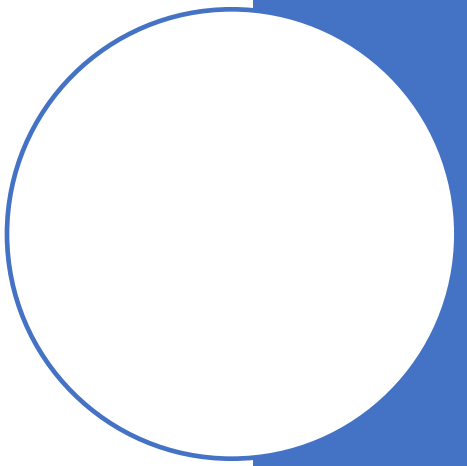
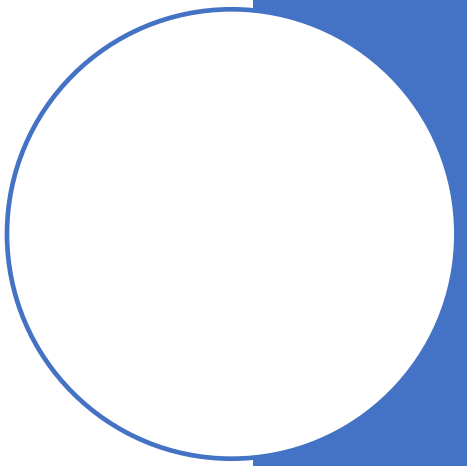
Design Patterns



Book: Head First Design Pattern

Design Pattern

- Avoid reinventing the wheel. Someone else has already solved the problem.
- Exploit the wisdom/lessons learned by those who have faced the same software design problem and survived.
- With design patterns, you get to take the advantage of the best practices and experience of others, so that you can focus on something else.
- Instead of *code* reuse, with patterns you get *experience* reuse.

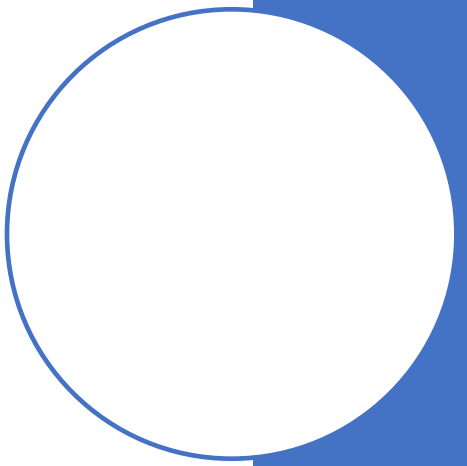


Book: Head-First Design Pattern

Observer Pattern

Keeps your objects in the know when something they might care about happens.

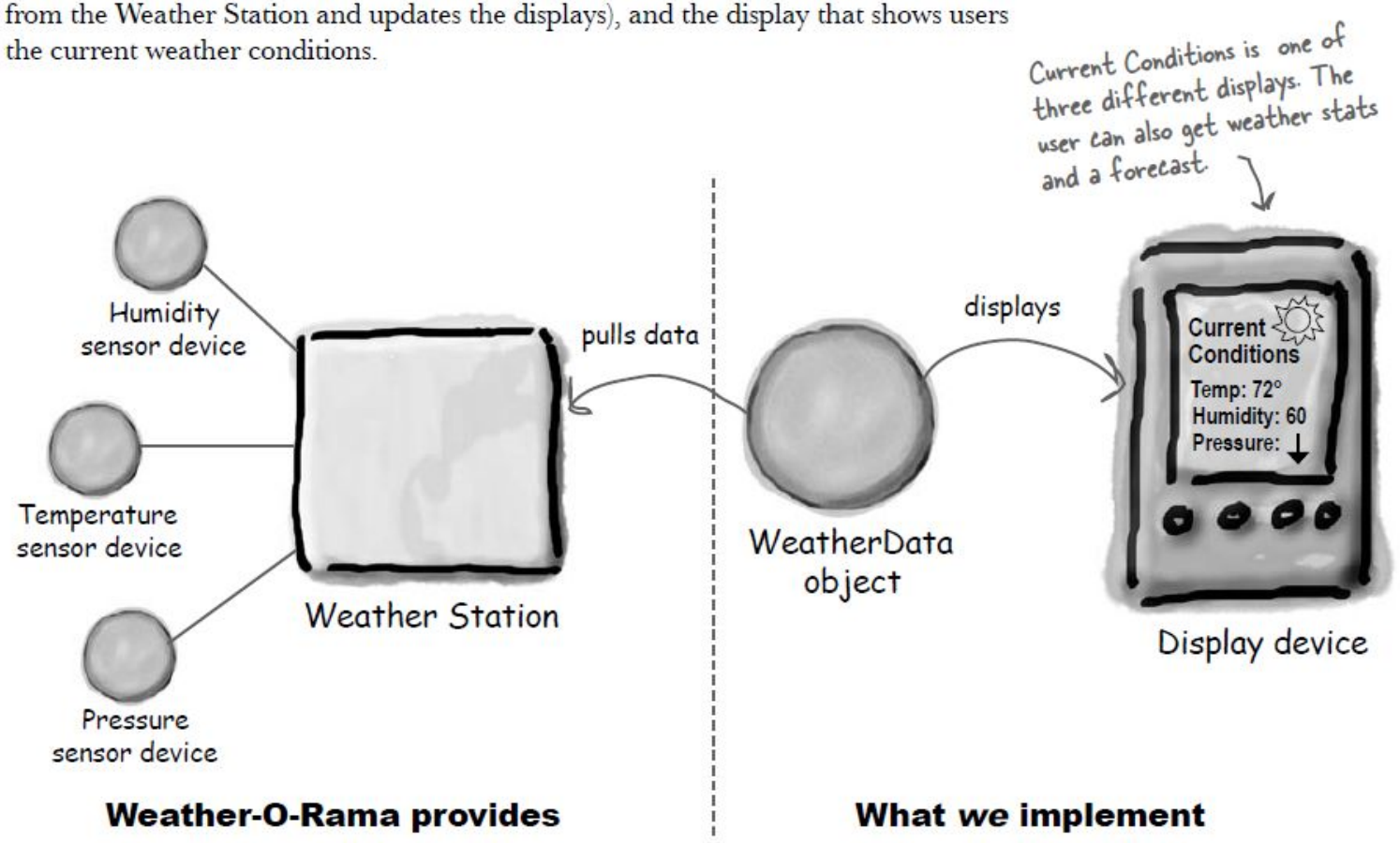
Don’t miss out when something interesting happens!



Observer Pattern

The Weather Monitoring application overview

The three players in the system are the weather station (the physical device that acquires the actual weather data), the WeatherData object (that tracks the data coming from the Weather Station and updates the displays), and the display that shows users the current weather conditions.



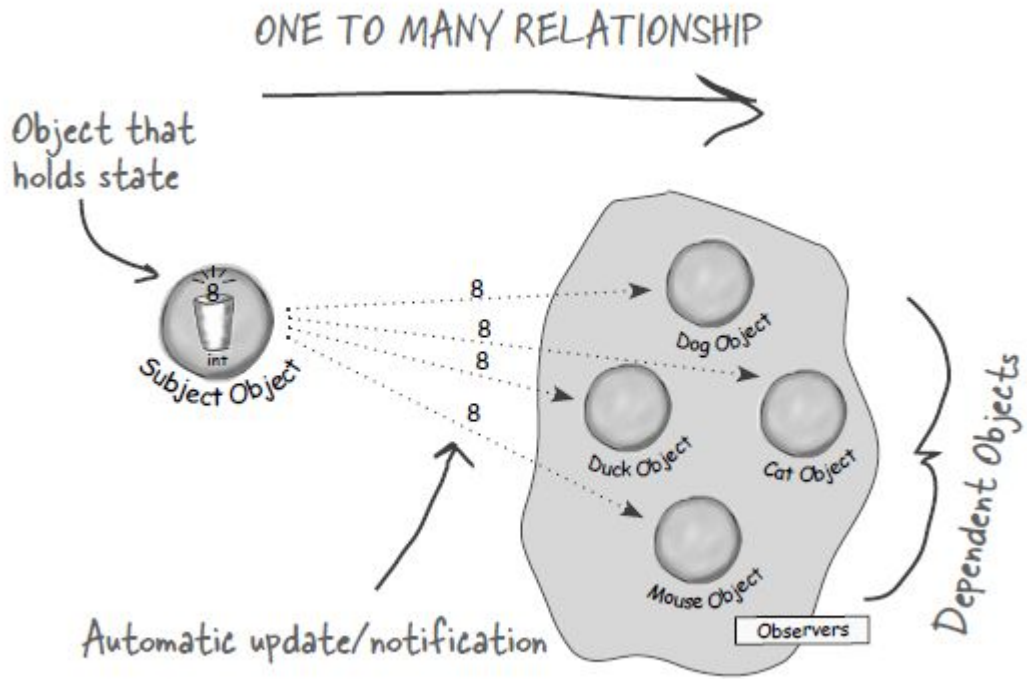
Book: Head First Design Pattern

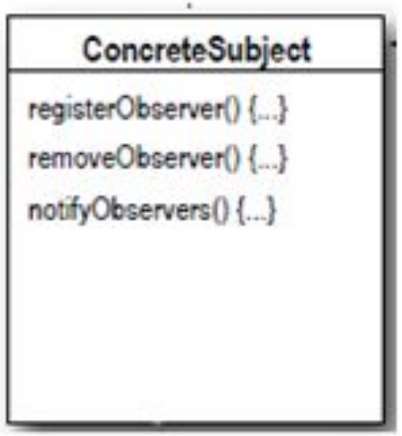
Observer Pattern
Publisher – Subscribers = Observer Pattern

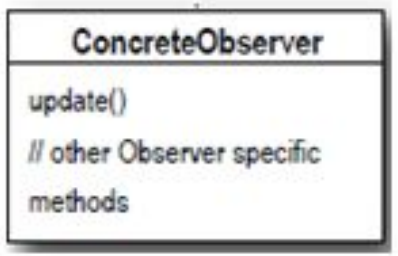
Example: Newspaper subscription

The Observer Pattern defines a one-to-many relationship between a set of objects.

When the state of one object changes, all of its dependents are notified.

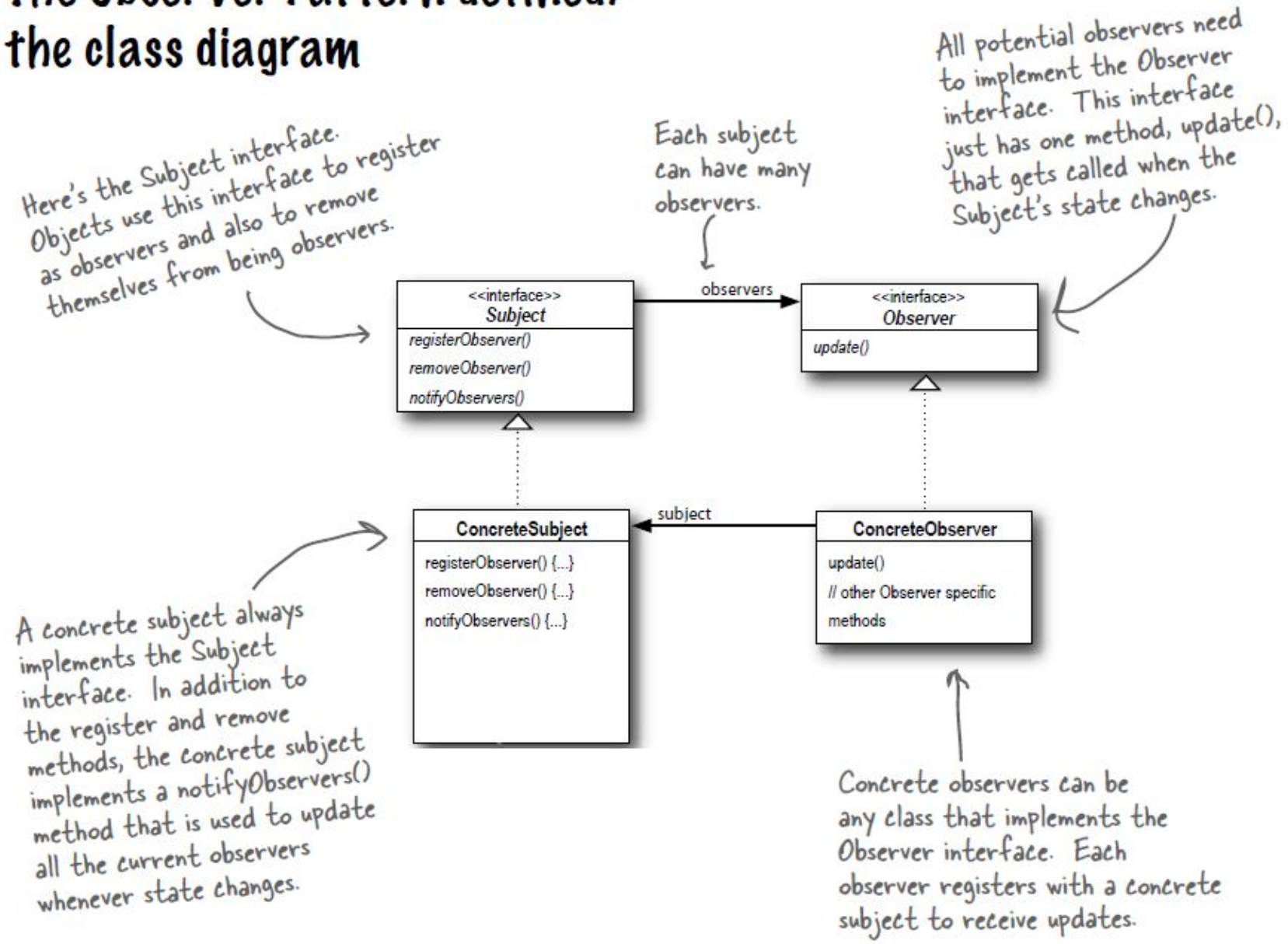






Book: Head First Design Pattern

The Observer Pattern defined: the class diagram

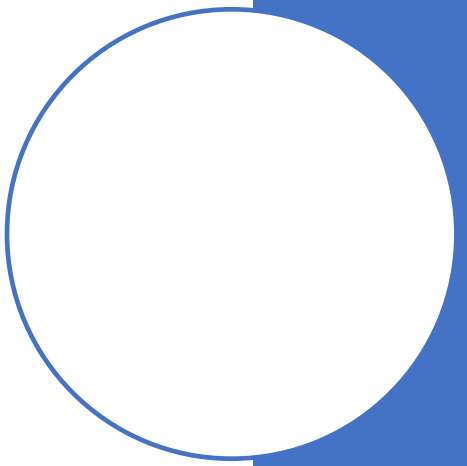


Book: Head First Design Pattern

Our system is now loosely coupled

Question

So, what are the advantages of being loosely coupled?



Our system is now loosely coupled

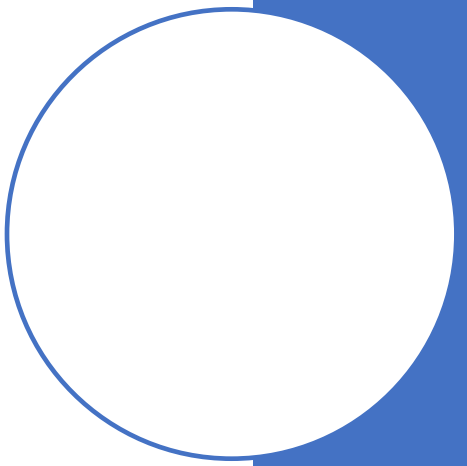
i) The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface).

ii) We can add new observers at any time.

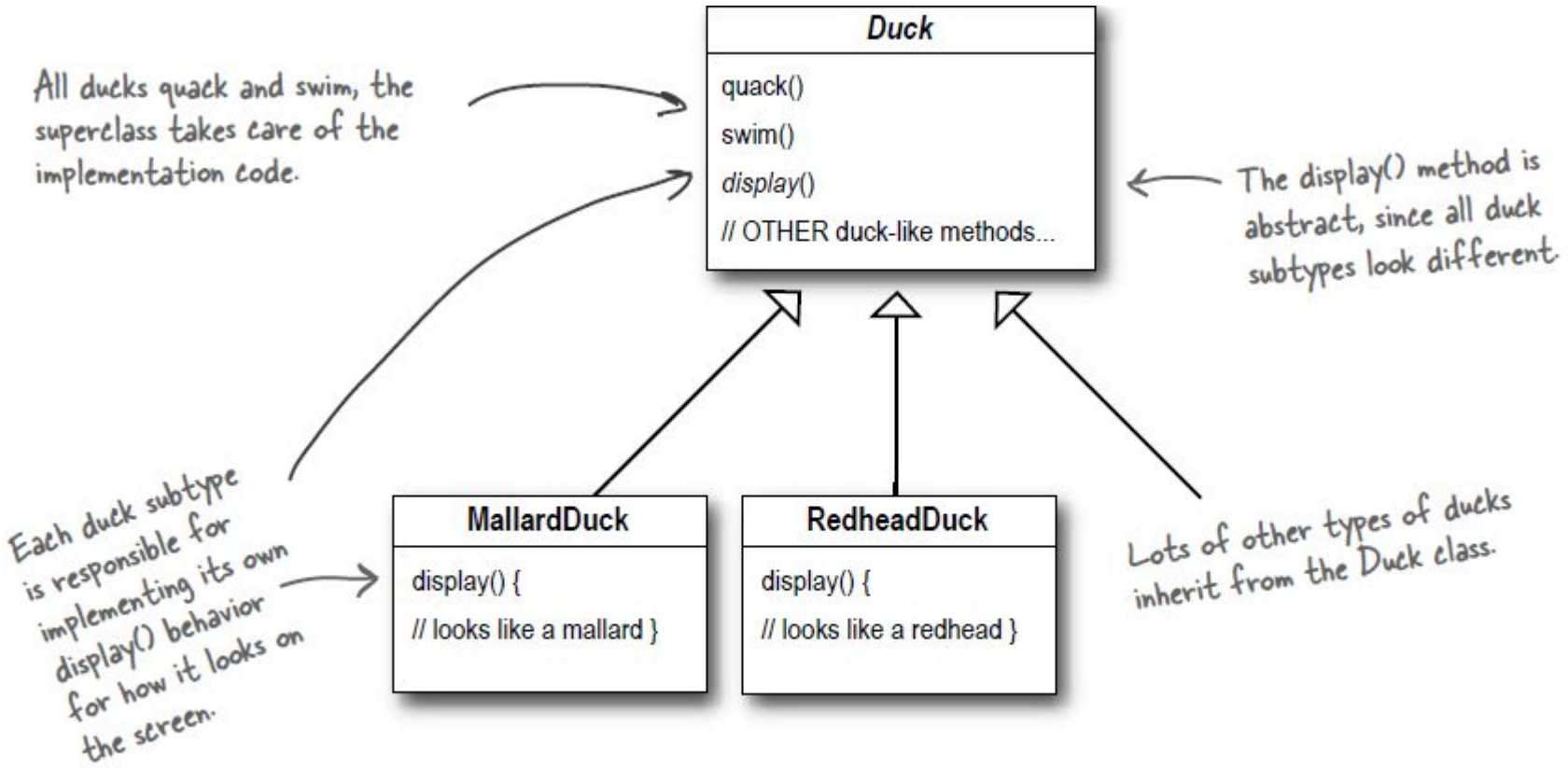
iii) We never need to modify the subject to add new types of observers.

iv) We can reuse subjects or observers independently of each other.

v) Changes to either the subject or an observer will not affect the other.



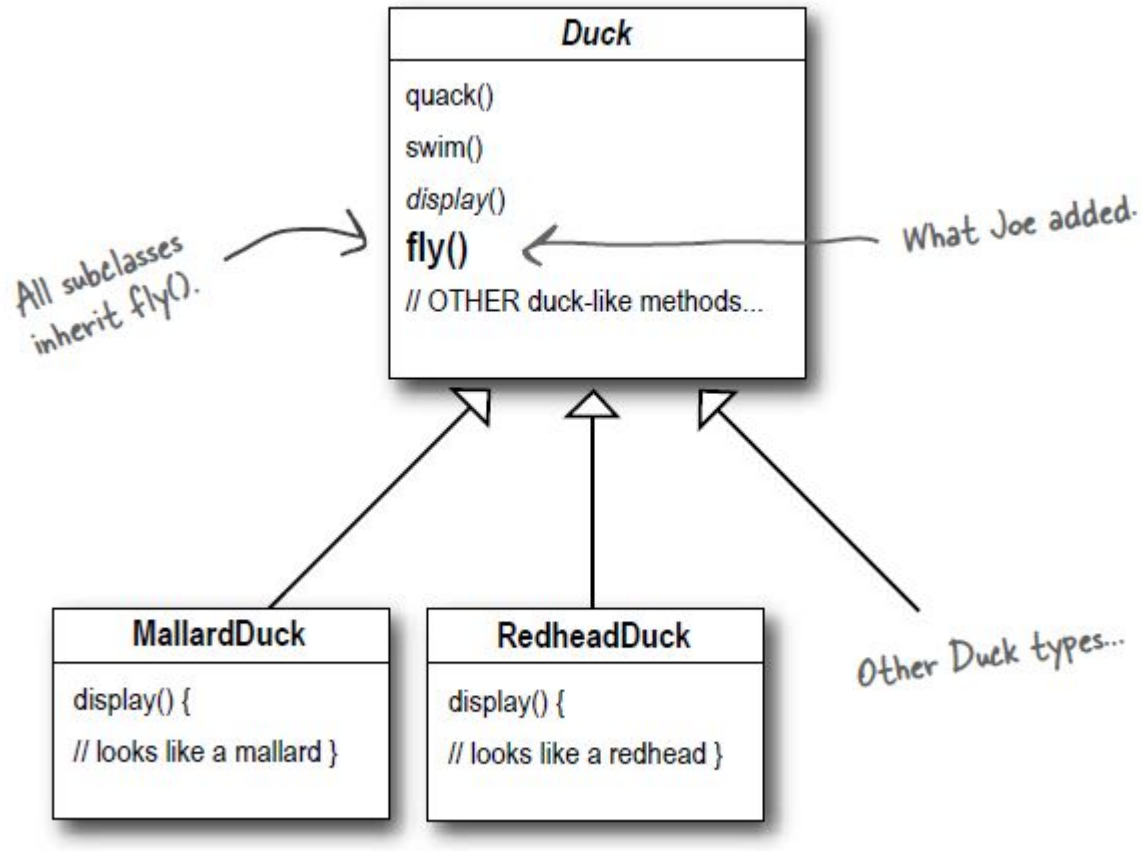
Strategy Pattern Motivation



Book: Head First Design Pattern

Now, we need the ducks to fly as well

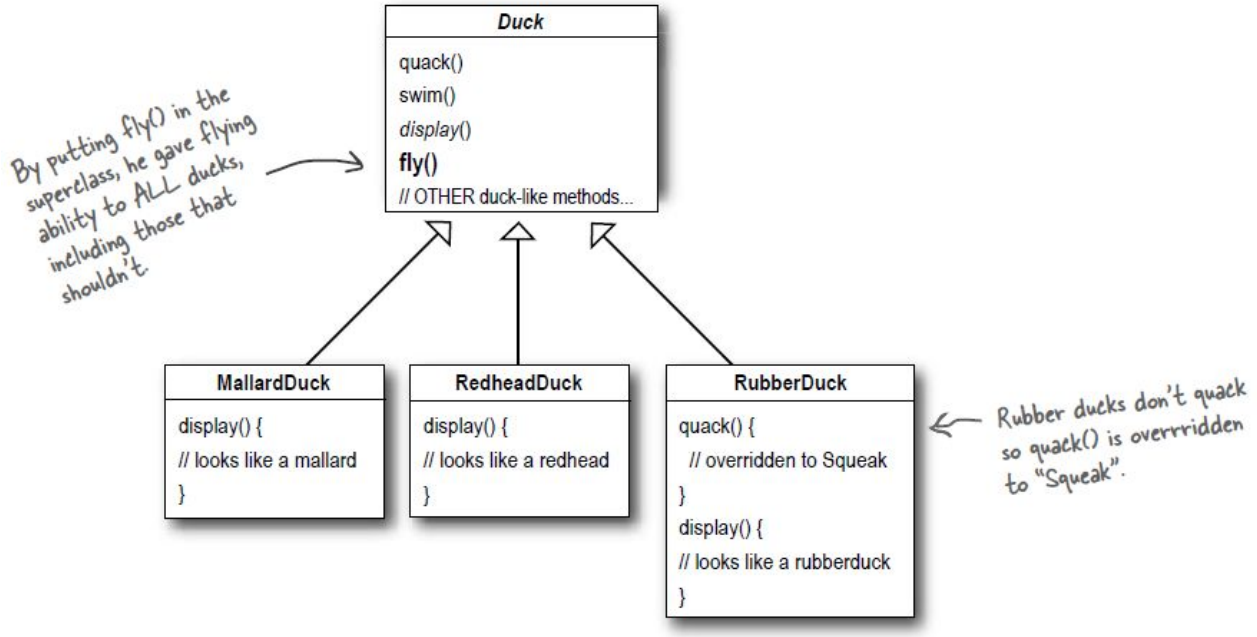
..



Book: Head First Design Pattern

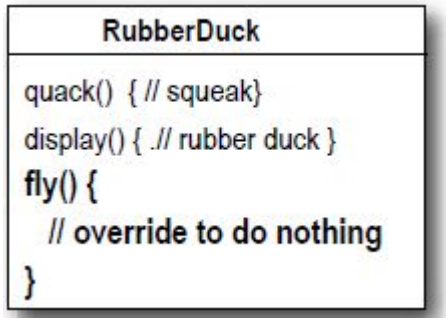
Now, we need the ducks to fly as well

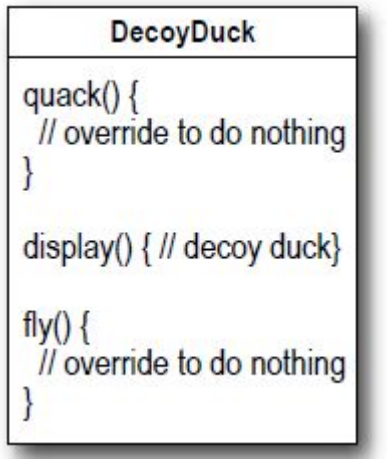
Problem: We didn't notice that now all subclasses of Duck would start flying.



Book: Head First Design Pattern

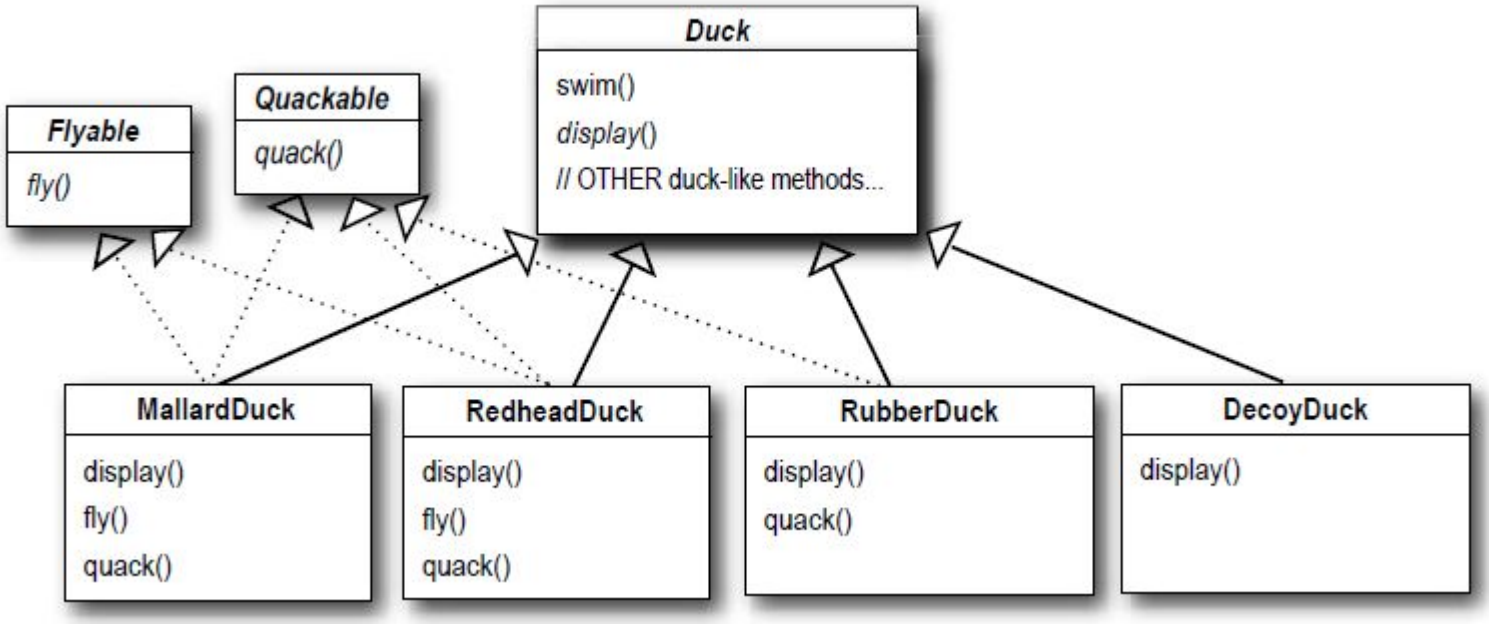
Solution to previous problem





But again, we are now in new problem.

How about using interface?



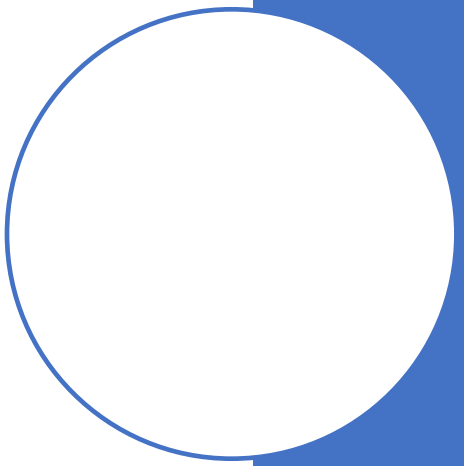
Is there any problem with using this approach?

Book: Head First Design Pattern

How about using interface?

Earlier, 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?!

It just creates a different maintenance nightmare.

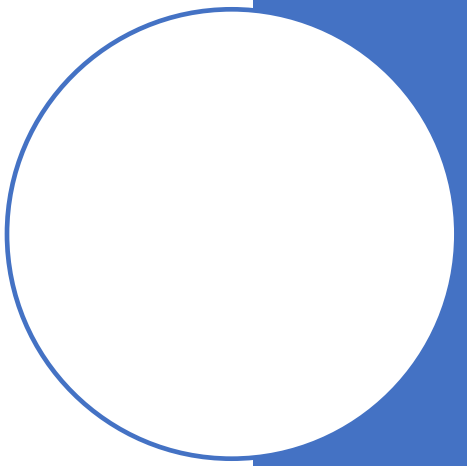


Strategy Pattern

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

Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t.

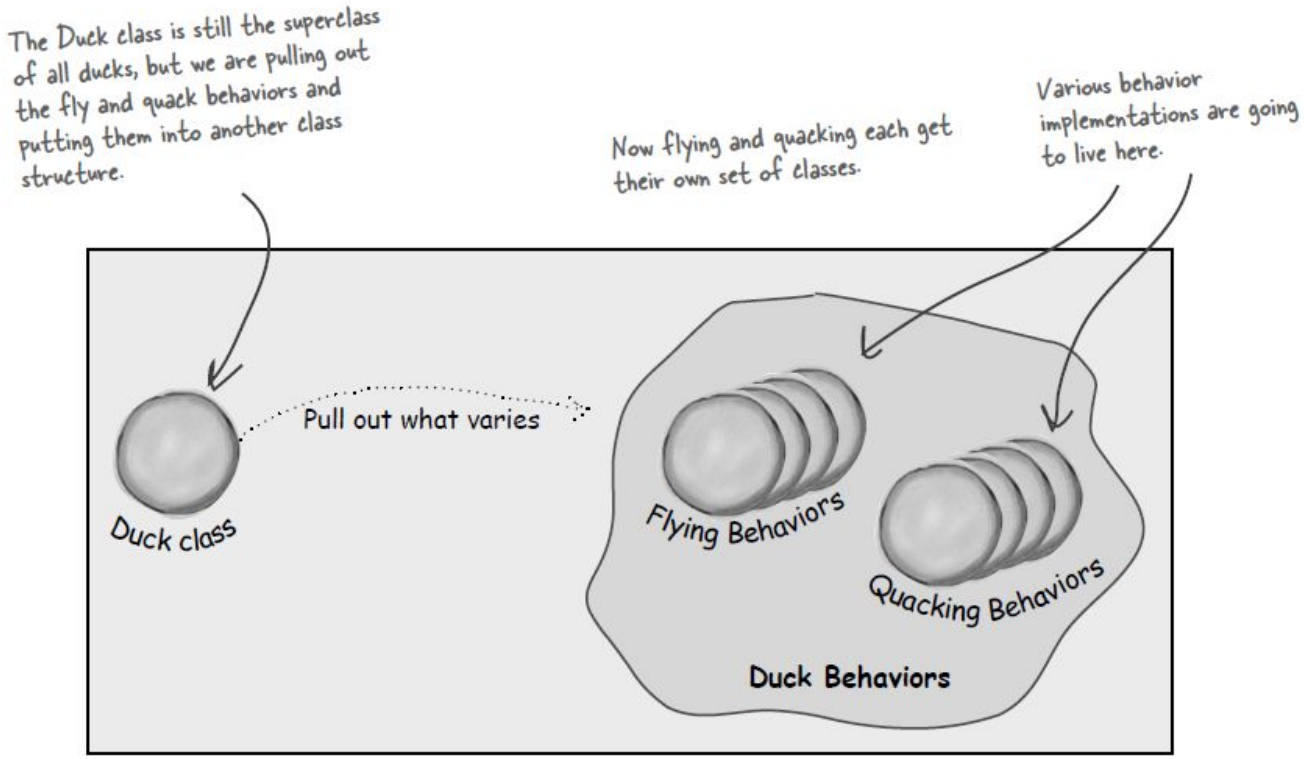
In the duck example, pull the duck behavior (that varies) out of the Duck classes e.g., fly, quack etc.



Separating what changes from what stays the same

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.



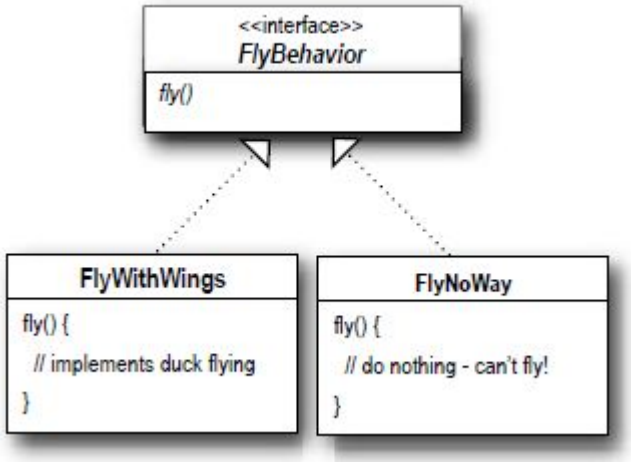
Book: Head First Design Pattern

Separating what changes from what stays the same

From now on, the Duck behaviors will live in a separate class—a class that implements a particular behavior interface.

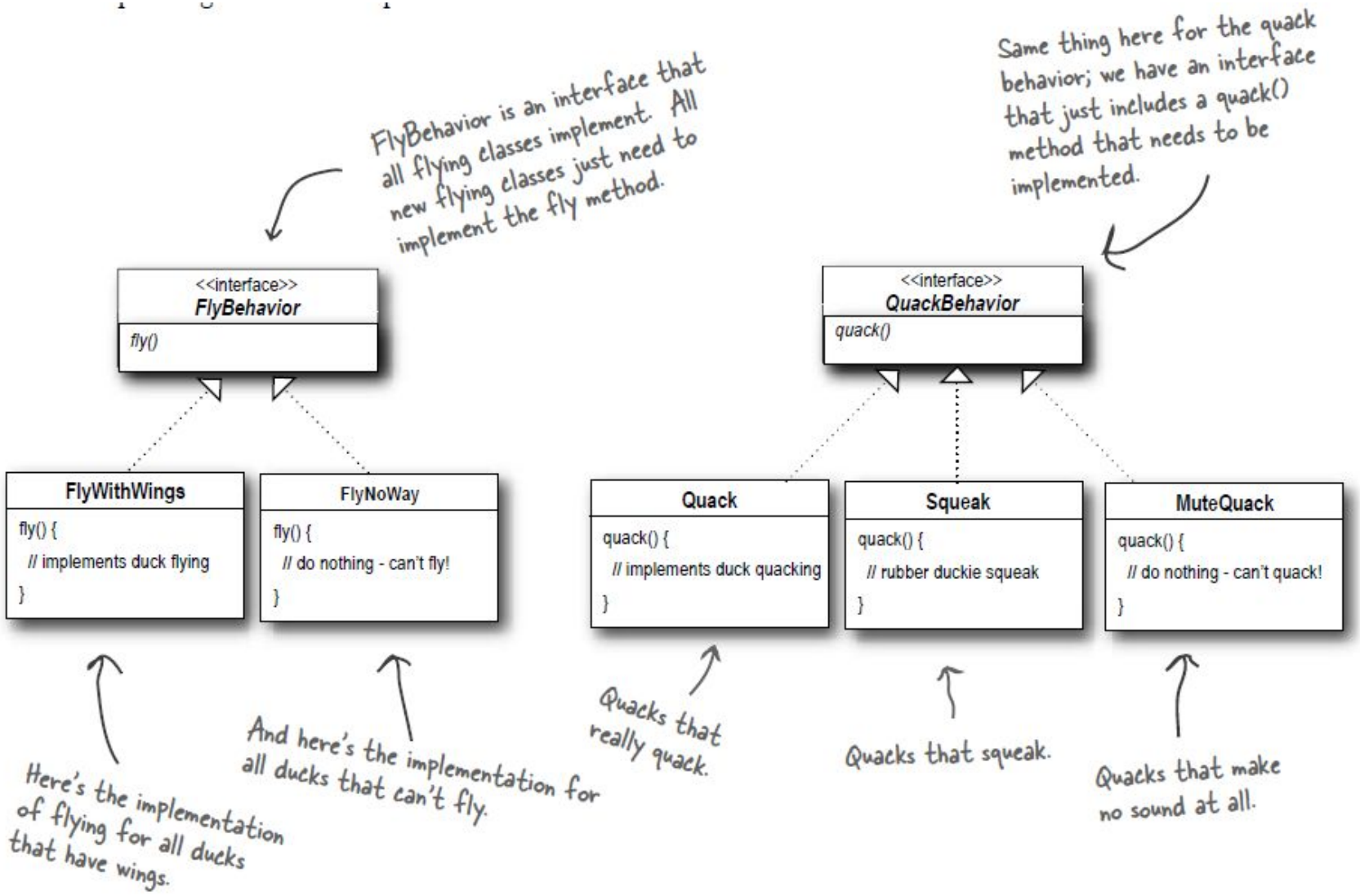
We'll make a set of classes whose entire reason for living is to represent a behavior (for example, "squeaking"), and it's the **behavior class**, rather than the Duck class, that will implement the behavior interface.

For example, we might want to instantiate a new MallardDuck instance and initialize it with a specific type of flying behavior.



Book: Head First Design Pattern

Implementing the Duck Behaviors

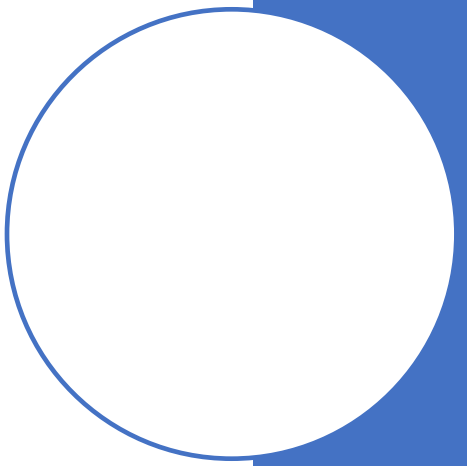


Book: Head First Design Pattern

Benefits of Using This New Approach

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!

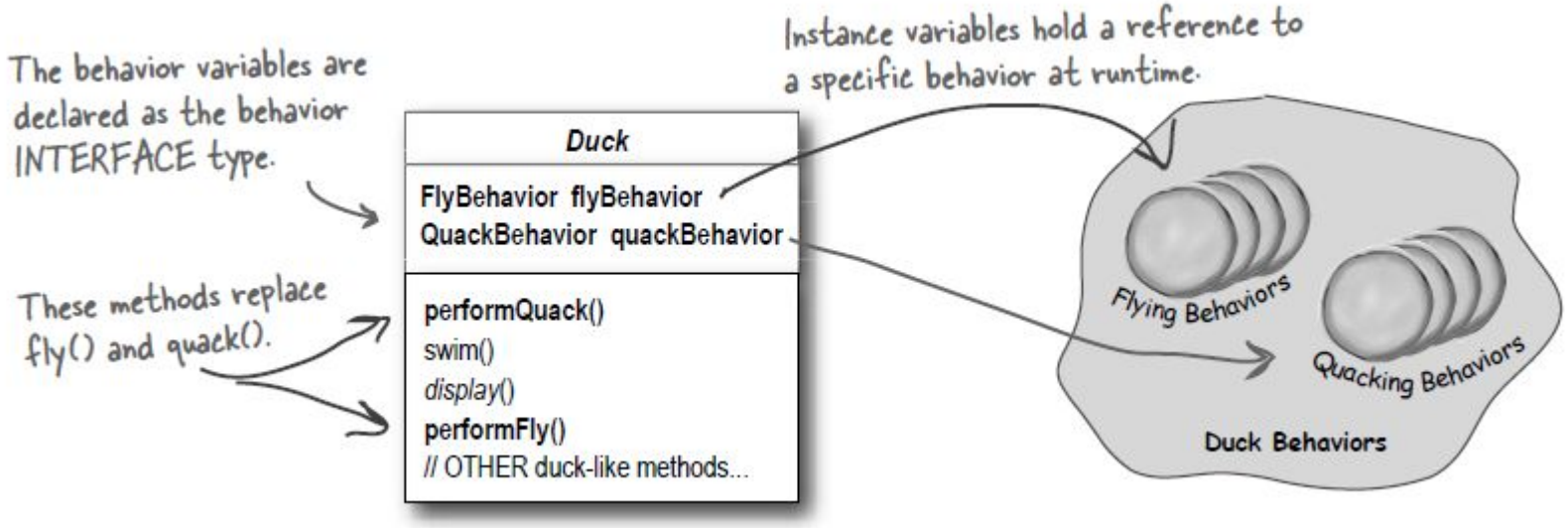
And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that use flying behaviors.



Step#1

Add two instance variables to the Duck class called flyBehavior and quackBehavior (Interfaces).

Each duck object will set these variables polymorphically to reference the specific behavior type it would like at runtime (FlyWithWings, Squeak, etc.).



Book: Head First Design Pattern

Step#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.

Step#2
How Duck class looks like

```
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.

Step#2

Okay, time to worry about how the flyBehavior and quackBehavior instance variables are set. MallardDuck's quack is a real live duck quack, not a squeak and not a mute quack.

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.

Simulator

```
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.