Introduction

Thursday, May 18, 2023 7:13 PM

(REF BOOK credit : Head First Design Pattern)

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.

Okay ,chill!, you didn’t understand what I'm saying, I also don’t understand this defination written in the book either.

But no worries, we will get go with some easy example to get into the Strategy pattern

Let's Get into the pattern-iformally(:P)

Thursday, May 18, 2023 8:31 PM

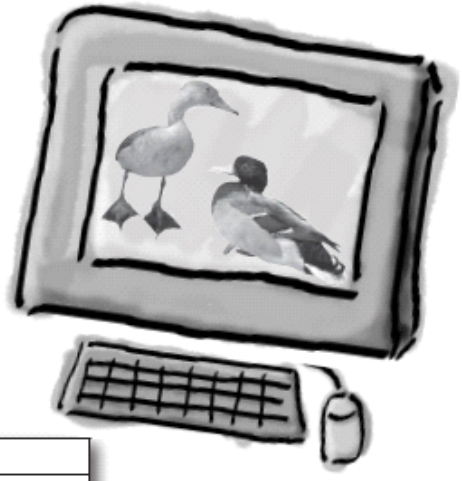
Let's Make a Project first.

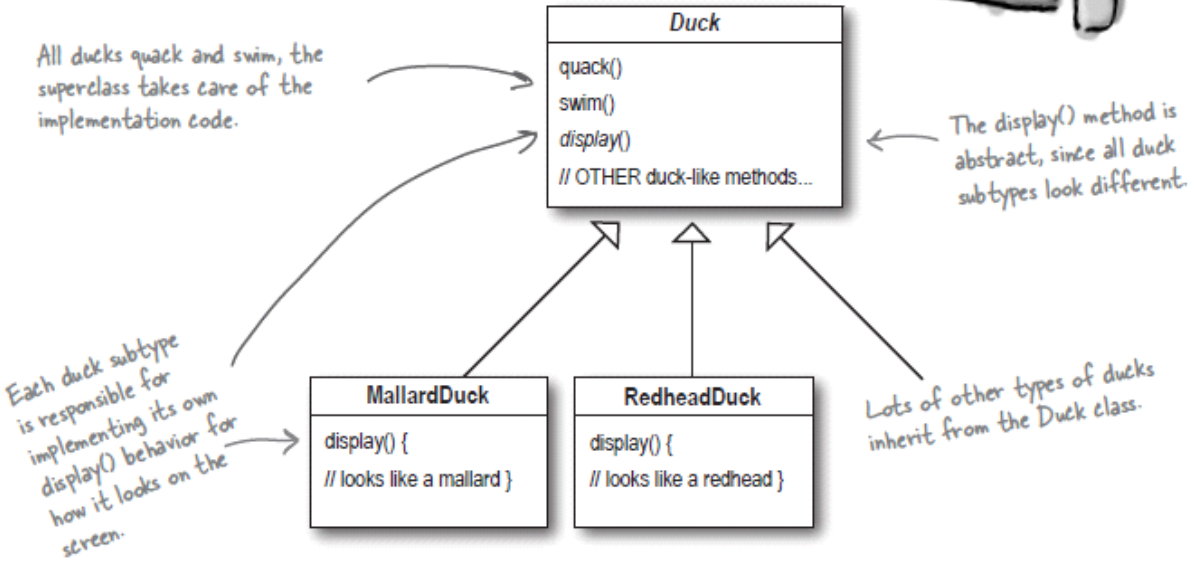
-> We will make a Simulator of Duck, it will show some duck implementation(quacking,flying etc)

Okay, Let’s do this first.

It started with a simple SimUDuck app

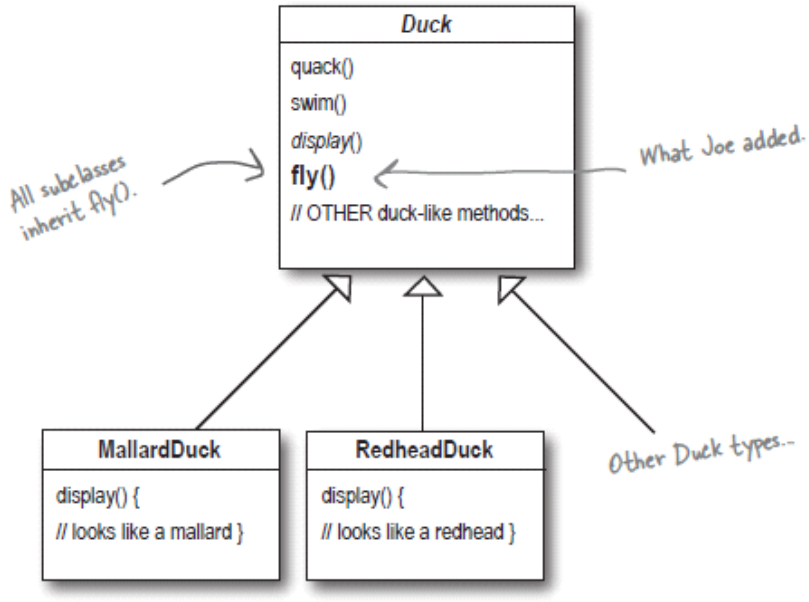
Joe works for a company that makes a highly successful duck pond simulation game, *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit.





NOW , One day , The manager Ordered Joe , to add a new feature fly()

Joe thought like this,just add fly() method in the duck super class.So all the subclass will get it.


Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
All subclasses inherit fly().
What Joe added.
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
Other Duck types...

But something went horribly wrong...

What happened?

Joe failed to notice that not *all* subclasses of Duck should *fly*. When Joe added new behavior to the Duck superclass, he was also adding behavior that was *not* appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.

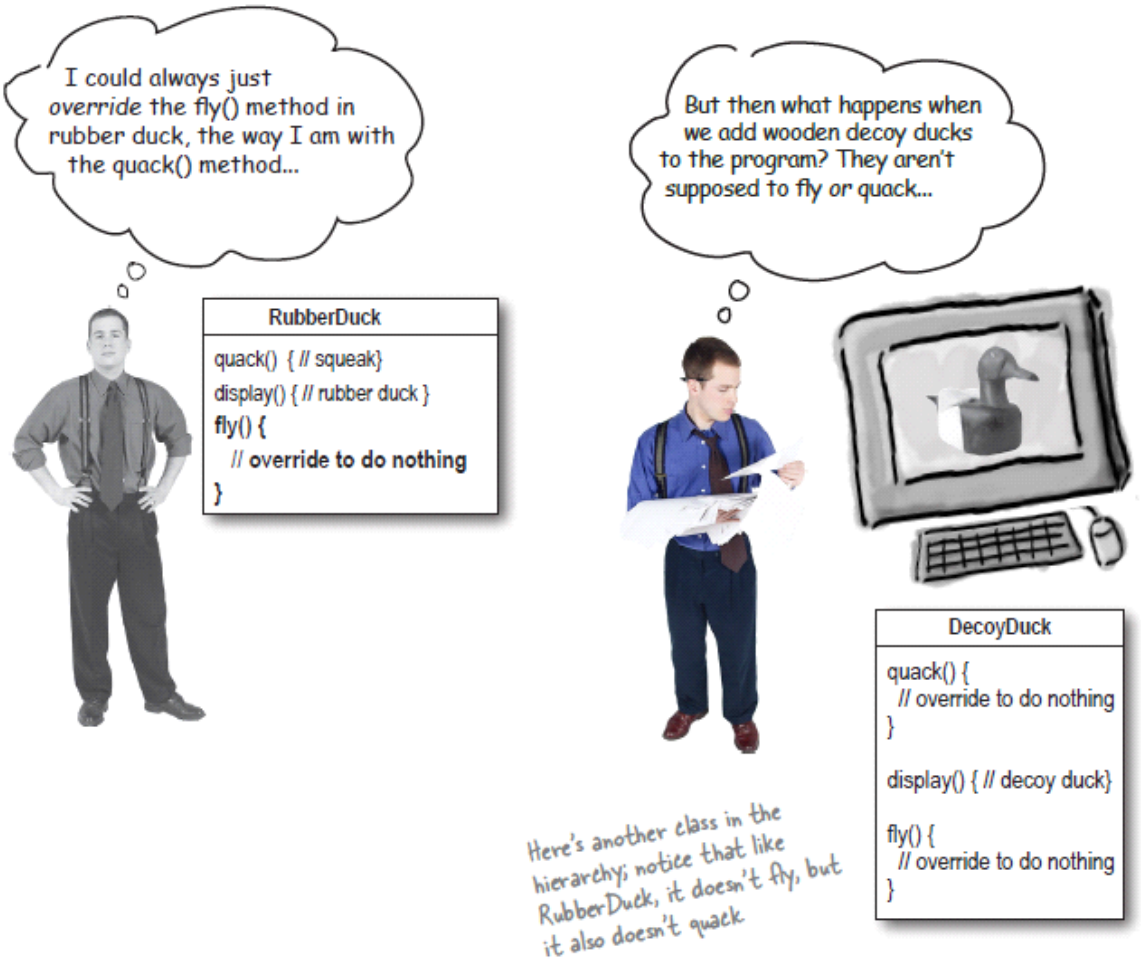
A localized update to the code caused a non-local side effect (flying rubber ducks)!

What he thought was a great use of inheritance for the purpose of reuse hasn't turned out so well when it comes to maintenance.

So, the Rubber Duck cannot fly or quack

Then Joe came up with an idea-

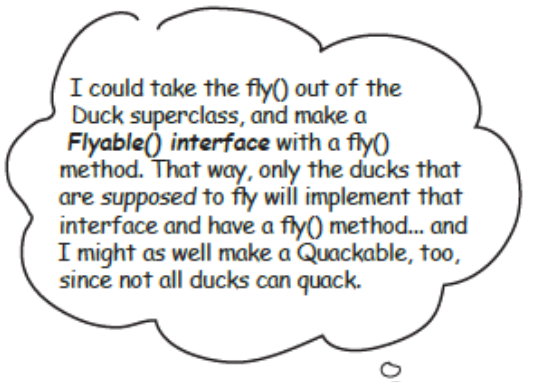
Joe thinks about inheritance...



How about an interface?

Joe realized that inheritance probably wasn't the answer, because he just got a memo that says that the executives now want to update the product every six months (in ways they haven't yet decided on). Joe knows the spec will keep changing and he'll be forced to look at and possibly override fly() and quack() for every new Duck subclass that's ever added to the program... *forever.*

So, he needs a cleaner way to have only *some* (but not *all*) of the duck types fly or quack.



What do YOU think about this design?

But it will also create problem, we cannot implement every new type of duck ,because there will no code reuse
So inheritance is not the right solution always

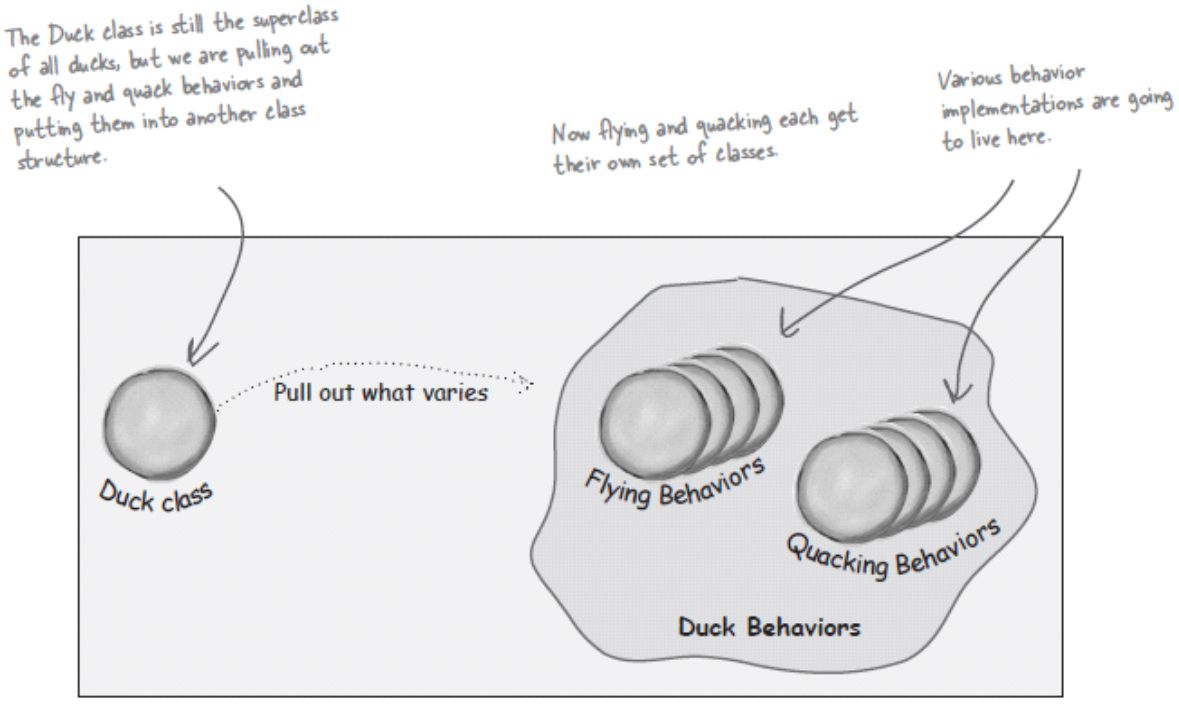
Design Principle

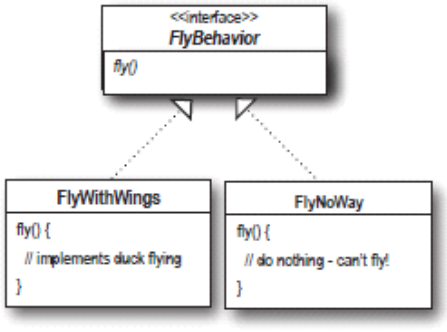
Identify the aspects of your application that vary and separate them from what stays the same.

The first of many design principles. We'll spend more time on these throughout the book.

We know that fly() and quack() are the parts of the Duck class that vary across ducks.

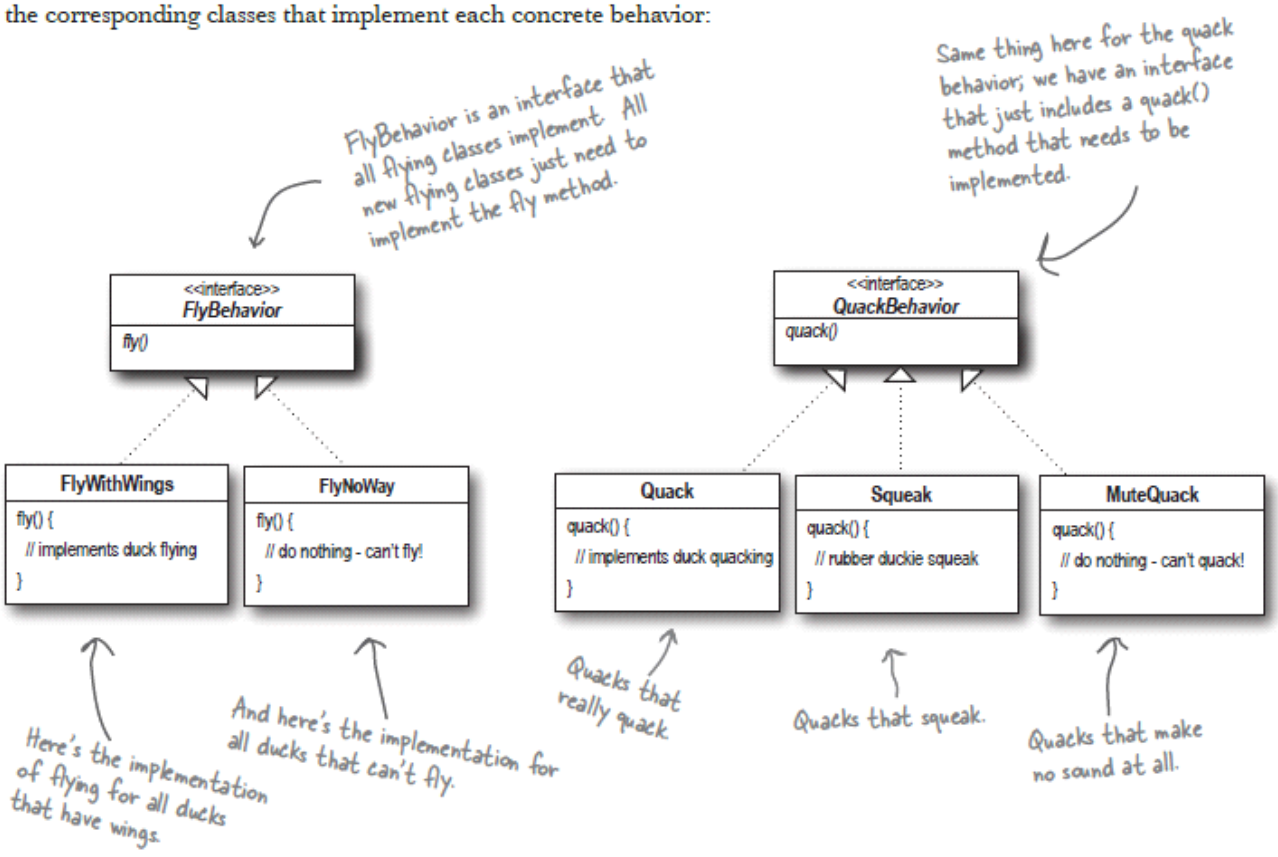
To separate these behaviors from the Duck class, we'll pull both methods *out* of the Duck class and create a new set of classes to represent each behavior.





Implementing the Duck Behaviors

Here we have the two interfaces, FlyBehavior and QuackBehavior along with the corresponding classes that implement each concrete behavior:



With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes!

And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that *use* flying behaviors.

So we get the benefit of REUSE without all the baggage that comes along with inheritance.

Integrating the Duck Behavior

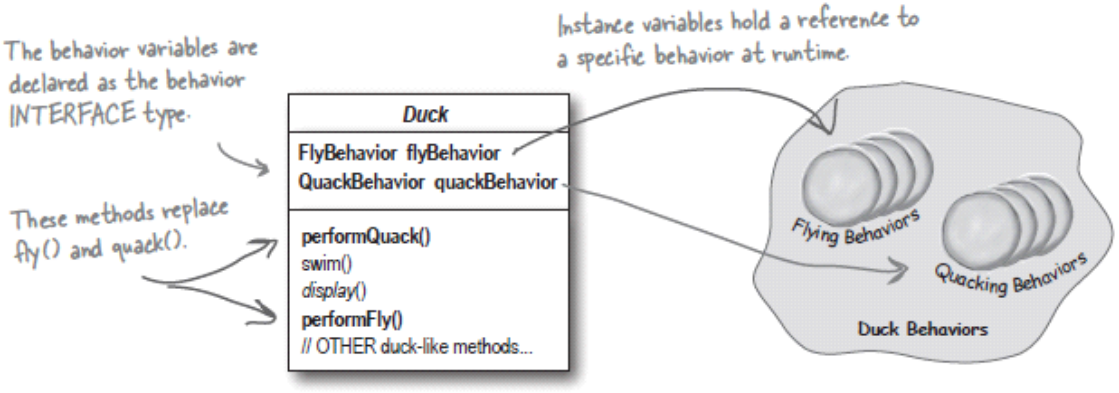
The key is that a Duck will now *delegate* its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass).

Here's how:

1. **First we'll add two instance variables** to the Duck class called *flyBehavior* and *quackBehavior*, that are declared as the interface type (not a concrete class implementation type). Each duck object will set these variables polymorphically to reference the *specific* behavior type it would like at runtime (FlyWithWings, Squeak, etc.).

   We'll also remove the fly() and quack() methods from the Duck class (and any subclasses) because we've moved this behavior out into the FlyBehavior and QuackBehavior classes.

   We'll replace fly() and quack() in the Duck class with two similar methods, called performFly() and performQuack(); you'll see how they work next.



3. **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.

Pretty simple, huh? To perform the quack, a Duck just allows the object that is referenced by quackBehavior to quack for it.
In this part of the code we don't care what kind of object it is, ***all we care about is that it knows how to quack()!***

❸ Okay, time to worry about **how the flyBehavior and quackBehavior instance variables are set**. Let's take a look at the MallardDuck class:

```
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.

So MallardDuck's quack is a real live duck **quack**, not a **squeak** and not a **mute quack**. So what happens here? When a MallardDuck is instantiated, its constructor initializes the MallardDuck's inherited quackBehavior instance variable to a new instance of type Quack (a QuackBehavior concrete implementation class).

And the same is true for the duck's flying behavior – the MallardDuck's constructor initializes the flyBehavior instance variable with an instance of type FlyWithWings (a FlyBehavior concrete implementation class).

Let's Code

Friday, May 19, 2023 10:43 AM

Testing the Duck code

1 Type and compile the Duck class below (Duck.java), and the MallardDuck class from two pages back (MallardDuck.java).

```
public abstract class Duck {

    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;
    public Duck() {
    }

    public abstract void display();

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

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

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

Declare two reference variables for the behavior interface types. All duck subclasses (in the same package) inherit these.

Delegate to the behavior class.

2 Type and compile the FlyBehavior interface (FlyBehavior.java) and the two behavior implementation classes (FlyWithWings.java and FlyNoWay.java).

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

The interface that all flying behavior classes implement.

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

Flying behavior implementation for ducks that DO fly...

```
public class FlyNoWay implements FlyBehavior {
    public void fly() {
        System.out.println("I can't fly");
    }
}
```

Flying behavior implementation for ducks that do NOT fly (like rubber ducks and decoy ducks).

3 Type and compile the QuackBehavior interface (QuackBehavior.java) and the three behavior implementation classes (Quack.java, MuteQuack.java, and Squeak.java).

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

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

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
       System.out.println("<< Silence >>");
   }
}
```

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

4 Type and compile the test class (MiniDuckSimulator.java).

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

This calls the MallardDuck's inherited performQuack() method, which then delegates to the object's QuackBehavior (i.e. calls quack() on the duck's inherited quackBehavior reference).

Then we do the same thing with MallardDuck's inherited performFly() method.

5 Run the code!

```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```

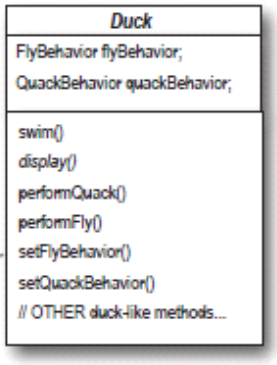
Setting behavior dynamically

What a shame to have all this dynamic talent built into our ducks and not be using it! Imagine you want to set the duck's behavior type through a setter method on the duck subclass, rather than by instantiating it in the duck's constructor.

1 Add two new methods to the Duck class:

```
public void setFlyBehavior(FlyBehavior fb) {
    flyBehavior = fb;
}

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



We can call these methods anytime we want to change the behavior of a duck on the fly.

editor note: gratuitous pun - fix

2 Make a new Duck type (ModelDuck.java).

```
public class ModelDuck extends Duck {
    public ModelDuck() {
       flyBehavior = new FlyNoWay();
       quackBehavior = new Quack();
    }

    public void display() {
       System.out.println("I'm a model duck");
    }
}
```

Our model duck begins life grounded... without a way to fly.

3 Make a new FlyBehavior type (FlyRocketPowered.java).

That's okay, we're creating a rocket powered flying behavior.

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

4 Change the test class (MiniDuckSimulator.java), add the ModelDuck, and make the ModelDuck rocket-enabled.

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

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

before

The first call to performFly() delegates to the flyBehavior object set in the ModelDuck's constructor, which is a FlyNoWay instance.

This invokes the model's inherited behavior setter method, and...voila! The model suddenly has rocket-powered flying capability!

after

If it worked, the model duck dynamically changed its flying behavior! You can't do THAT if the implementation lives inside the duck class.

5 Run it!

```
File Edit Window Help Yabadabadoo
%java MiniDuckSimulator
Quack
I'm flying!!
I can't fly
I'm flying with a rocket!
```

To change a duck's behavior at runtime, just call the duck's setter method for that behavior.

Analysis

Friday, May 19, 2023 10:45 AM