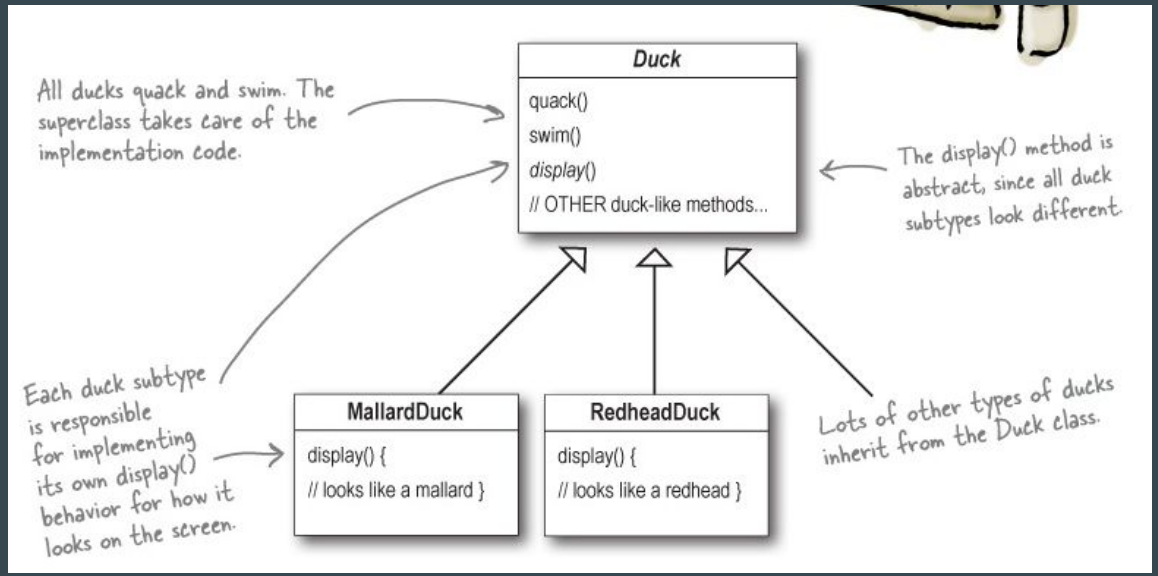
Patterns 1

The Strategy Pattern

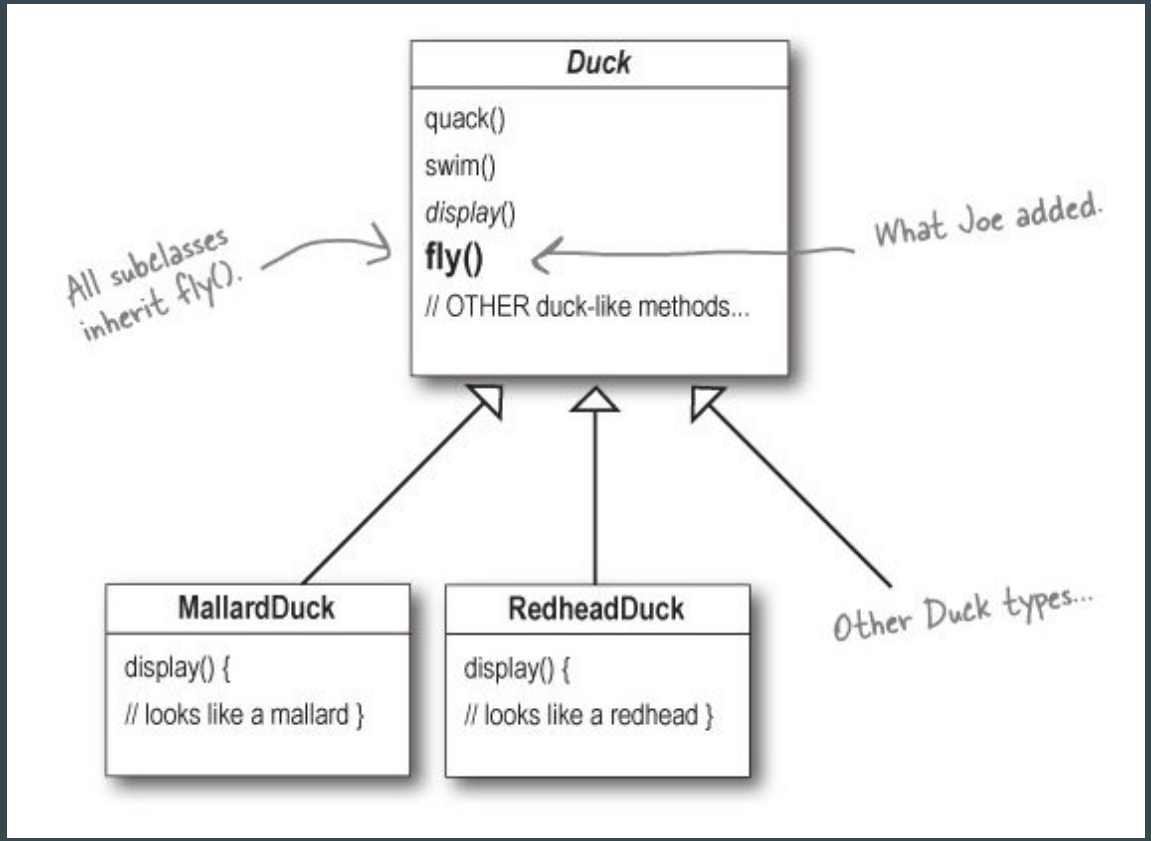
The Program: A duck pond simulation game

- Has a large variety of duck species swimming and making different sounds.
- At first, standard OO techniques were used: a Duck superclass was made from which all the other duck classes will inherit:



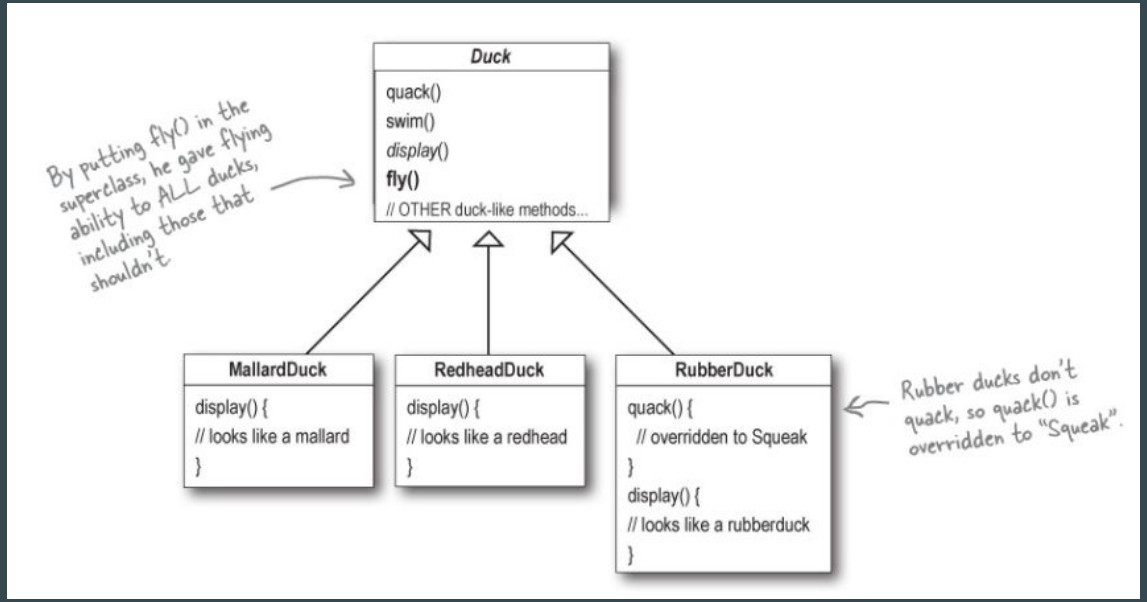
But now the ducks need to fly!

An easy approach would be to simply add and implement a fly() method in the Duck superclass



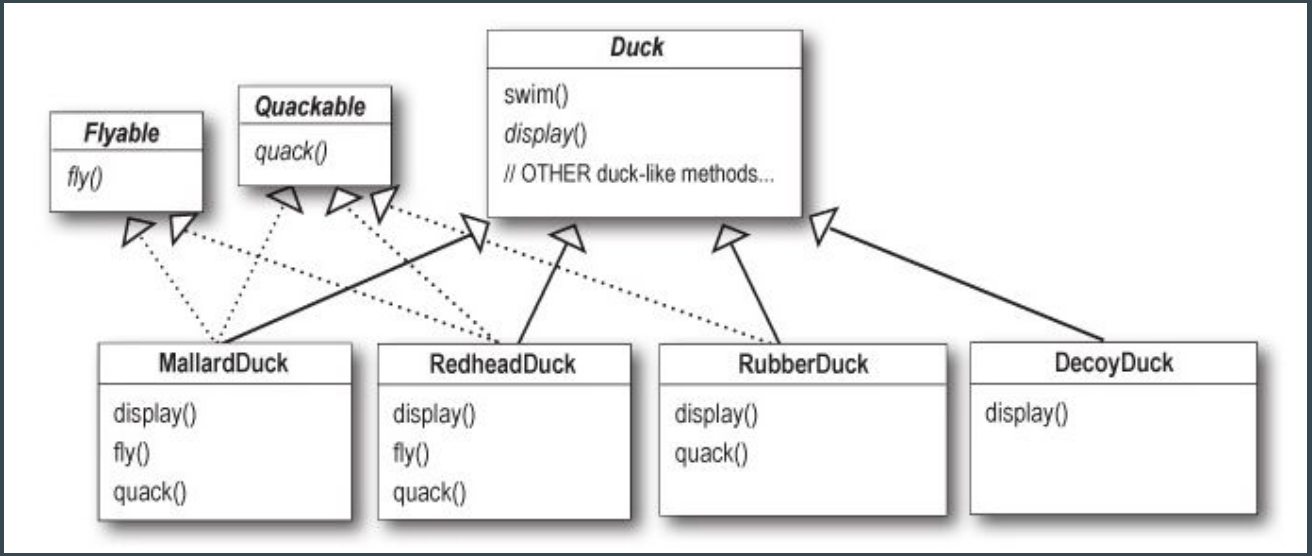
Problem: The rubber duckies are also flying!

- These kinds of ducks do not fly!
- Are we to override the fly method a non-functioning implementation?
- This kind of situation happens when not all method implementations in the superclass are appropriate for all subclasses



- The flying rubber ducky effect is an instance of a non-local side effect
- This problem would also occur with, for example, decoy ducks which neither quack nor fly
- Also, be ready for this fact: Software always changes
- We need a way to reuse code, and also be able to make changes to code quickly

What about an interface?



But we lose the code reuse of fly() and quack() for all those kinds of ducks?

Also, we must be able to change flying/quacking behavior at run time

- The executives now want to update the product every 6 months in unknown ways
- To handle this, we take fly and quack out of the class and put them in interfaces

We need another solution: Let's think

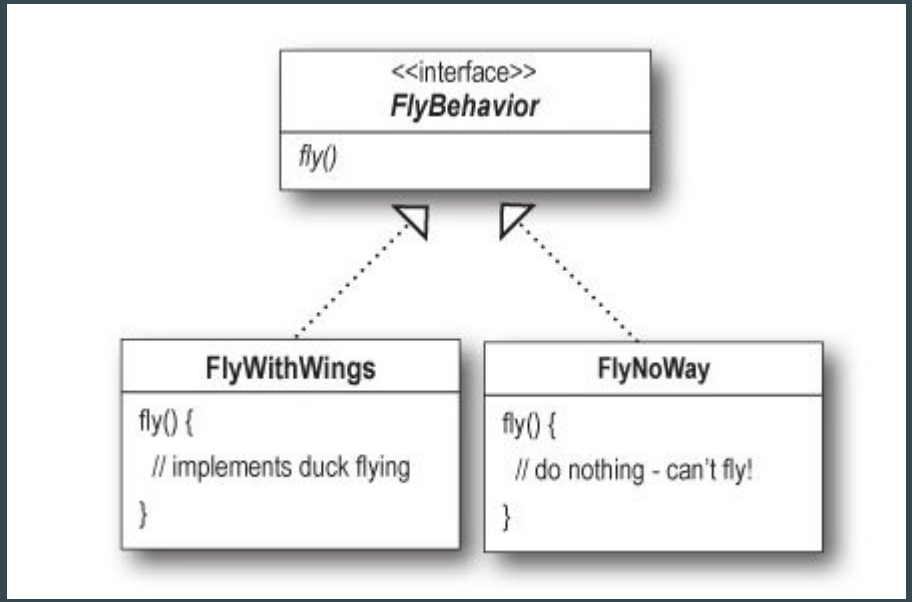
- A design principle: identify the parts of the application that can change and those that will always stay the same
- Take what varies and encapsulate it so that changes in it don't effect the rest of your code: more flexibility and less unintended side effects
- All patterns provide a way to let some part of a system vary independently of all other parts
- We need to do this to our ducks

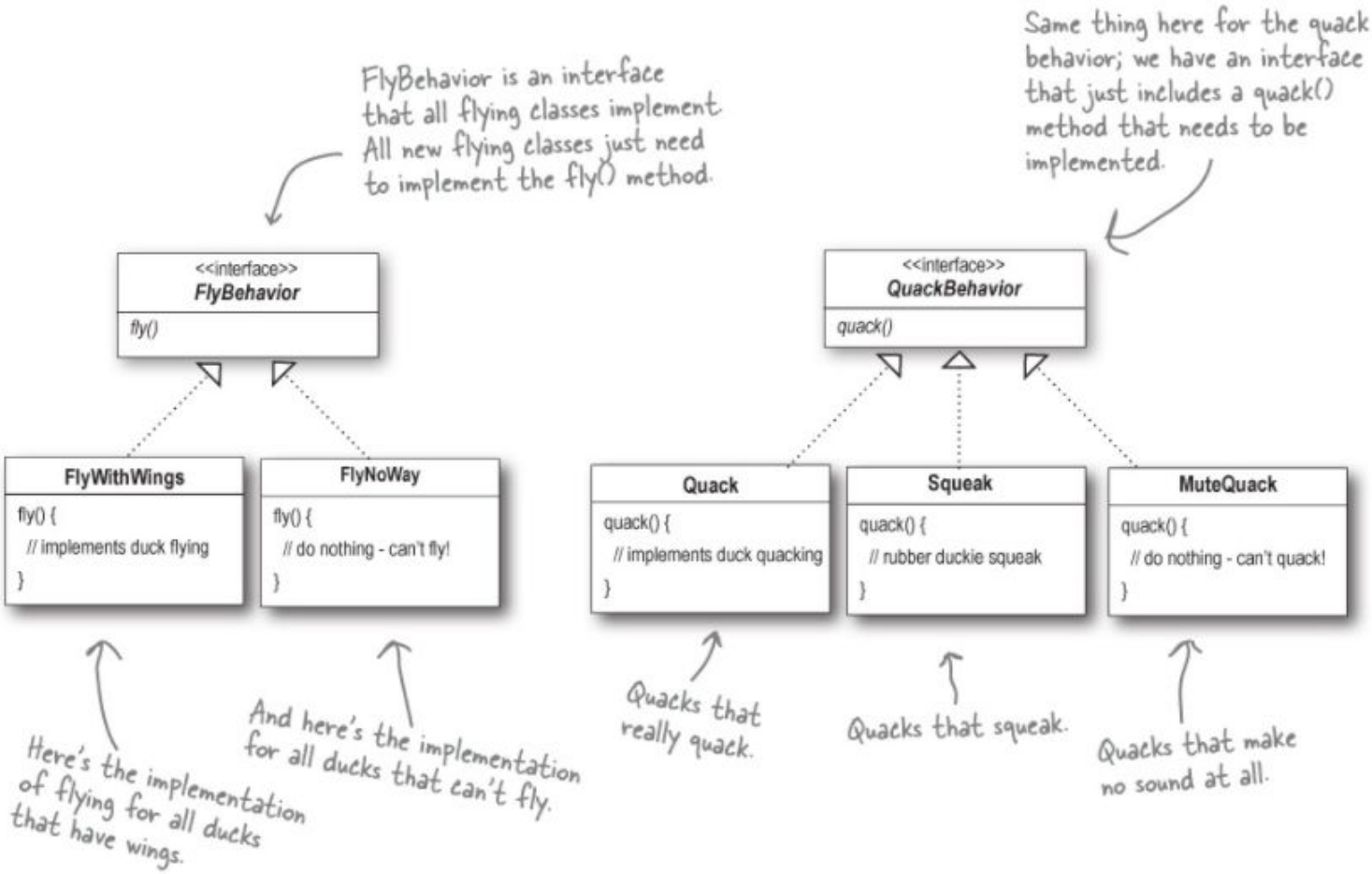
Rethinking the Duck Class

- What are behaviors some ducks have and others don't? Flying and quacking. Leave the rest alone
- Regarding the behaviors that change, we will make separate interfaces for them
  - Program to an interface, rather than a class (or a superclass/abstract class)
- The duck classes then will use *implementations* of these interfaces for their behaviors: they don't need to know the implementations of these behaviors
  - This will allow for code reuse
- Create interfaces FlyBehavior and QuackBehavior
- We'll separate the code for the behaviors from the code of the Duck subclasses; their code won't be locked into the Duck class

That is, we are making separate classes to handle each behavior that changes

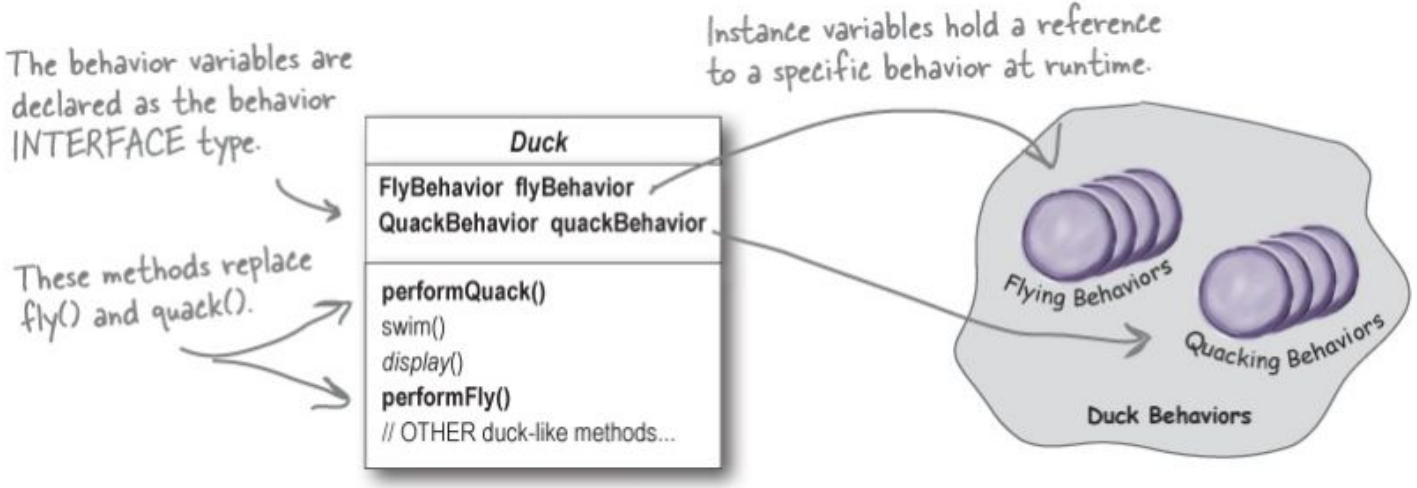
- With this we still get reuse (we can use FlyWithWings in many classes)
- We can change the behavior of many classes at the same time
- We keep flying code separate from Duck code, we don't have to touch them for different flying behaviors




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.
<<interface>>
FlyBehavior
fly()
<<interface>>
QuackBehavior
quack()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
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.

Integrating the new classes into Duck

- Duck will now have two instance variables, one for the flying behavior, and one for the quacking behavior
  - The type of these variables will be the interface or superclass
- Replace the fly() and quack() methods with performFly() and performQuack()
- See the next slide for an illustration



② **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.

Example subclass of 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");
    }
}
```

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.

Overview of the code:

- Create the behavior interfaces
- Implement some classes from the behavior interfaces
- Implement some Duck subclasses, creating the preferred behavior classes in the constructor
- Write a test program

See code demo for whole program