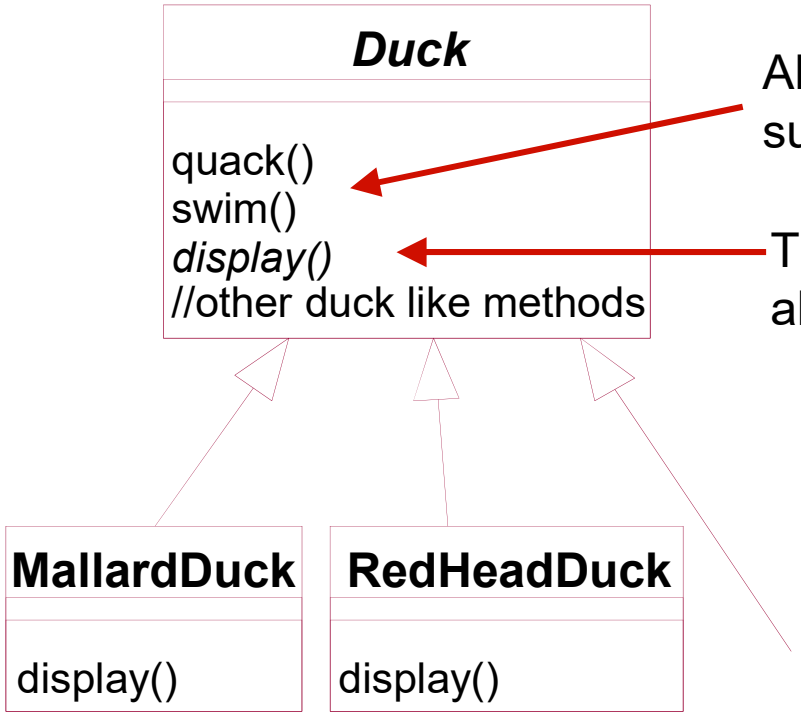
The Strategy Pattern

The Specifications

- Joe works at a company that produces a simulation game called *SimUDuck.*
  He is an OO Programmer and his duty is to implement the necessary functionality for the game

- The game should have the following specifications:
  - **A variety of different ducks should be integrated into the game**
  - **The ducks should swim**
  - **The ducks should quack**

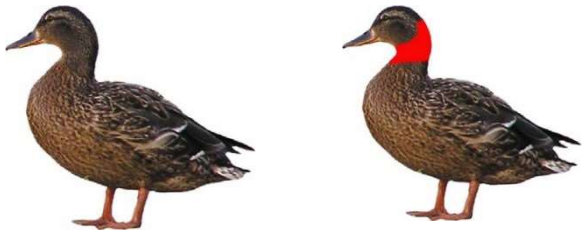
A First Design for the Duck Simulator Game



All ducks `quack()` and `swim()`. The superclass takes care of the implementation

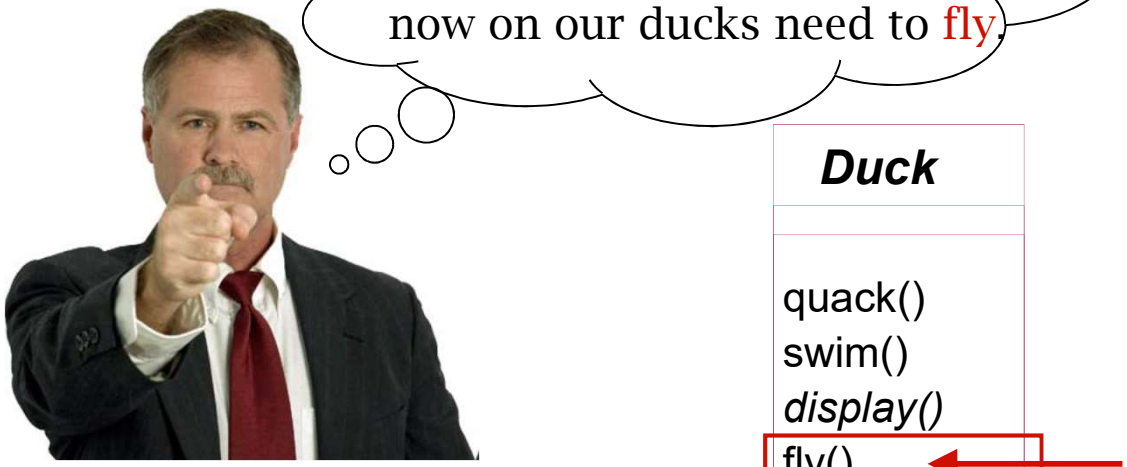
The `display()` method is **abstract** since all the duck subtypes look different

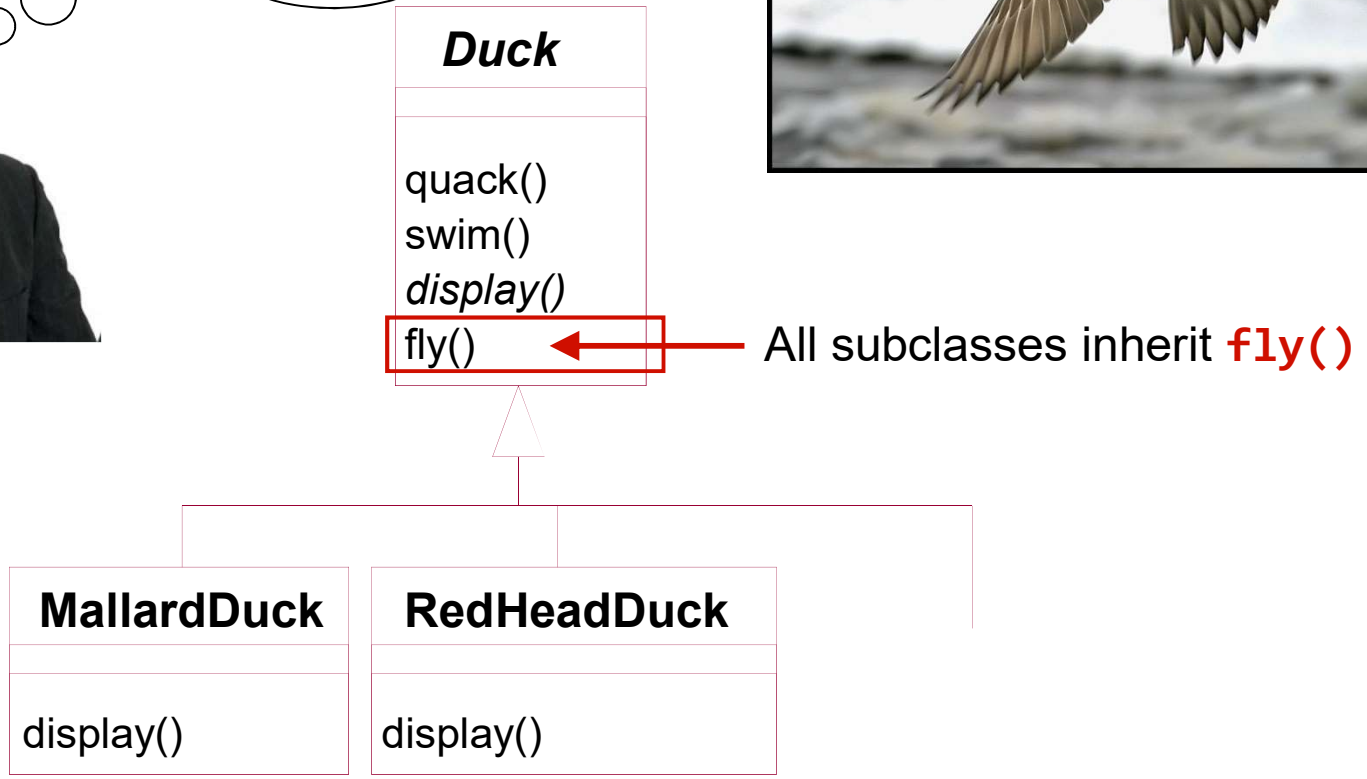
Each duck subtype is responsible for implementing its own `display()` behavior for how it looks on the screen



Lots of other types of ducks inherit from the Duck type

But now we need the ducks to fly…

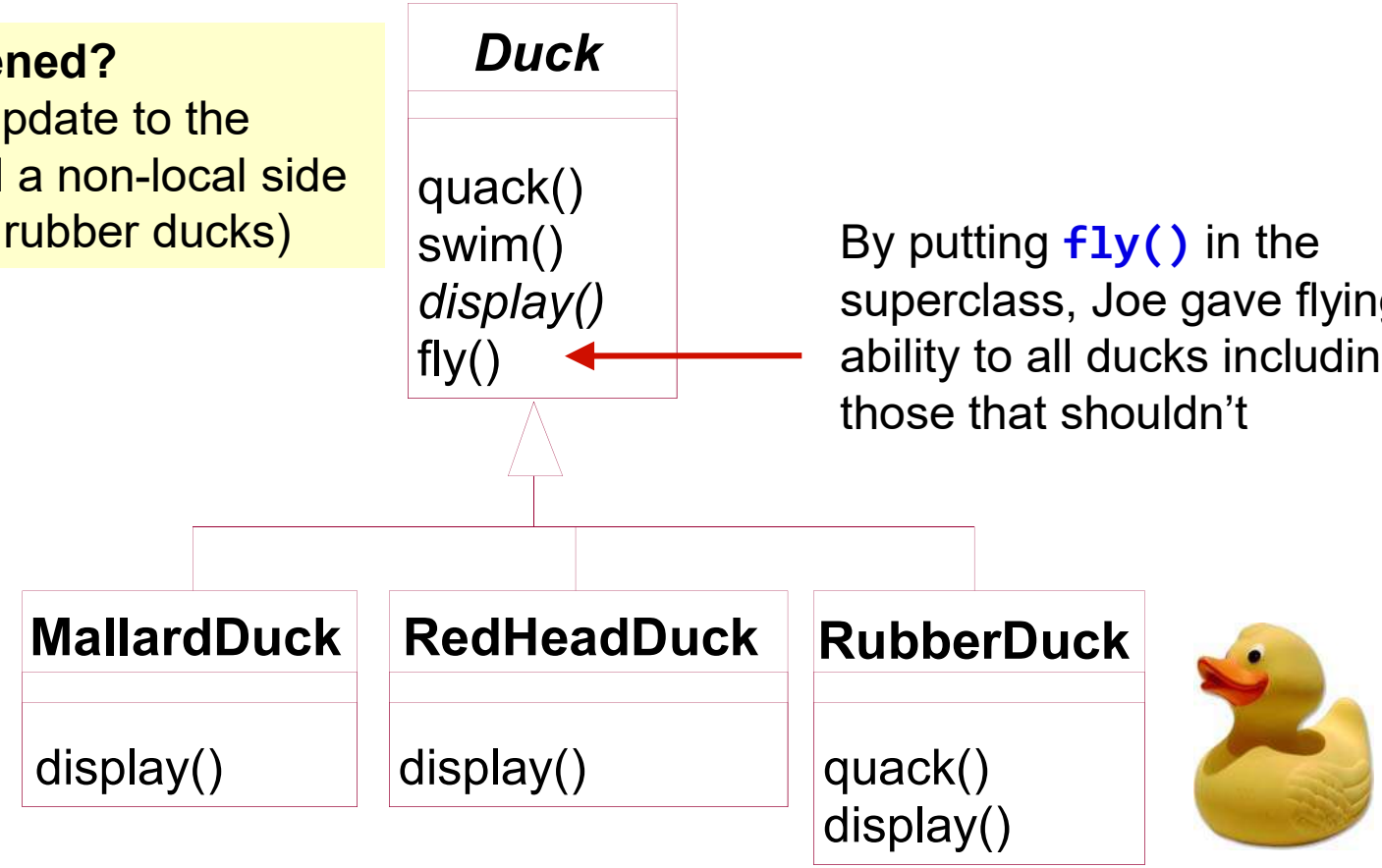




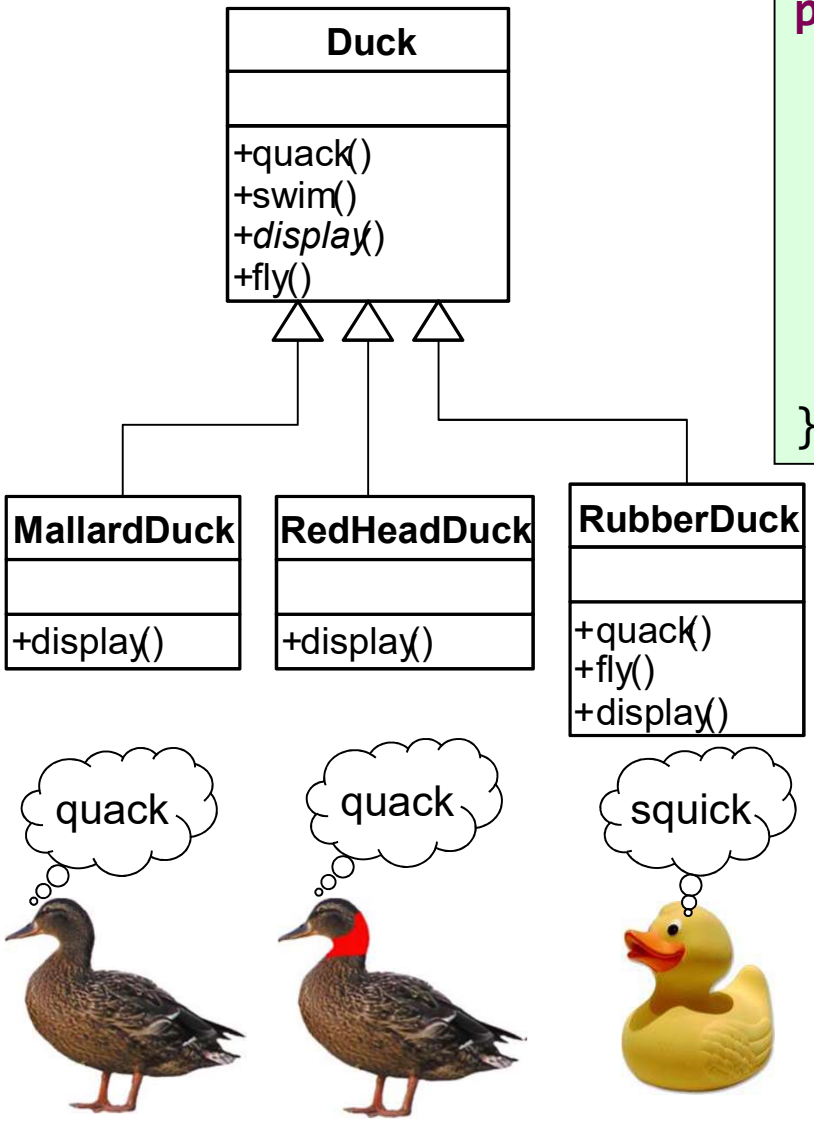
But something went horribly wrong..

- At a demo the program failed to impress anyone – There were rubber ducks flying across the screen!

What happened?
A localized update to the code caused a non-local side effect (flying rubber ducks)



Inheritance at Work

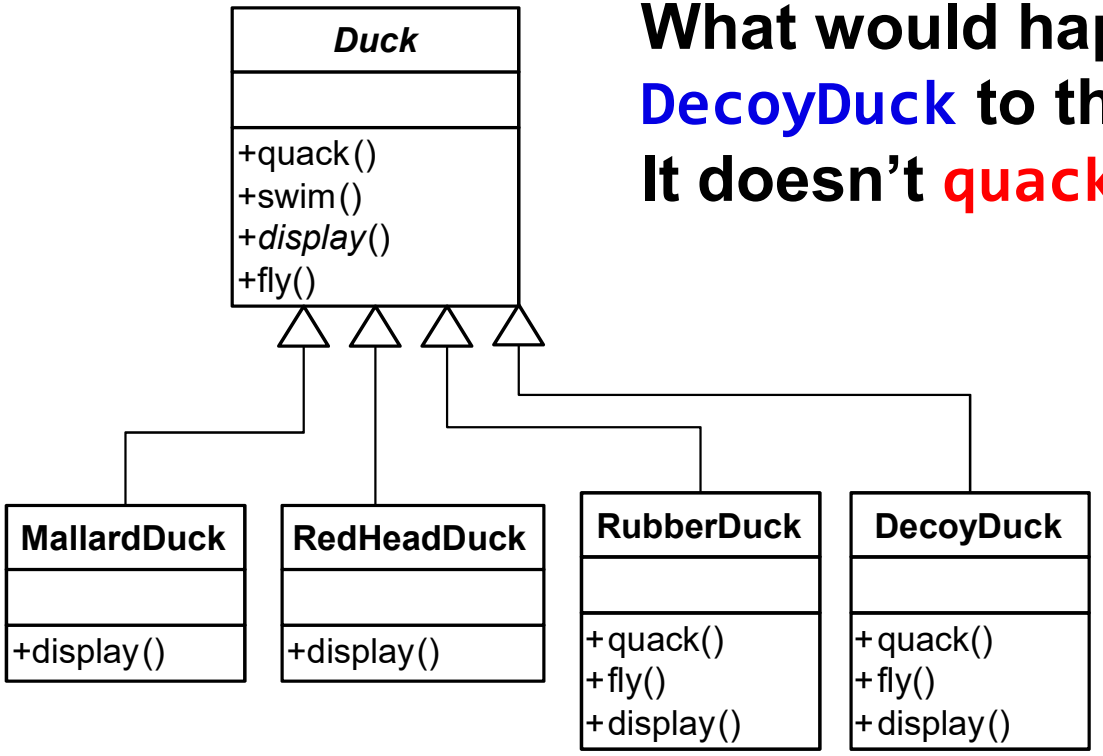



```
public class Duck {
   ...
   public void fly() {
     // fly implementation
   }
   public void quack() {
     System.out.println("quack, quack");
   }
}
```

We can override the **fly()** method in the rubber duck in a similar way that we override the **quack()** method

```
public class RubberDuck
                    extends Duck {
   ...
   public void fly() { }
   public void quack() {
     System.out.println(
             "squick, squick");
   }
}
```

Yet Another Duck is Added to the Application



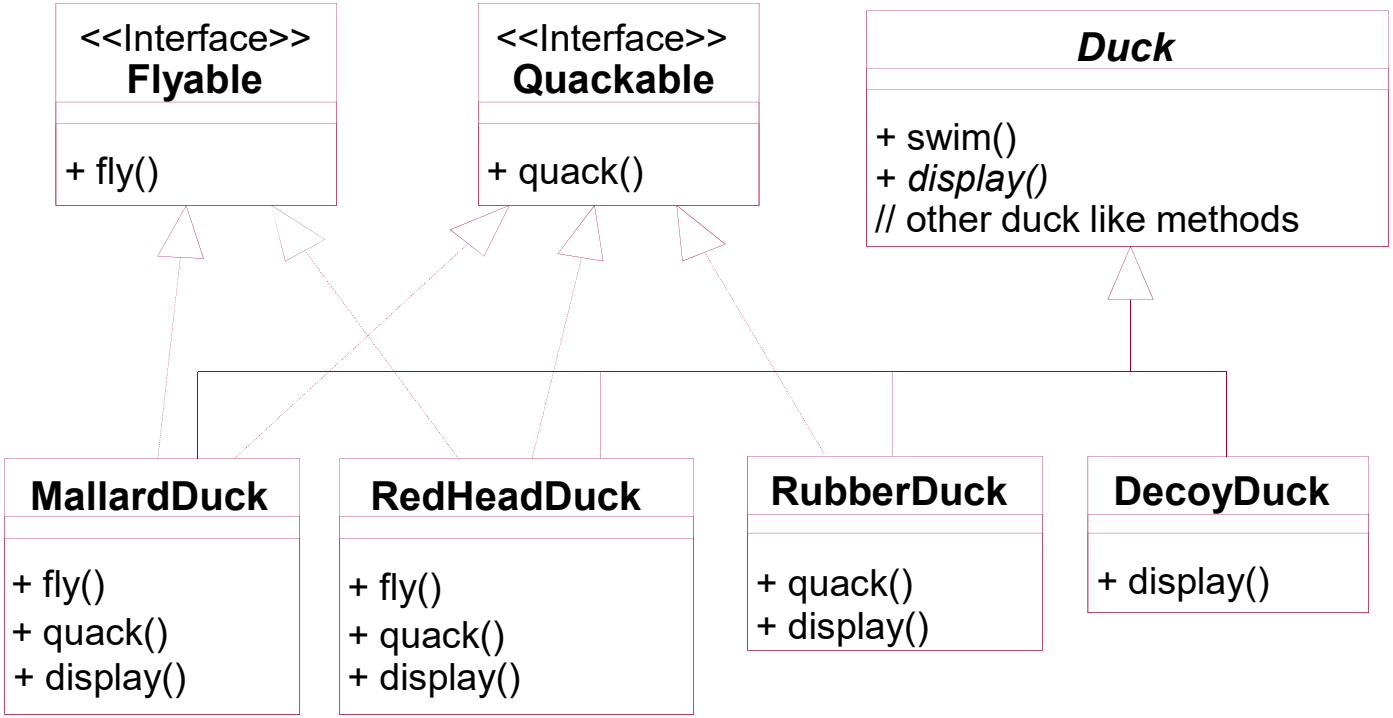
What would happen if we added a `DecoyDuck` to the class hierarchy? It doesn't `quack()` or `fly()`.

```
public class DecoyDuck
    extends Duck {
  ...

  public void fly() {
    // do nothing
  }

  public void quack() {
    // do nothing
  }
}
```

How about an interface?

- Need a cleaner way to make some ducks fly or quack.
  - Could take the `fly()` out of the superclass and make an `Flyable` interface with a `fly()` method. Each duck that is supposed to fly will implement that interface
  - and maybe a `Quackable` interface as well.

What do you think about this design?



What do you think?

- Dumb!!!!

- “Duplicate code” all over the place.
  - Interface not reuse code
  - A small change to the flying behavior will require changing all 48 of the Duck subclasses!

Embracing Change

- In SOFTWARE projects you can count on one thing that is constant:

CHANGE

- Solution
  - Deal with it.
    - Make CHANGE part of your design.
    - Identify what vary and separate from the rest.

Let's shoot some ducks!

Change is a taste of life

Sharpen your pencil

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).

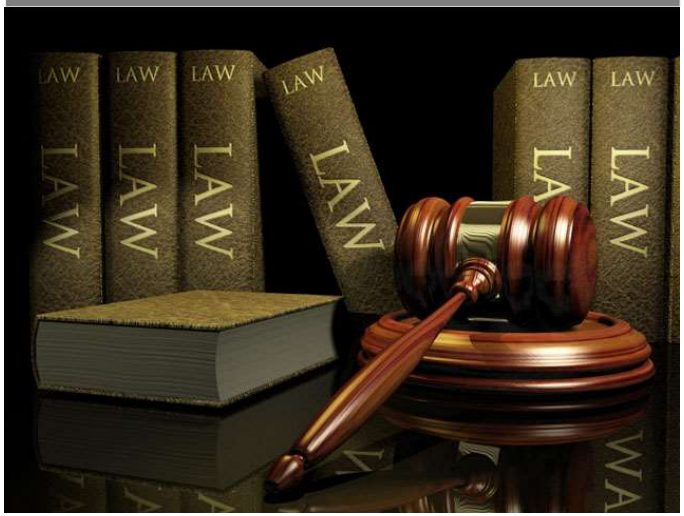
My customers or users decide they want something else, or they want new functionality.

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!

Design Principle

Encapsulate what varies

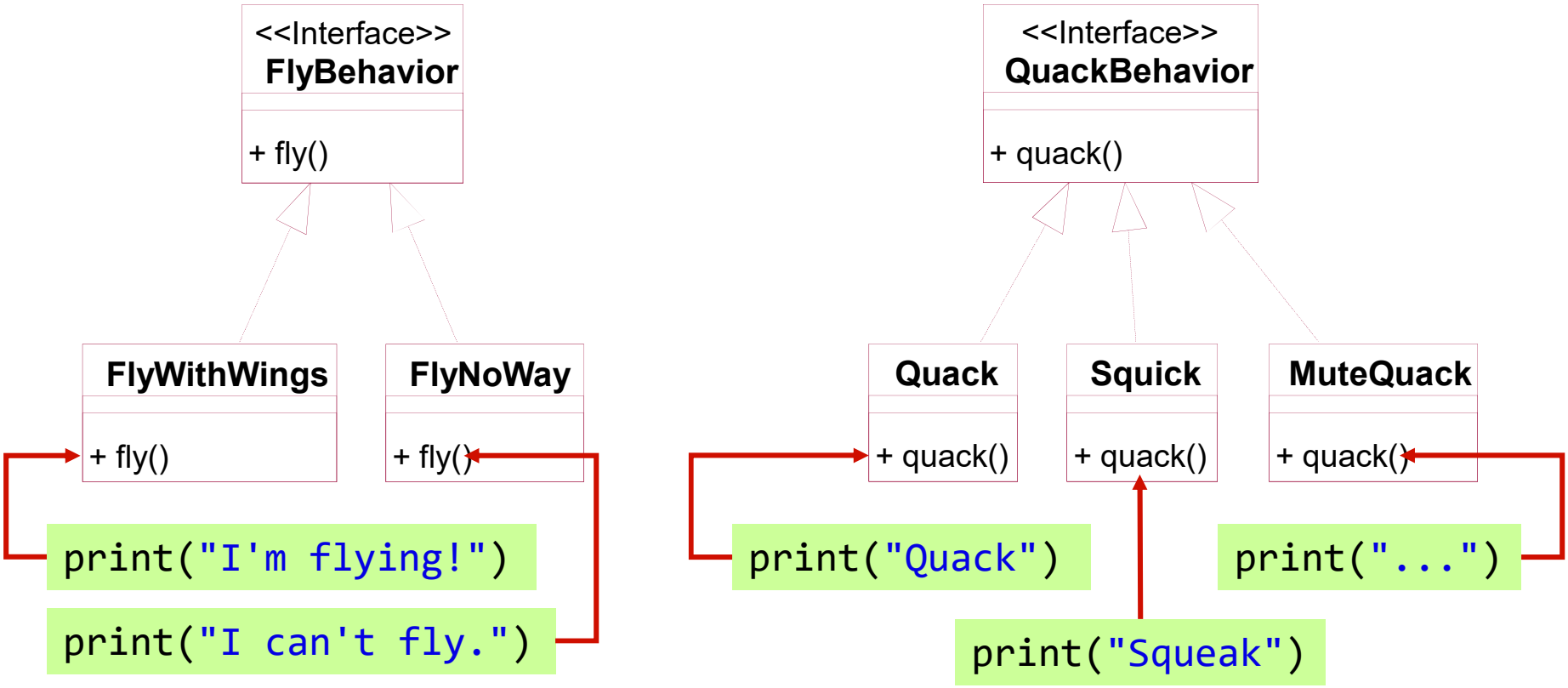
The Constitution of Software Architects



- Encapsulate what varies.
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????

Embracing Change in Ducks

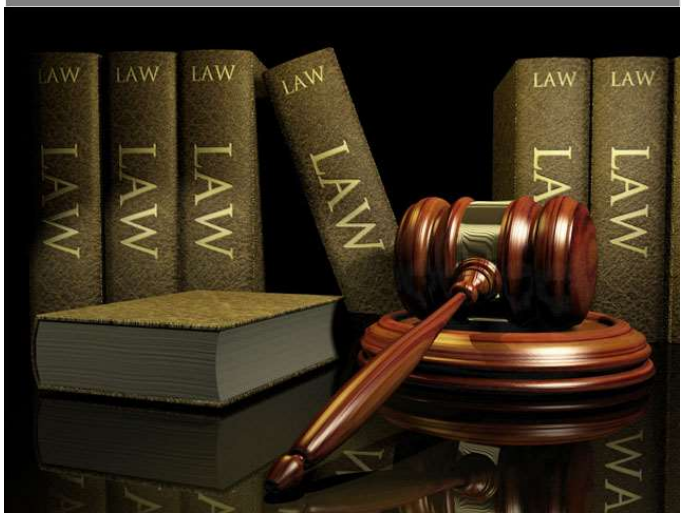
- `fly()` and `quack()` are the parts that vary
- We create a new set of classes to represent each behavior



Design Principle

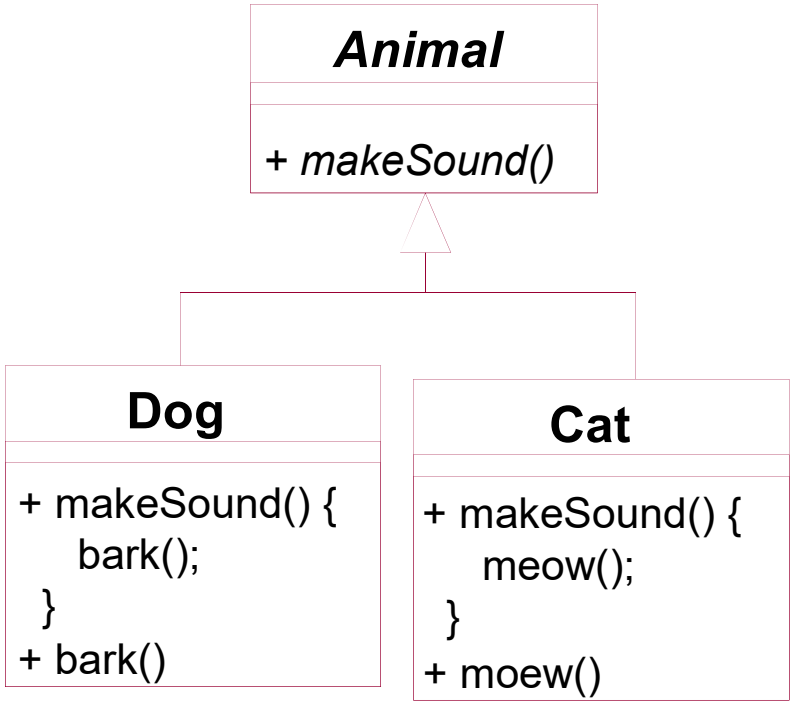
Program to an interface not to an implementation

The Constitution of Software Architects



- Encapsulate what varies.
- Program through an interface not to an implementation
- ??????????
- ??????????
- ??????????
- ??????????
- ??????????
- ??????????
- ??????????

Design Principle Example



Program through an implementation

```
Dog dog = createDog();
dog.bark();
```

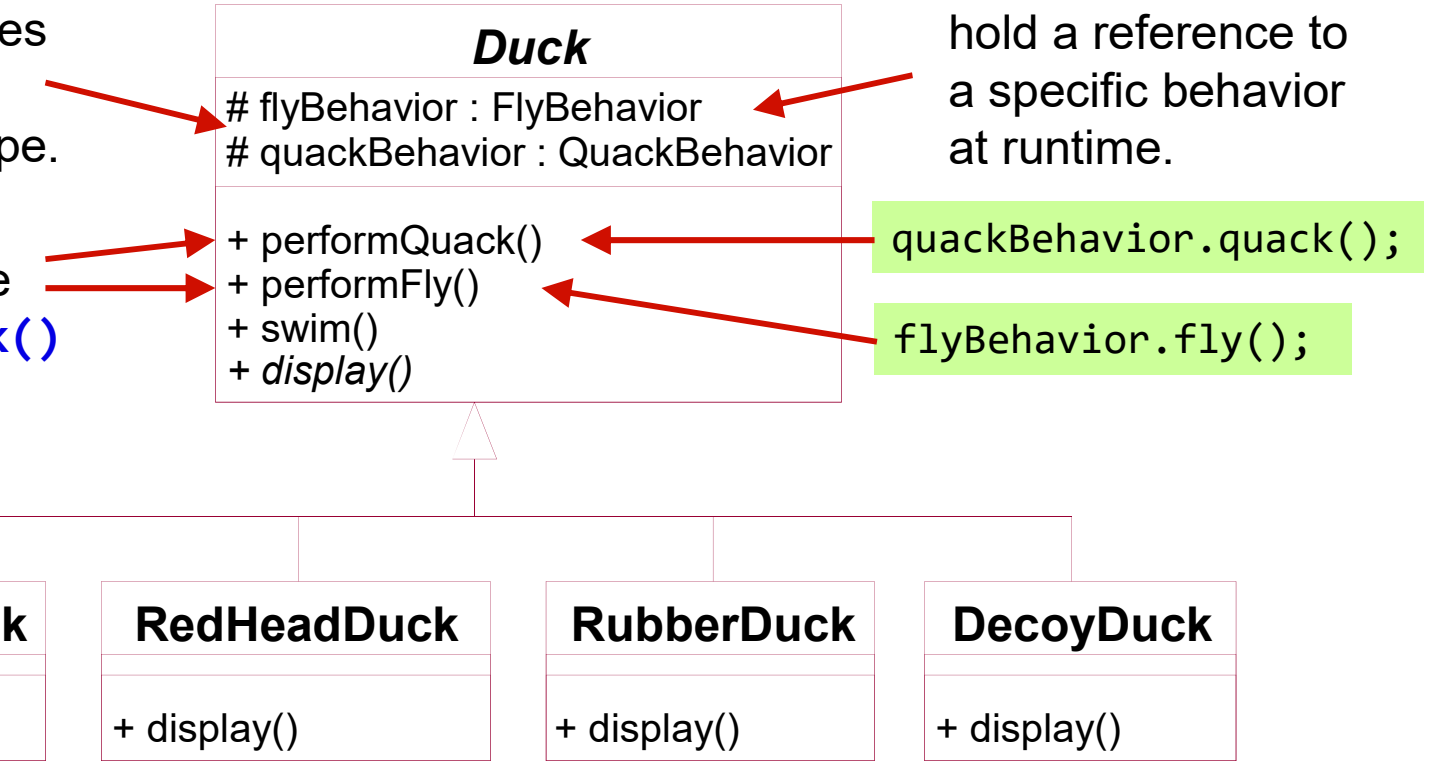
Program through an interface

```
Animal dog = createDog();
dog.makeSound();
```

Integrating the Duck Behavior

The behavior variables are declared as the behavior interface type.

These replace the `fly()` and `quack()` methods.



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

```
quackBehavior.quack();
```

```
flyBehavior.fly();
```

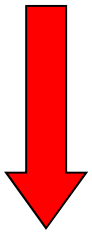
```
class MallardDuck extends Duck
  public MallardDuck(){
    flyBehavior = new FlyWithWings();
    quackBehavior = new Quack();
  }
}
```

```
class DecoyDuck extends Duck
  public DecoyDuck(){
    flyBehavior = new FlyNoWay();
    quackBehavior = new MuteQuack();
  }
}
```

Design Principle Ahead

Duck
flyBehavior : FlyBehavior # quackBehavior : QuackBehavior
+ performQuack() + performFly() + swim() + *display()*

Each Duck **HAS A** `FlyingBehavior` and a `QuackBehavior` to which it delegates flying and quacking behaviors



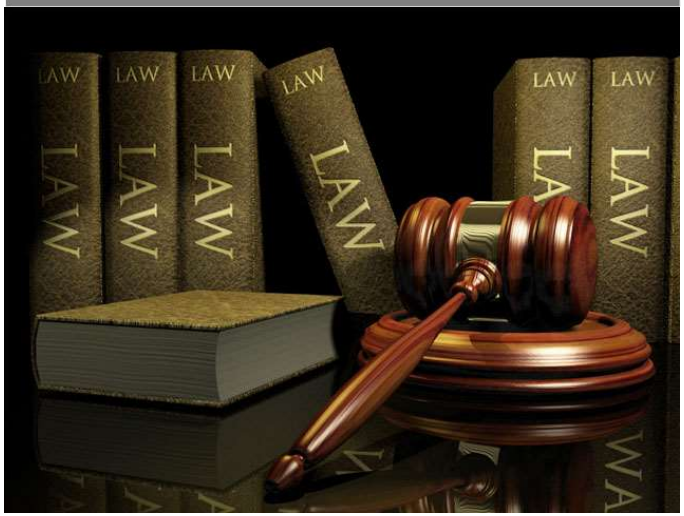
Composition

Instead of inheriting behavior, the duck get their behavior by being composed with the right behavior object

Design Principle

Favor Composition over Inheritance

The Constitution of Software Architects



- Encapsulate what varies.
- Program through an interface not to an implementation
- Favor Composition over Inheritance
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????

Putting it together…

```
public interface QuackBehavior {
    void quack();
}
```

```
public interface FlyBehavior {
    void fly();
}
```

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

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

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

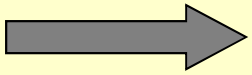
Putting it together…

```
public class MallardDuck extends Duck {
  public MallardDuck() {
    quackBehavior = new Quack();
    flyBehavior = new FlyWithWings();
  }
}
```

```
public class Quack implements QuackBehavior {
  public void quack() {
    System.out.println("Quack");
  }
}
```

```
public class FlyWithWings implements FlyBehavior {
  public void fly() {
    System.out.println("I'm flying!!");
  }
}
```

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



```
I'm a real Mallard duck
Quack
I'm flying!!
```

Setting Behavior Dynamically!

Duck

- flyBehavior : FlyBehavior
- quackBehavior : QuackBehavior

+ performFly()
+ performQuack()
+ swim()
+ display()
+ setFlyBehavior(fb : FlyBehavior)
+ setQuackBehavior(qb : QuackBehavior)

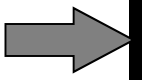
```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}
public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```

// Test it out in main

```
Duck model = new ModelDuck();
model.performFly();
model.setFlyBehavior(
          new FlyRocketPowered());
model.performFly();
```

// Create a new type of Duck

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        setFlyBehavior(new FlyNoWay());
        setQuackBehavior(new Quack());
    }
    public void display() {
        System.out.println("I'm a model duck");
    }
}
```

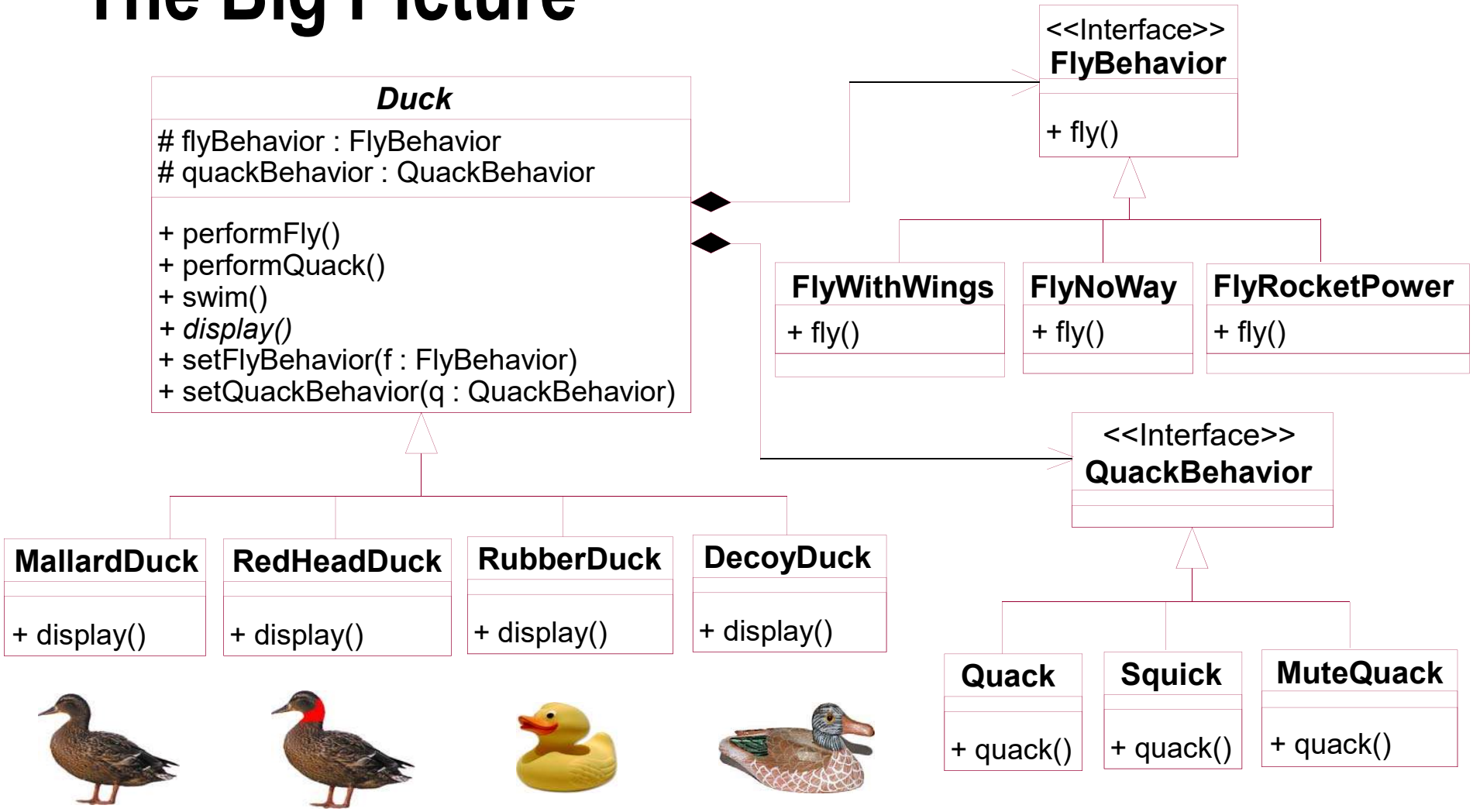


I'm a model duck
I can't fly
I'm flying with a rocket

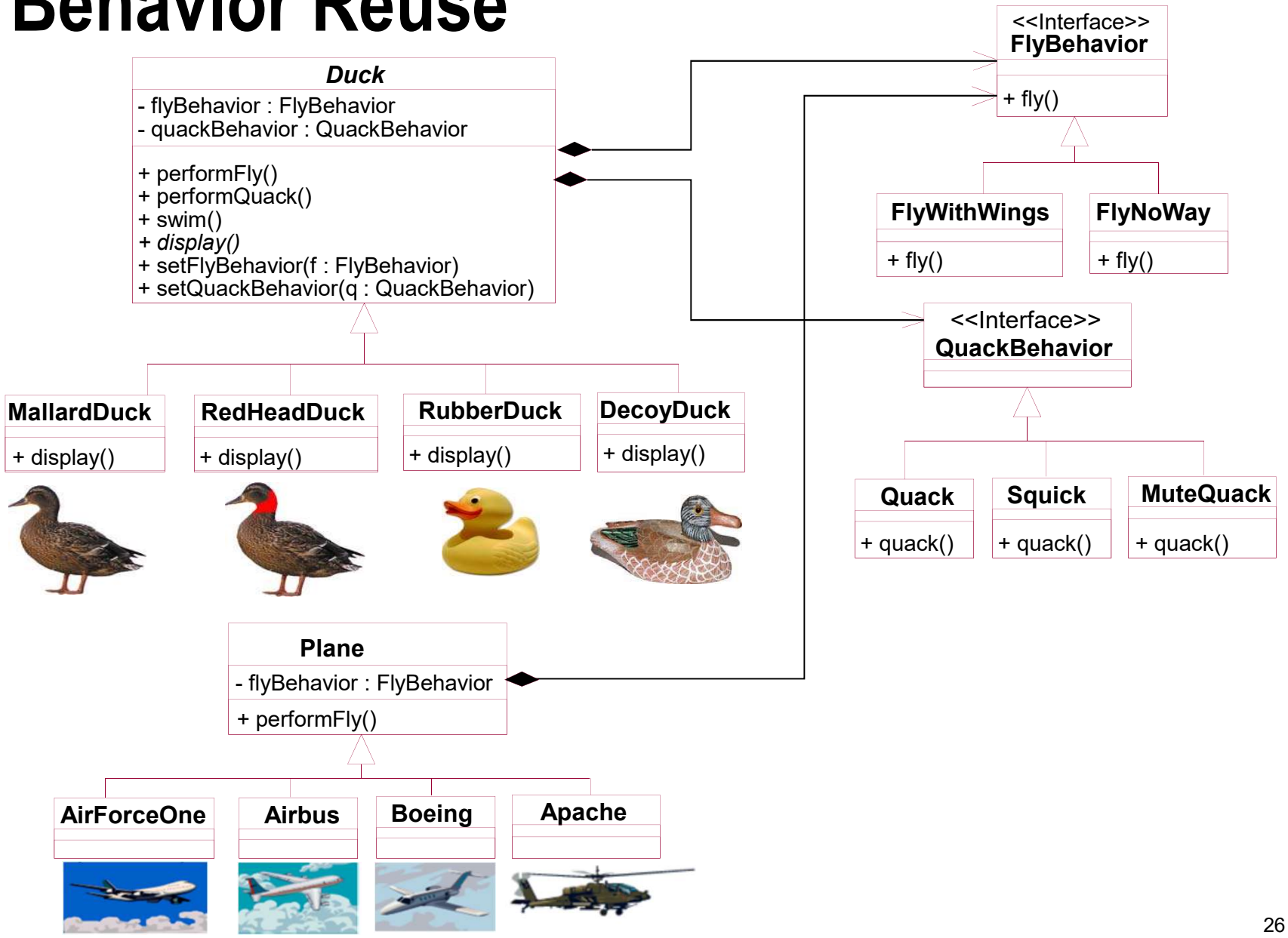
// Make a new FlyBehavior type

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying with a rocket");
    }
}
```

The Big Picture



Behavior Reuse

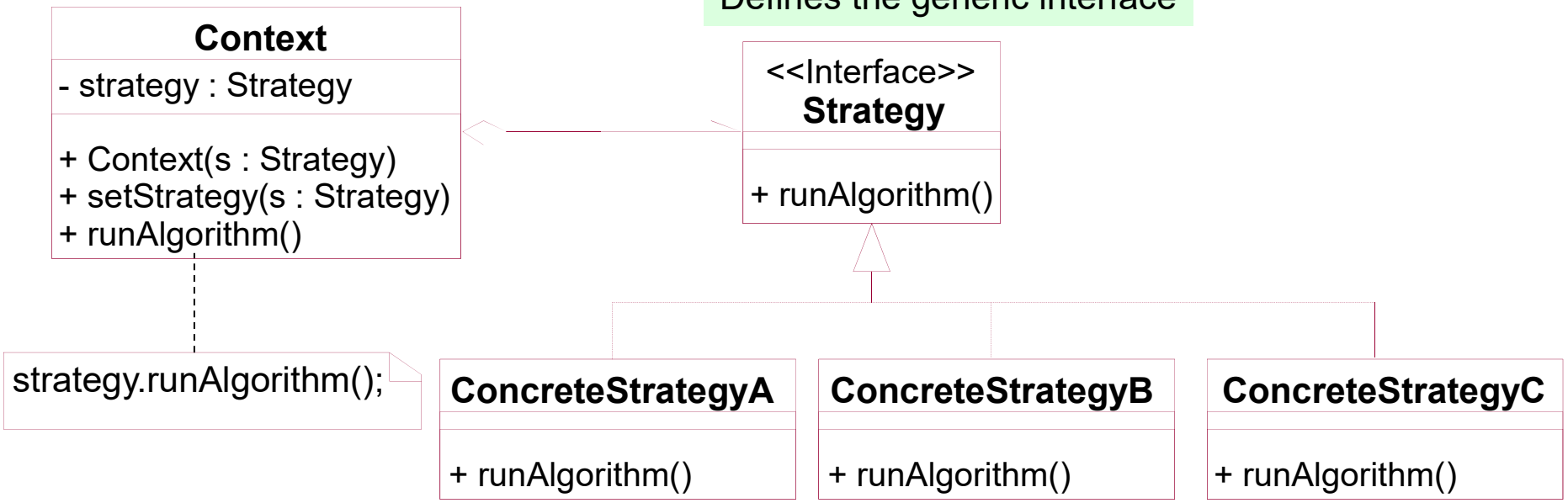


The Strategy Design Pattern

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

Context manages the data structures that a concrete strategy operates on.

Defines the generic interface



ConcreteStrategy classes provide the implementations of the different strategies. These operate on the data structures in the the **Context**, and can be set dynamically.

Summary

- Strategy pattern allows selection of one of several algorithms dynamically.
- Algorithms may be related via an inheritance hierarchy or unrelated [must implement the same interface]
- Strategies don’t hide everything -- client code is typically aware that there are a number of strategies and has some criteria to choose among them -- shifts the algorithm decision to the client.