

# INTRODUCTION TO DESIGN PATTERNS

---

Chandan R. Rupakheti

Week 1-1

# Overview

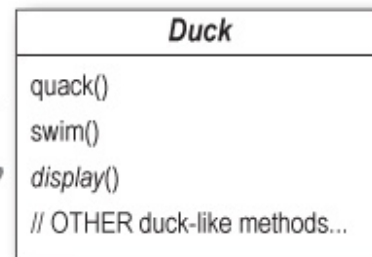
- Why Design Patterns?
- Some key OO design principles
- Identifying variability and invariant of software system
- The Strategy Pattern



# SimUDuck App

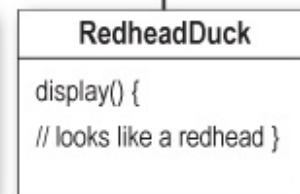
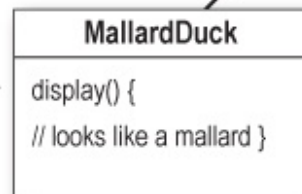


All ducks quack and swim. The superclass takes care of the implementation code.



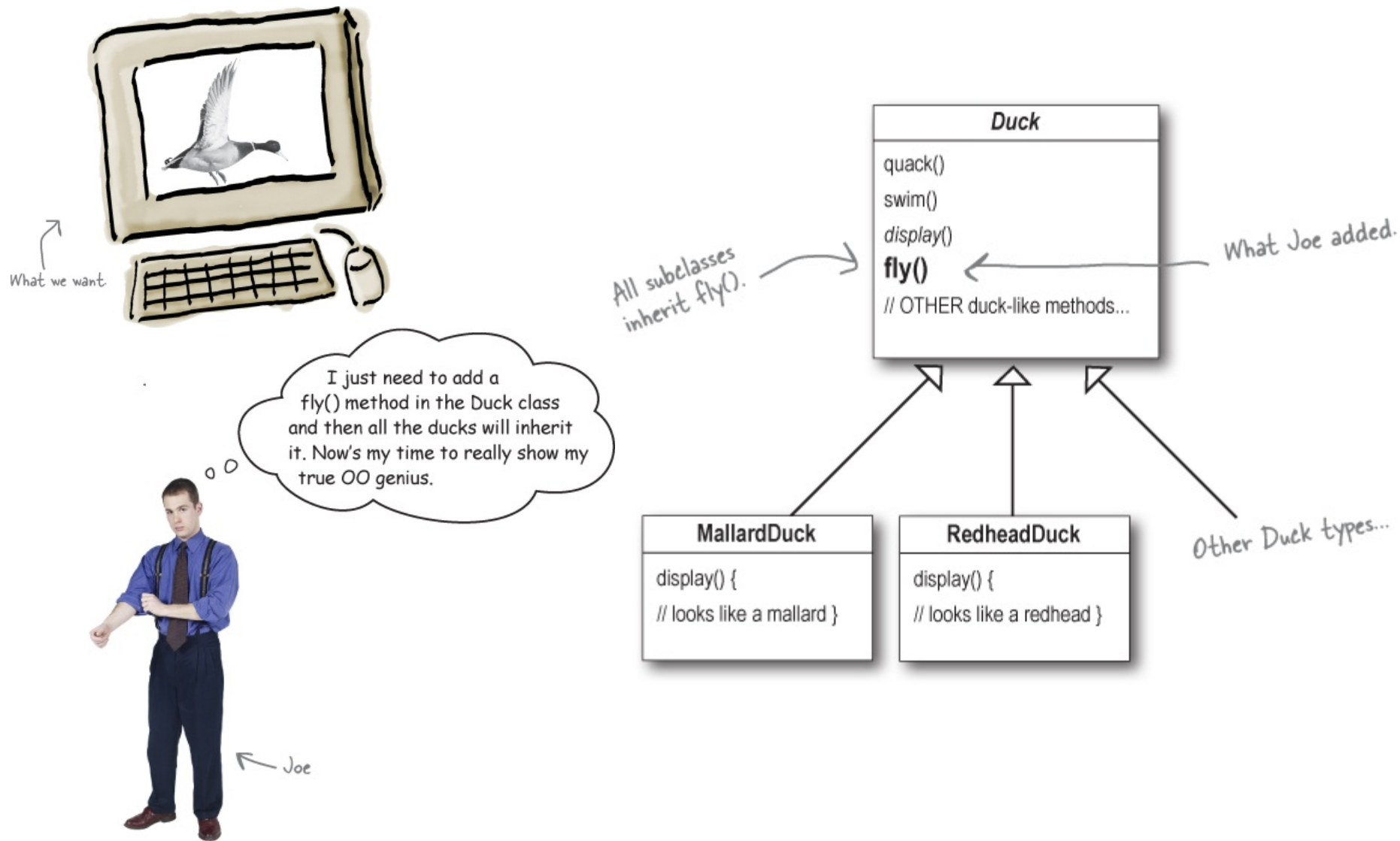
The display() method is abstract, since all duck subtypes look different.

Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen.



Lots of other types of ducks inherit from the Duck class.

# But now we need the ducks to FLY!



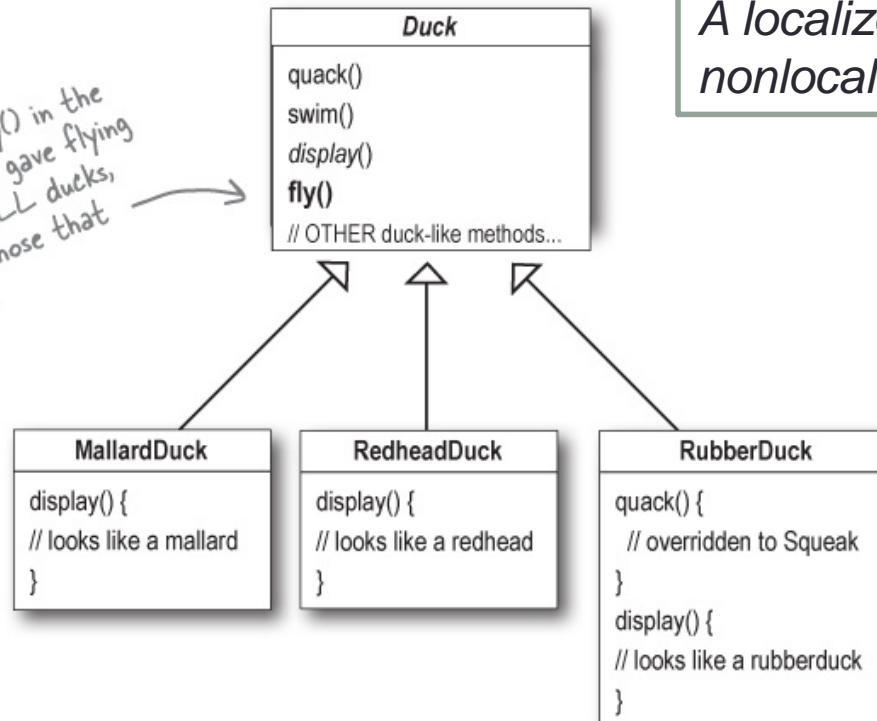
# But something went horribly wrong ...

Joe, I'm at the shareholder's meeting. They just gave a demo and there were rubber duckies flying around the screen. Was this your idea of a joke? You might want to spend some time on [Monster.com](http://Monster.com)...



# What Happened?

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



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

OK, so there's a slight flaw in my design. I don't see why they can't just call it a "feature." It's kind of cute...

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

*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 ...

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...



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

display() { // decoy duck }

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

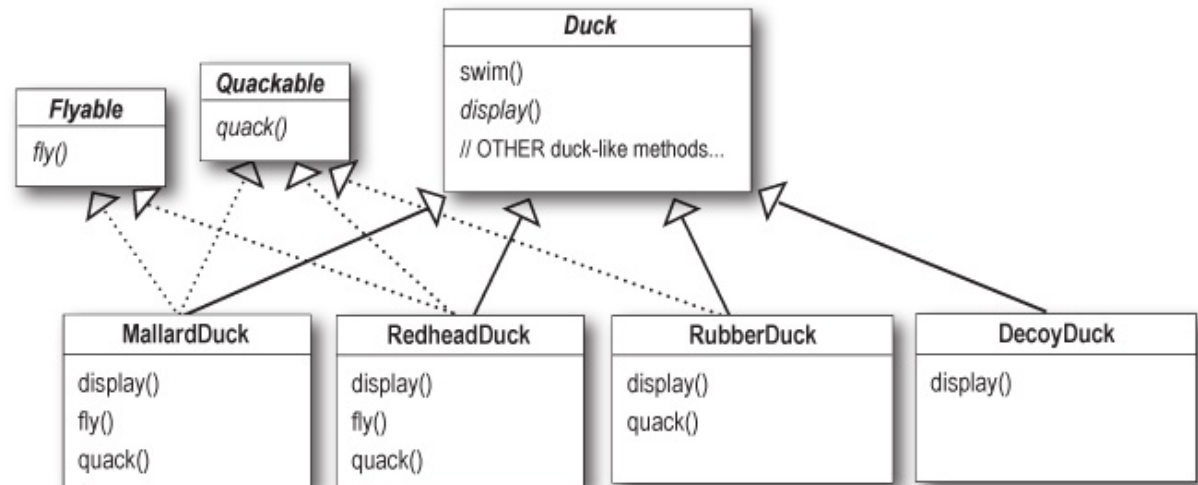
Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.





# 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...





# 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:**

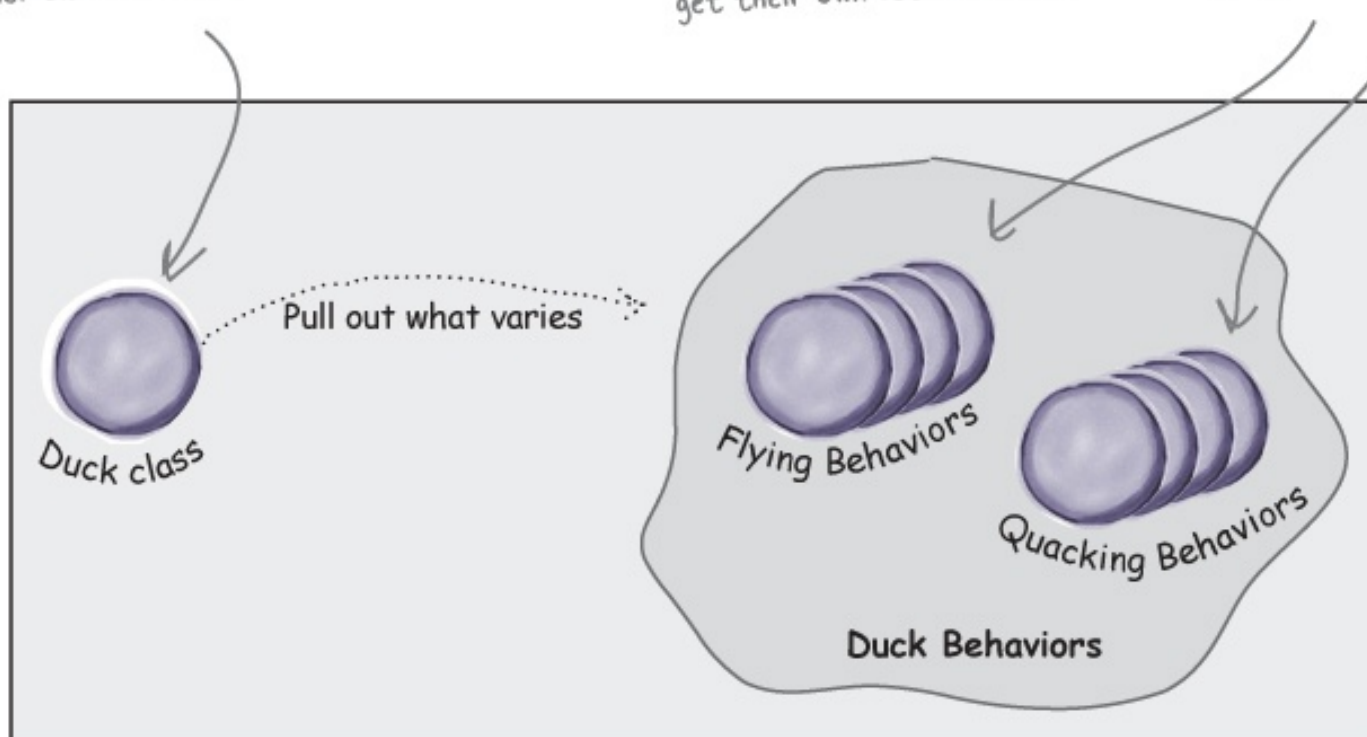
Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.

# Applying the principle

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.

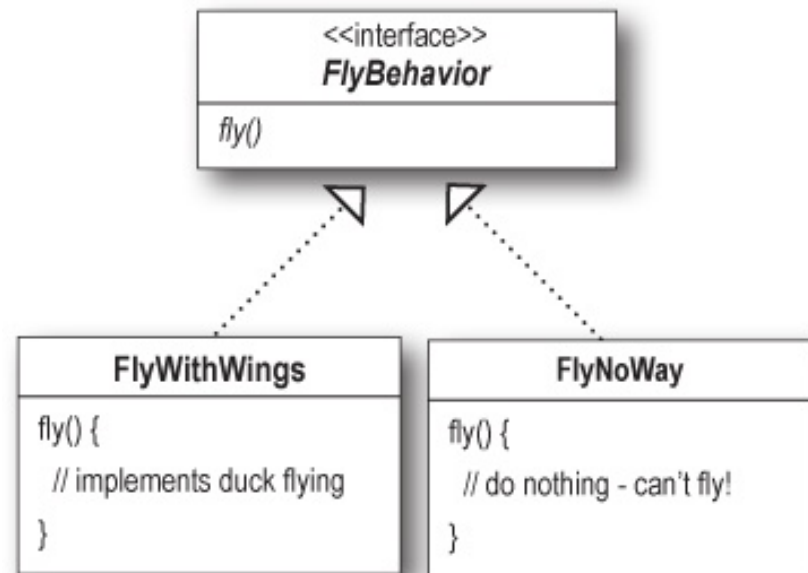


# Designing the Duck Behaviors

## 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.



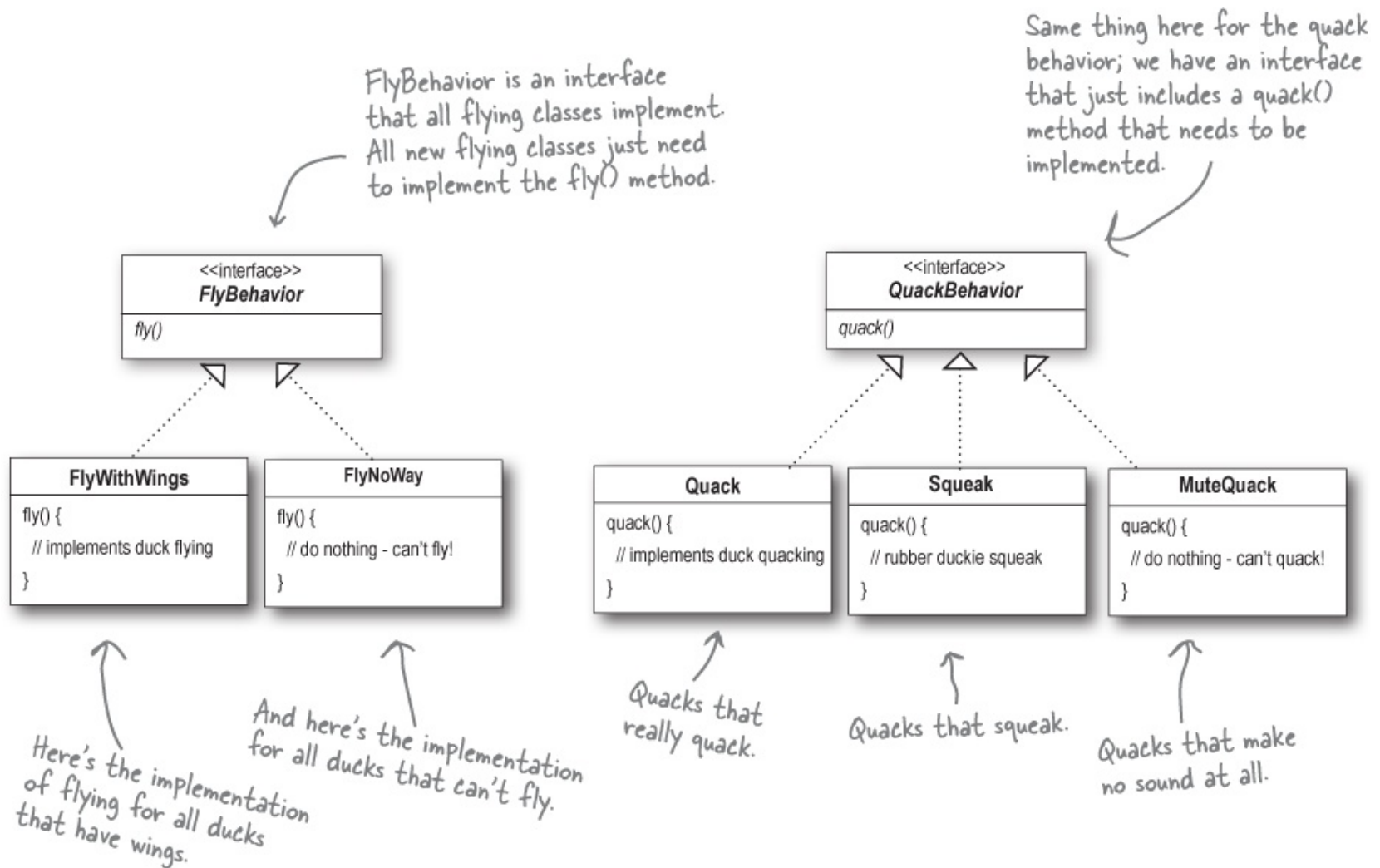
# Humm, Interface?

- “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.

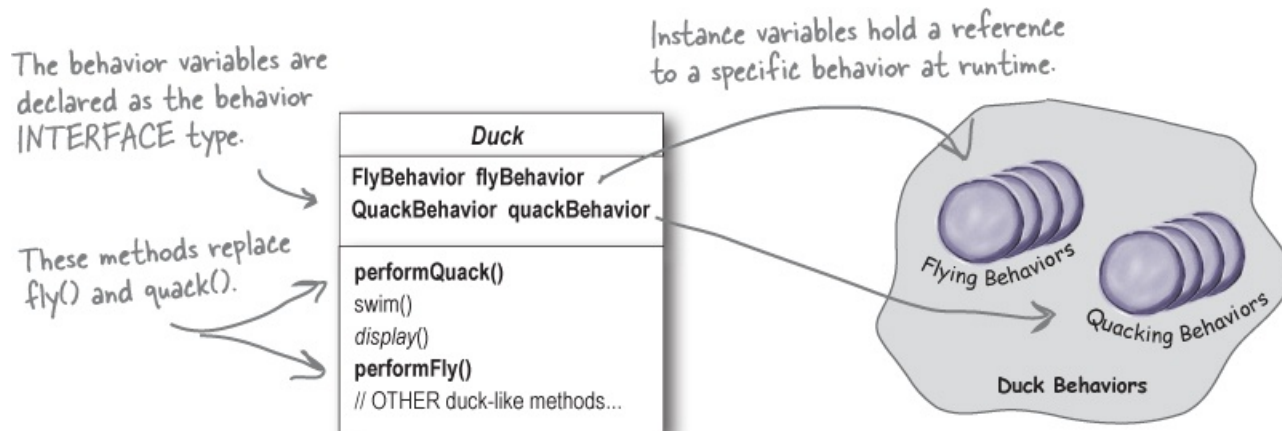


I don't see why you have to use an *interface* for FlyBehavior. You can do the same thing with an abstract superclass. Isn't the whole point to use polymorphism?

# 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();  
    }
```

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

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

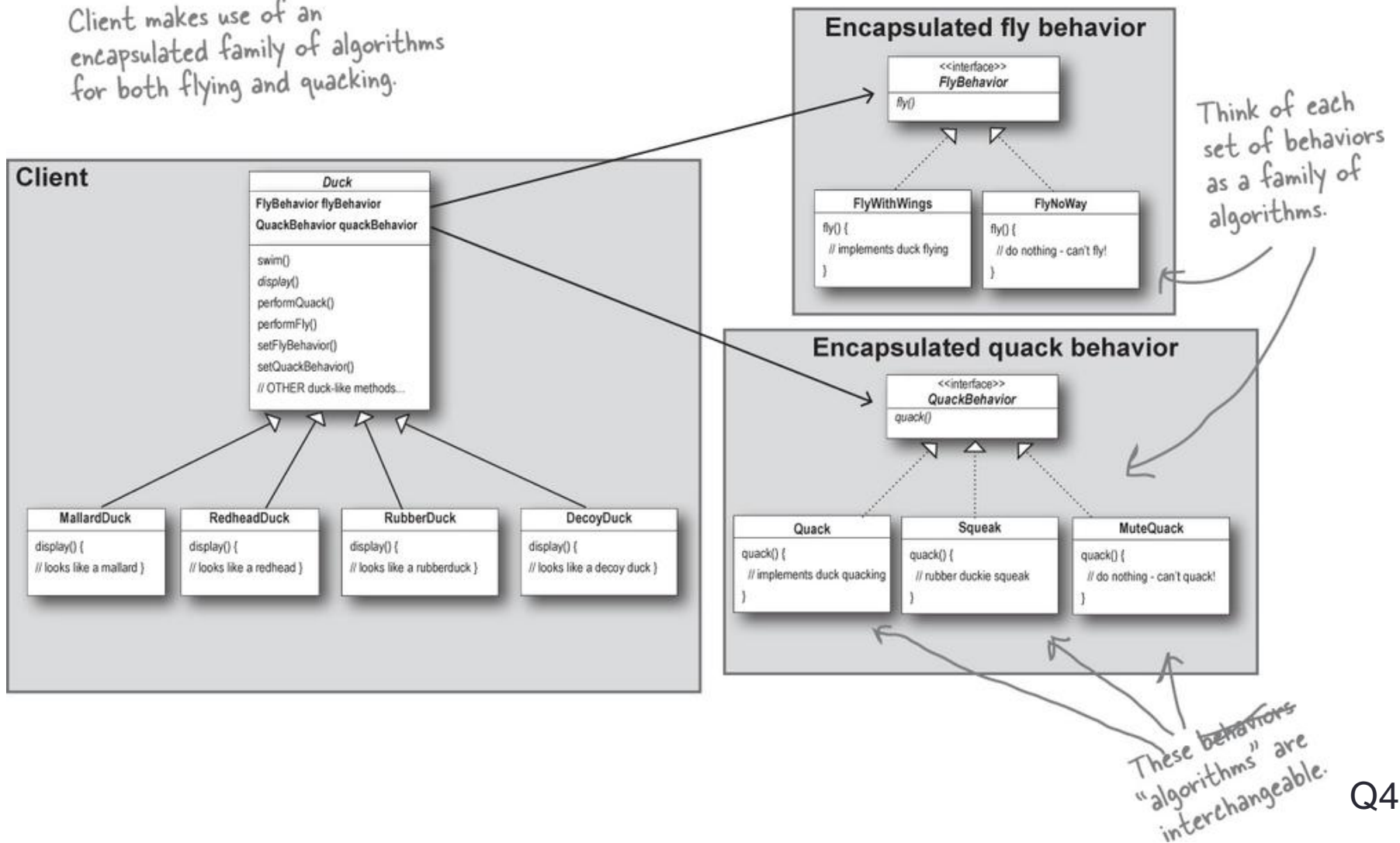
Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

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

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



# HAS-A can be better than IS-A

## Design Principle:

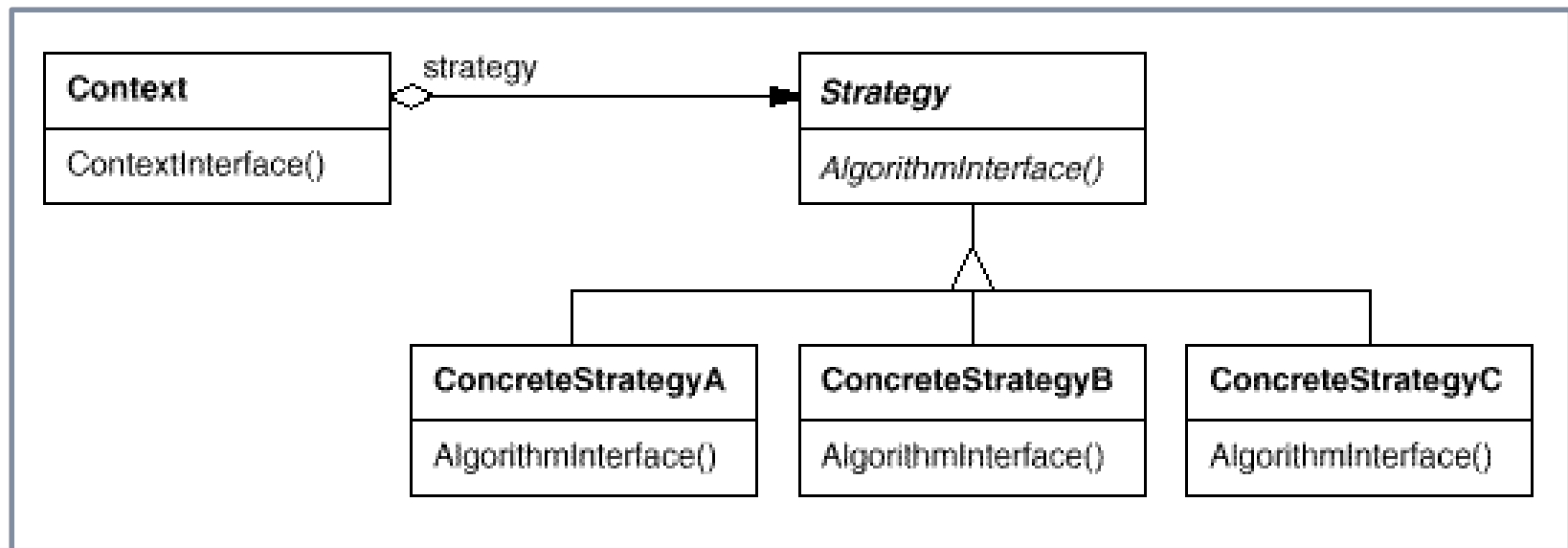
*Favor composition over inheritance.*

- 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

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

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

