Lecture 01: Introduction to Design Patterns

SE313, Software Design and Architecture
Damla Oguz

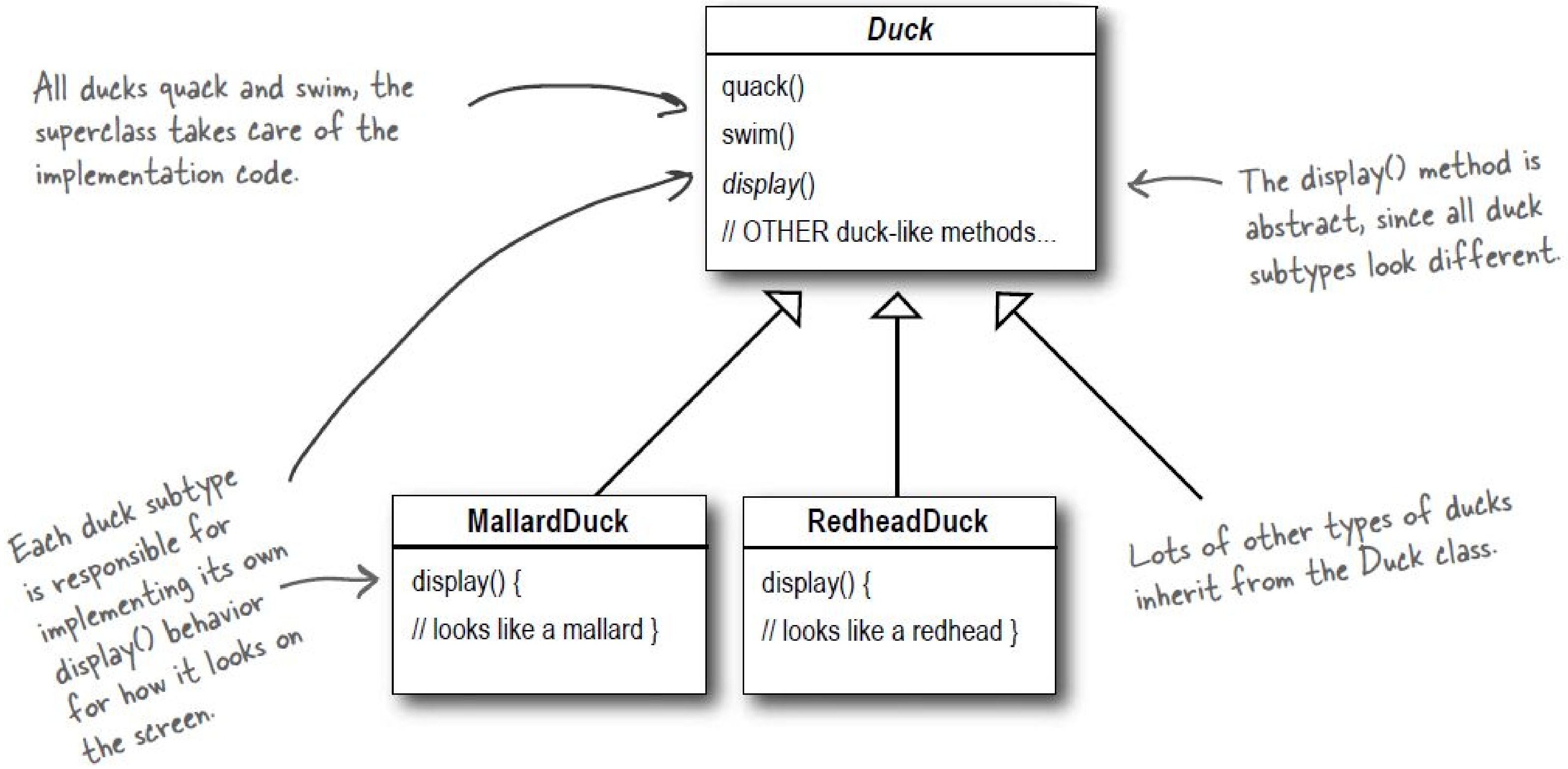
Chapter 1: Introduction to Design Patterns

- **Someone has already solved your problems**.
- In this lecture we will learn why (and how) you can exploit the wisdom and lessons learned by other developers who have been down the same design problem road and survived the trip.
- The best way to use patterns is
  - to ***load your brain*** with them and then
  - ***recognize places*** in your designs and existing applications where you can apply them.
- Instead of ***code* reuse**, with patterns you get ***experience* reuse**.

It started with a simple SimUDuck app

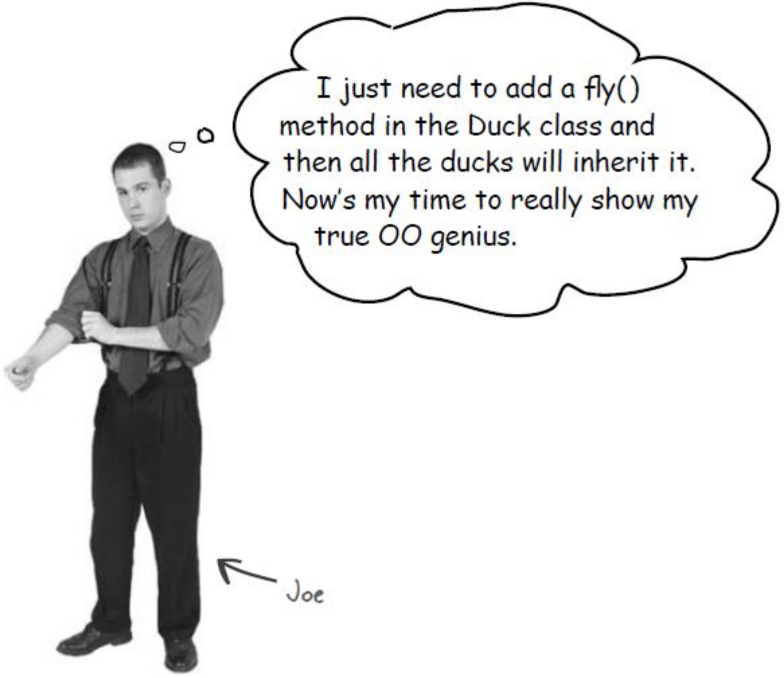
Joe works for a company that makes a highly successful duck pond simulation game, *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system uses standard OO techniques and created one Duck superclass from which all other duck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

It started with a simple SimUDuck app (cont.)

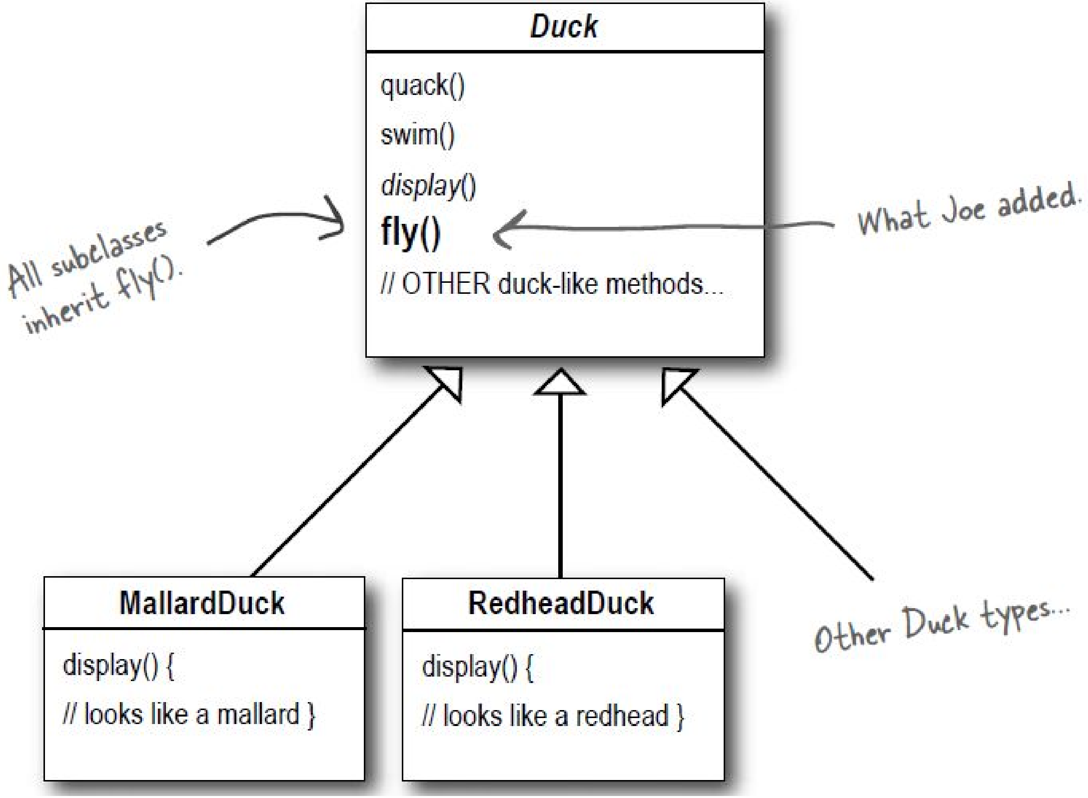


But now we need the ducks to FLY

The executives decided that flying ducks is just what the simulator needs to blow away the other duck sim competitors.



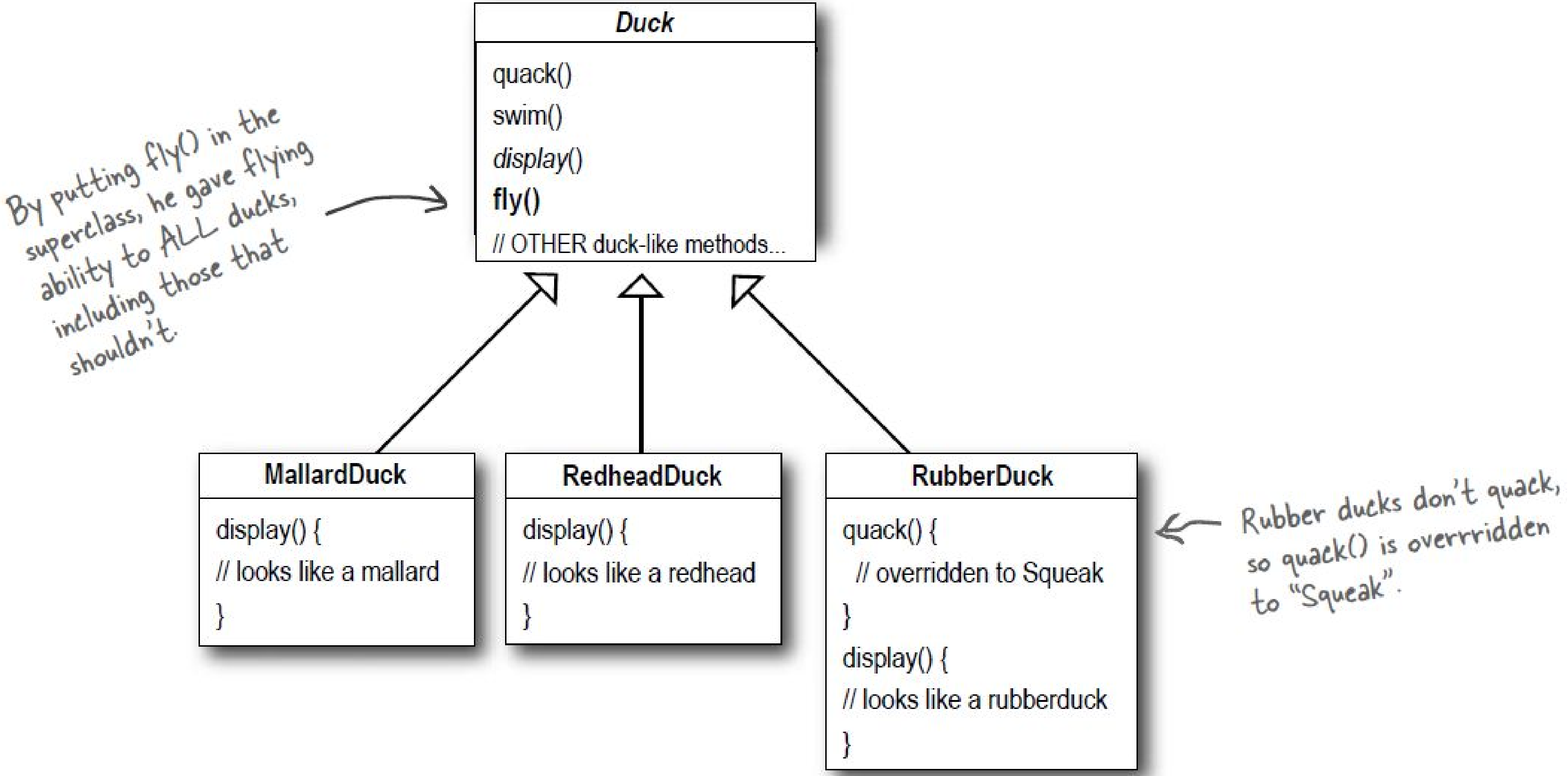
But now we need the ducks to FLY (cont.)



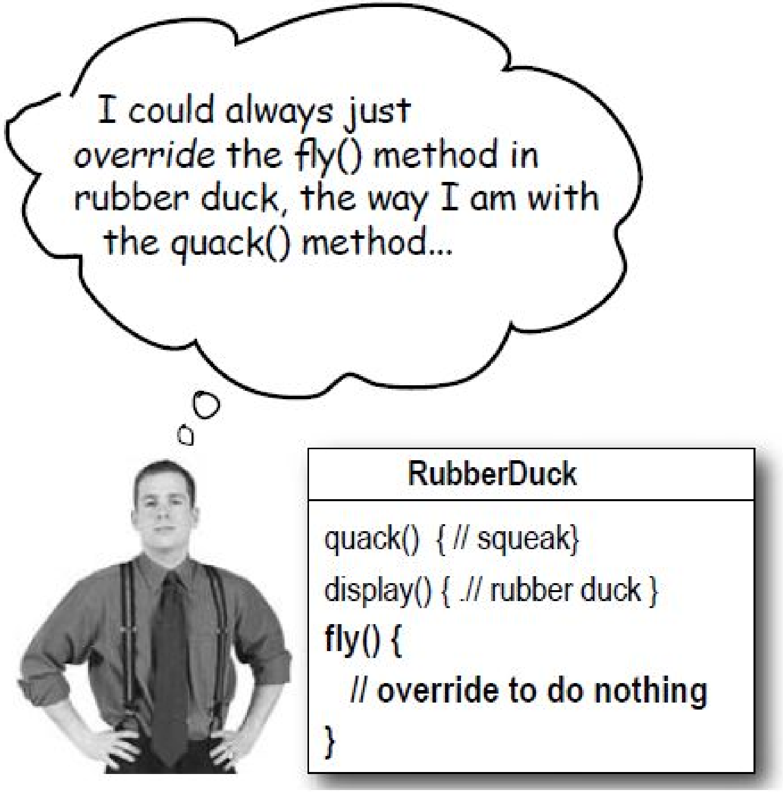
But something went horribly wrong...

- Joe failed to notice that not *all* subclasses of Duck should *fly*.
- Joe added new behavior to the Duck superclass.
  - He was also adding behavior that was *not* appropriate for some Duck subclasses.
- A localized update to the code caused a non-local side effect
  - Flying rubber ducks!

But something went horribly wrong... (cont.)



Joe thinks about inheritance…



Joe thinks about inheritance... (cont.)

DecoyDuck

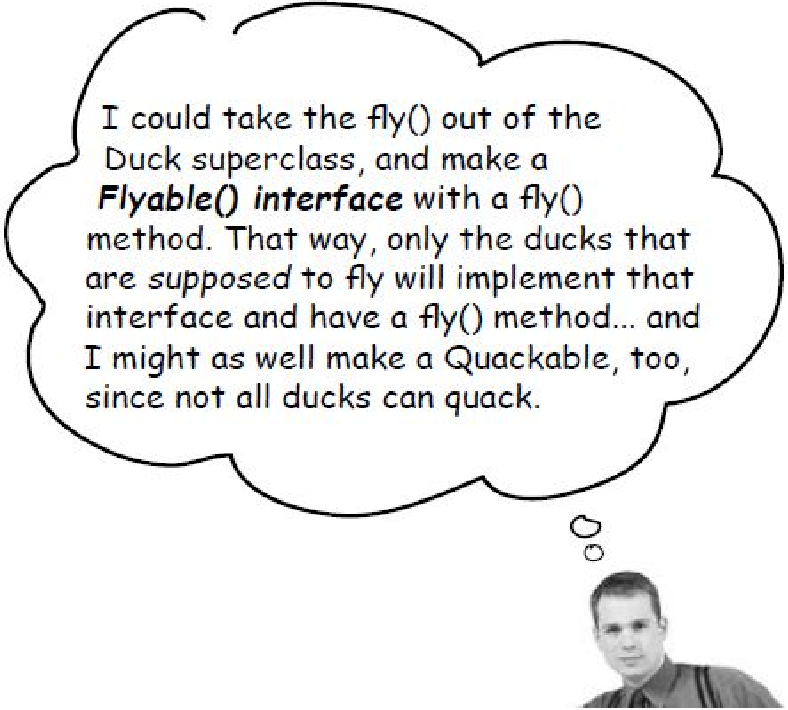
```
quack() {
  // override to do nothing
}

display() { // decoy 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.

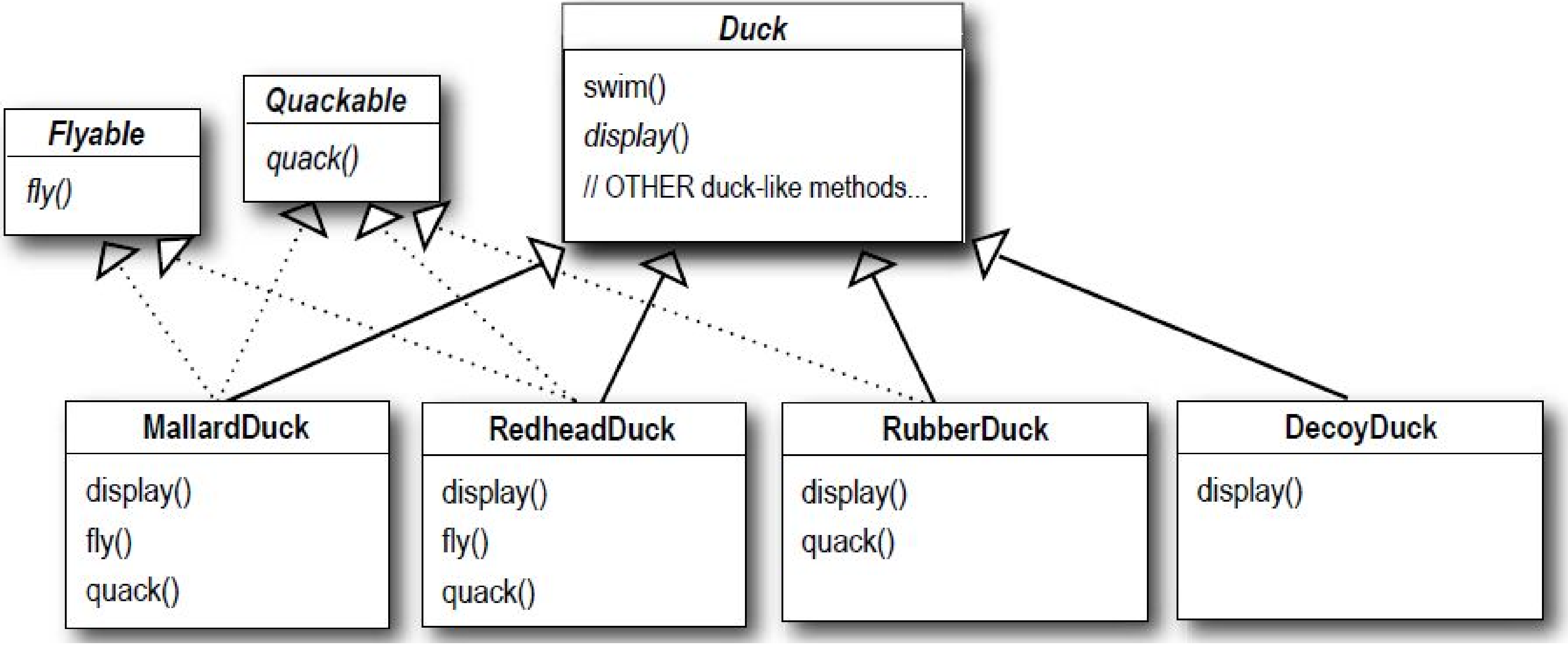
Inheritance was not the answer...

- Inheritance was not the answer because not *all* of the subclasses should have flying or quacking behavior.
- Moreover, the executives now want to update the product every six months (in ways they haven't yet decided on).
- For every new Duck subclass that's ever added to the program
  - Joe will be forced to look at and possibly override fly() and quack() ... *forever*.
- So, he needs a cleaner way to have only *some* (but not *all*) of the duck types fly or quack.

How about an interface?



What do you think about this design?



What do you think about this design? (cont.)

- Is the problem solved while having the subclasses implement Flyable and/or Quackable?
  - Part of the problem is solved.
    - No inappropriately flying rubber ducks

What do you think about this design? (cont.)

- Duplicate code
  - Joe thought having to override a few methods was bad.
  - What if he needs to make a little change to the flying behavior?
  - Maintenance nightmare!
  - And of course there might be more than one kind of flying behavior even among the ducks that *do* fly...
- What would you do if you were Joe?
  - Applying good OO software design principles?

The one constant in software development

- The one true constant that will be with you always is CHANGE.
- No matter how well you design an application, over time an application
  - must grow and change or
  - it will die

Zeroing in on the problem...

- Using inheritance hasn't worked out very well
  - the duck behavior keeps changing across the subclasses, and
  - it's not appropriate for all subclasses to have those behaviors
- The Flyable and Quackable interface sounded promising at first
  - Java interfaces have no implementation code, so no code reuse.
    - That means that whenever you need to modify a behavior, you're forced to track down and change it in all the different subclasses where that behavior is defined.
- Luckily, there's a design principle for just this situation.

Design Principle

- Design Principle: **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.

Take what varies and "encapsulate" it so it won't affect the rest of your code.

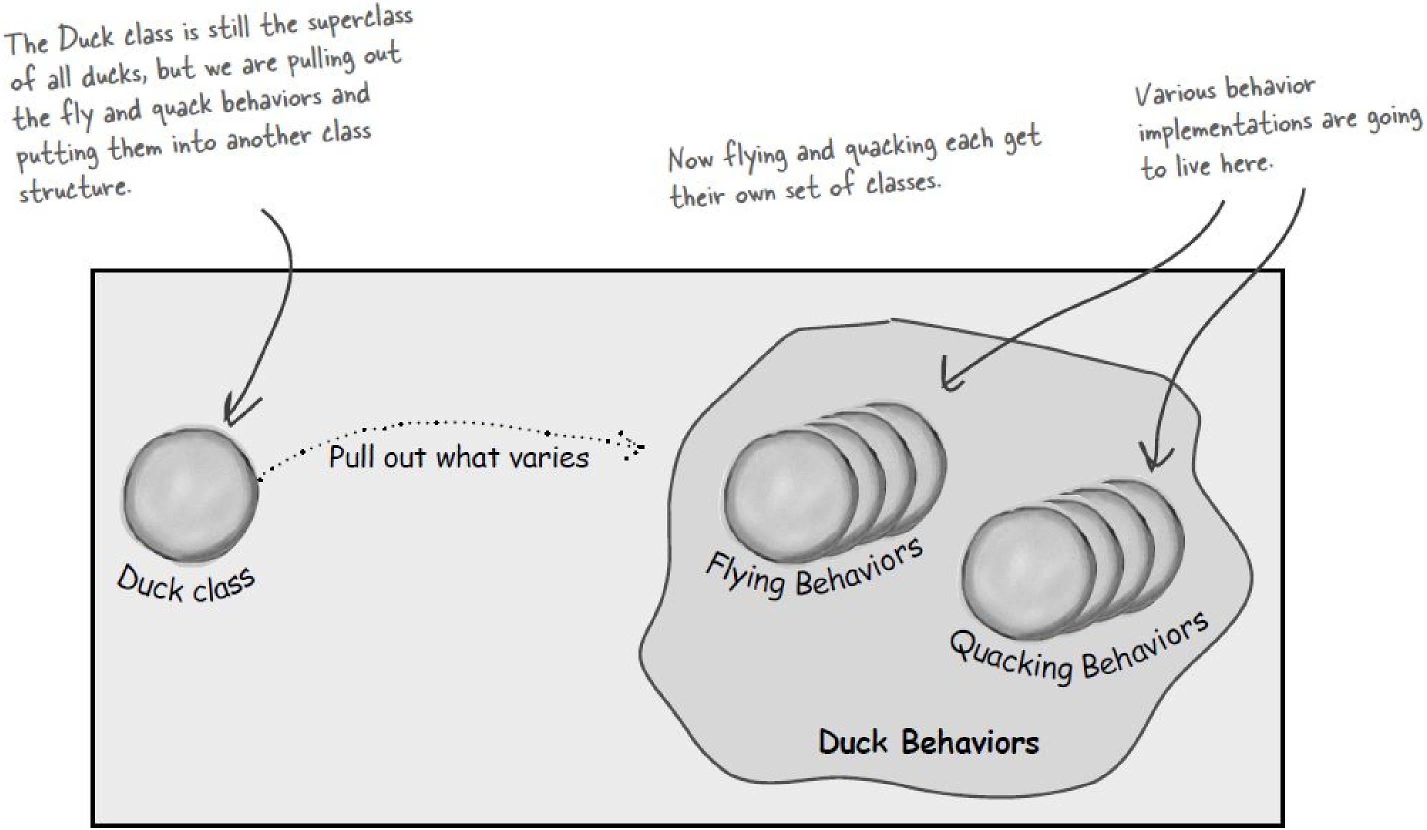
The result? Fewer unintended consequences from code changes and more flexibility in your systems!

- Okay, time to pull the duck behavior out of the Duck classes!

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.

Separating what changes from what stays the same



Designing the Duck Behaviors

- We'd like to keep things flexible
- We want to *assign* behaviors to the instances of Duck
- We want to provide changing the behavior of a duck dynamically
  - We should include behavior setter methods in the Duck classes

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

- We'll use an interface to represent each behavior – for instance, FlyBehavior and QuackBehavior
- Each implementation of a behavior will implement one of those interfaces

Designing the Duck Behaviors (cont.)

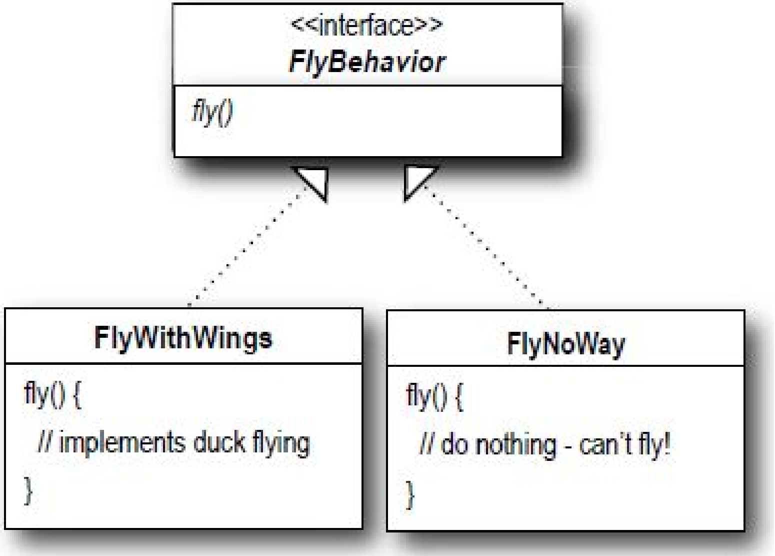
- So this time it won't be the *Duck* classes that will implement the flying and quacking interfaces.
- Instead, we'll make a set of classes whose entire reason is to represent a behavior (for example, "squeaking"), and it's the *behavior* class, that will implement the behavior interface.

Designing the Duck Behaviors (cont.)

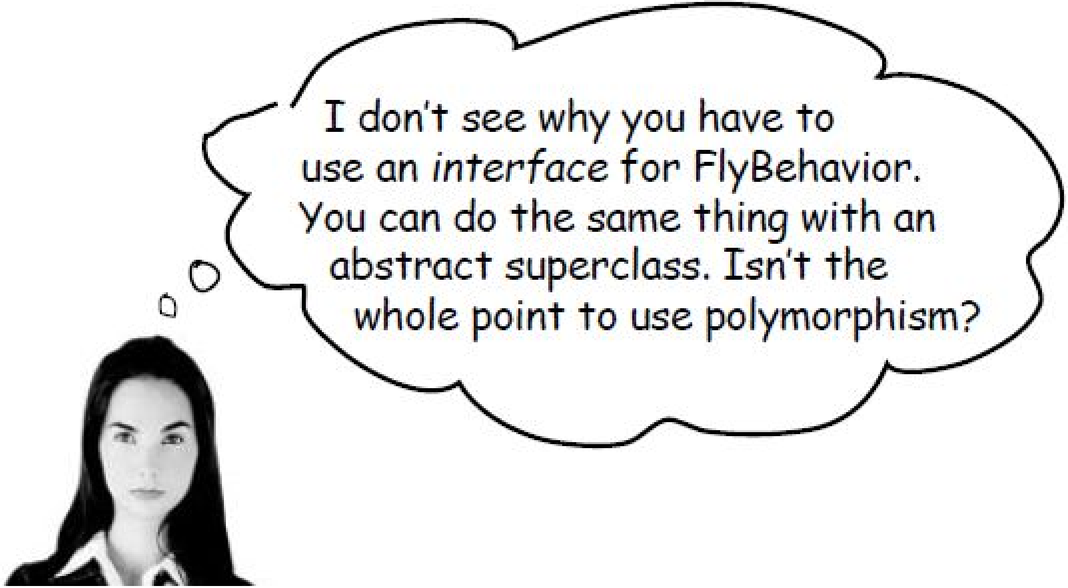
- In our previous design ideas, a behavior either came from
  - a concrete implementation in the superclass Duck,
  - or by providing a specialized implementation in the subclass itself.
- In both cases we were relying on an implementation.
- We were locked into using that specific implementation.
- There was no room for changing out the behavior (other than writing more code).

Designing the Duck Behaviors (cont.)

- With our new design the Duck subclasses will use a behavior represented by an interface (FlyBehavior and QuackBehavior),
  - the actual implementation of the behavior won't be locked into the Duck subclass.



A question

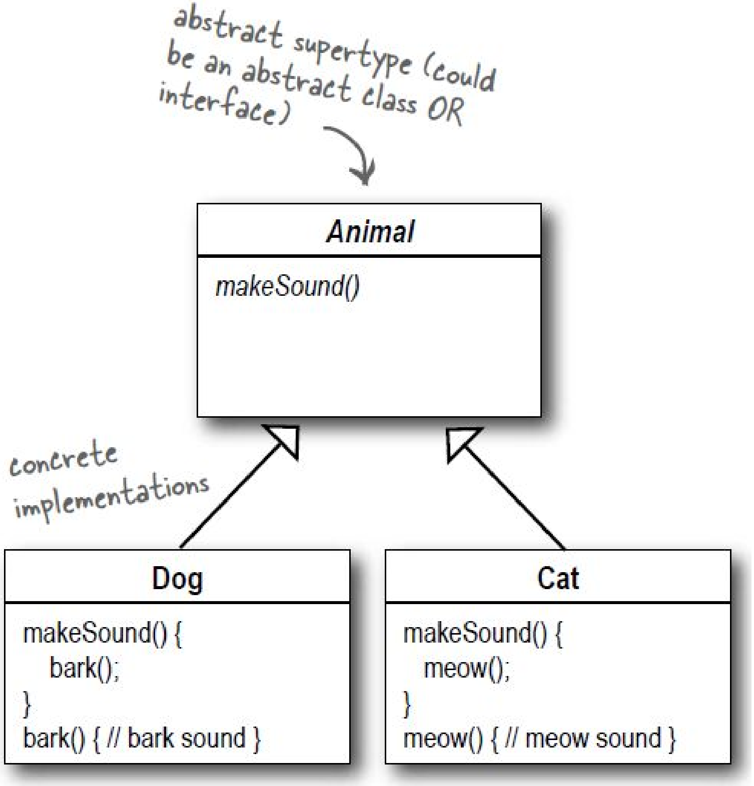

I don't see why you have to use an *interface* for FlyBehavior. You can do the same thing with an abstract superclass. Isn't the whole point to use polymorphism?

Program to an interface

- **“Program to an interface” really means “Program to a supertype.”**
  - The word interface is overloaded here.
  - There’s the concept of interface, but there’s also the Java construct interface.
  - You can program to an interface, without having to actually use a Java interface.
  - The point is to exploit polymorphism by programming to a supertype so that the actual runtime object isn’t locked into the code.
- “**Program to a supertype**” can be rephrased as “**the declared type of the variables should be a supertype usually an abstract class or interface**”
  - The class declaring the objects doesn’t have to know about the actual object types!”

Program to an interface (cont.)

- Imagine an abstract class Animal, with two concrete implementations, Dog and Cat.



Program to an interface (cont.)

- Programming to an implementation:

```
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.

- Programming to an interface:

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

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

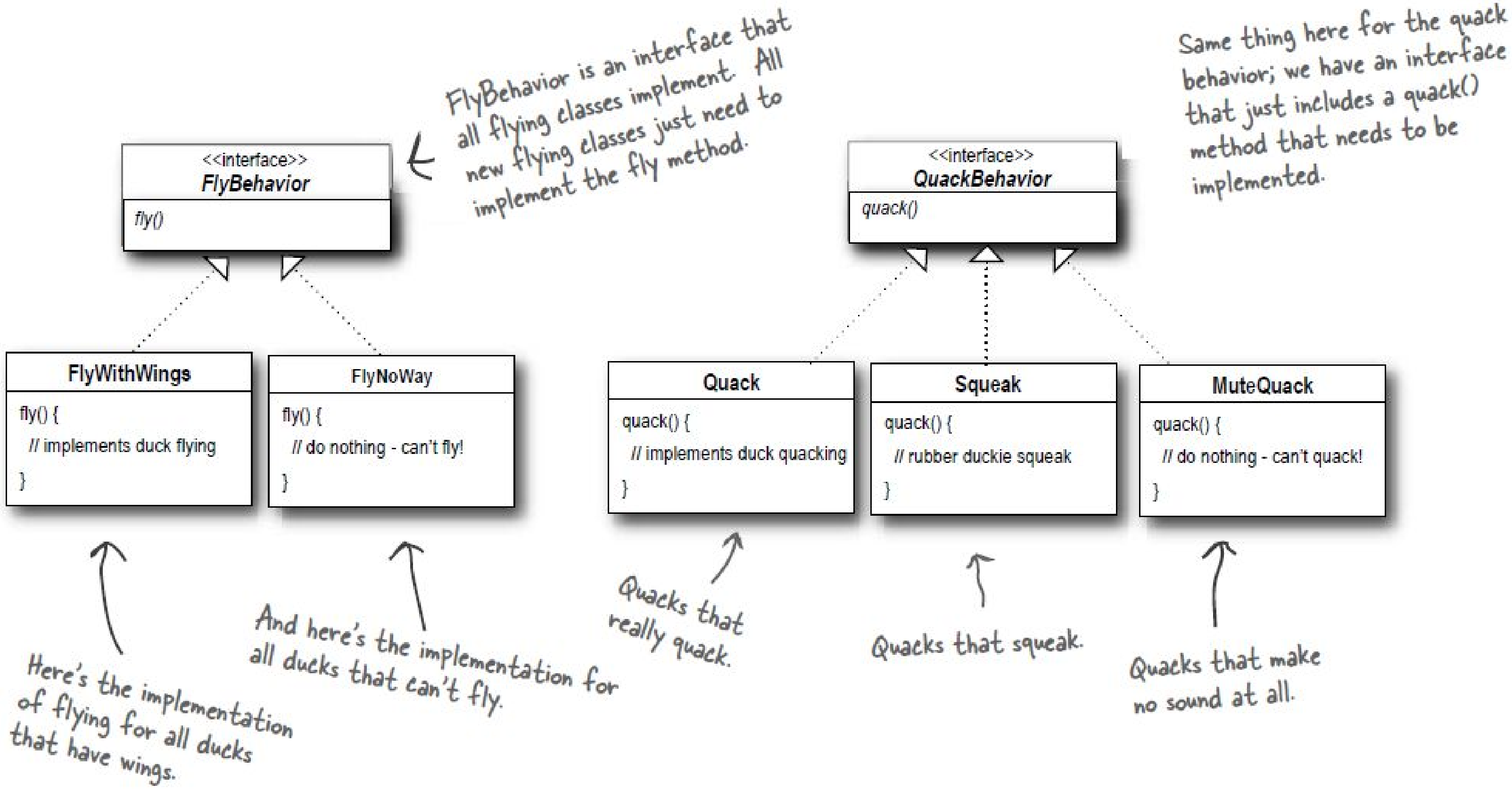
Program to an interface (cont.)

- Even better, rather than hard-coding 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



Implementing the Duck Behaviors (cont.)

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.

A question

Q: Should we make Duck an interface too?

A question

Q: Should we make Duck an interface too?

A: Not in this case. As you'll see once we've got everything hooked together, we do benefit by having Duck not be an interface and having specific ducks, like MallardDuck, inherit common properties and methods. Now that we've removed what varies from the Duck inheritance, we get the benefits of this structure without the problems.

Integrating the Duck Behavior

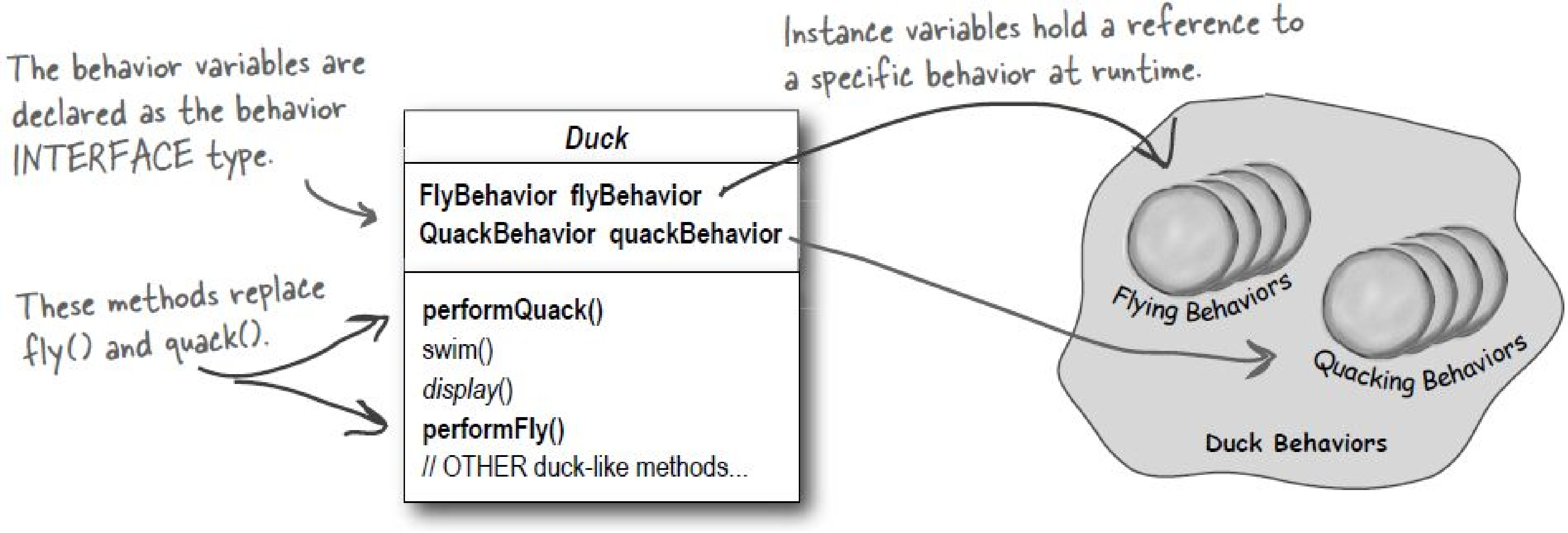
- The key is that a Duck will now delegate its flying and quacking behavior, instead of using quacking and flying methods defined in the Duck class (or subclass).
- How?

Integrating the Duck Behavior (cont.)

1. **First we'll add two instance variables** to the Duck class called ***flyBehavior*** and ***quackBehavior***, that are declared as the interface type.

- We'll also remove the fly() and quack() methods from the Duck class (and any subclasses)
- We'll replace fly() and quack() in the Duck class with two similar methods, called **performFly()** and **performQuack()**;

Integrating the Duck Behavior (cont.)



Integrating the Duck Behavior (cont.)

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.

- 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()*****!**

Integrating the Duck Behavior (cont.)

3. 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 quack-Behavior and flyBehavior instance variables from class Duck.

Testing the Duck code

- FlyBehavior.java (interface)
  - FlyNoWay.java
  - FlyWithWings.java
- QuackBehavior.java (interface)
  - Quack.java
  - Squeak.java
  - MuteQuack.java
- Duck.java (abstract class)
  - MallardDuck.java
- MiniDuckSimulator.java (test class)

Eclipse...

Setting behavior dynamically

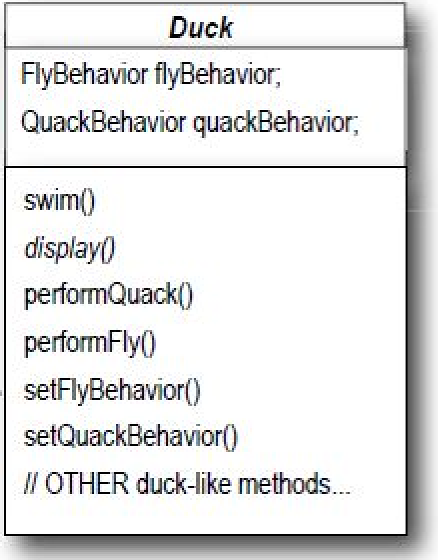
Imagine you want to set the duck's behavior type through a setter method on the duck subclass, rather than by instantiating it in the duck's constructor.

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

Setting behavior dynamically (cont.)

1. **Add two new methods to the Duck class:**

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

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



Setting behavior dynamically (cont.)

2. Make a new Duck type: ModelDuck

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

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

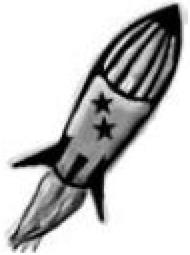
Our model duck begins life grounded... without a way to fly.

Setting behavior dynamically (cont.)

3. Make a new FlyBehavior type: FlyRocketPowered

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!”);
    }
}
```



Setting behavior dynamically (cont.)

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

5. Run it!

Setting behavior dynamically (cont.)

```
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.

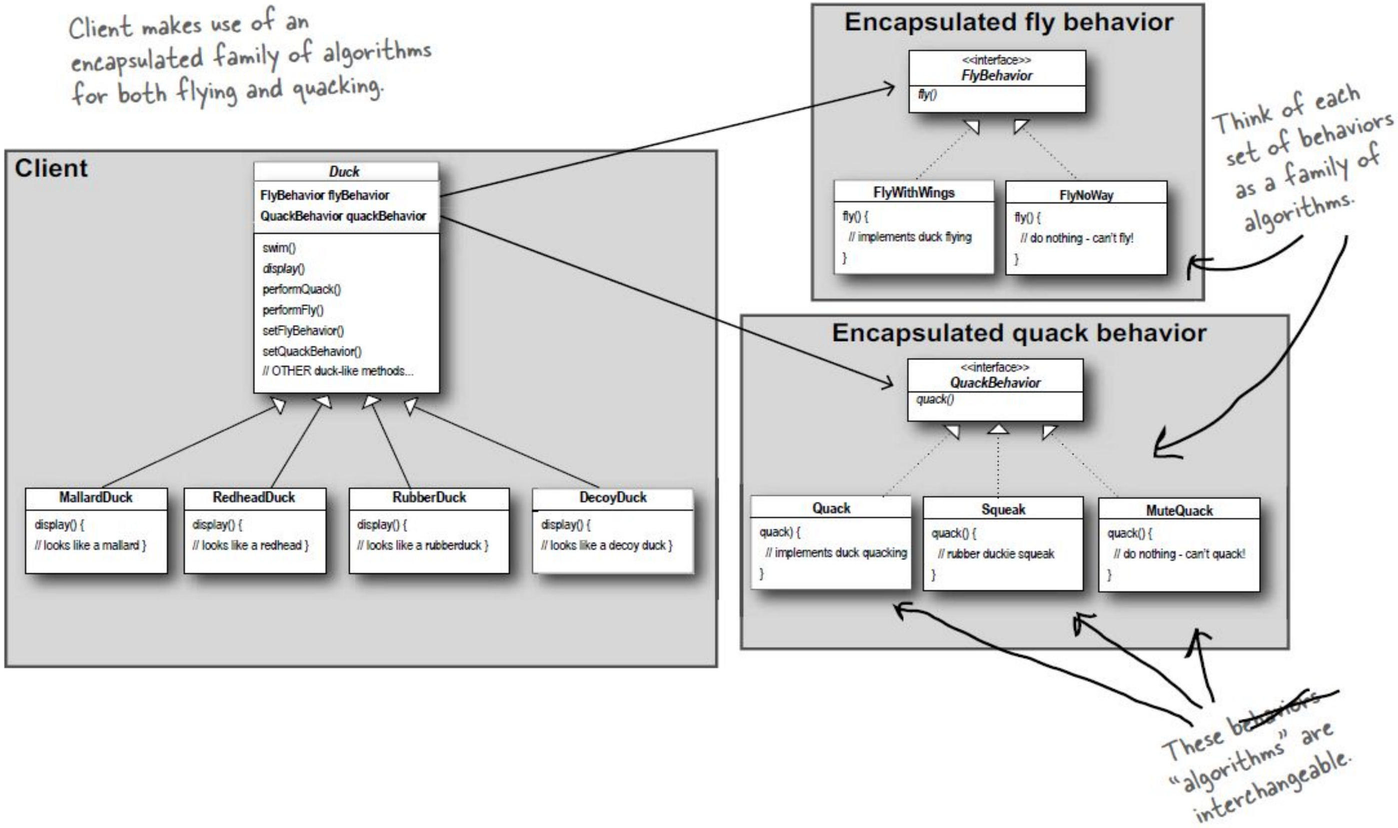
This invokes the model's inherited behavior setter method, and...voila! The model suddenly has rocket-powered flying capability!

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

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

after

The Big Picture on encapsulated behaviors



HAS-A can be better than IS-A

- The HAS-A relationship is an interesting one: each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking.
- When you put two classes together like this you're using ***composition***.
- Instead of *inheriting* their behavior, the ducks get their behavior by being composed with the right behavior object.
- This is an important technique.
- Design Principle: **Favor composition over inheritance.**

HAS-A can be better than IS-A (cont.)

- As you’ve seen, creating systems using composition
  - gives you a lot more flexibility
  - also lets you ***change behavior at runtime*** as long as the object you’re composing with implements the correct behavior interface
- Composition is used in many design patterns and you’ll see a lot more about its advantages and disadvantages throughout the course.

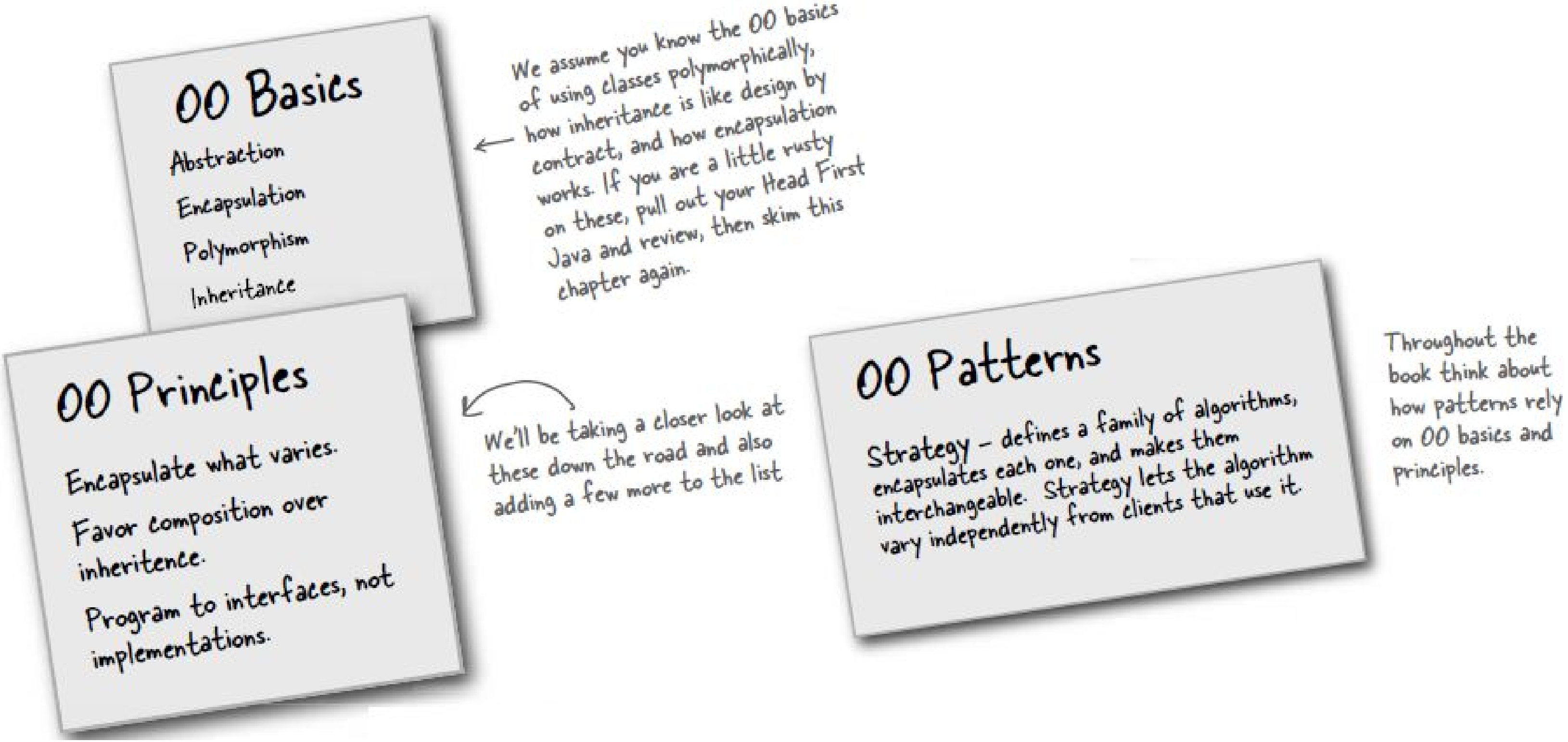
Strategy Pattern

- 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.

Strategy Pattern (cont.)

- You just applied your first design pattern: the STRATEGY pattern.
- You used the Strategy Pattern to rework the SimUDuck app.
- Thanks to this pattern, the simulator is ready for any changes.

Tools for your Design Toolbox



References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2004.