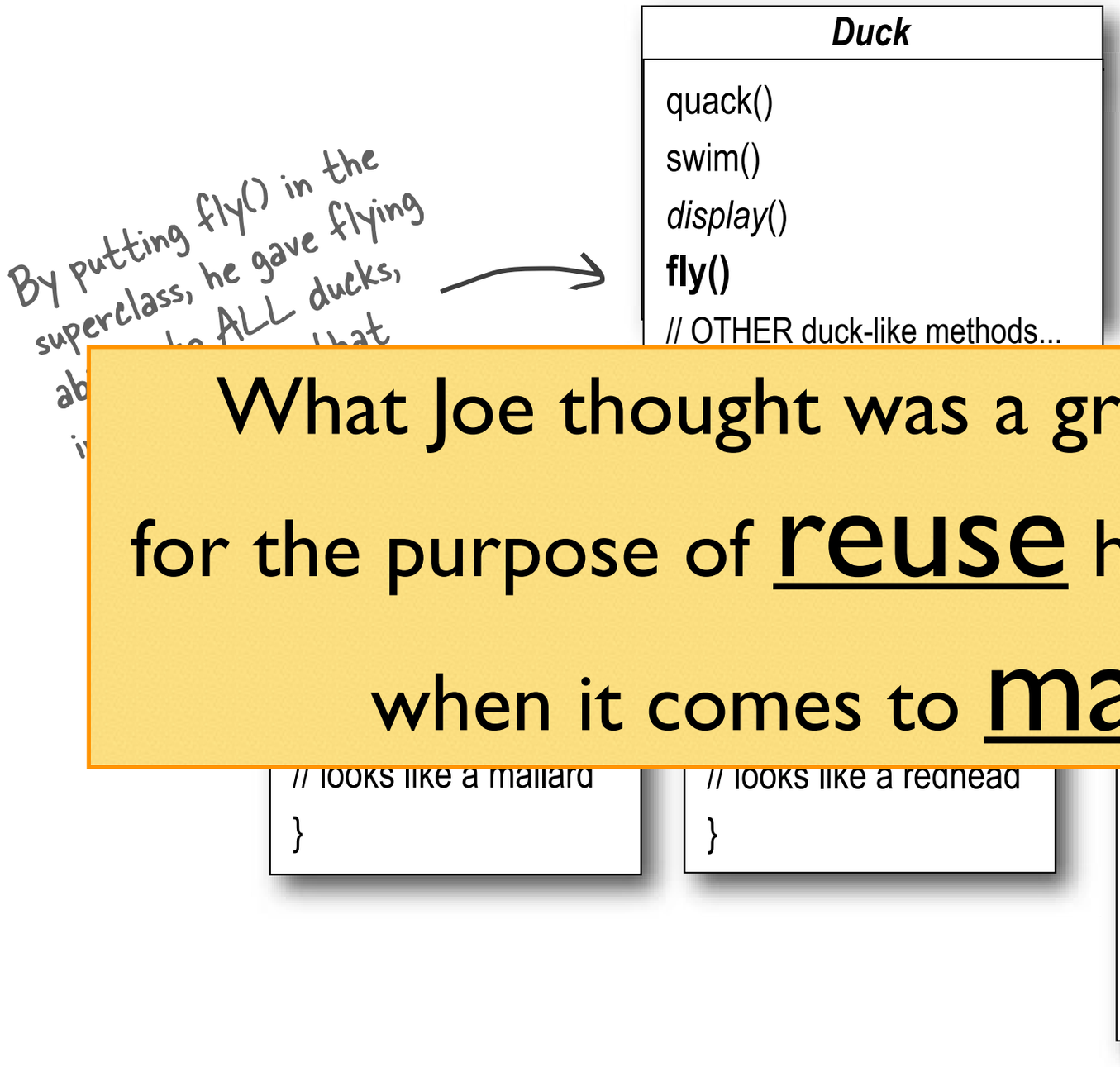
What happened?

By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't.



Rubber ducks don't quack so quack() is overrridden to "Squeak".

What happened?



What Joe thought was a great use of inheritance for the purpose of **reuse** hasn't turned out so well when it comes to **maintenance.**

Joe thinks...

I could always just *override* the fly() method in rubber duck, the way I am with the quack() method...

RubberDuck

```
quack()  { // squeak}
display() { .// rubber duck }
fly() {
   // override to do nothing
}
```

But then what happens when we add wooden decoy ducks to the program? They aren't supposed to fly or quack...

Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

DecoyDuck

```
quack() {
  // override to do nothing
}

display() { // decoy duck}

fly() {
  // override to do nothing
}
```

Maintenance Nightmare

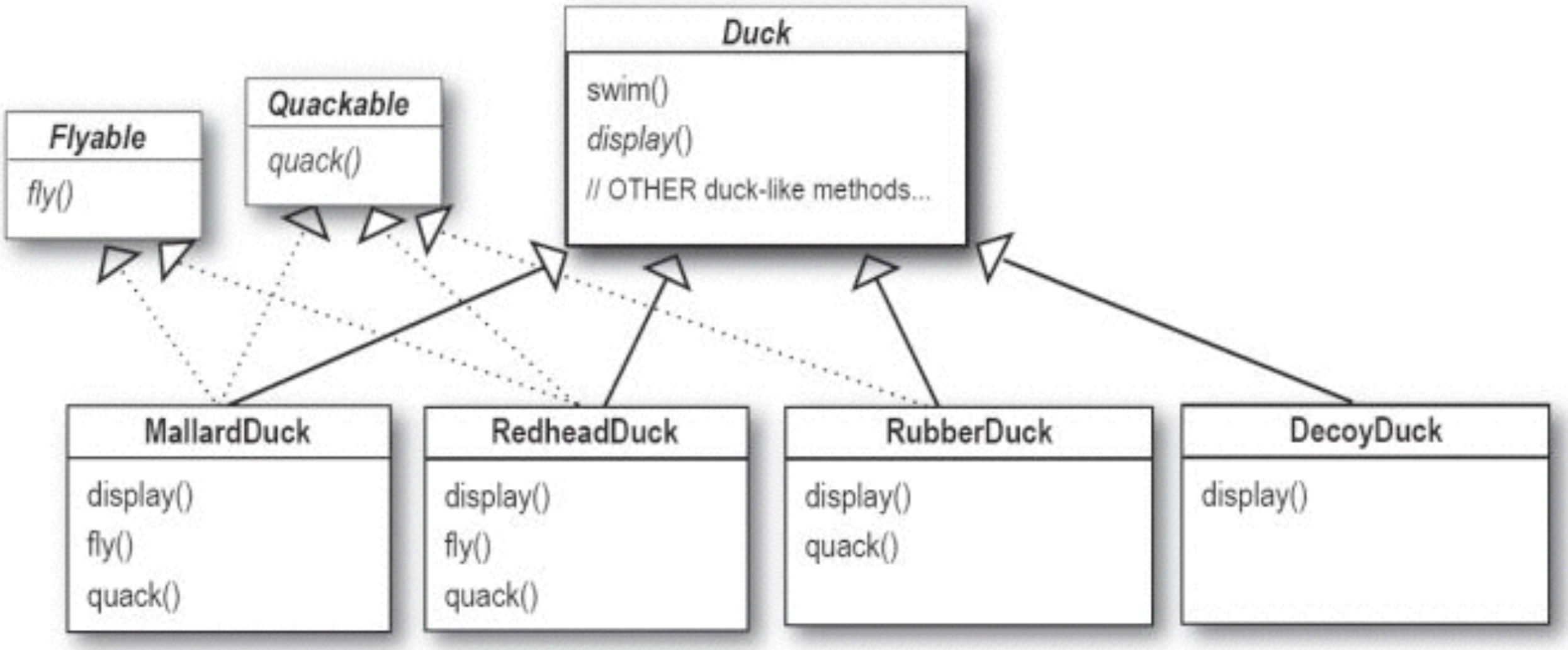
- What about RubberMallardDuck?
  - They quack, so don't override
  - They can't fly, so override
  - They look like Mallard, duplicate the display code
- What about the Duck King?
  - The Duck King can change from Mallard to Redhead to Rubber at any time
  - Joe is fired.

Sharpen your pencil

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

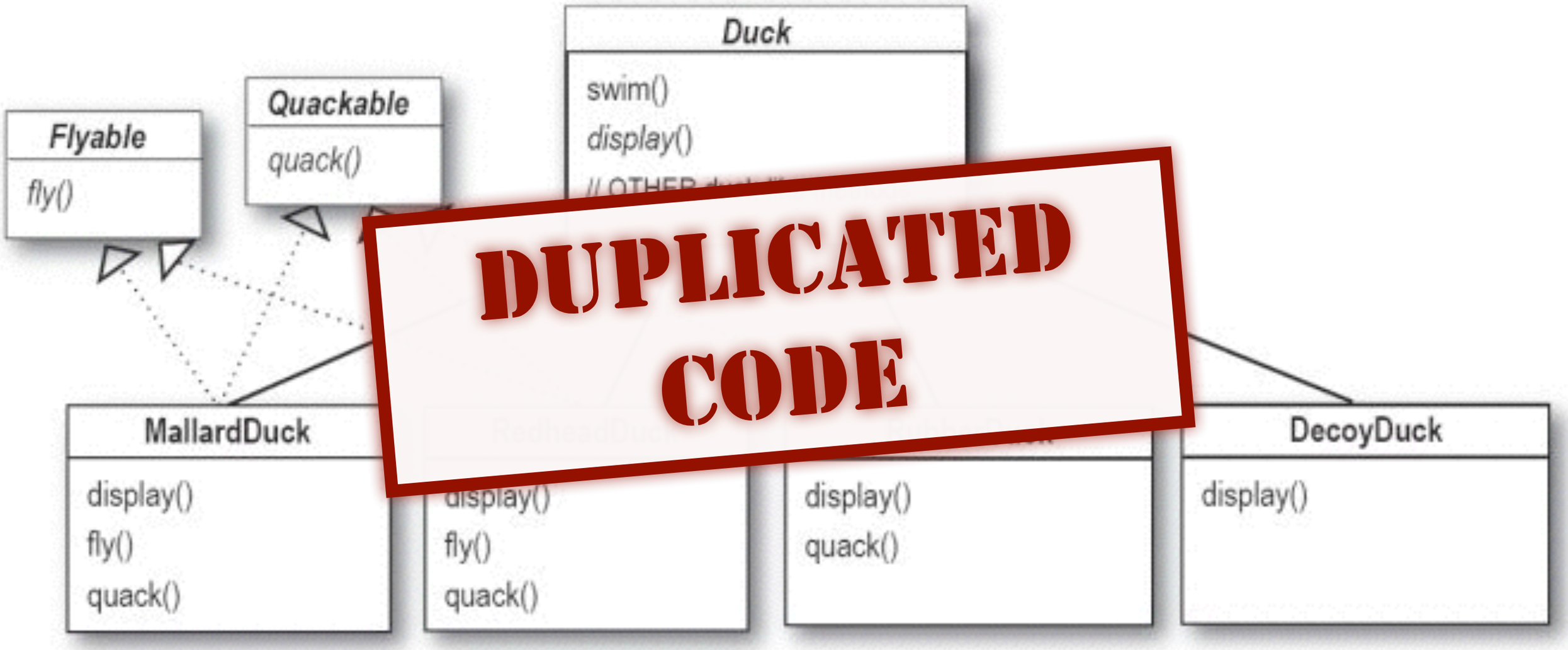
- ❑ A. Code is duplicated across subclasses.
- ☒ B. Runtime behavior changes are difficult.
- ❑ C. We can't make ducks dance.
- ☒ D. Hard to gain knowledge of all duck behaviors.
- ❑ E. Ducks can't fly and quack at the same time.
- ☒ F. Changes can unintentionally affect other ducks.

How about an interface?



What do YOU think about this design?

How about an interface?



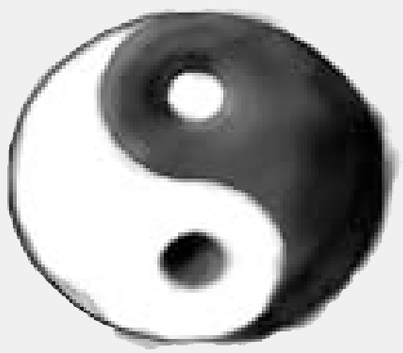
What do YOU think about this design?

Duplicated Code

That is, like, the dumbest idea you've come up with. **Can you say, "duplicate code"?** If you thought having to *override a few methods* was bad, how are you gonna feel when you need to make a little change to the flying behavior... *in all 48 of the flying Duck subclasses?!*

Encapsulation

Take what varies and "encapsulate" it so it won't affect the rest of your code.



Design Principle

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

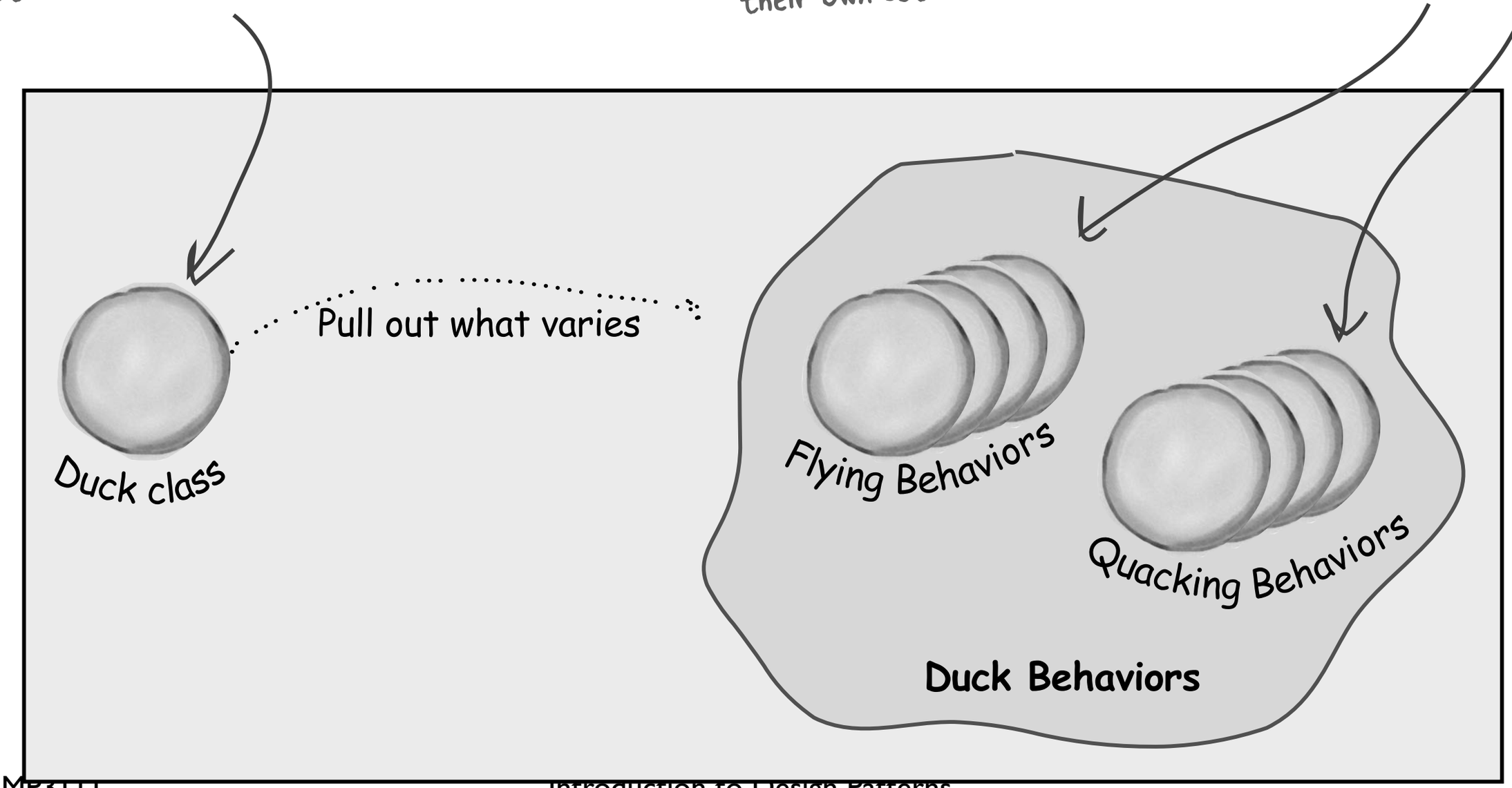
The result? Fewer unintended consequences from code changes and more flexibility in your systems

Separating the duck behaviors

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.

Now flying and quacking each get their own set of classes.

Various behavior implementations are going to live here.



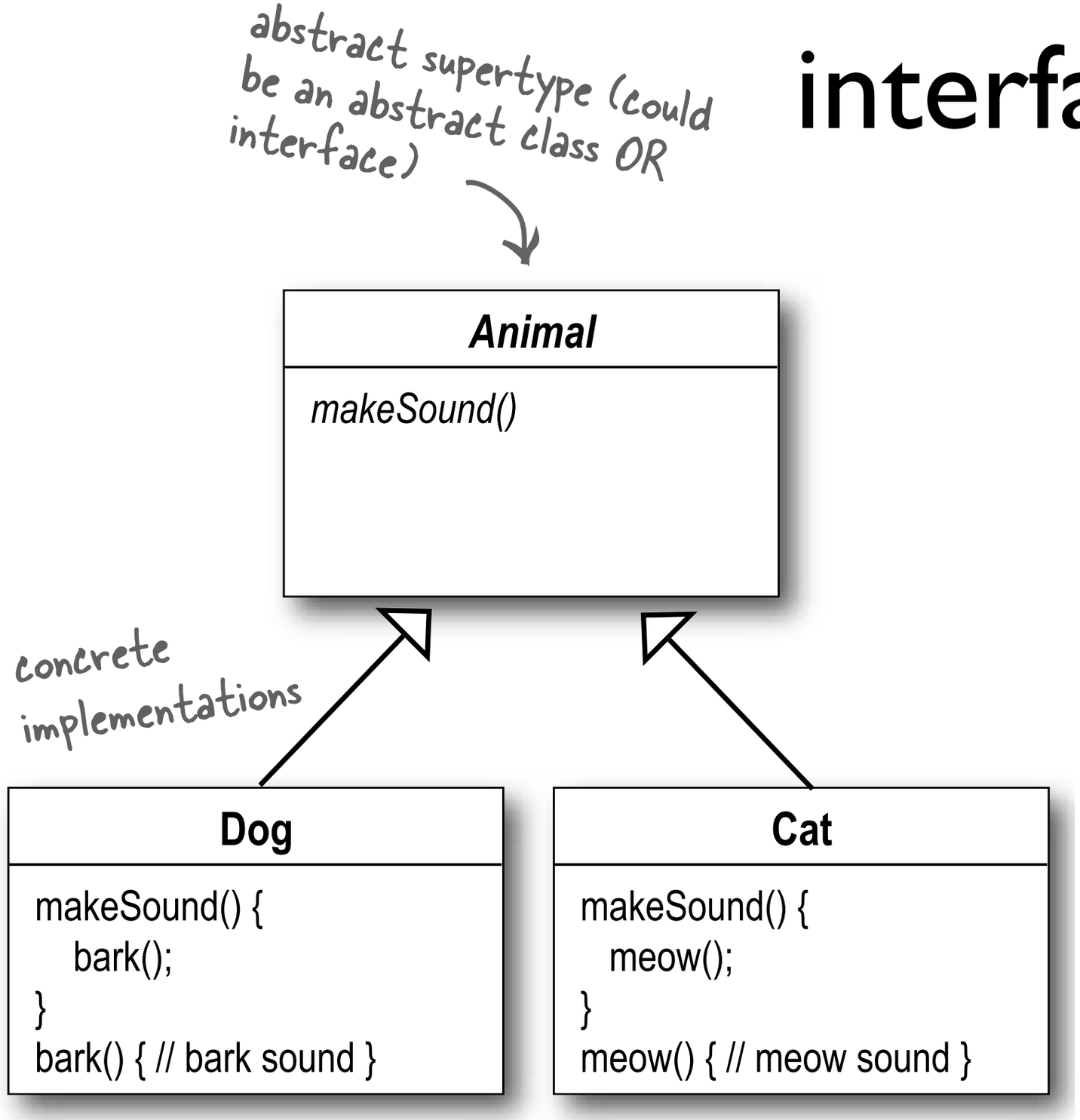
Program to an interface

Design Principle

Program to an interface, not an implementation.

That way, the Duck classes won't need to know any of the implementation details of their behaviors.

Program to an interface/supertype

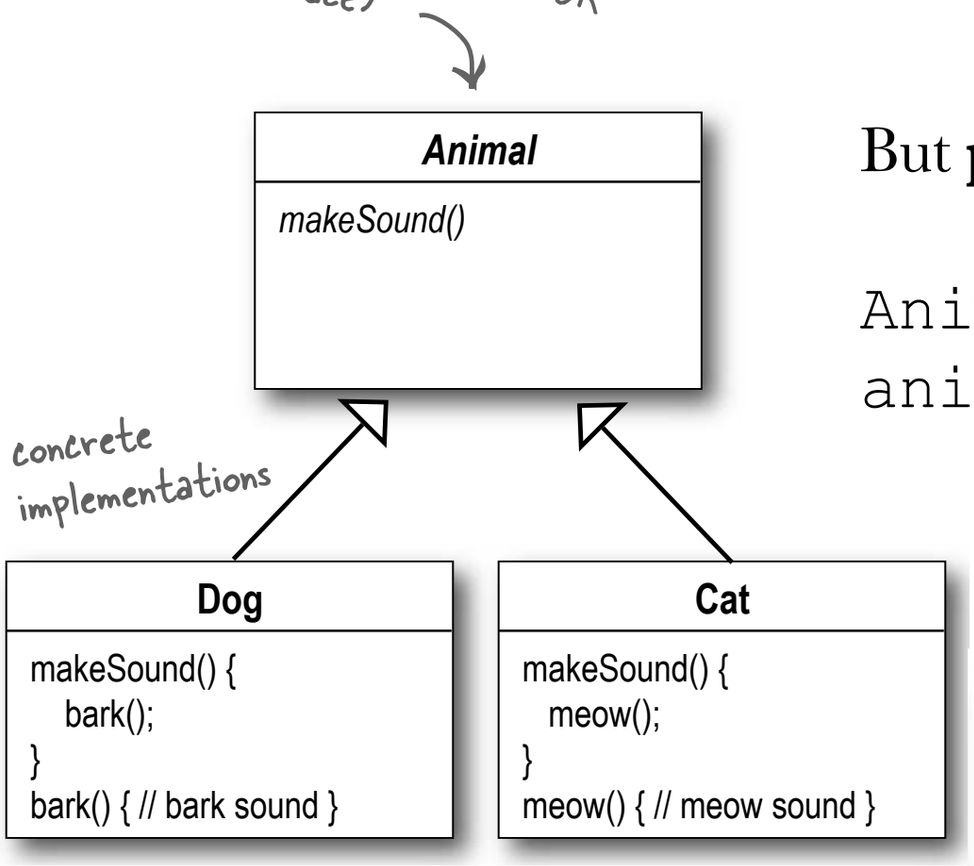


Program to an interface/supertype

Programming to an implementation would be:

```
Dog d = new Dog();
d.bark();
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.



But **programming to an interface/supertype** would be:

```
Animal animal = new Dog();
animal.makeSound();
```

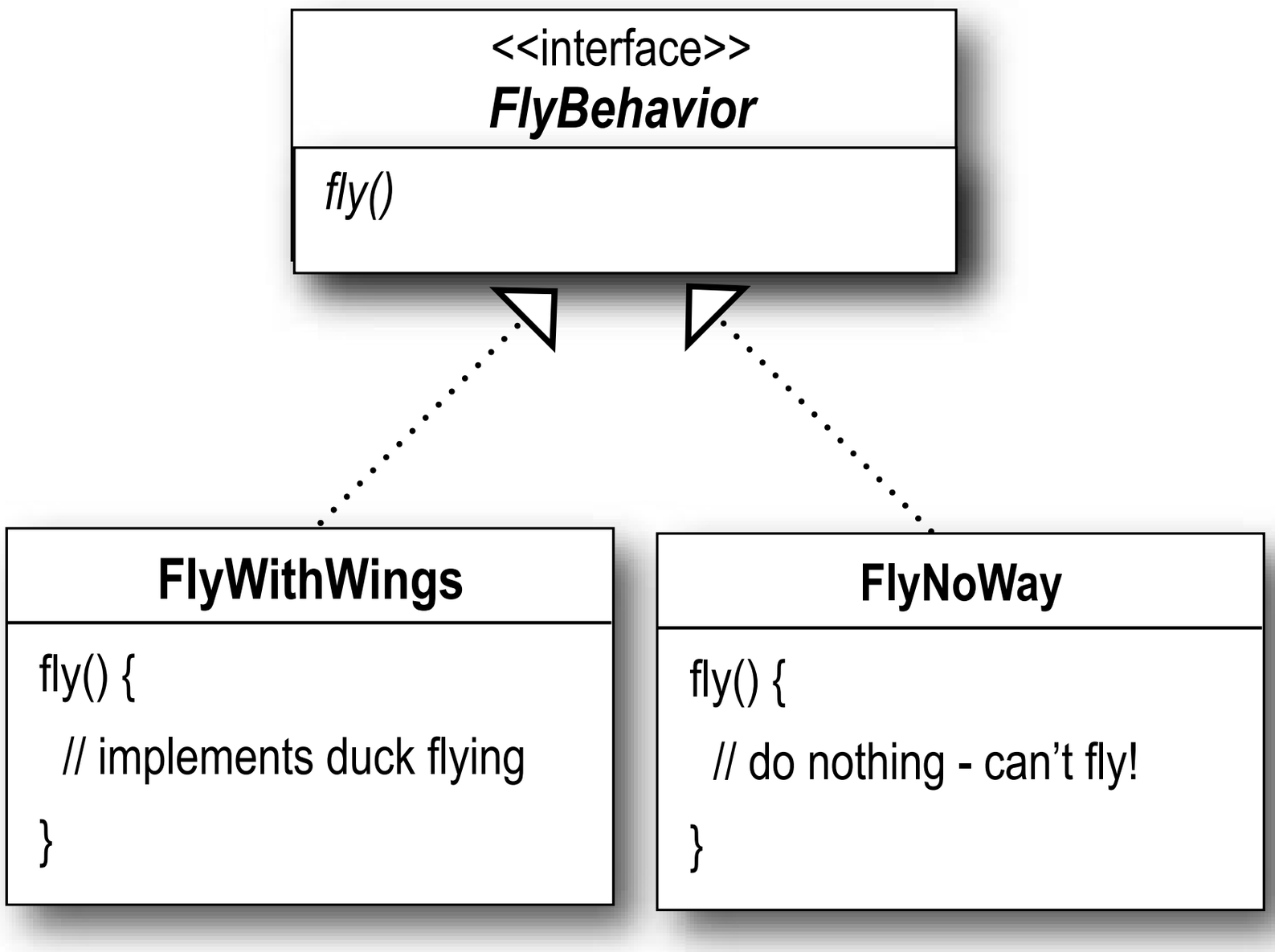
We know it's a Dog, but we can now use the animal reference polymorphically.

Even better, rather than hard-coding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime:**

```
a = getAnimal()
a.makeSound();
```

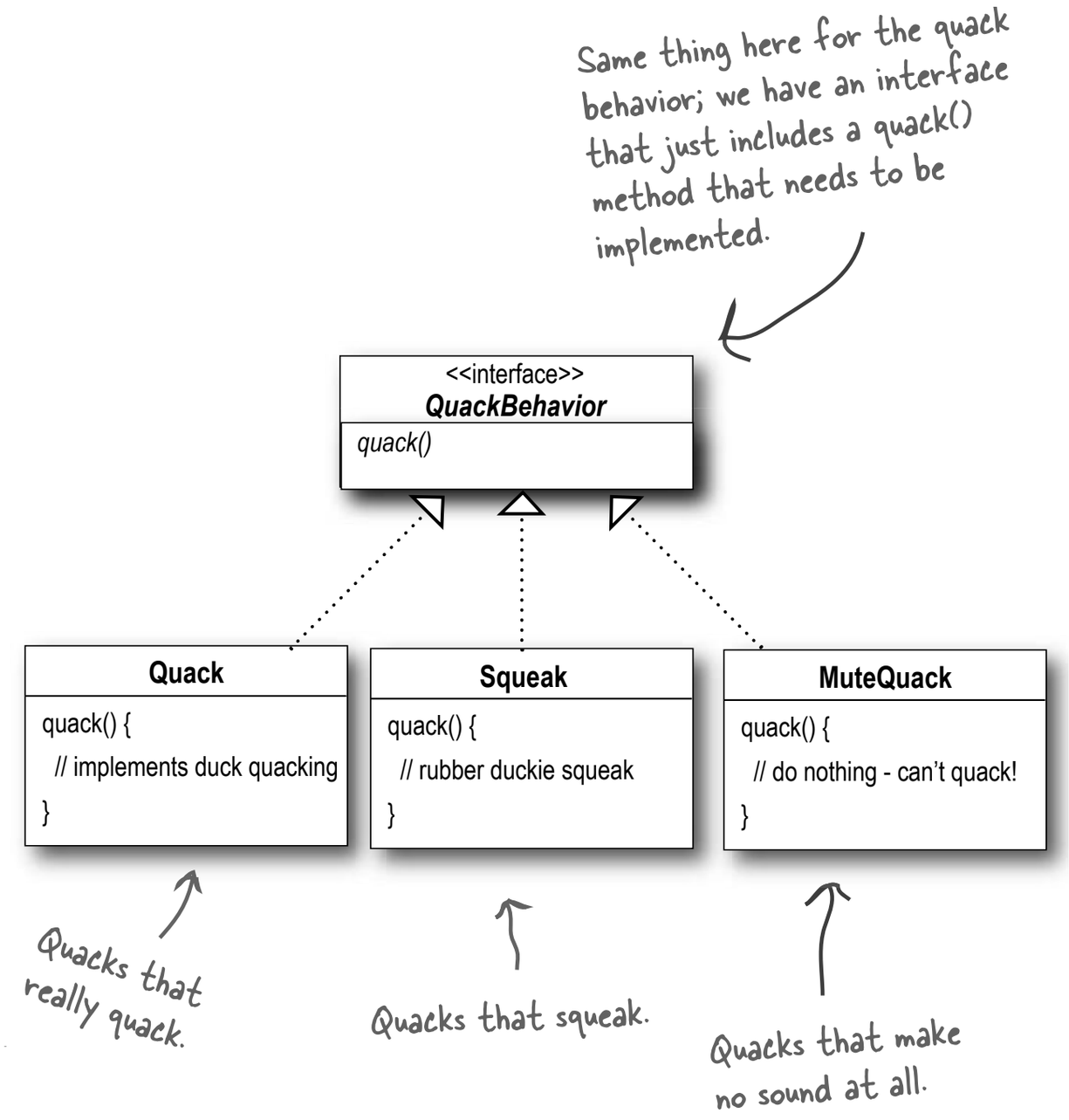
We don't know WHAT the actual animal subtype is... all we care about is that it knows how to respond to makeSound().

Use interfaces for behaviors



similar for QuackBehavior

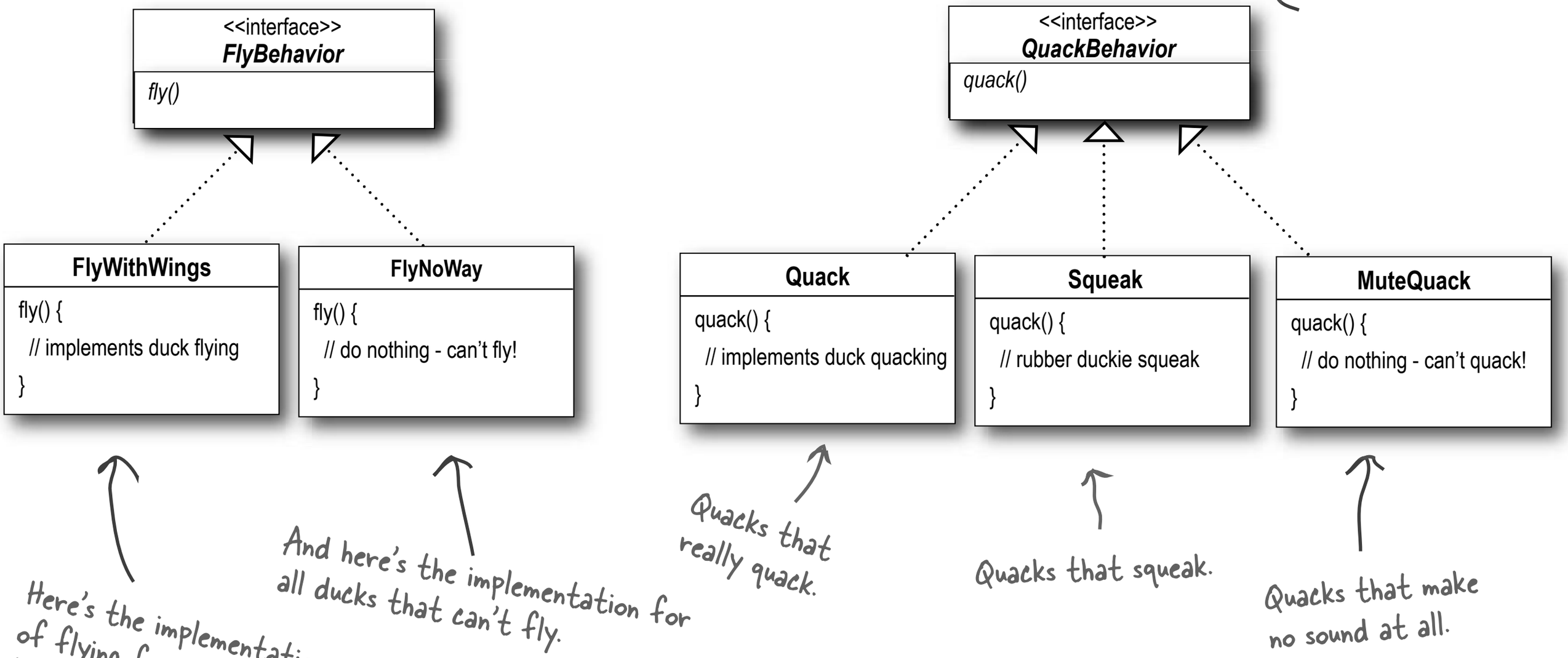
Use interfaces for behaviors



Duck Behaviors

FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly method.

Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.



Here's the implementation of flying for all ducks that have wings.

And here's the implementation for all ducks that can't fly.

Quacks that really quack.

Quacks that squeak.

Quacks that make no sound at all.

Our design rocks

- We can reuse the fly and quack behaviors
  - not hidden inside Duck anymore
- We can add new behaviors
  - without modifying any existing behavior classes or Duck classes that use behaviors
- SimUDuck.app now has the benefit of REUSE while still being MAINTAINABLE

Sharpen your pencil

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

Create a class FlyRocketPowered that implements interface FlyBehavior

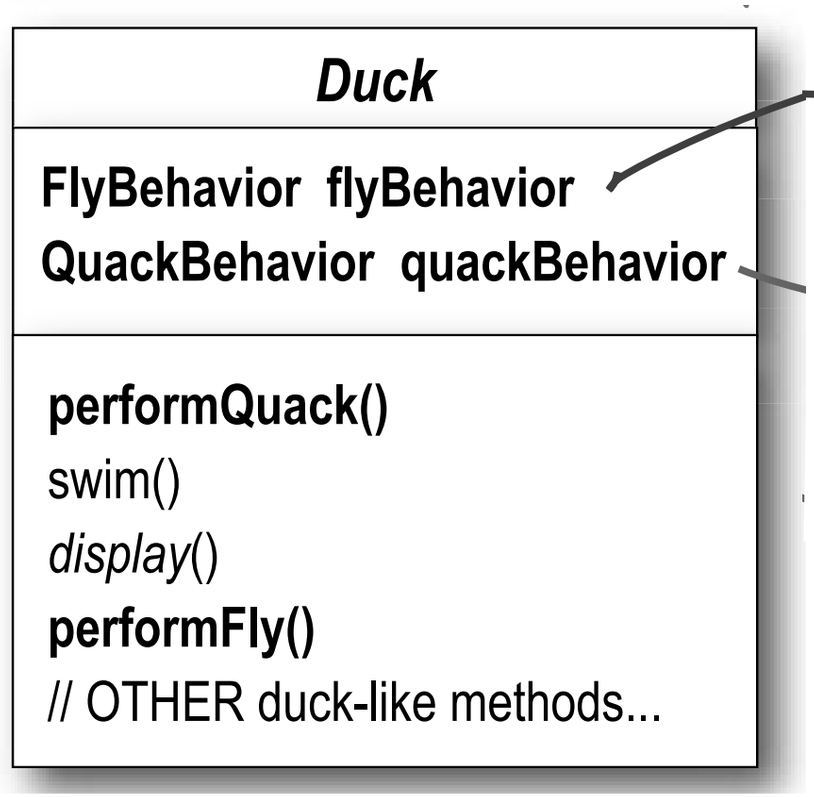
Use FlyRocketPowered

Integrating the duck behavior

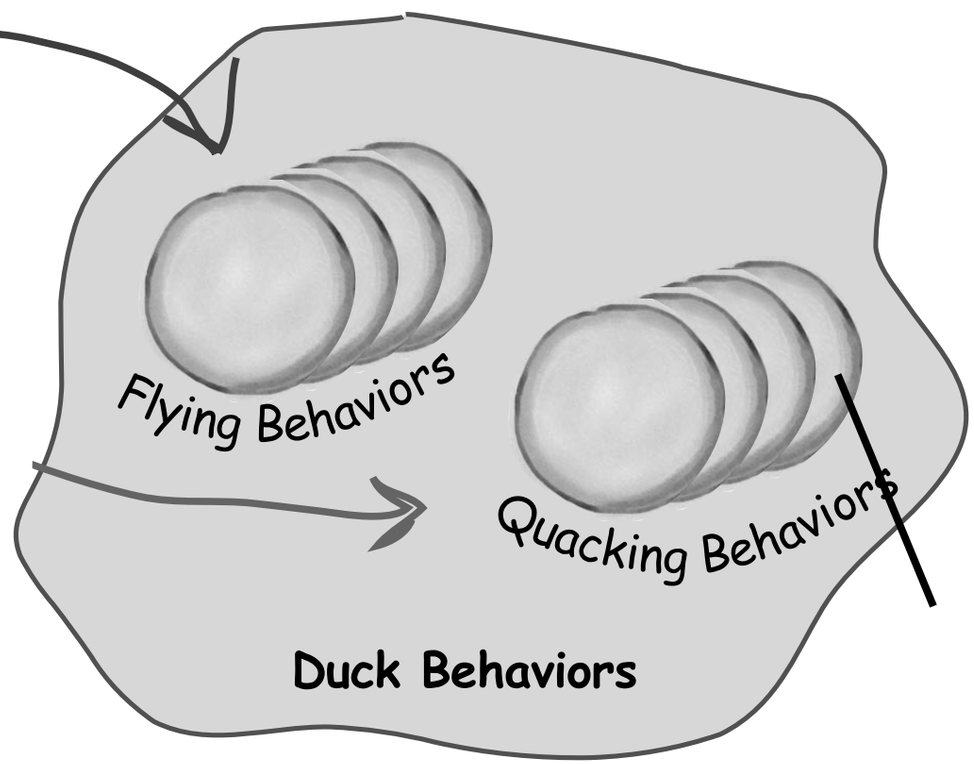
1 **First we'll add two instance variables**

The behavior variables are declared as the behavior INTERFACE type.

Instance variables hold a reference to a specific behavior at runtime.



These methods replace fly() and quack().



Add two instance variables to Duck class

Integrating the duck behavior

❷ **Now we implement performQuack():**

```
public class Duck {
    QuackBehavior quackBehavior;
    // more

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

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

Delegate the quack and fly to the behaviors

Integrating the duck behavior

```
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");
    }
}
```

A MallardDuck uses the Quack class to handle its quack, so when performQuack is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

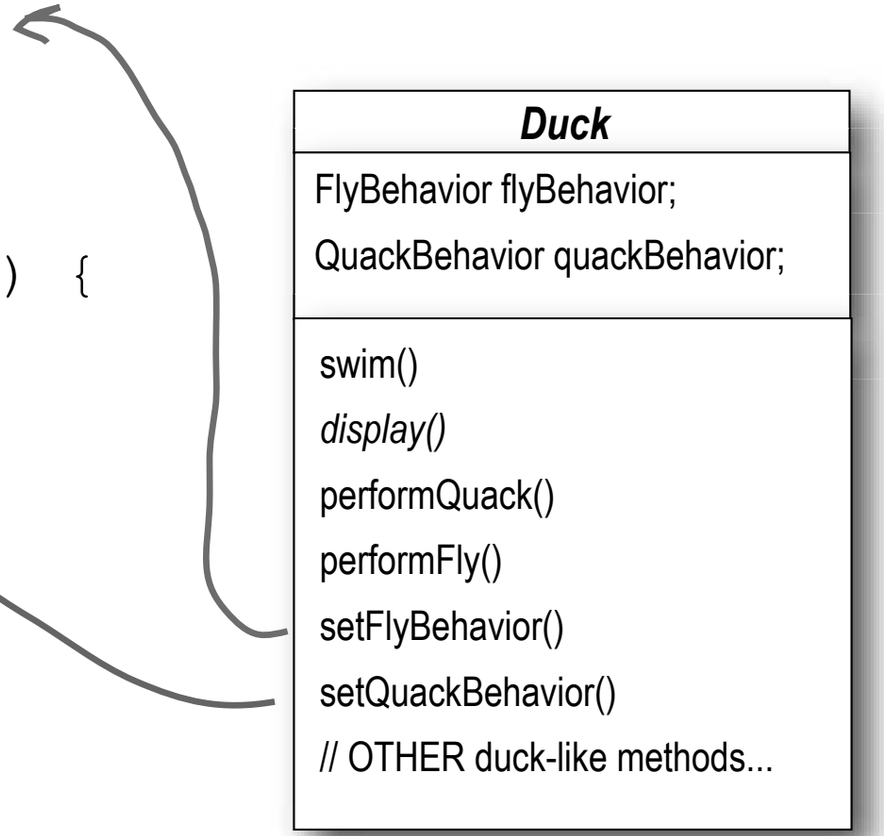
And it uses FlyWithWings as its FlyBehavior type.

Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

Initialize the behaviors in subclasses

Setting behavior dynamically

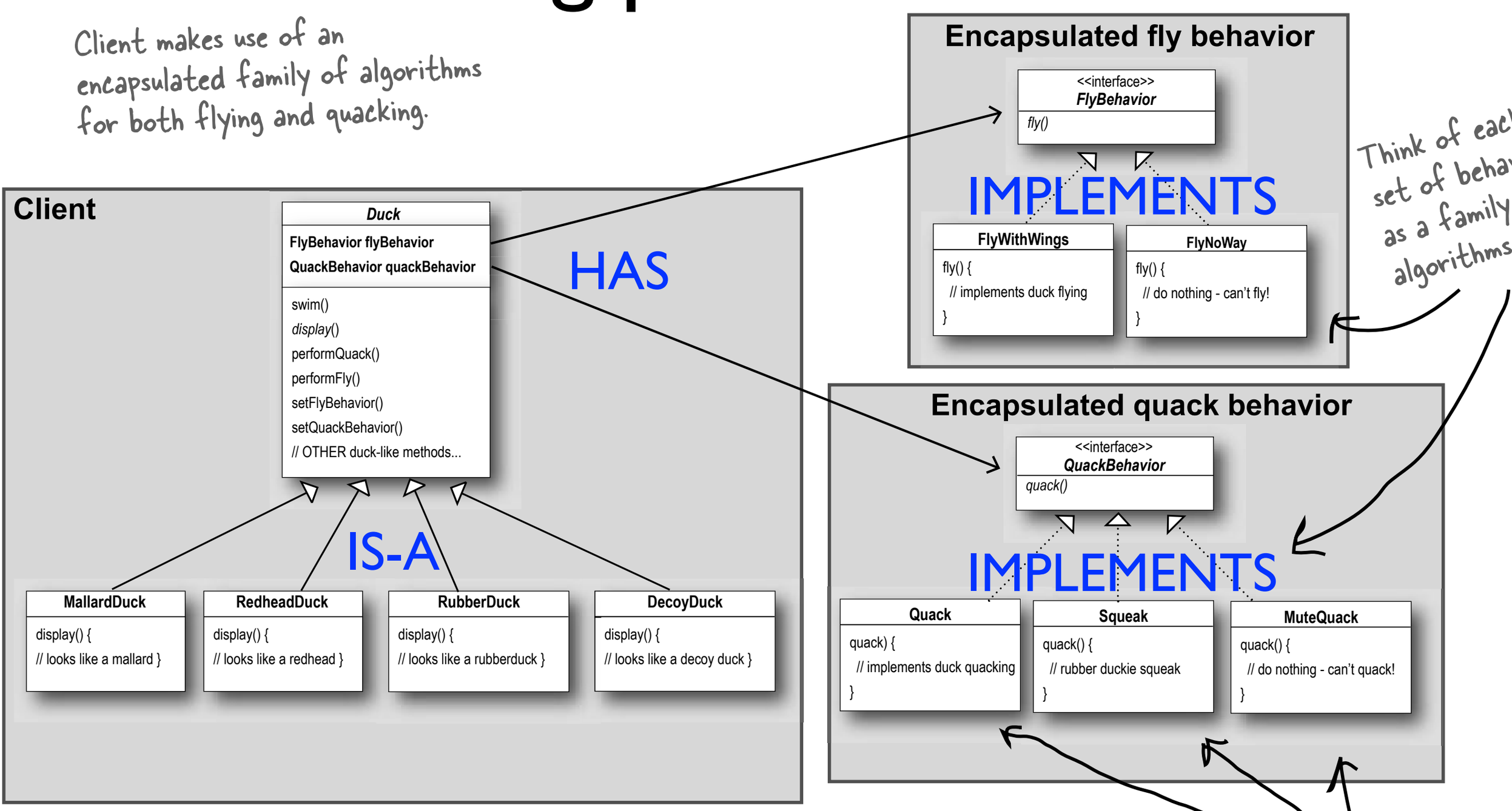
```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



We can call these methods anytime we want to change the behavior of a duck on the fly.

The big picture

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



... aka the **strategy** pattern

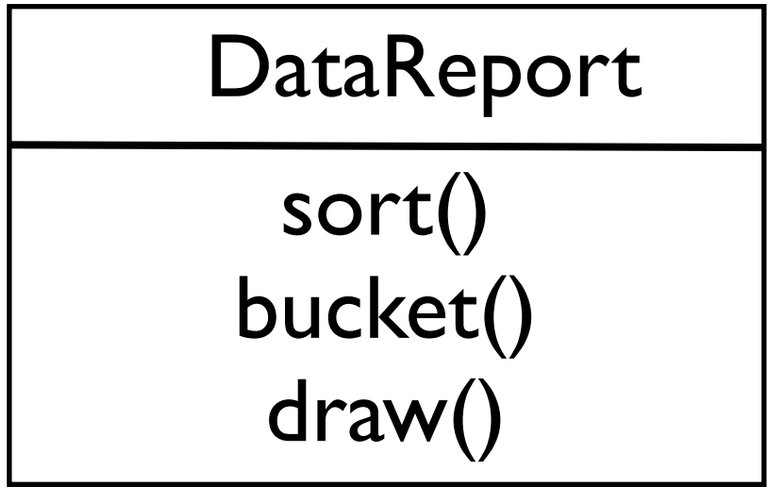
The strategy pattern

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

HAS-A can be better than IS-A

A data reporting program

- Features
  - Can sort the data in different ways
  - Can bucket the data with different categories
  - Can draw different diagrams: Pie/Bar/Line



Design patterns

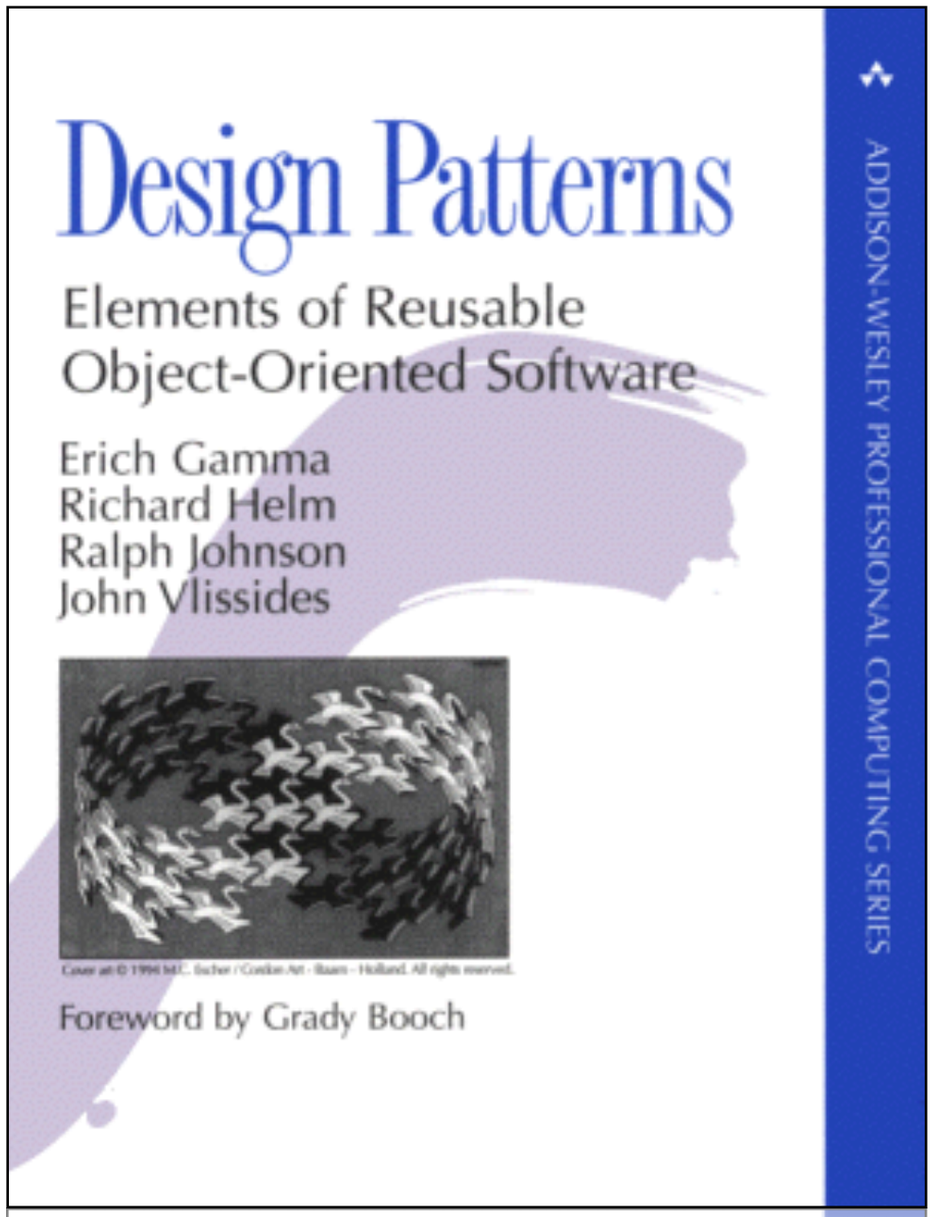
Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

— Alexander: “A

Different kinds

- Creational patterns
  - describe the process of object creation
- Structural patterns
  - composition of classes or objects
- Behavioral patterns
  - how classes or objects interact
- Concurrency patterns

The GoF book



- Catalog of over 20 design patterns
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
- Published in 1994

Alice

I need a Cream cheese with jelly on white bread, a chocolate soda with vanilla ice cream, a grilled cheese sandwich with bacon, a tuna fish salad on toast, a banana split with ice cream & sliced bananas and a coffee with a cream and two sugars, ... oh, and put a hamburger on the grill!

走狗

Flo

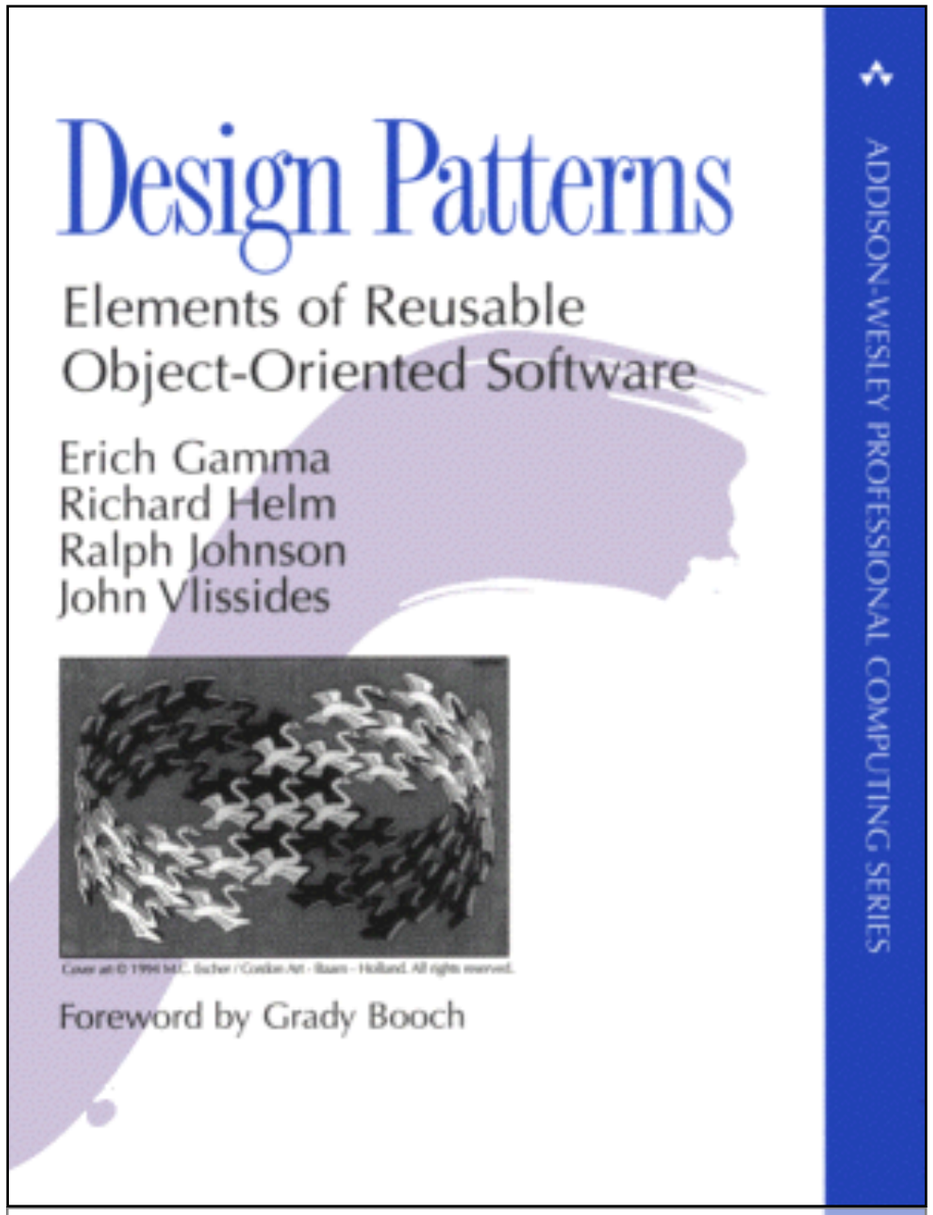
Give me a C.J. White, a black & white, a Jack Benny, a radio, a house boat, a coffee regular and burn one!

豬意

A shared pattern vocabulary

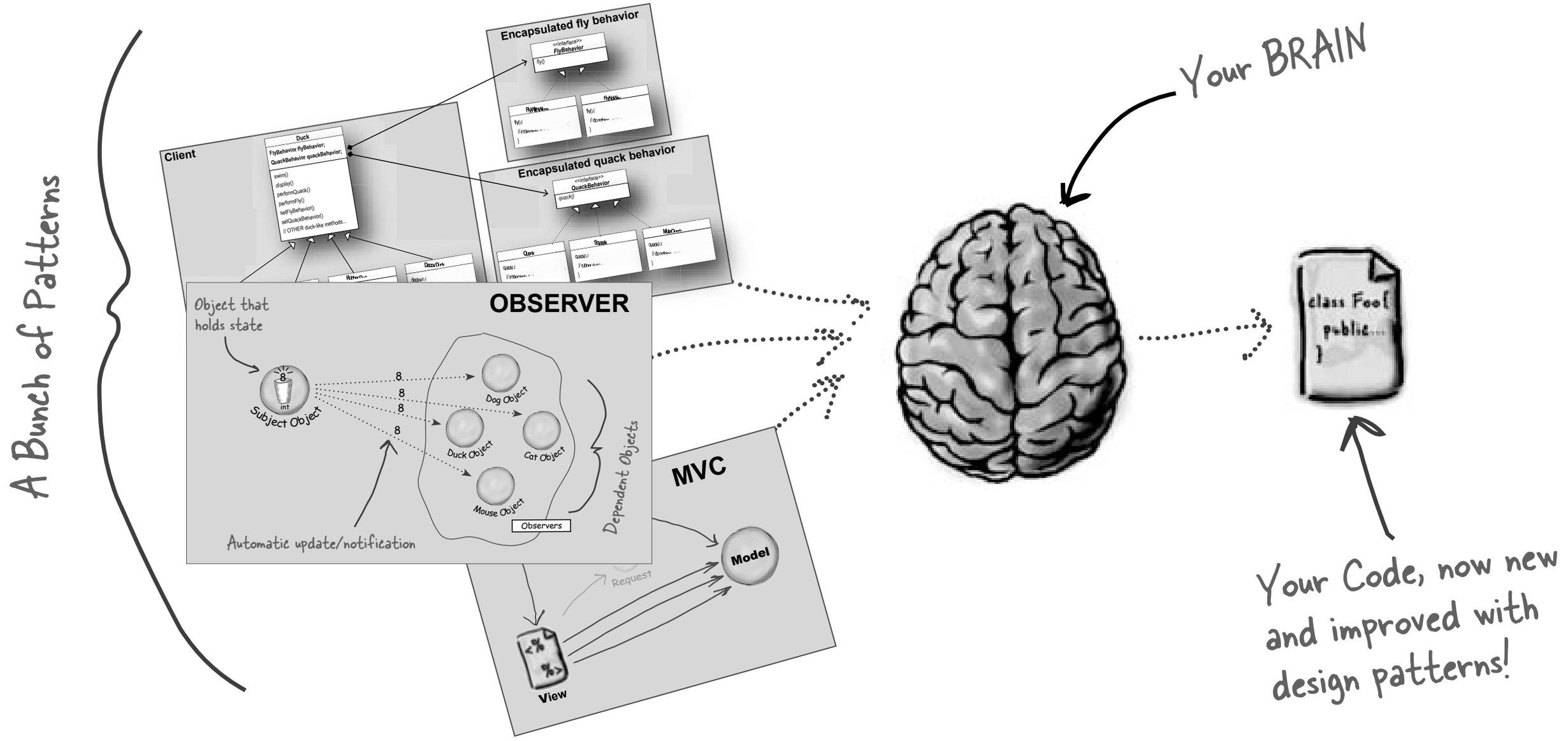
- Shared pattern vocabularies are powerful
- Patterns allow you to say more with less
- Allow you to stay “in the design” longer
- Turbo charge your development team
- Junior developers get up to speed

How to describe patterns



- Pattern Name and Classification
- Intent:
- Also Known As
- Motivation (Forces)
- Applicability
- Structure
- Participants
- Collaboration
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

How to use design patterns



Patterns are nothing more than using OO design principles...

Skeptical Developer

A common misconception, Grasshopper, but it's more subtle than that. You have much to learn...

Friendly Patterns Guru

Remember, knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.

Summary

- Encapsulate what varies
- Program to an interface/supertype
- Favor composition of inheritance
- Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.

One down, many to go!

Summary

OO Basics

Abstraction

Encapsulation

Polymorphism

OO Principles

Encapsulate what varies.

Favor composition over inheritence.

Program to interfaces, not implementations.

OO Patterns

Strategy – defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

One down, many to go!