INTRODUCTION TO DESIGN PATTERNS

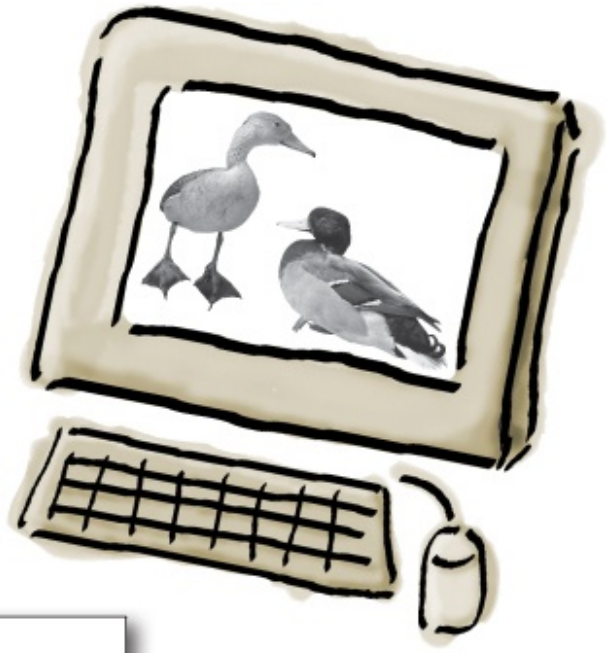
Chandan R. Rupakheti

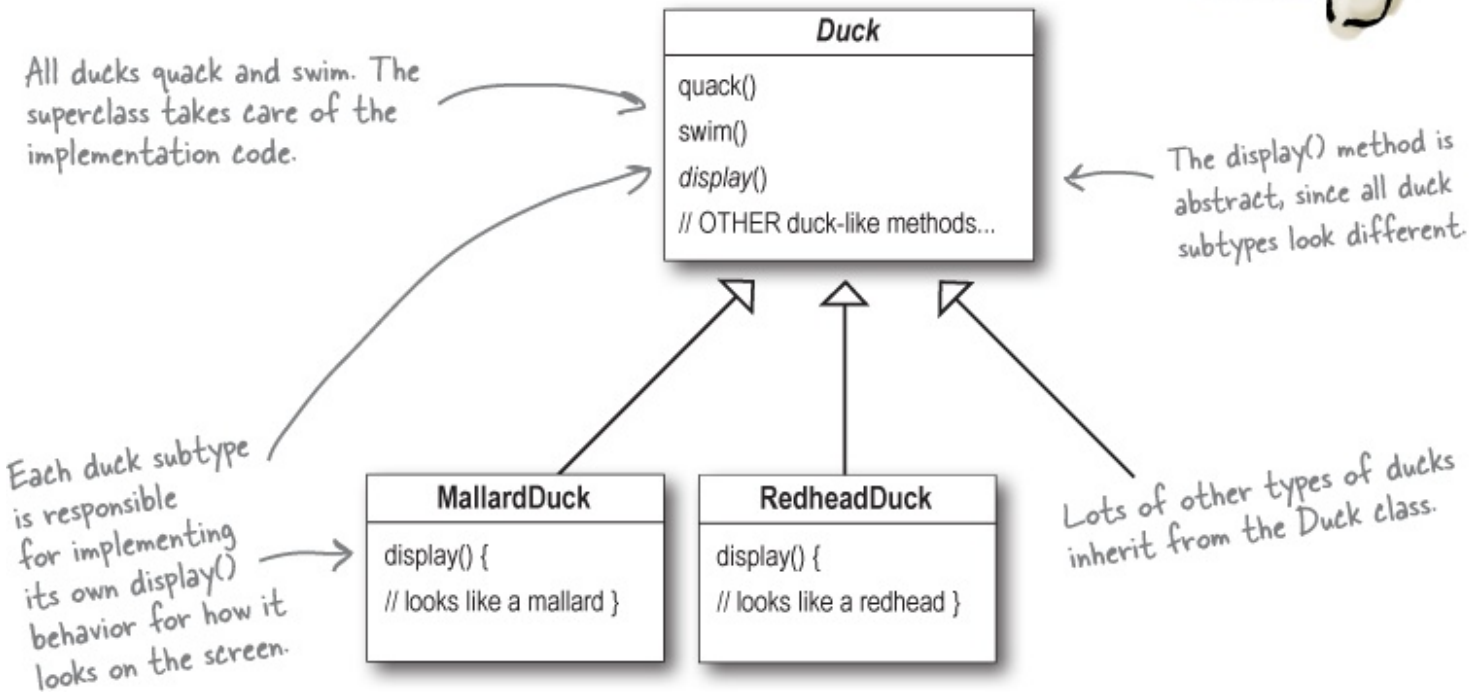
Week 1-1

Overview

- Why Design Patterns?
- Some key OO design principles
- Identifying variability and invariant of software system
- The Strategy Pattern

SimUDuck App



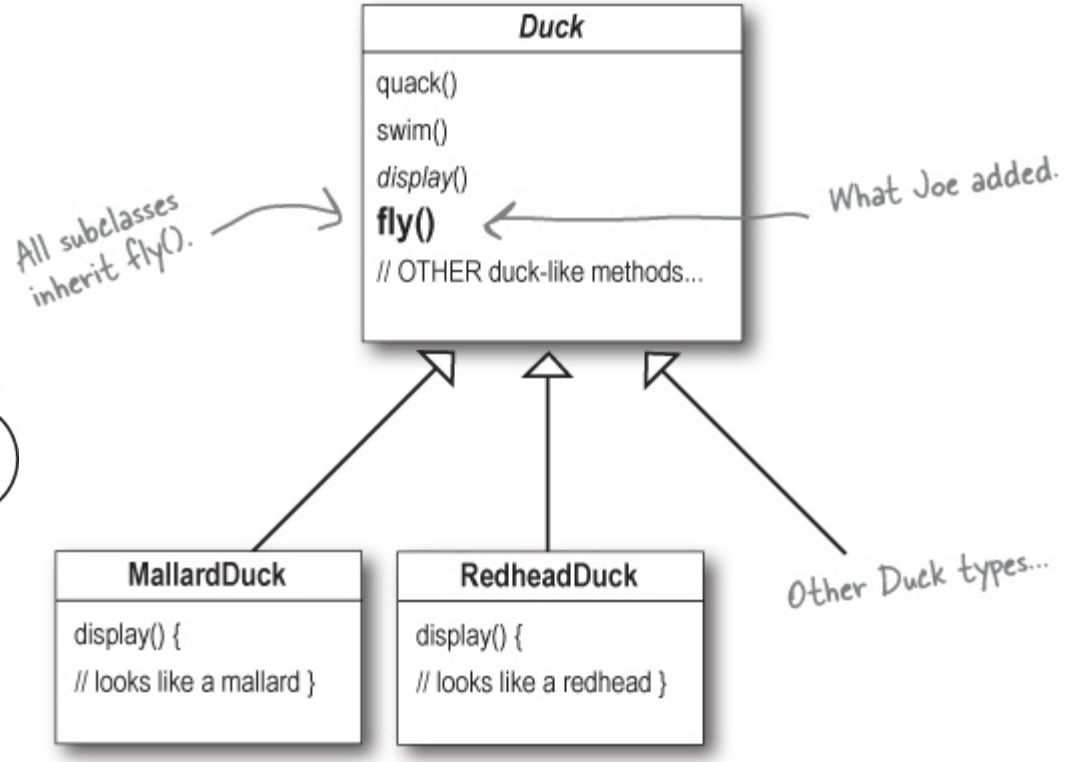


But now we need the ducks to FLY!

What we want

I just need to add a fly() method in the Duck class and then all the ducks will inherit it. Now's my time to really show my true OO genius.

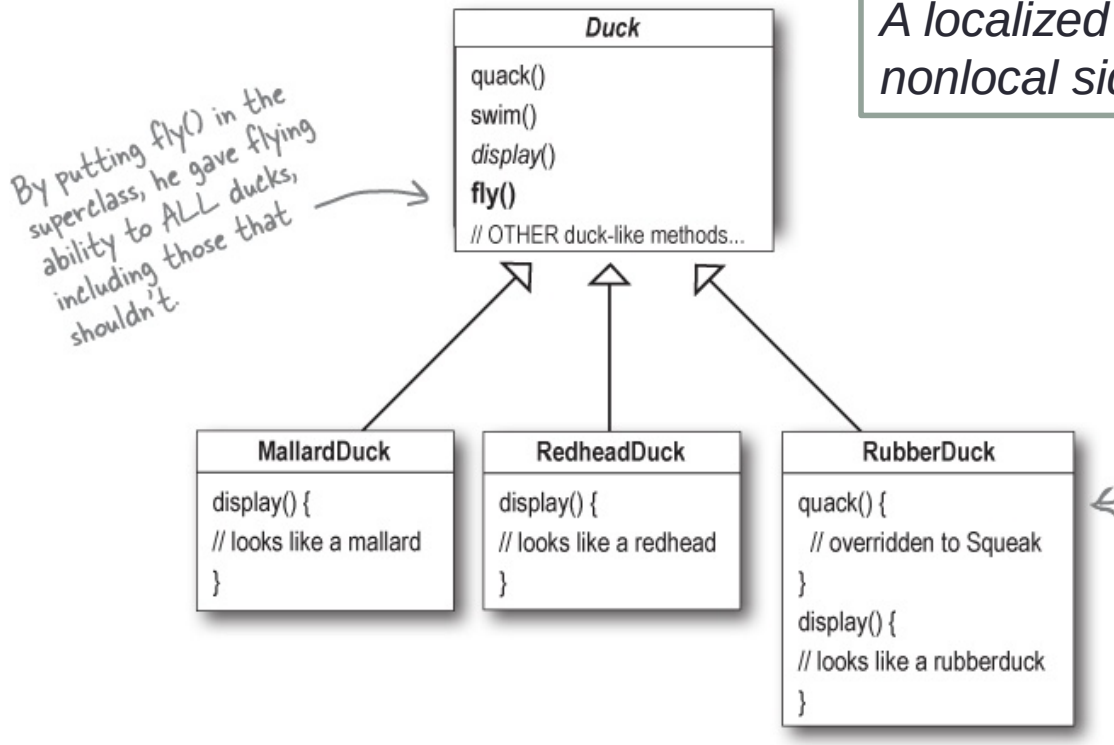
Joe



But something went horribly wrong …

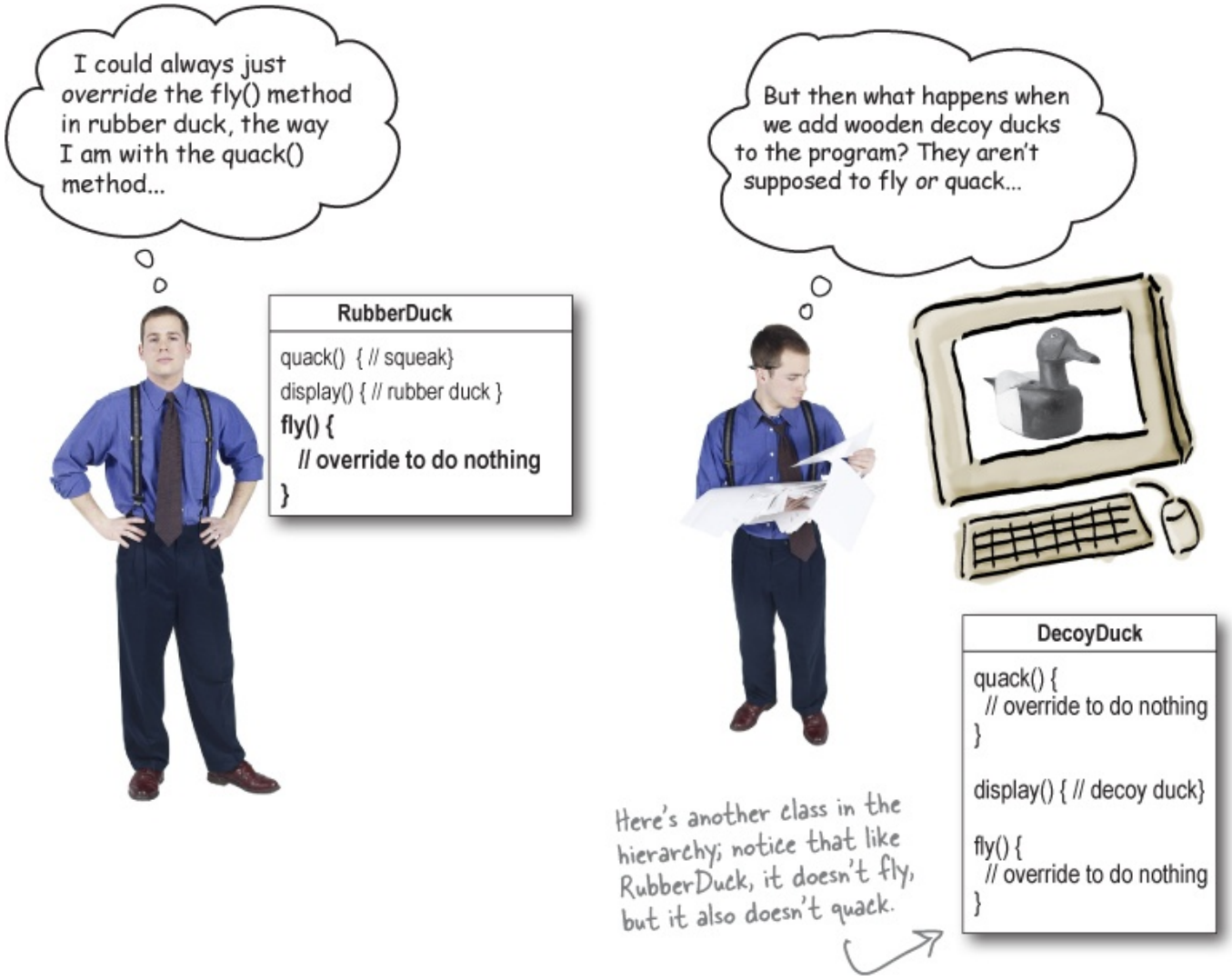
What Happened?

A localized update to the code caused a nonlocal side effect (flying rubber ducks)!



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.

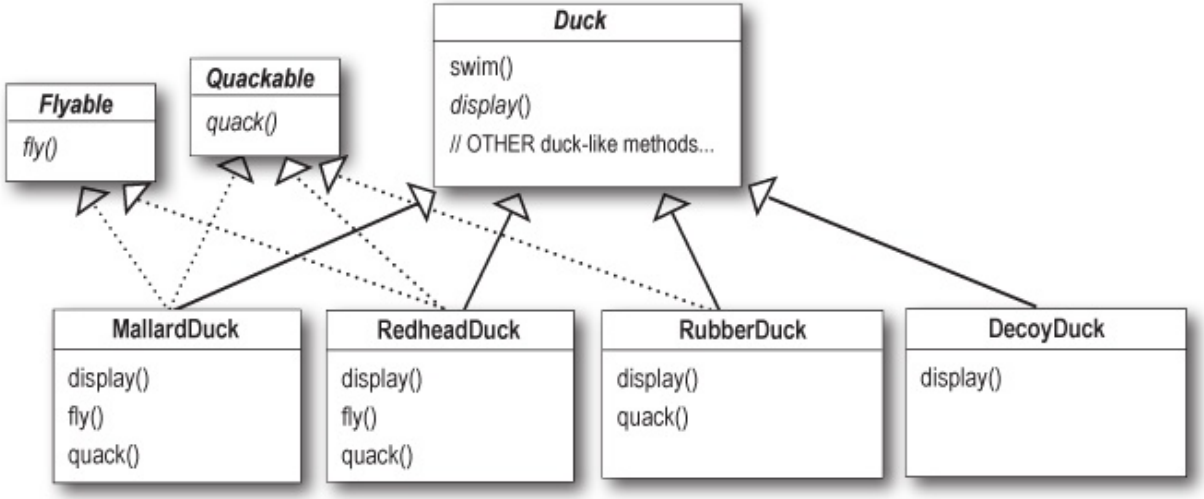
Joe thinks about inheritance …



Q1

How about an interface?

I could take the fly() out of the Duck superclass, and make a ***Flyable() interface*** with a fly() method. That way, only the ducks that are *supposed* to fly will implement that interface and have a fly() method... and I might as well make a Quackable, too, since not all ducks can quack.



What do YOU think about this design?

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?!*

Wouldn't it be dreamy if there were a way to build software so that when we need to change it, we could do so with the least possible impact on the existing code? We could spend less time reworking code and more making the program do cooler things...

Q2

Zeroing in on the problem...

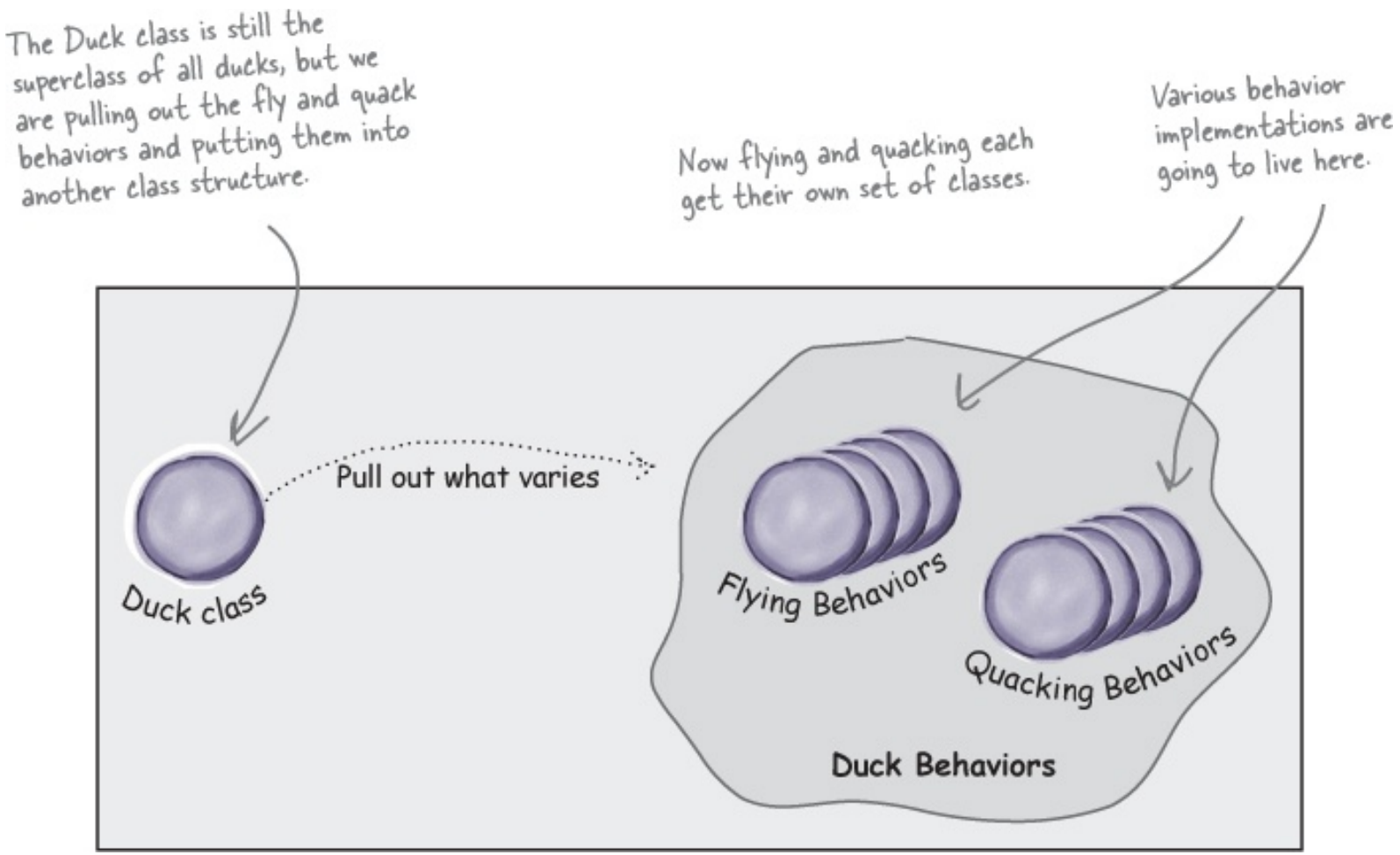
First Design Principle:

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

Another way to think about this principle:

Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.

Applying the principle

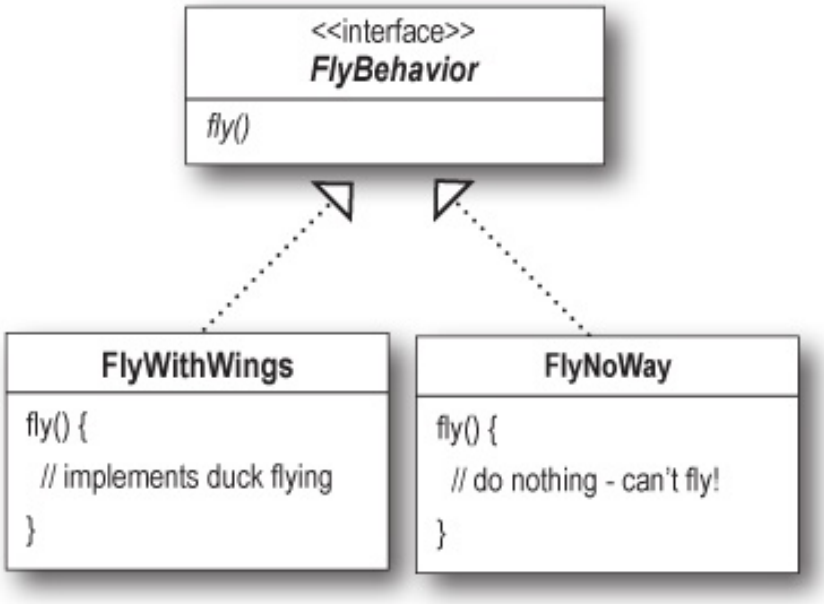


Designing the Duck Behaviors

Dependency Inversion Principle:

Program to an interface, not an implementation.

- From now on, the Duck behaviors will live in a separate class – a class that implements a particular behavior interface.
- That way, the Duck classes won't need to know any of the implementation details for their own behaviors.

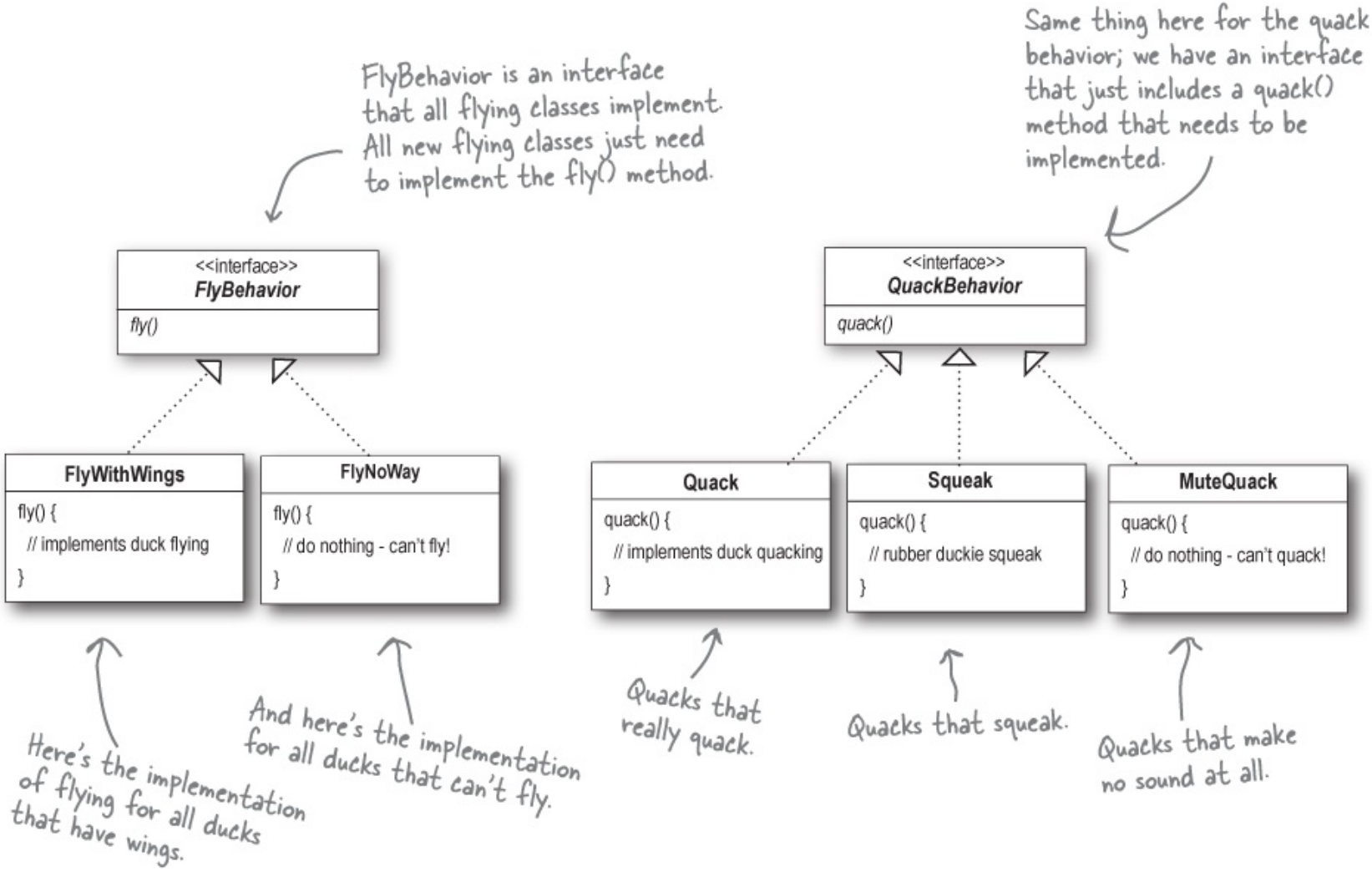


Humm, Interface?

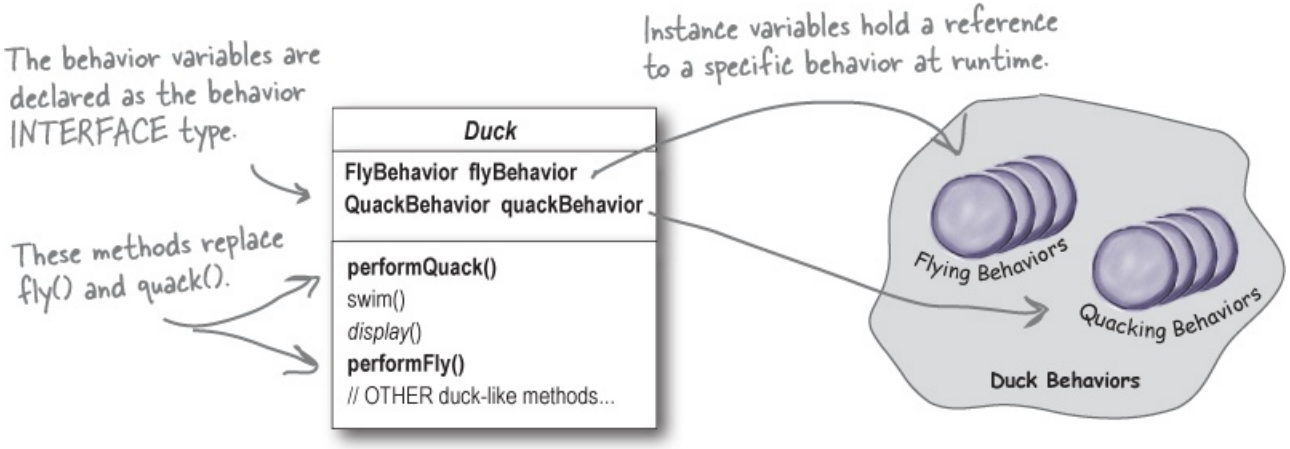
- "Program to an *interface*" really means "Program to a *supertype*."
- You can *program to an interface*, without having to actually use a Java interface.
- The point is to exploit polymorphism by programming to a supertype so that the actual runtime object isn't locked into the code.

Q3

Implementing the Duck Behaviors



Integrating the Duck Behavior




```
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.

More Integration

```
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.

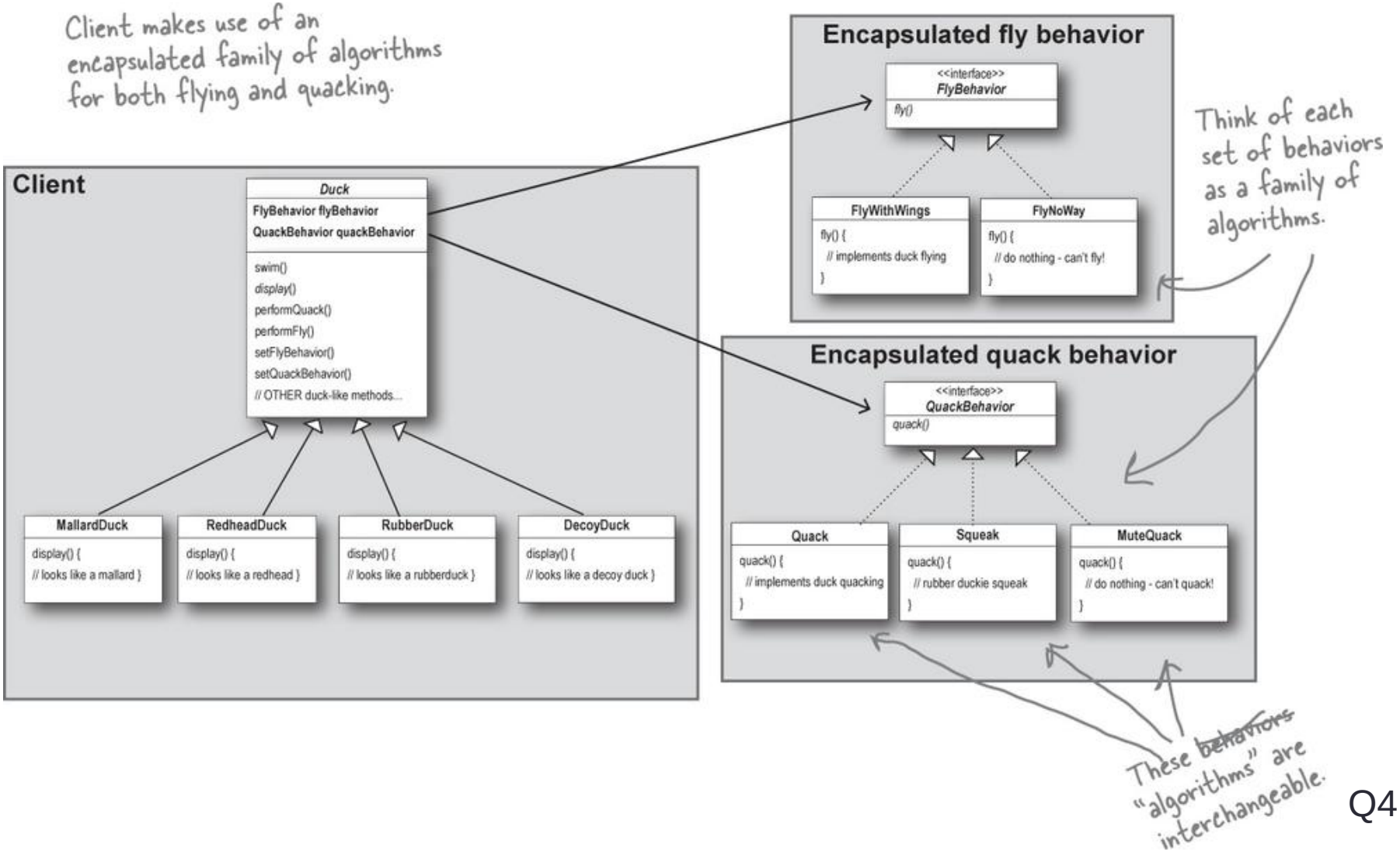
Humm, what about DIP in the constructor?

```
public class MallardDuck extends Duck {

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

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

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

Wait a second, didn't you say we should NOT program to an implementation? But what are we doing in that constructor? We're making a new instance of a concrete Quack implementation class!

The Big Picture



Q4

HAS-A can be better than IS-A

Design Principle:

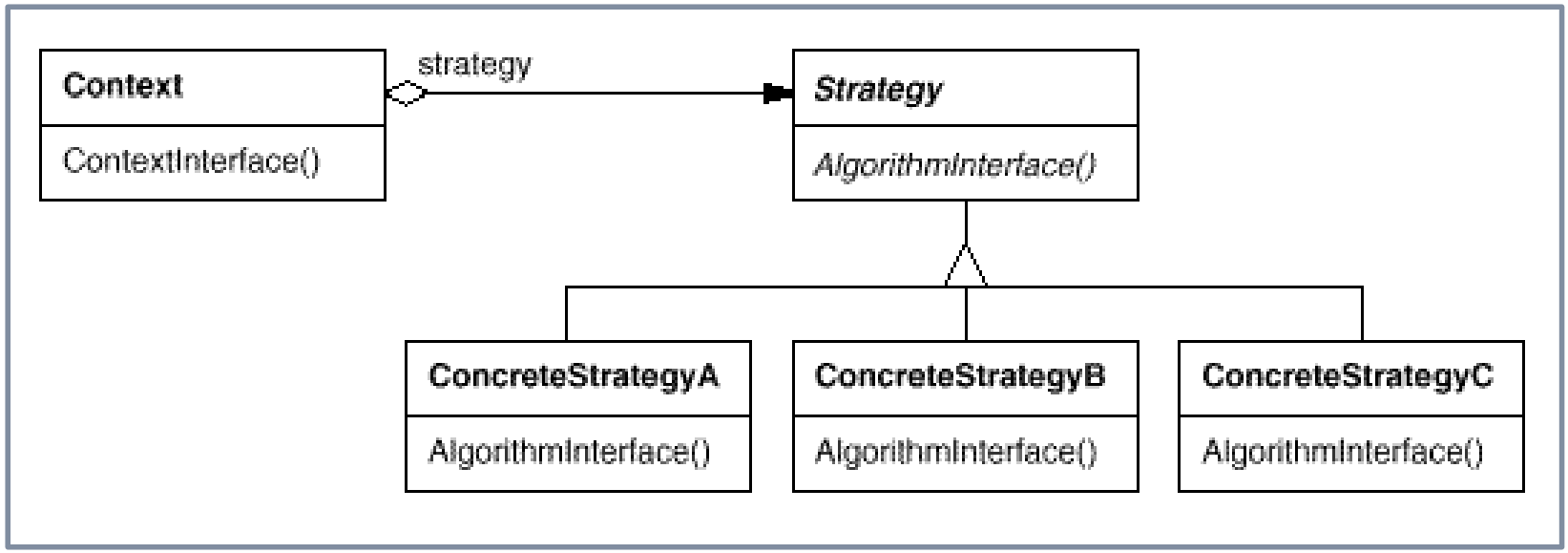
Favor composition over inheritance.

- Each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking.
- When you put two classes together like this you're using **composition**. Instead of *inheriting* their behavior, the ducks get their behavior by being *composed* with the right behavior object.

Recap

Design Principle:

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



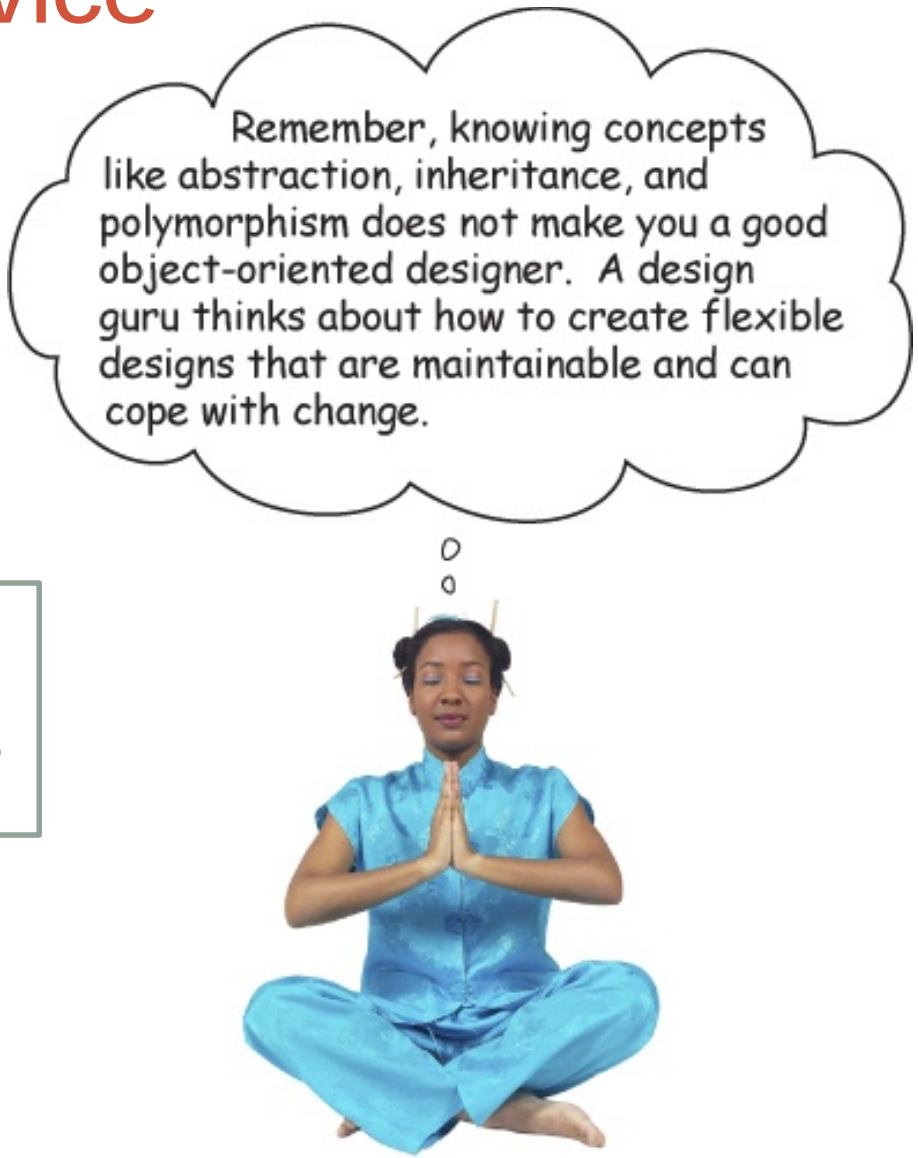
Design Principle (DIP):

Program to an interface, not an implementation.

Design Principle:

Favor composition over inheritance.

Final Words of Advice



Take a 5 mins. break and let's work on the lab problems after you return.