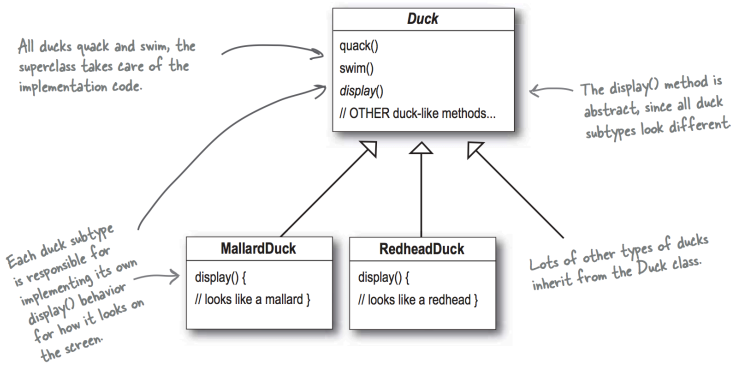
Strategy
Example: SimUDuck

From *Head First Design Patterns*

CS 534 | Ozyegin University

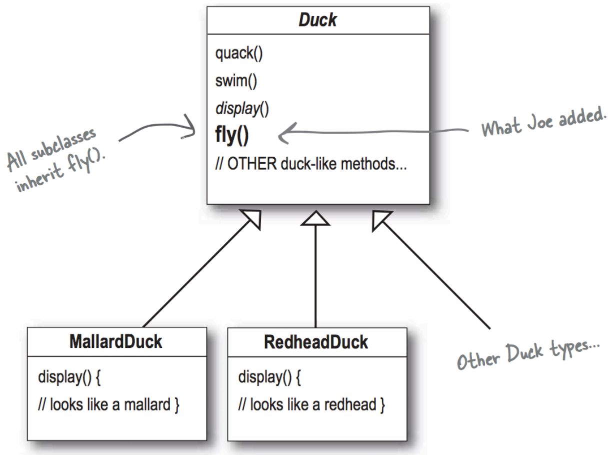
Version 1

- Requirements
  - Simulate ducks
  - All ducks swim
  - All ducks quack
  - All ducks have an appearance, but Redhead, Mallard, YeşilbaşlıGövel etc. ducks look different from each other.


All ducks quack and swim, the superclass takes care of the implementation code.
Duck
quack()
swim()
display()
// OTHER duck-like methods...
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.
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
Lots of other types of ducks inherit from the Duck class.

Version 2

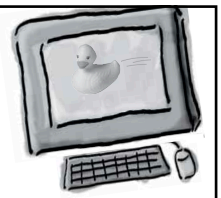
- New requirements
  - All ducks fly

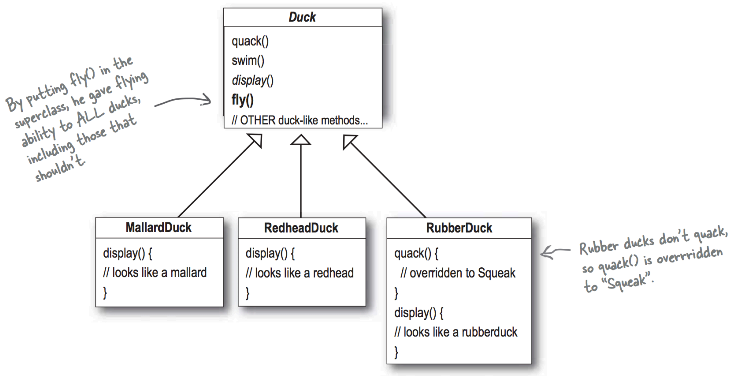


3

Version 3

- New requirements
  - Simulate rubber ducks
  - Problem: rubber ducks can't fly

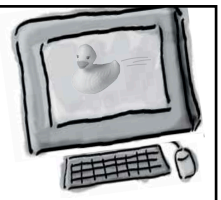


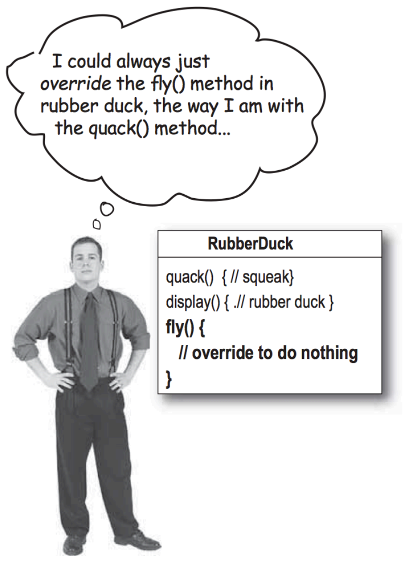

Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't.
MallardDuck
display() {
// looks like a mallard
}
RedheadDuck
display() {
// looks like a redhead
}
RubberDuck
quack() {
// overridden to Squeak
}
display() {
// looks like a rubberduck
}
Rubber ducks don't quack, so quack() is overridden to "Squeak".

4

Version 4

- Fix the flying rubber duck problem

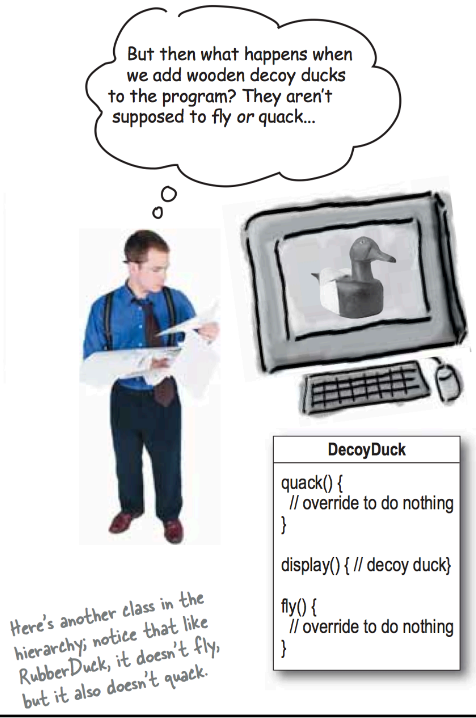



I could always just *override* the fly() method in rubber duck, the way I am with the quack() method...

```
RubberDuck
quack()  { // squeak}
display() { .// rubber duck }
fly() {
   // override to do nothing
}
```

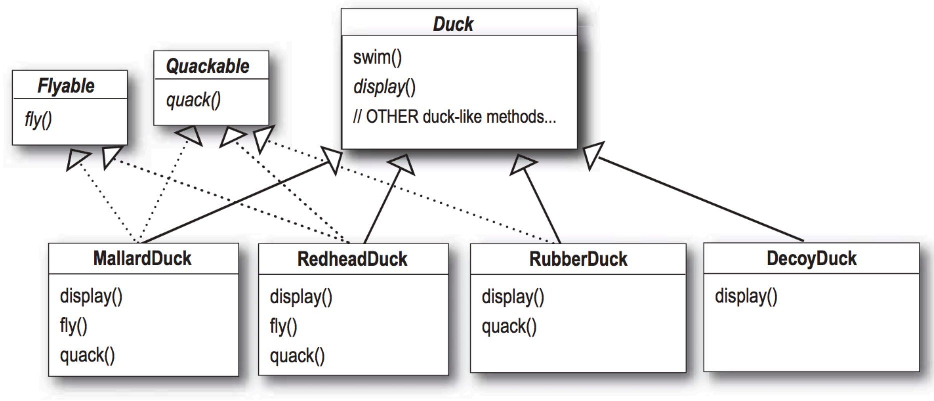
5

- The “fix” is not maintainable


But then what happens when we add wooden decoy ducks to the program? They aren’t supposed to fly *or* quack...
Here’s another class in the hierarchy; notice that like RubberDuck, it doesn’t fly, but it also doesn’t quack.

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

display() { // decoy duck}

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

Version 5



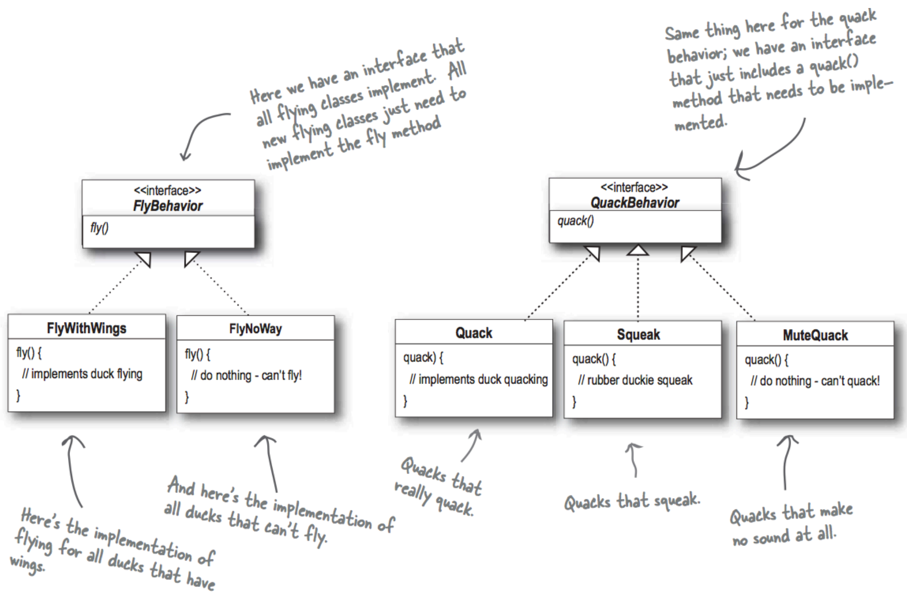
7

Version 6

- New requirements
  - Ducks can be equipped with a rocket to make them fly
- Solution: Separate the flying behaviour to make it configurable at runtime.

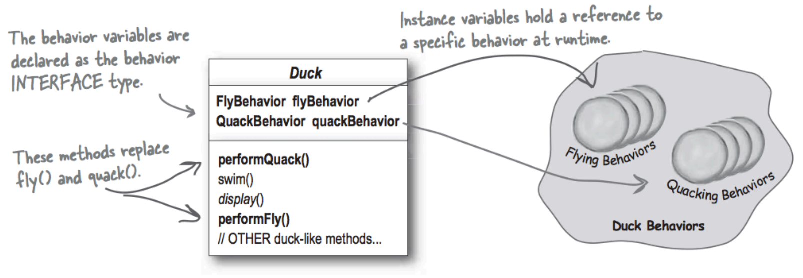
8

Version 6


Here we have 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 of all ducks that can't fly.
Quacks that really quack.
Quacks that squeak.
Quacks that make no sound at all.

9

Version 6


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

❸ **Now we implement performQuack():**

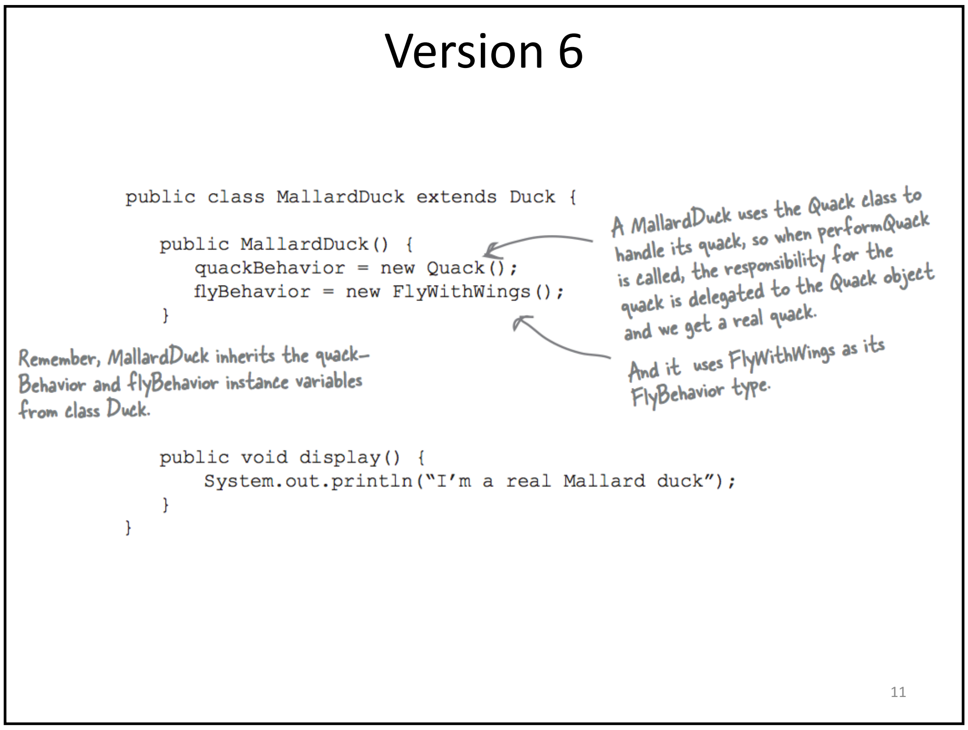
```
public class Duck {
   QuackBehavior quackBehavior;
   // more

   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()!***

10



Version 6

```
public class MallardDuck extends Duck {

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

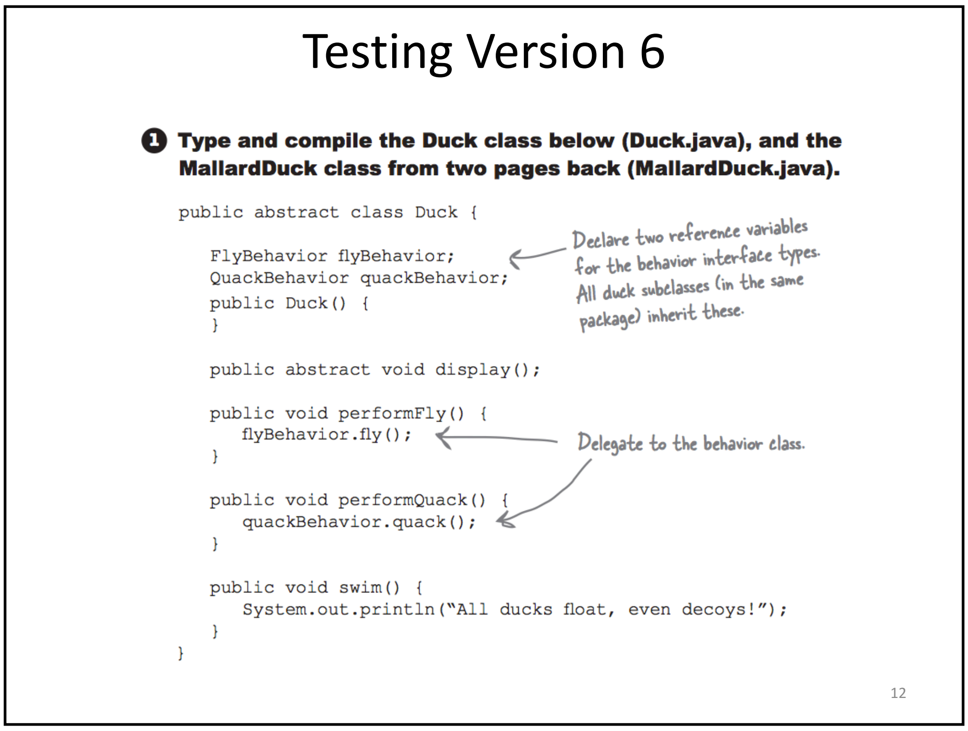
A MallardDuck uses the Quack class to handle its quack, so when performQuack is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

Remember, MallardDuck inherits the quack-Behavior and flyBehavior instance variables from class Duck.

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

11



Testing Version 6

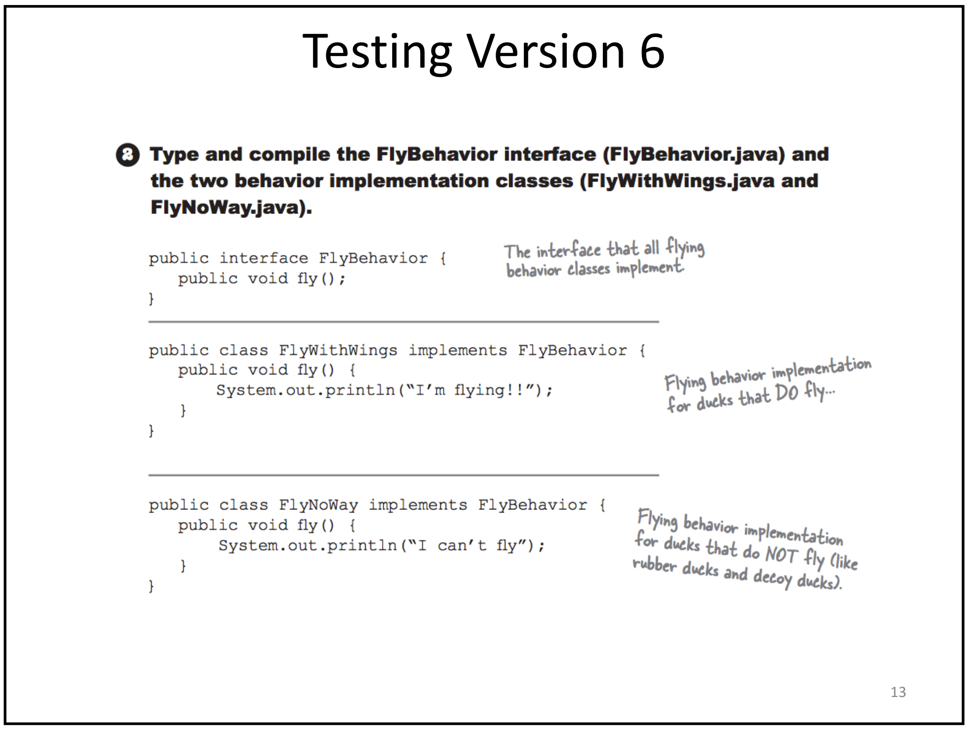
❶ Type and compile the Duck class below (Duck.java), and the MallardDuck class from two pages back (MallardDuck.java).

```
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.

12



Testing Version 6

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.

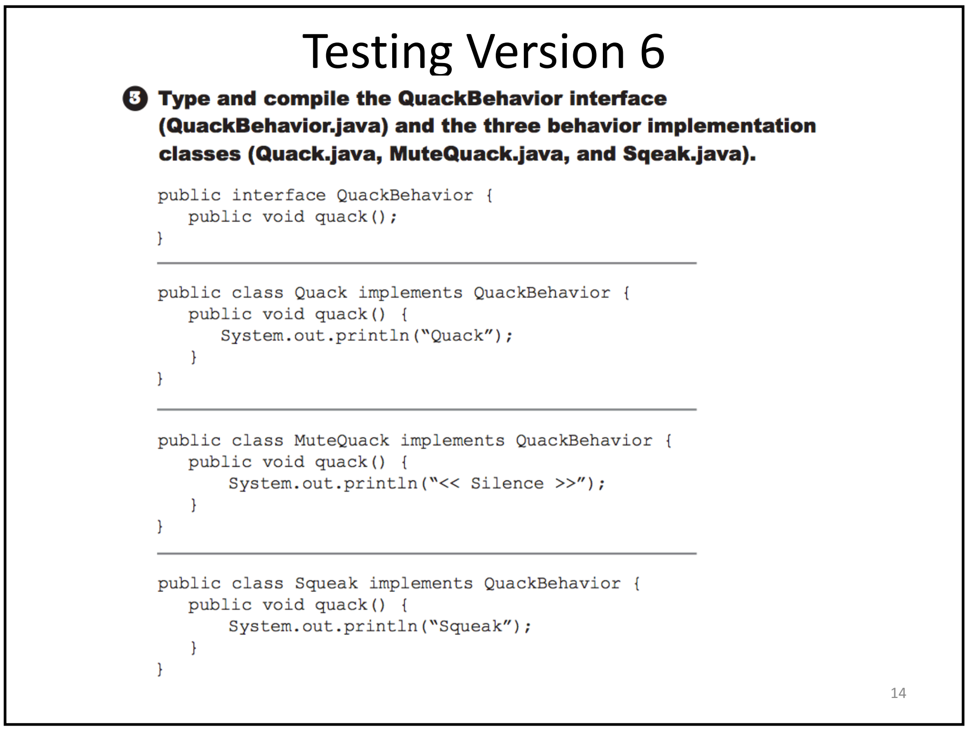
```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
       System.out.println("I'm flying!!");
   }
}
```

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).

13



Testing Version 6

3 Type and compile the QuackBehavior interface (QuackBehavior.java) and the three behavior implementation classes (Quack.java, MuteQuack.java, and Sqeak.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");
   }
}
```

14

Testing Version 6

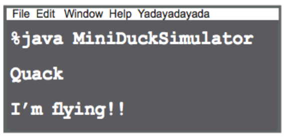
4 Type and compile the test class (MiniDuckSimulator.java).

```
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!



```
%java MiniDuckSimulator
Quack
I'm flying!!
```

Testing Version 6

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.

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!");
    }
}
```

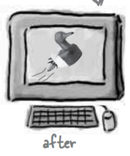
4 Change the test class (MiniDuckSimulator.java), add the ModelDuck, and make the ModelDuck rocket-enabled.

```
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();
   }
}
```

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!

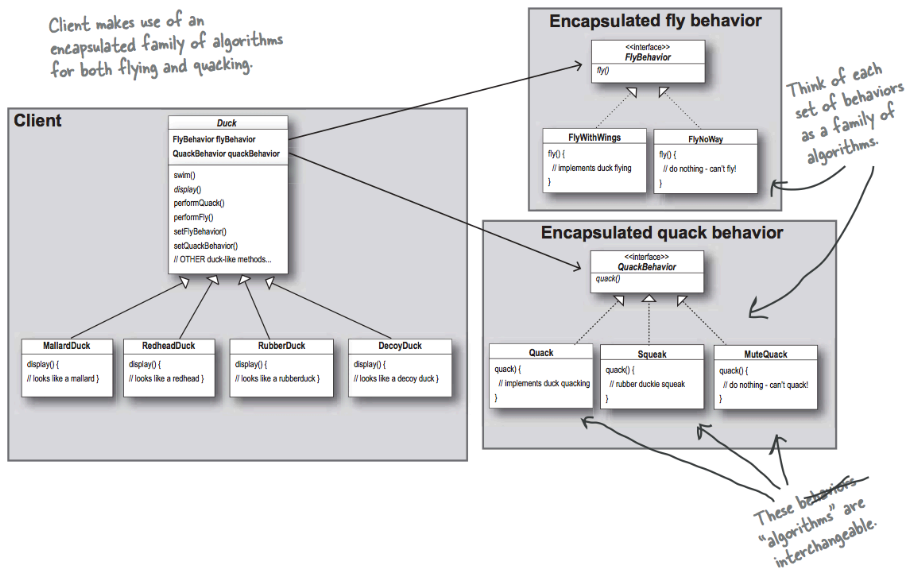
If it worked, the model duck dynamically changed its flying behavior! You can't do THAT if the implementation lives inside the duck class.



5 Run it!

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

Version 6 – The big picture


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.