Design Patterns: The Strategy Pattern

...

What are design patterns and why do we care?

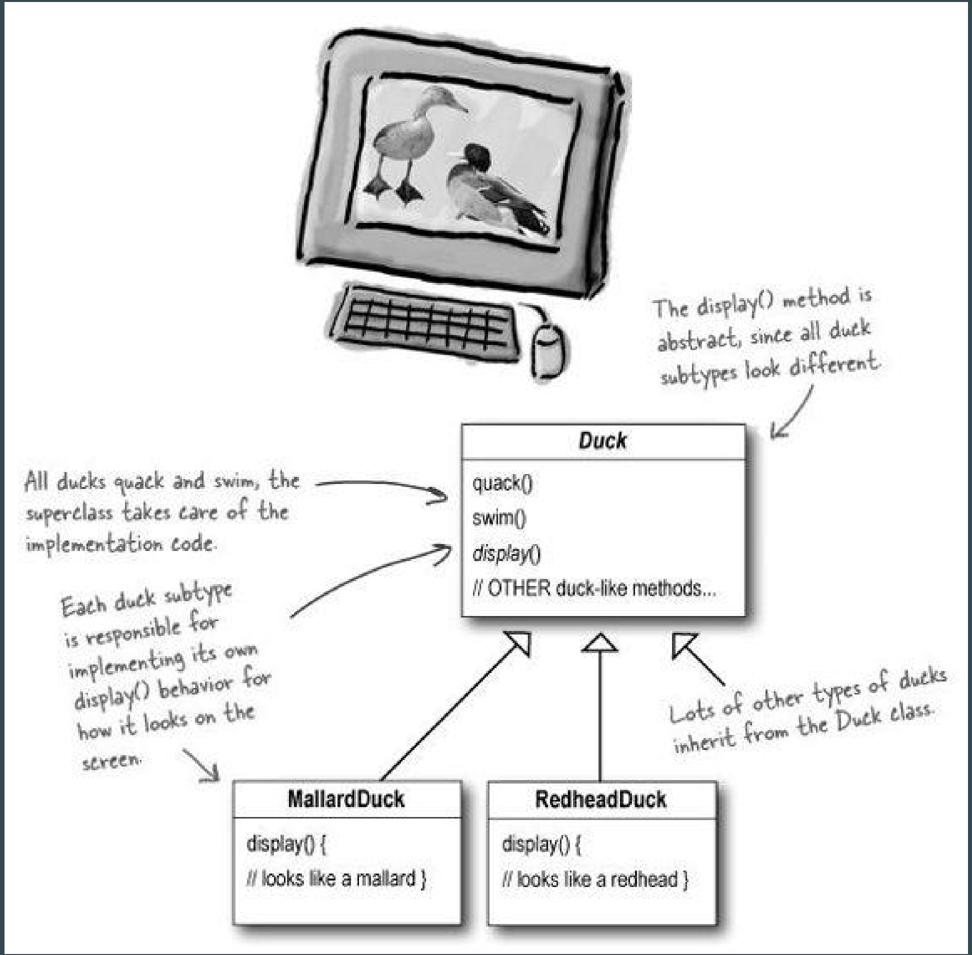
- abstract description of how to structure your code
- usually designed to solve a certain problem
- shared vocabulary
- ideally makes your code
  - easier to understand
  - more flexible
  - more maintainable

The SimUDuck App:

An application to simulate a pond with ducks.

This is the initial design.

Problems arise when we want to implement a new method `fly()`.



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

Duck

quack()

swim()

display()

// OTHER duck-like methods...

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

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.

MallardDuck

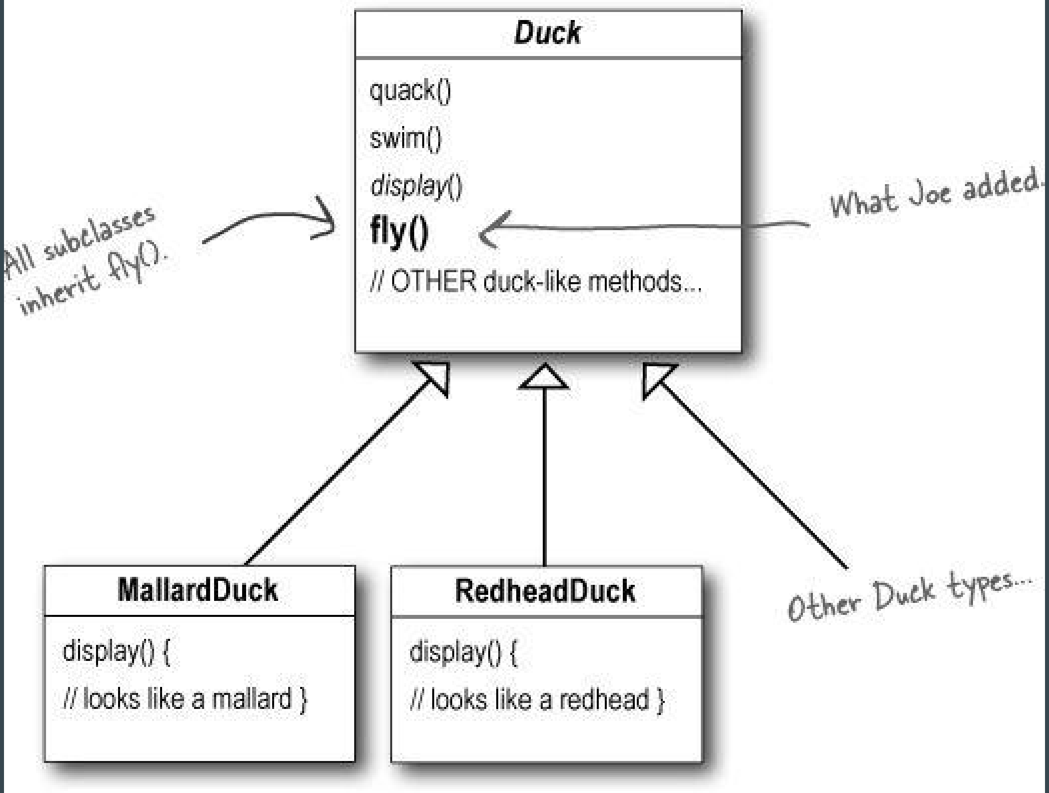
display() {
// looks like a mallard }

RedheadDuck

display() {
// looks like a redhead }

Extending the Duck Class

Putting `fly()` in the Duck class is not flexible enough.



Extending the Duck Class

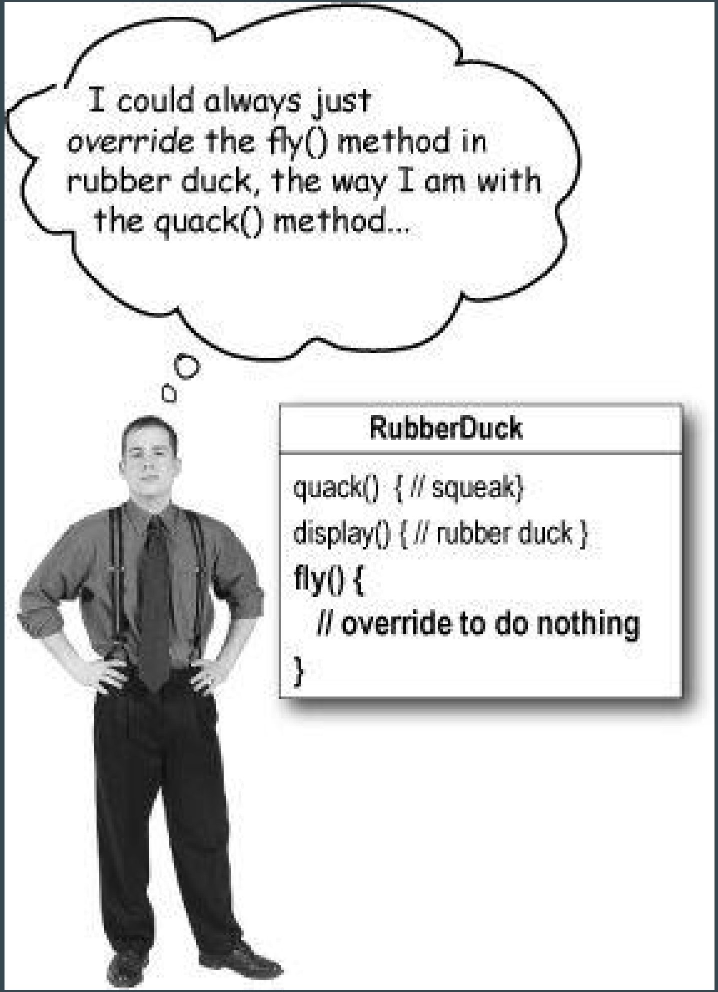
Putting `fly()` in the Duck class is not flexible enough.

Extending the Duck Class

Overriding the behavior for each duck type is difficult to maintain.

Code duplication.

Overriding is hard coded for each new class.


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
}
```

Let the subclasses implement the fly behavior

Even more code duplication.

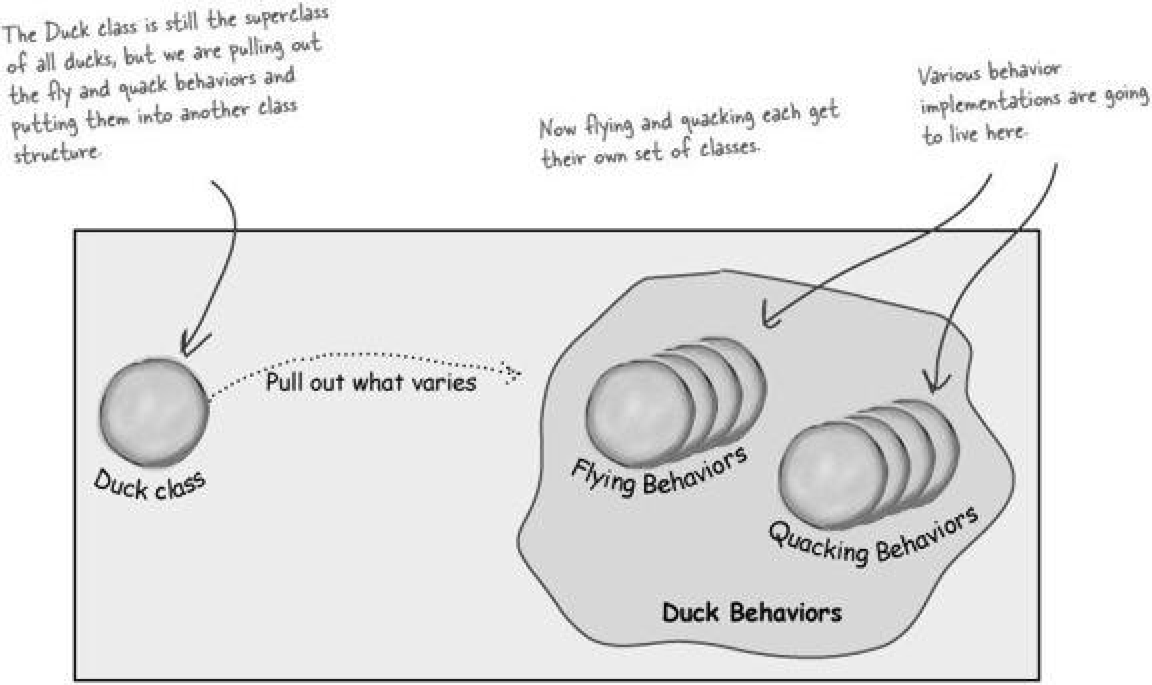
The challenge is to have different forms of ducks without too much code duplication.

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?!*

What to do?

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

Separating what changes from what stays the same

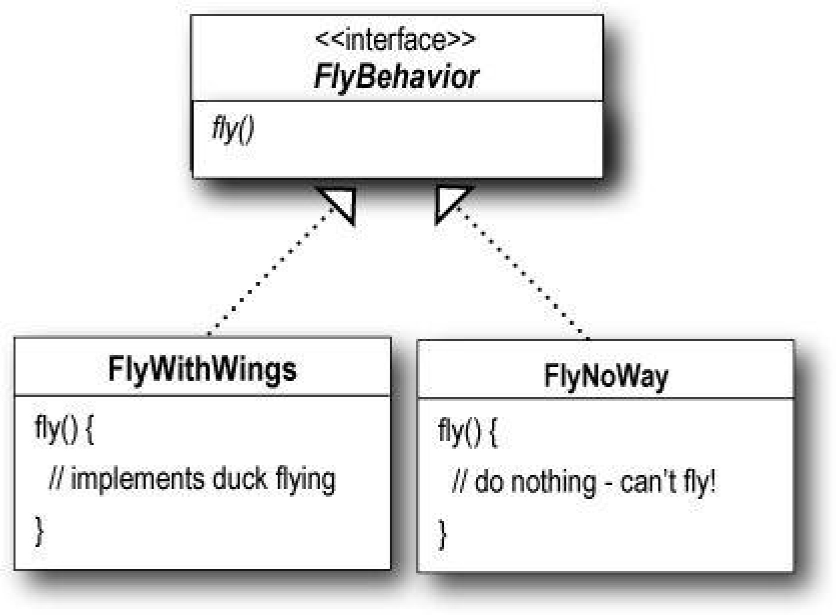

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.
Now flying and quacking each get their own set of classes.
Various behavior implementations are going to live here.
Pull out what varies
Duck class
Flying Behaviors
Quacking Behaviors
Duck Behaviors

How do we design these **sets** of classes?

Design Principle:
Program to an interface, not an implementation.

That means there will be a supertype that establishes the interface and subclasses implement the actual behavior.

Interface and implementation

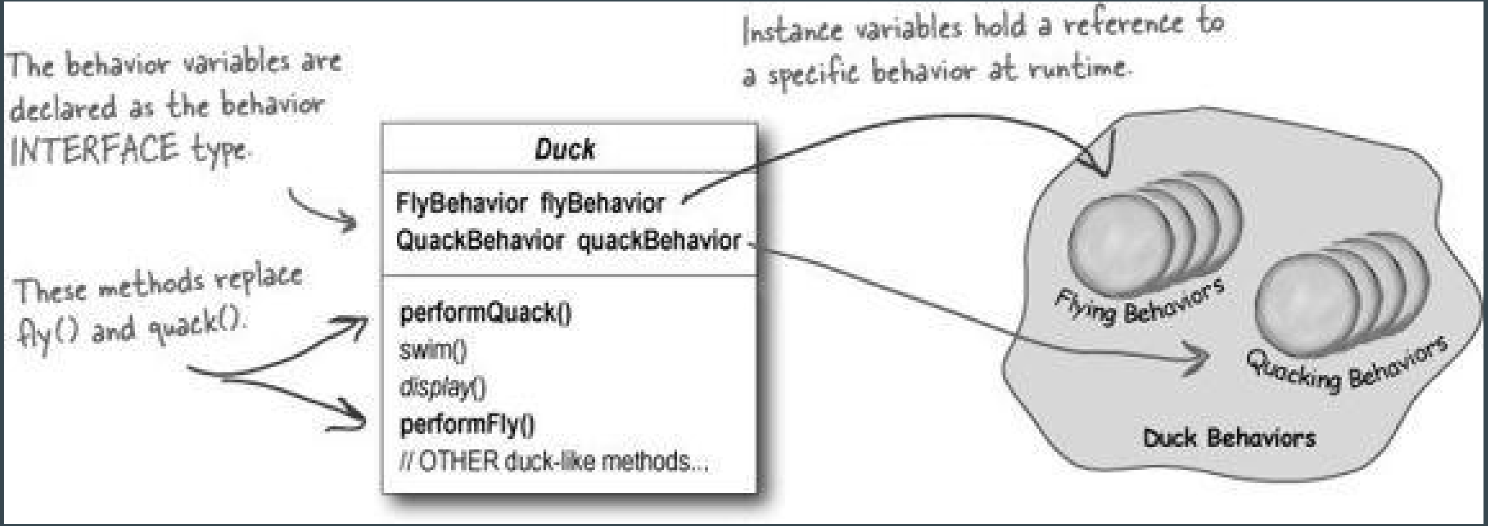


```
fly() {
  // implements duck flying
}
```

```
fly() {
  // do nothing - can't fly!
}
```

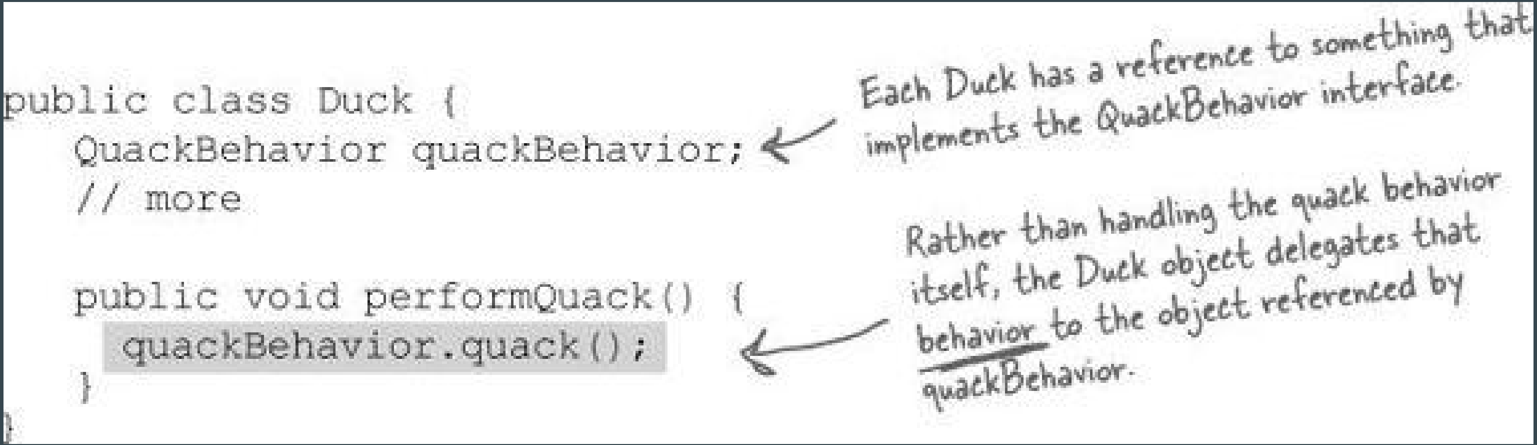
Integrating the Duck Behavior

1. First we'll add two instance variables


The behavior variables are declared as the behavior INTERFACE type.
These methods replace fly() and quack().
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
performQuack()
swim()
display()
performFly()
// OTHER duck-like methods...
Instance variables hold a reference to a specific behavior at runtime.
Flying Behaviors
Quacking Behaviors
Duck Behaviors

Integrating the Duck Behavior

2. Now we implement `performQuack()`:



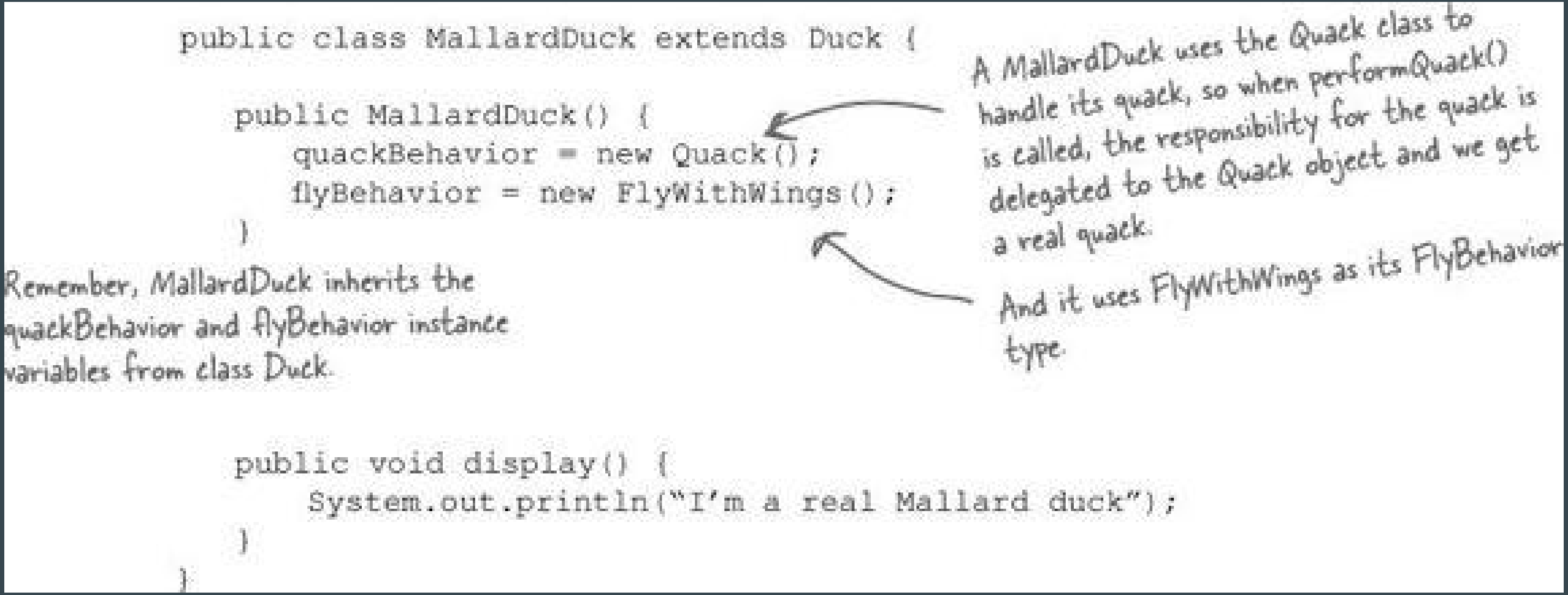
```
public class Duck {
   QuackBehavior quackBehavior;
   // more

   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.

The constructors of the duck subclasses implement a specific 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”);
    }
}
```

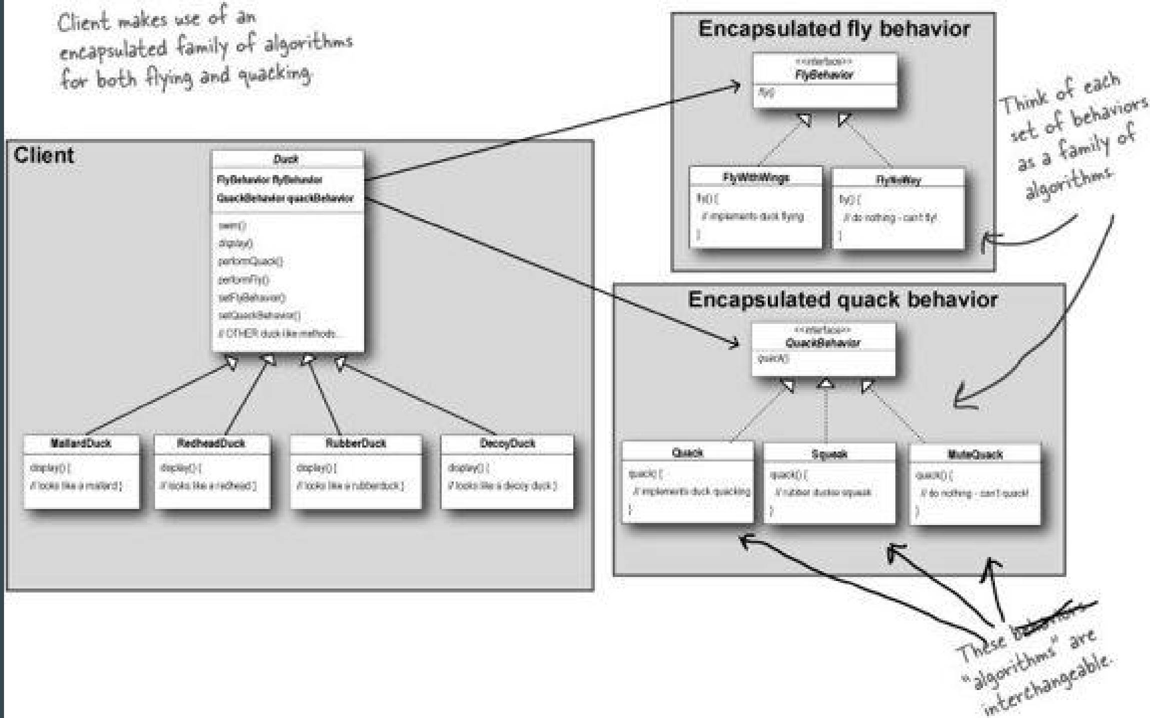
A MallardDuck uses the Quack class to handle its quack, so when performQuack() is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

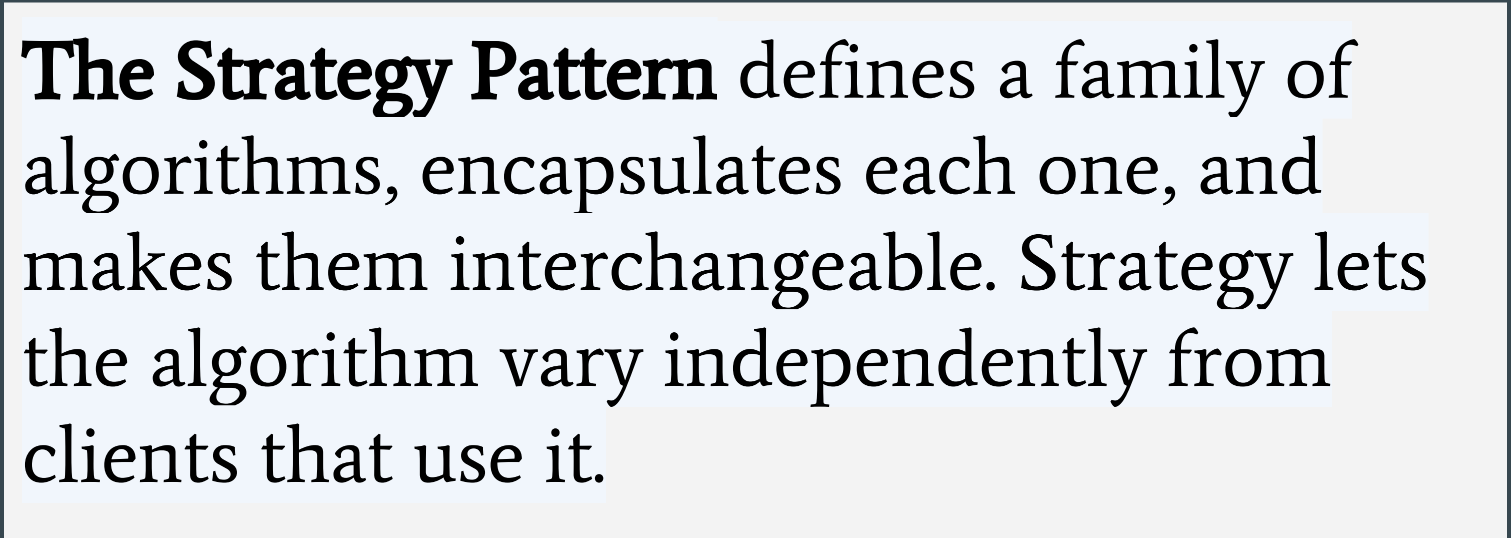
And it uses FlyWithWings as its FlyBehavior type.

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

Big Picture

Design Principle:
Favor composition over inheritance

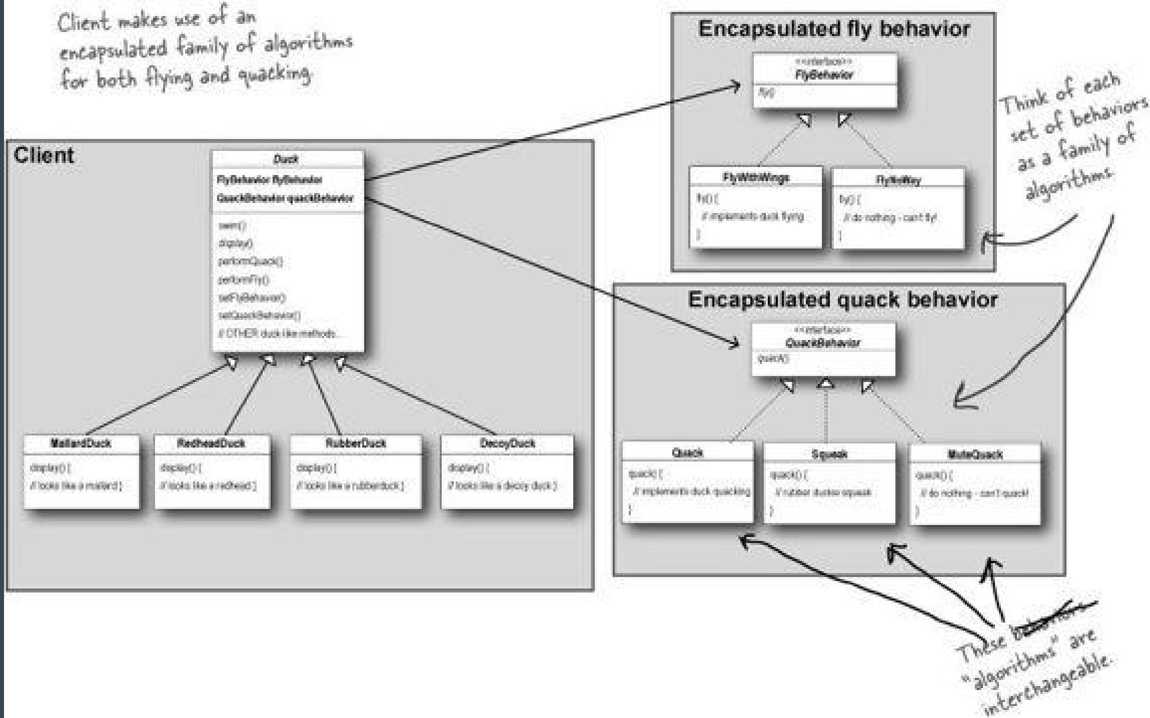

Client makes use of an encapsulated family of algorithms for both flying and quacking
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
These ~~behaviors~~ "algorithms" are interchangeable.



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.

Exercises

1. Get the code from /design-patterns/christian/usingstrategy/
2. Encapsulate the quack behavior.
3. Add a “FlyWithRockets” behavior.
4. Give a rubber duck rockets at runtime.


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
These ~~behaviors~~ “algorithms” are interchangeable.