Strategy Pattern

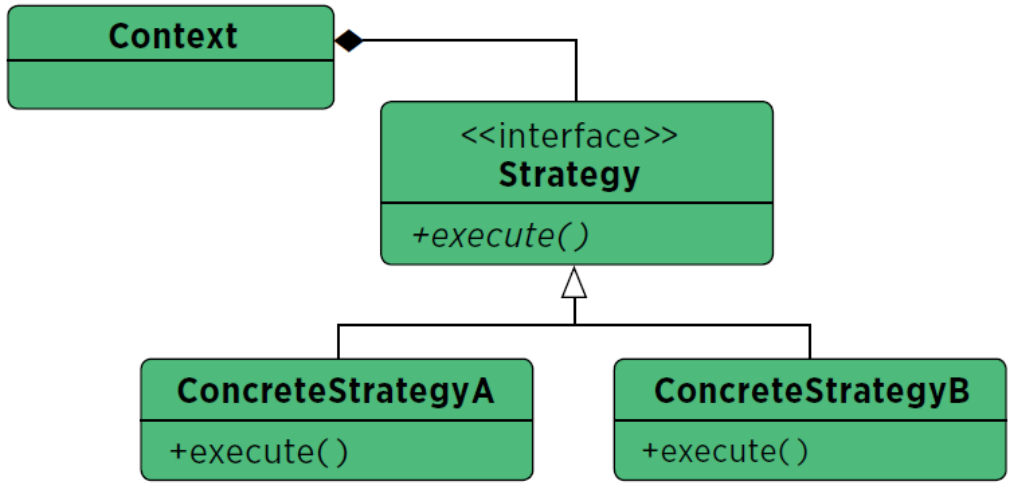
For the Complete Code, See the "Official" Head-First Design Patterns GitHub Repo:

https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/

And the course SVN repo:

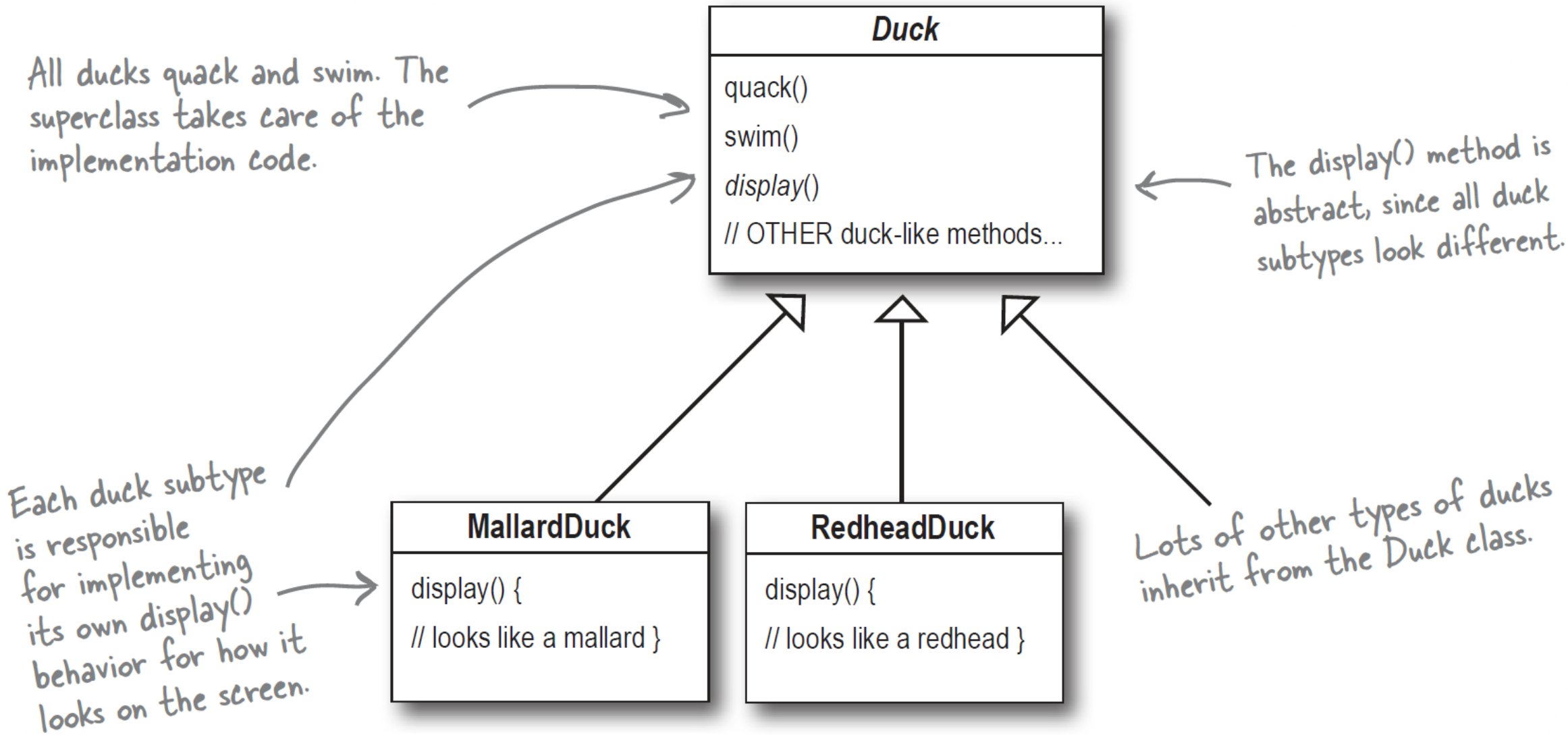
svn://cosc436.net:65436/Examples/trunk

STRATEGY — Object Behavioral

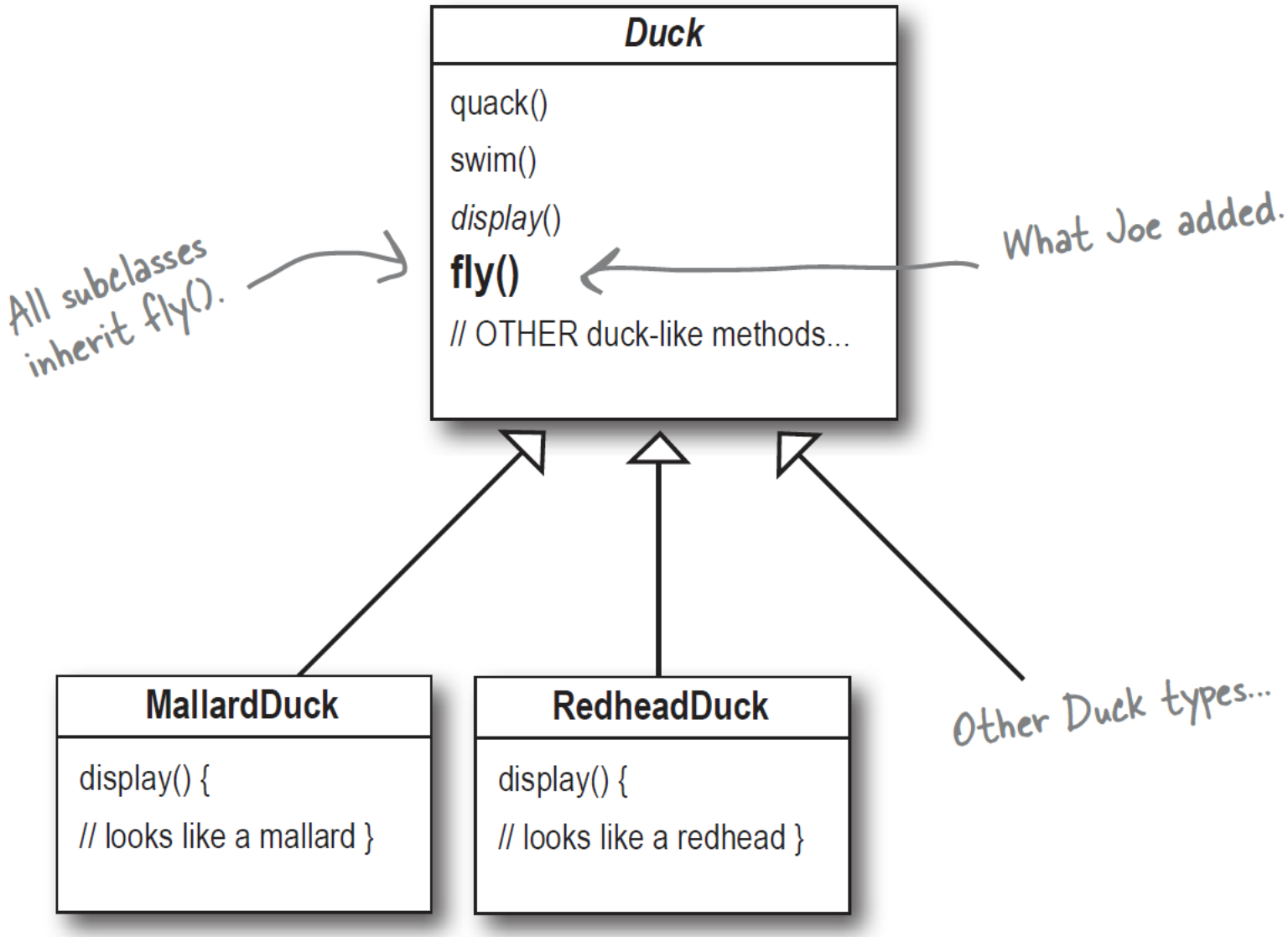


Purpose	Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.
Use When	• The only difference between many related classes is their behavior. • Multiple versions or variations of an algorithm are required. • Algorithms access or utilize data that calling code shouldn't be exposed to. • The behavior of a class should be defined at runtime. • Conditional statements are complex and hard to maintain.
Example	When importing data into a new system different validation algorithms may be run based on the data set. By configuring the import to utilize strategies the conditional logic to determine what validation set to run can be removed and the import can be decoupled from the actual validation code. This will allow us to dynamically call one or more strategies during the import.

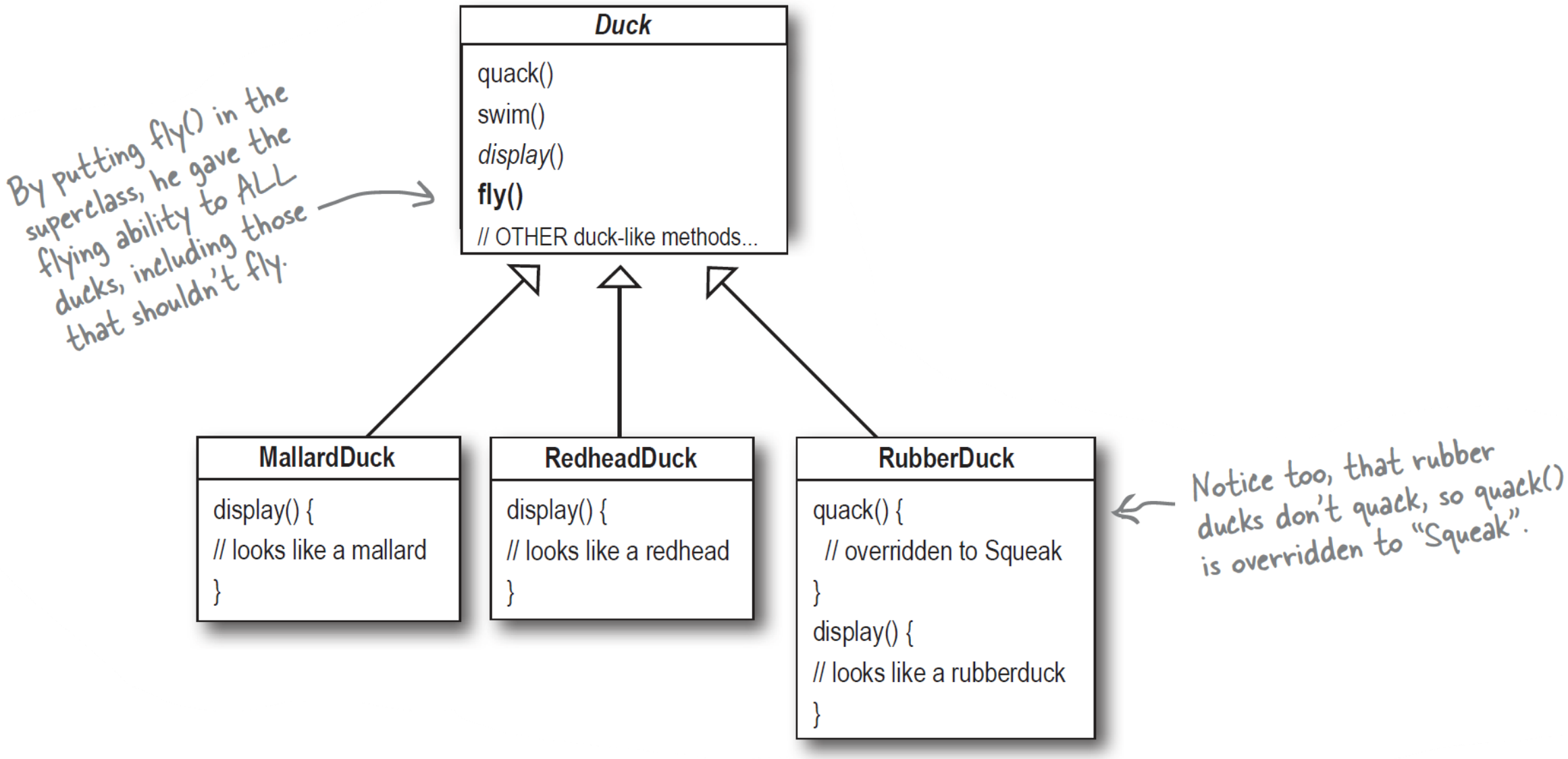
It started with a simple SimUDuck app



But now we need the ducks to FLY



But something went horribly wrong...



Brainstorming!

RubberDuck

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

Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

DecoyDuck

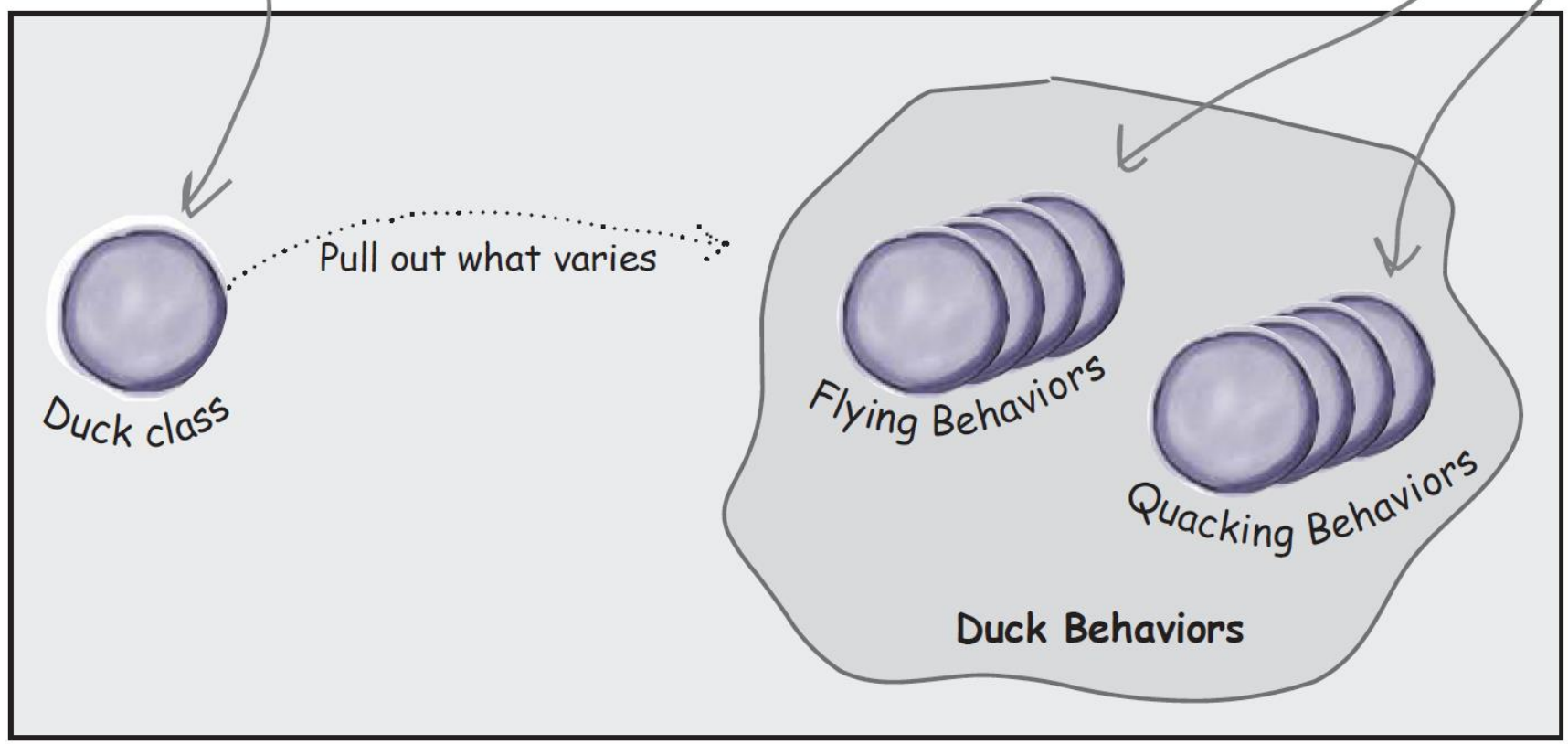
```
quack() {
  // override to do nothing
}

display() { // decoy duck}

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

Separating what changes from what stays the same

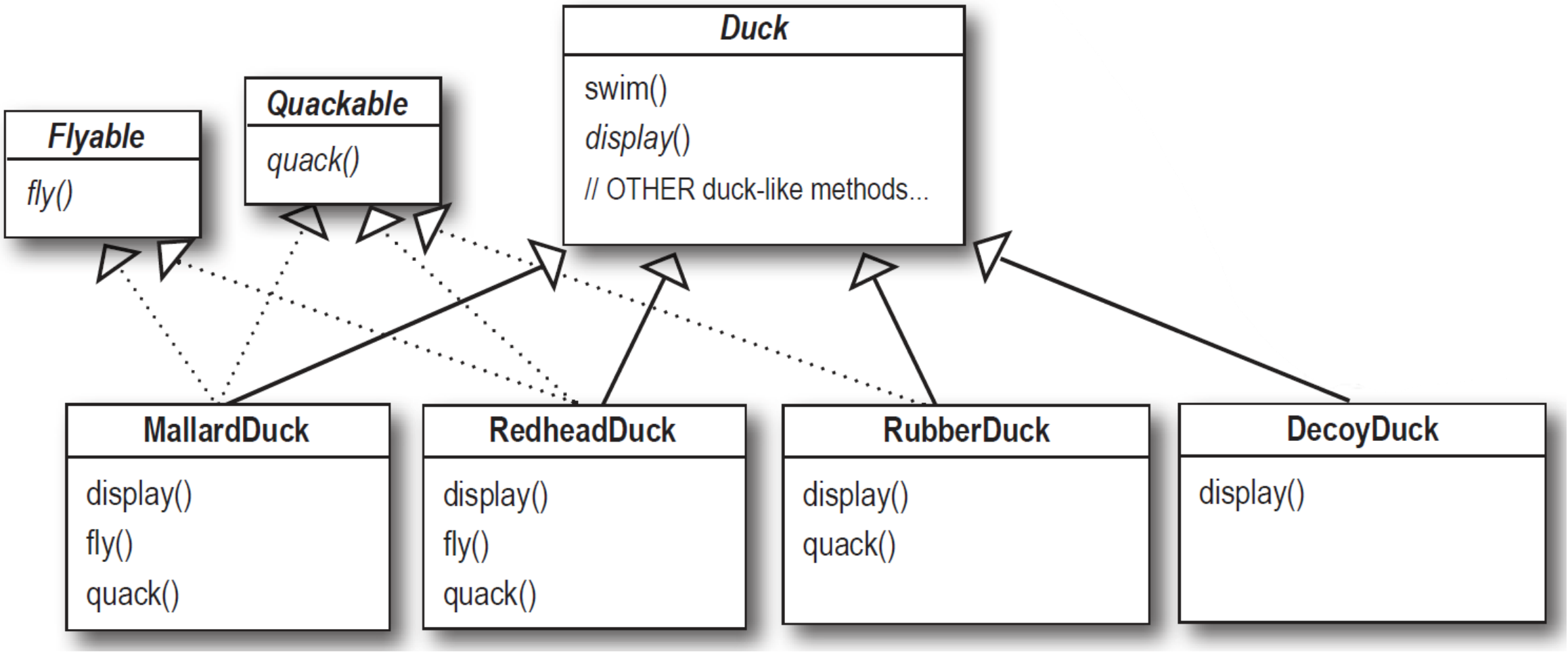
The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.

Now flying and quacking each get their own set of classes.

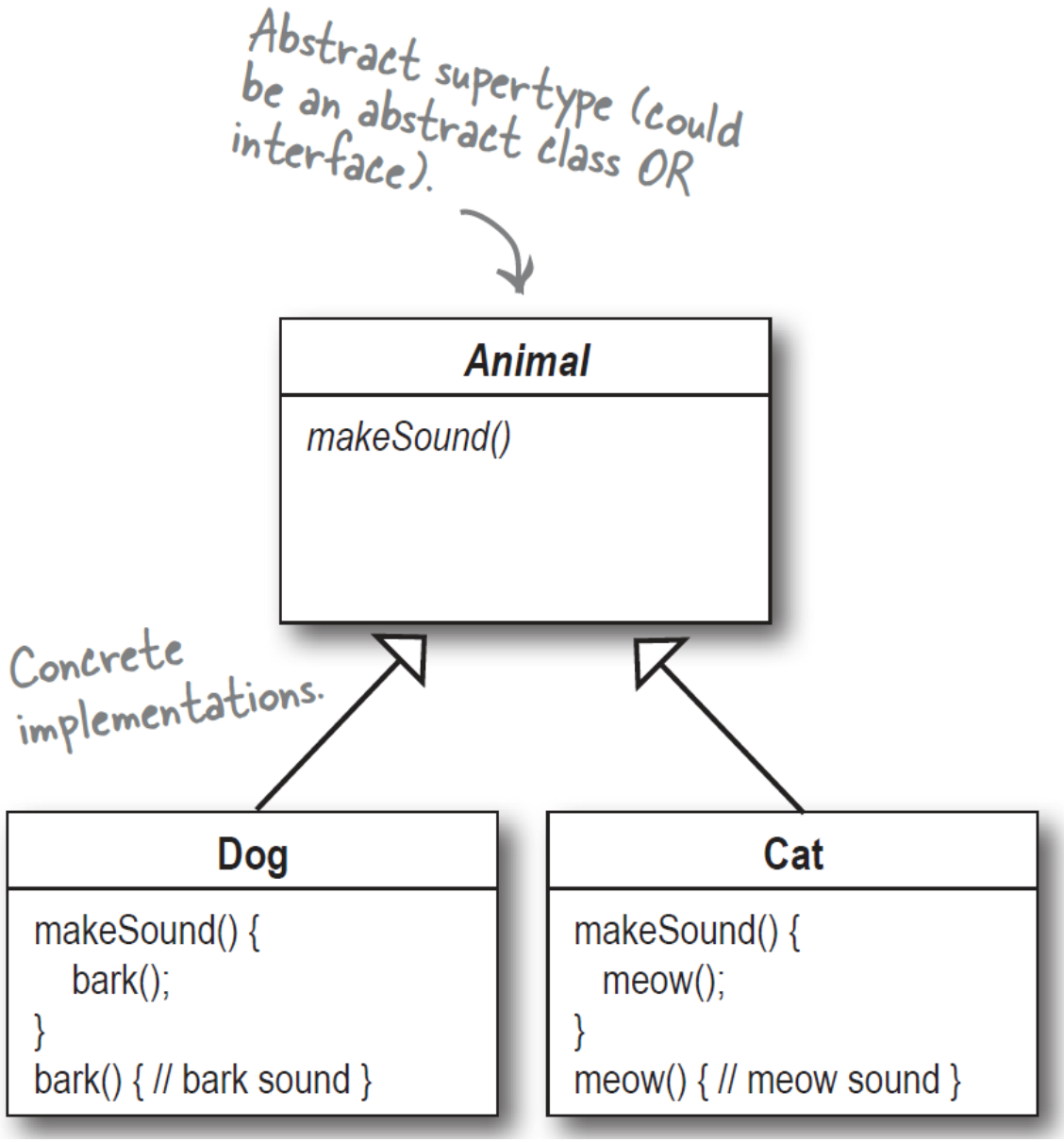
Various behavior implementations are going to live here.



How about an interface?



"Program to an interface" really means "Program to a supertype."



Programming to an implementation would be:

```
Dog d = new Dog();
d.bark();
```

Declaring the variable "d" as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

But **programming to an interface/supertype** would be:

```
Animal animal = new Dog();
animal.makeSound();
```

We know it's a Dog, but we can now use the animal reference polymorphically.

Even better, rather than hardcoding the instantiation of the subtype (like new Dog()) into the code, **assign the concrete implementation object at runtime:**

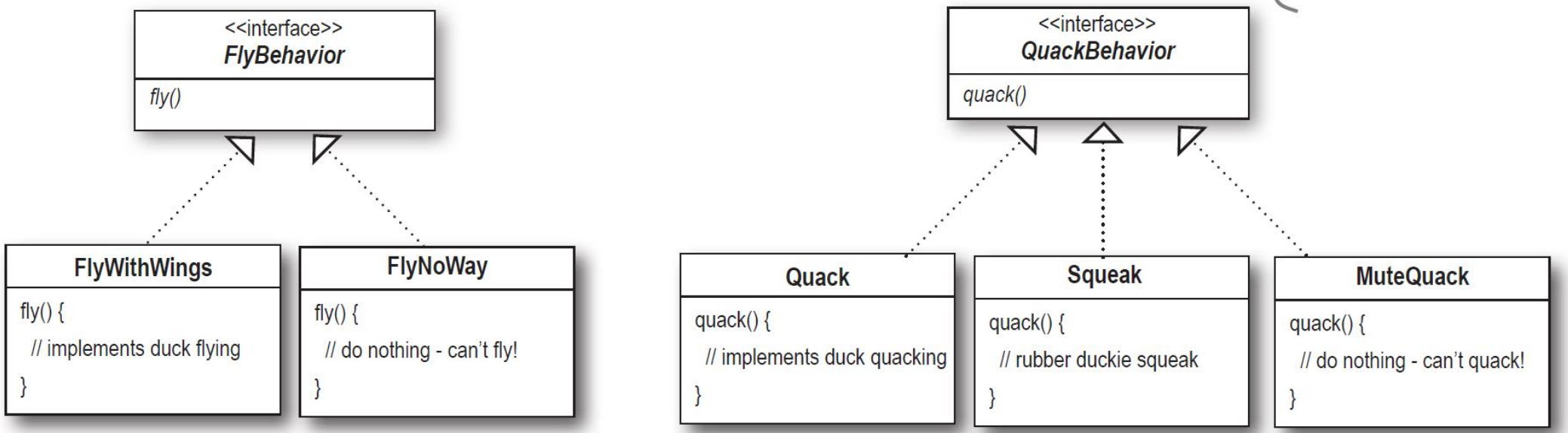
```
a = getAnimal();
a.makeSound();
```

We don't know WHAT the actual animal subtype is...all we care about is that it knows how to respond to makeSound().

Implementing the Duck 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.



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.

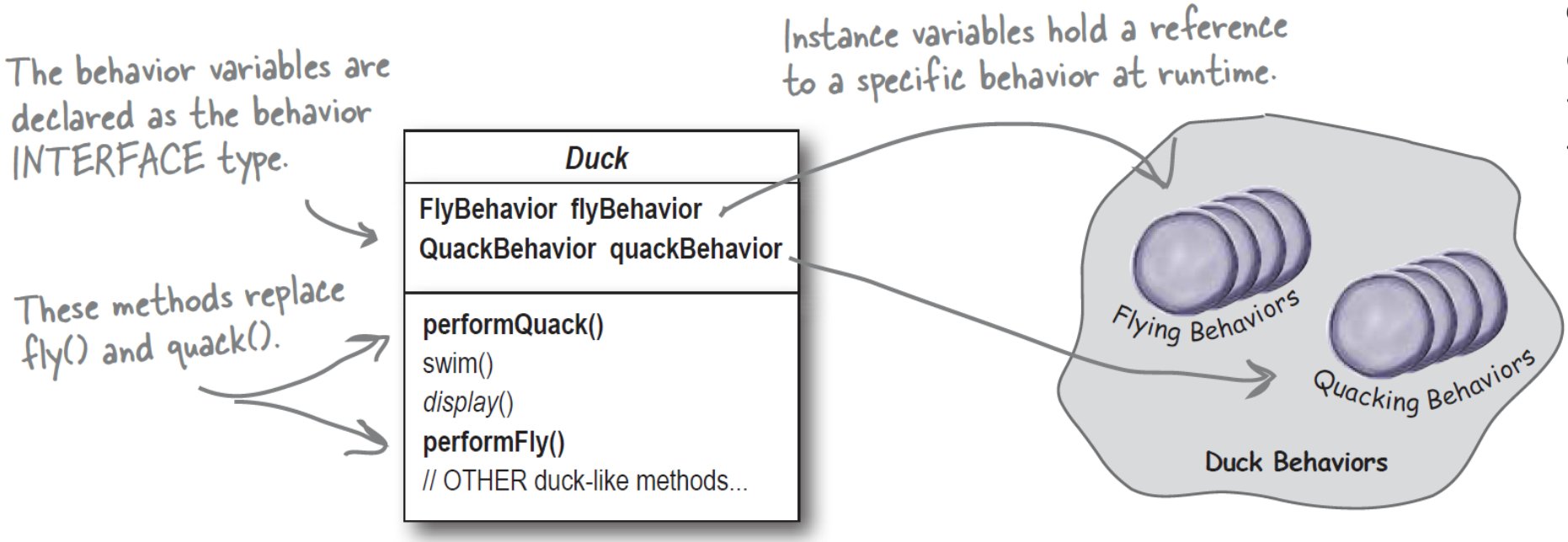
With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes!

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 Behaviors

Here's the key: A Duck will now delegate its flying and quacking behaviors, instead of using quacking and flying methods defined in the Duck class (or subclass).



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

```
public abstract 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.

More integration...

```
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.

The Big Picture on encapsulated behaviors