EECS3311 - Software Design

Strategy Design Pattern

Welcome to Design Patterns

(c) Head First Design Patterns chapter 1

- Someone has already solved your problem
- Exploit the wisdom and lessons learned by other developers who've been down the same design problem road and survived the trip.
- Instead of **code reuse**, with patterns you get **experience reuse**.

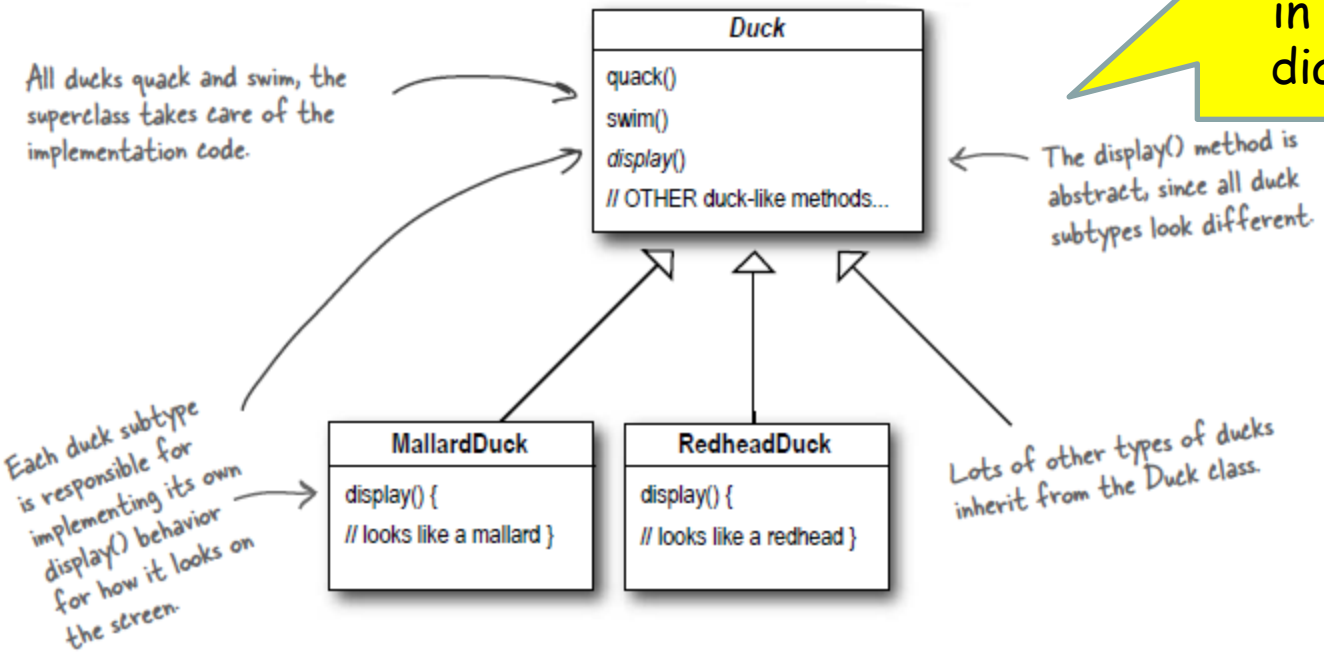
Joe's problem: SimUDuck

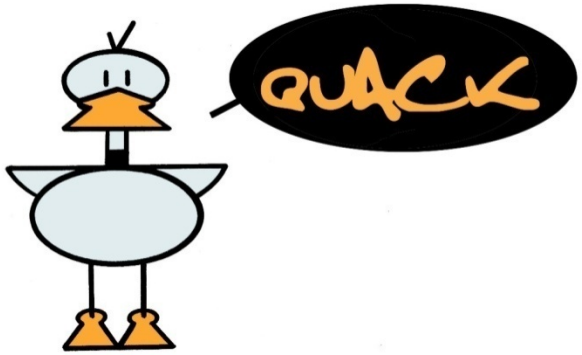
It started with a simple SimUDuck app

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.

superclass DUCK

How would this be represented in a BON diagram?



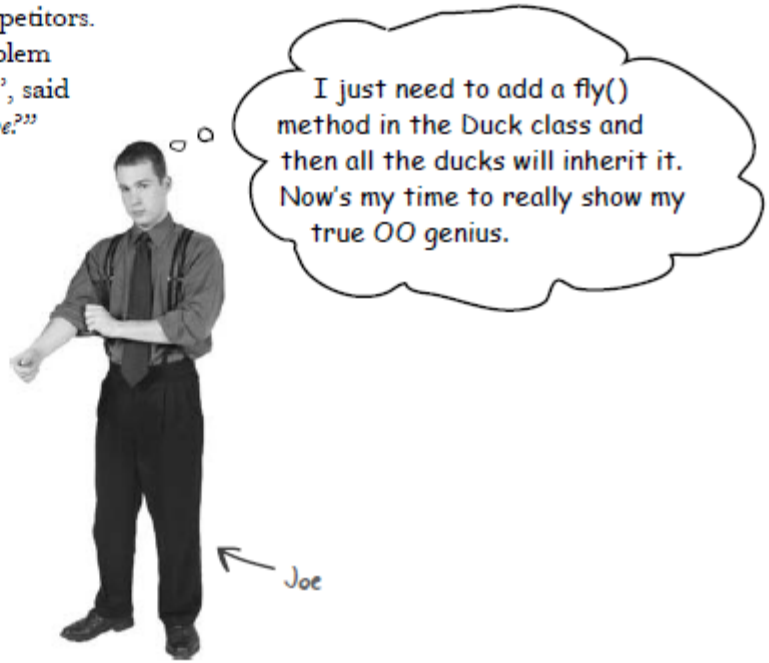


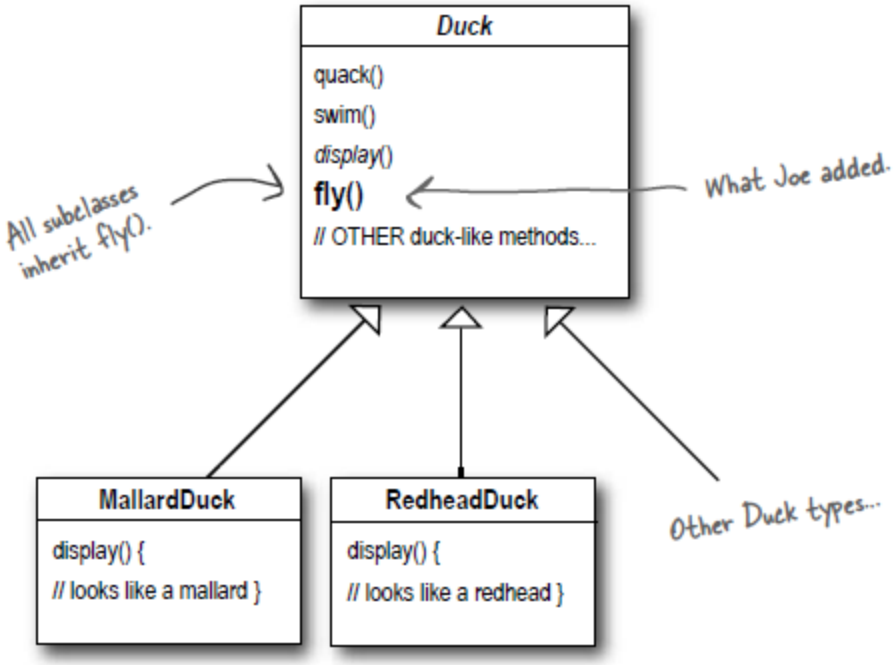
- In the last year, the company has been under increasing pressure from competitors.
- After a week long off-site brainstorming session over golf, the company executives think it's time for a big innovation.

- They need something *really impressive to show at the* upcoming shareholders meeting in Maui *next week.*

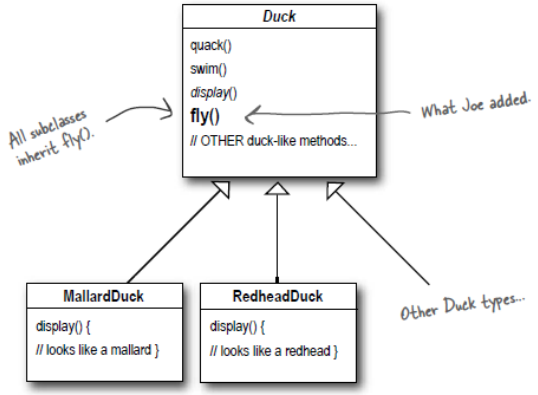
But now we need the ducks to FLY

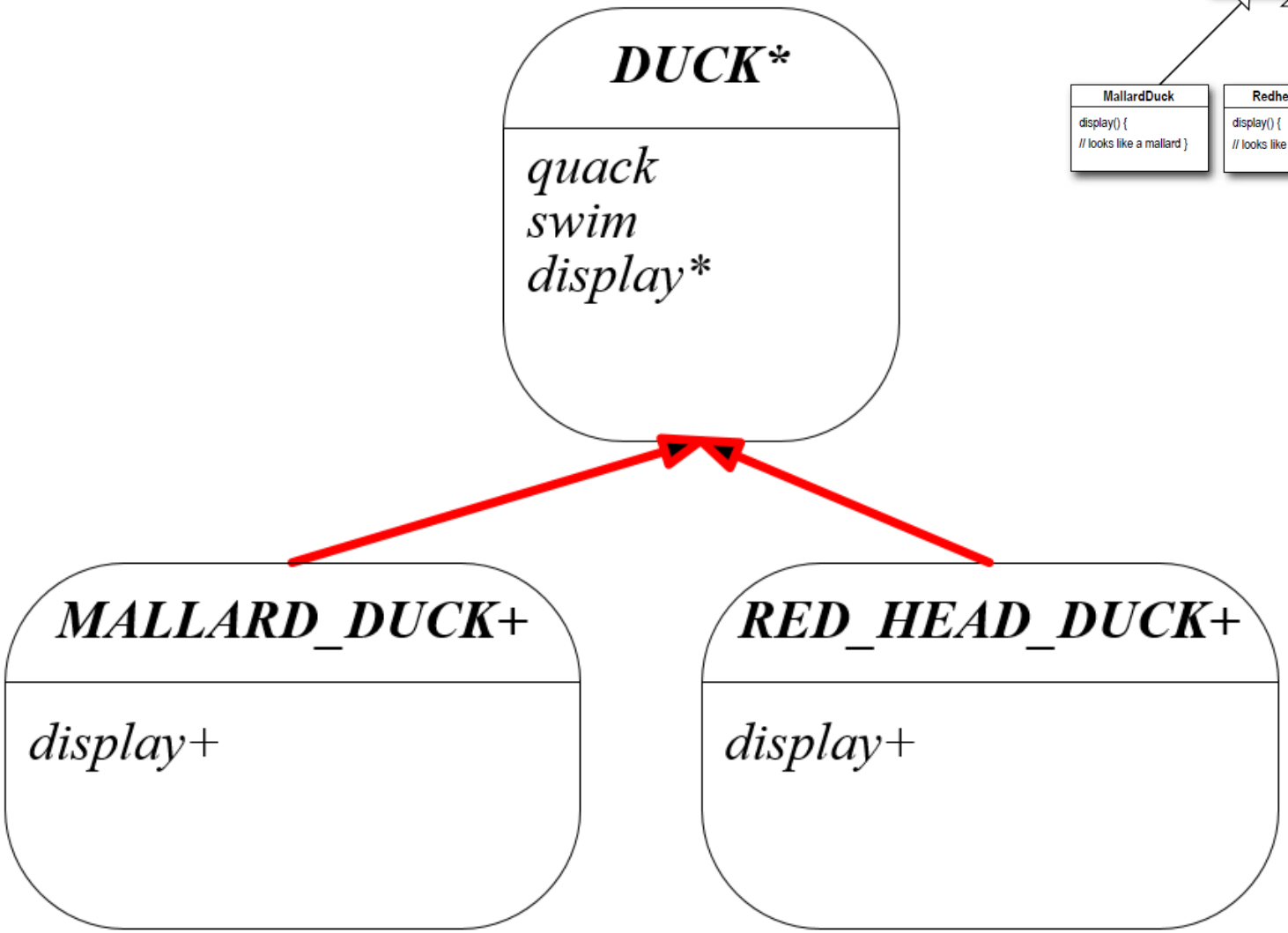
The executives decided that flying ducks is just what the simulator needs to blow away the other duck sim competitors. And of course Joe's manager told them it'll be no problem for Joe to just whip something up in a week. "After all", said Joe's boss, "he's an OO programmer... *how hard can it be?*"



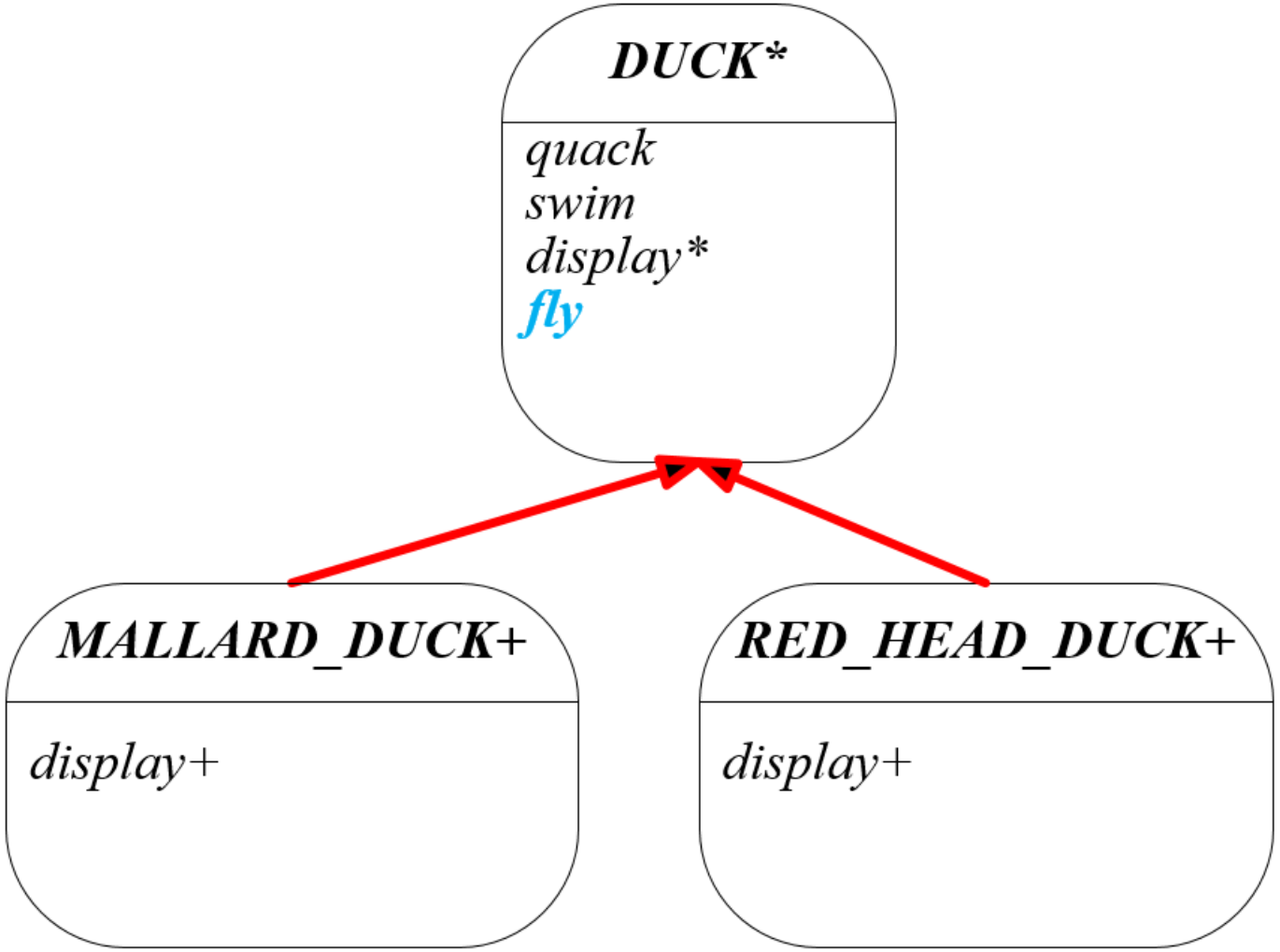

Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
All subclasses inherit fly().
What Joe added.
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
Other Duck types...

BON vs. UML

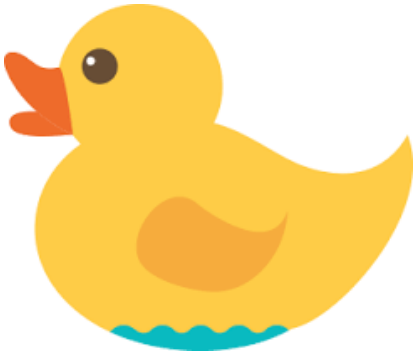

Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
All subclasses inherit fly().
What Joe added.
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
Other Duck types...

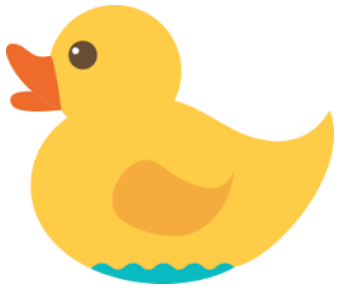

DUCK*
quack
swim
display*
MALLARD_DUCK+
display+
RED_HEAD_DUCK+
display+

- All ducks swim
- All ducks quack?
- All ducks fly?



But something went horribly wrong...



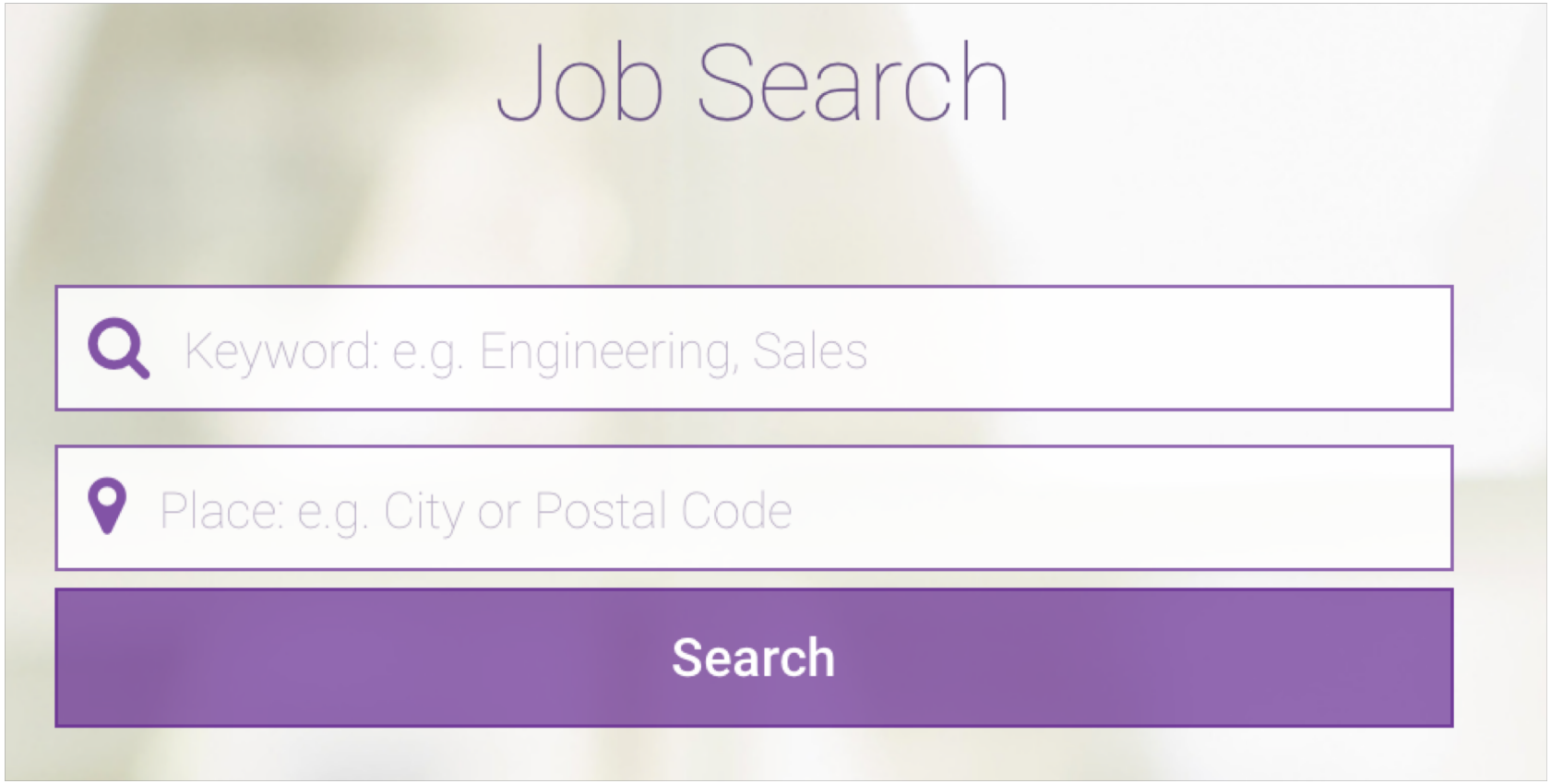


What happened?

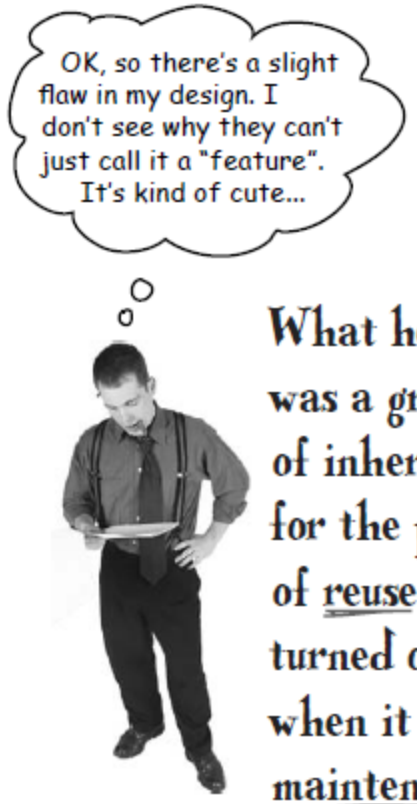
Joe failed to notice that not *all* subclasses of Duck should *fly*. When Joe added new behavior to the Duck superclass, he was also adding behavior that was *not* appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.

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

Joe searching Monster.com



But something went horribly wrong...

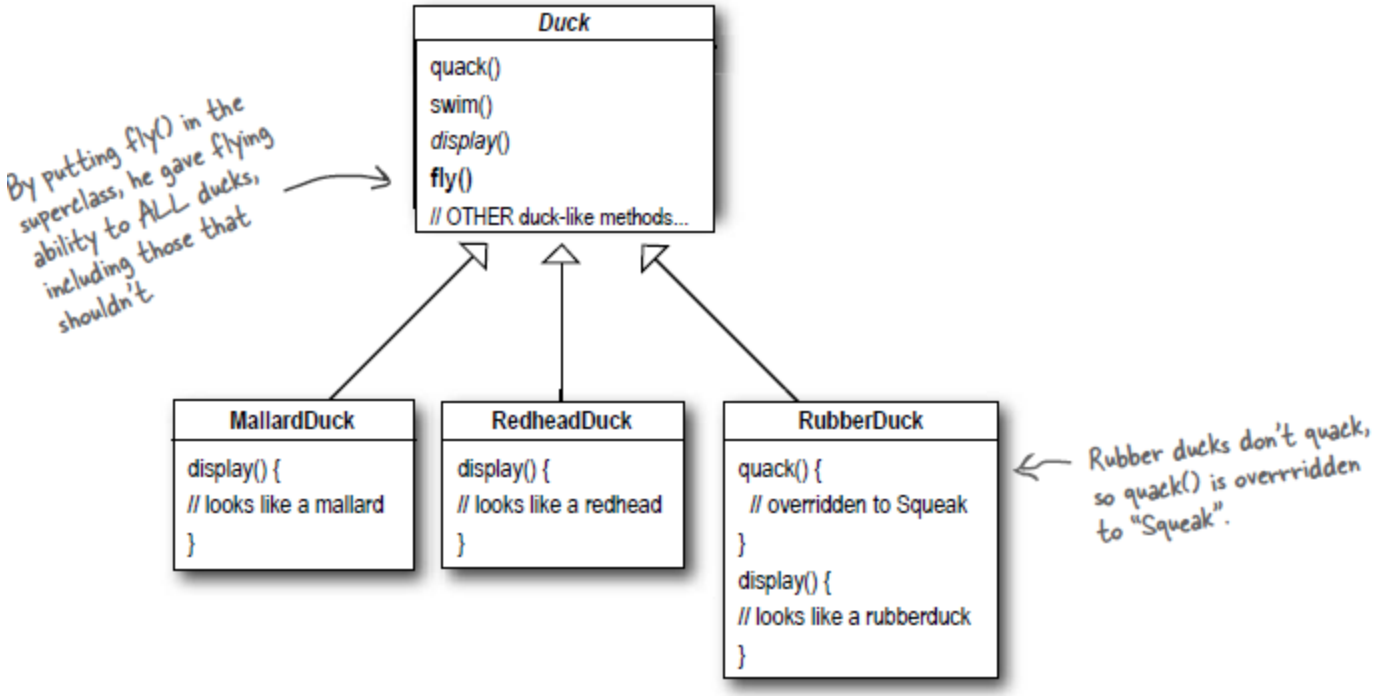


What happened?

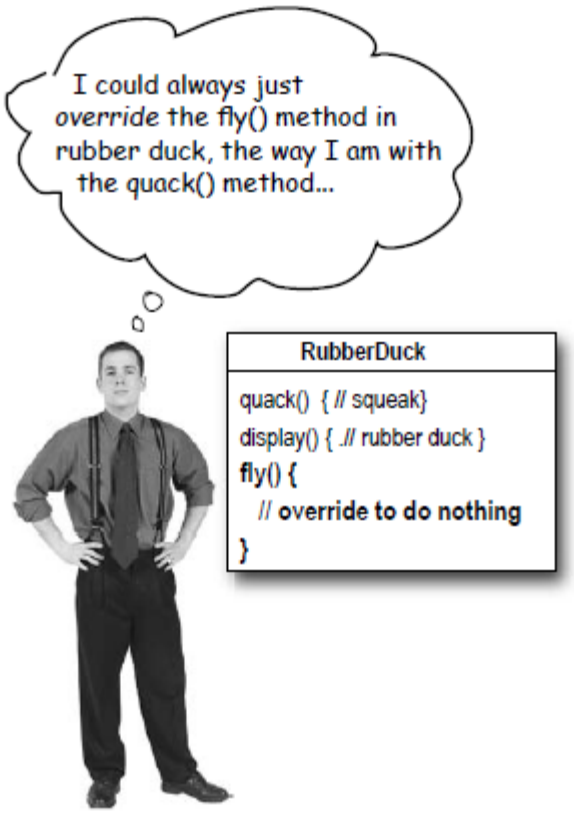
Joe failed to notice that not *all* subclasses of Duck should *fly*. When Joe added new behavior to the Duck superclass, he was also adding behavior that was *not* appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.

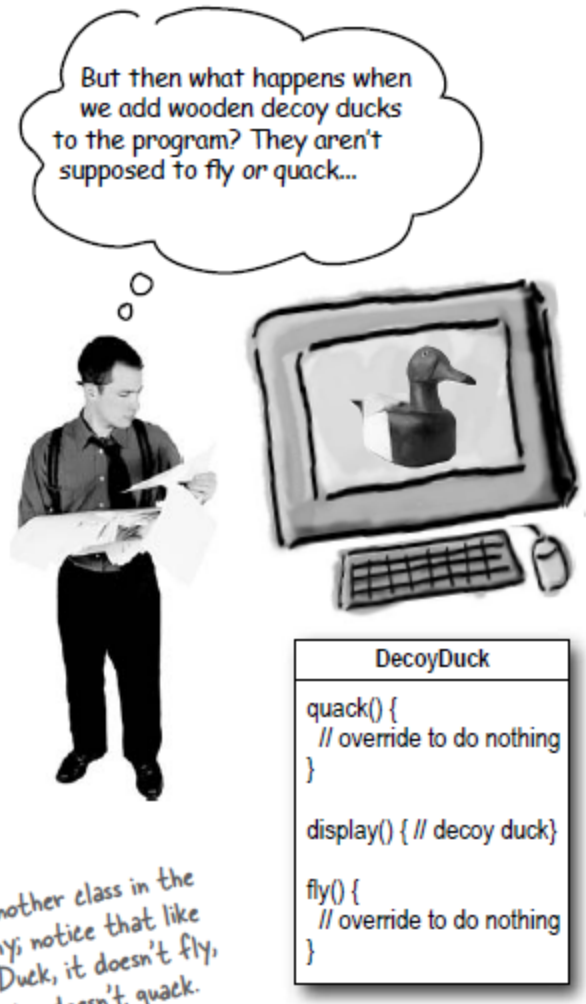
A localized update to the code caused a non-local side effect (flying rubber ducks)!

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.


By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't
Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
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 overrridden to "Squeak".

Joe thinks about inheritance...





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

Sharpen your pencil

Which of the following are disadvantages of using *inheritance* to provide Duck behavior? (Choose all that apply.)

- ❑ A. Code is duplicated across subclasses.
- ❑ B. Runtime behavior changes are difficult.
- ❑ C. We can't make ducks dance.
- ❑ D. Hard to gain knowledge of all duck behaviors.
- ❑ E. Ducks can't fly and quack at the same time.
- ❑ F. Changes can unintentionally affect other ducks.

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.*

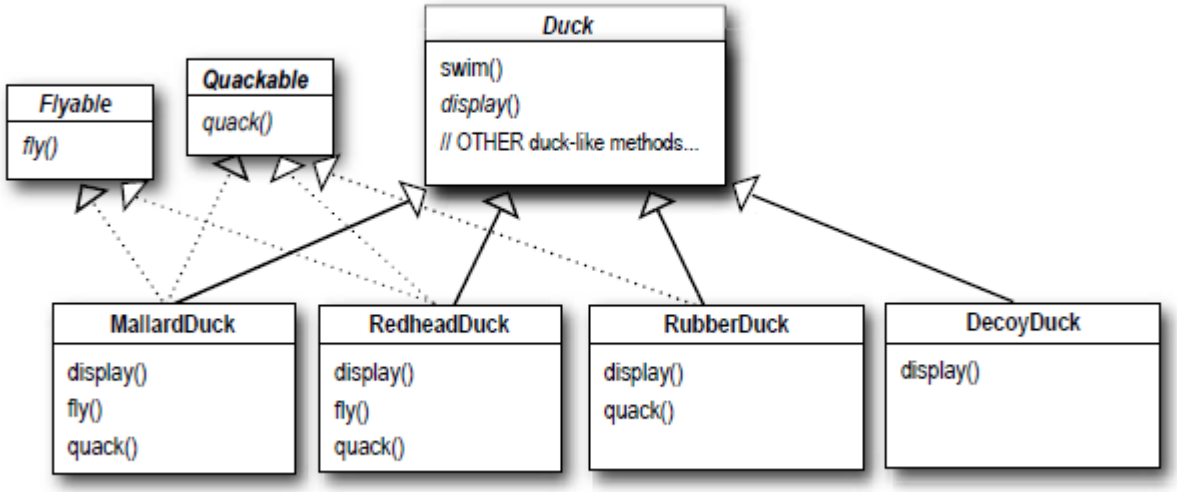
So, he needs a cleaner way to have only *some* (but not *all*) of the duck types fly or quack.

I could take the fly() out of the Duck superclass, and make a ***Flyable() interface*** with a fly() method. That way, only the ducks that are *supposed* to fly will implement that interface and have a fly() method... and I might as well make a Quackable, too, since not all ducks can quack.

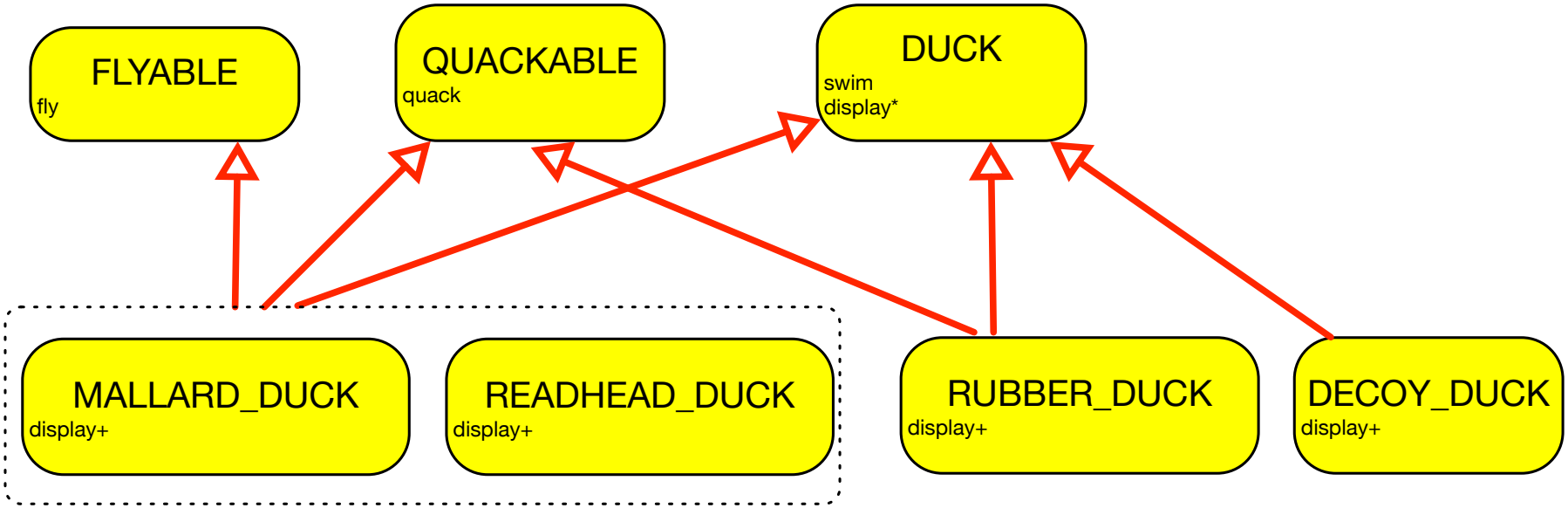
Java interface

What do you think about this design?

- We know that not all of the subclasses should have flying or quacking behavior, so inheritance isn't the right answer.
- But a Java interface has no implementation – so we have now destroyed code reuse
- What about true multiple inheritance (UML/Eiffel)?
  - Not a problem because we have full support for multiple inheritance



Multiple Imheritance



Note for design documents:
A BON diagram may present only some information relevant at that level of abstraction

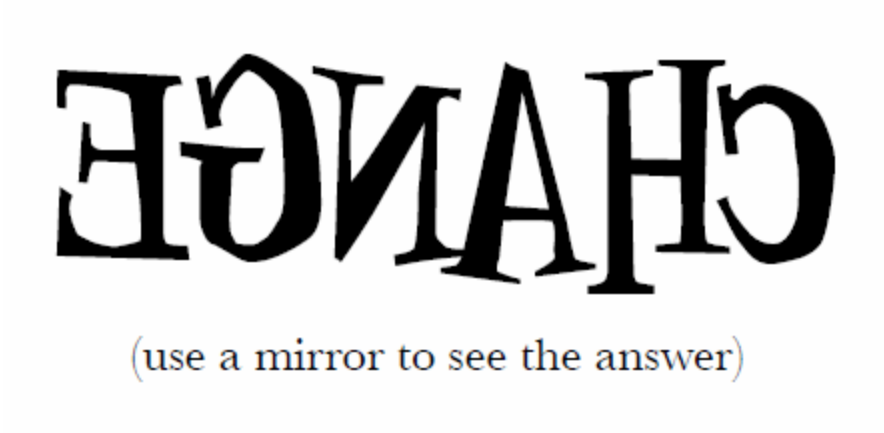
Problem: Find a design?

- Most ducks fly in the same way, but a few species of duck have different flyable behaviour, or perhaps they do not fly at all
- Most ducks quack in the same way, but a few species have a different quackable behaviour or they do not quack at all
- Must be able to change flyable and quackable behavior dynamically


Wouldn't it be dreamy if only there were a way to build software so that when we need to change it, we could do so with the least possible impact on the existing code? We could spend less time *reworking* code and more making the program do cooler things...

- Okay, what's the one thing you can always count on in software development?
- No matter where you work, what you're building, or what language you are programming in, what's the one true constant that will be with you always?

The one constant



- No matter how well you design an application, over time an application must grow and change or it will *die*.

- Inheritance hasn't worked out very well, since the duck behavior keeps changing across the subclasses, and it's not appropriate for all subclasses to have those behaviors.

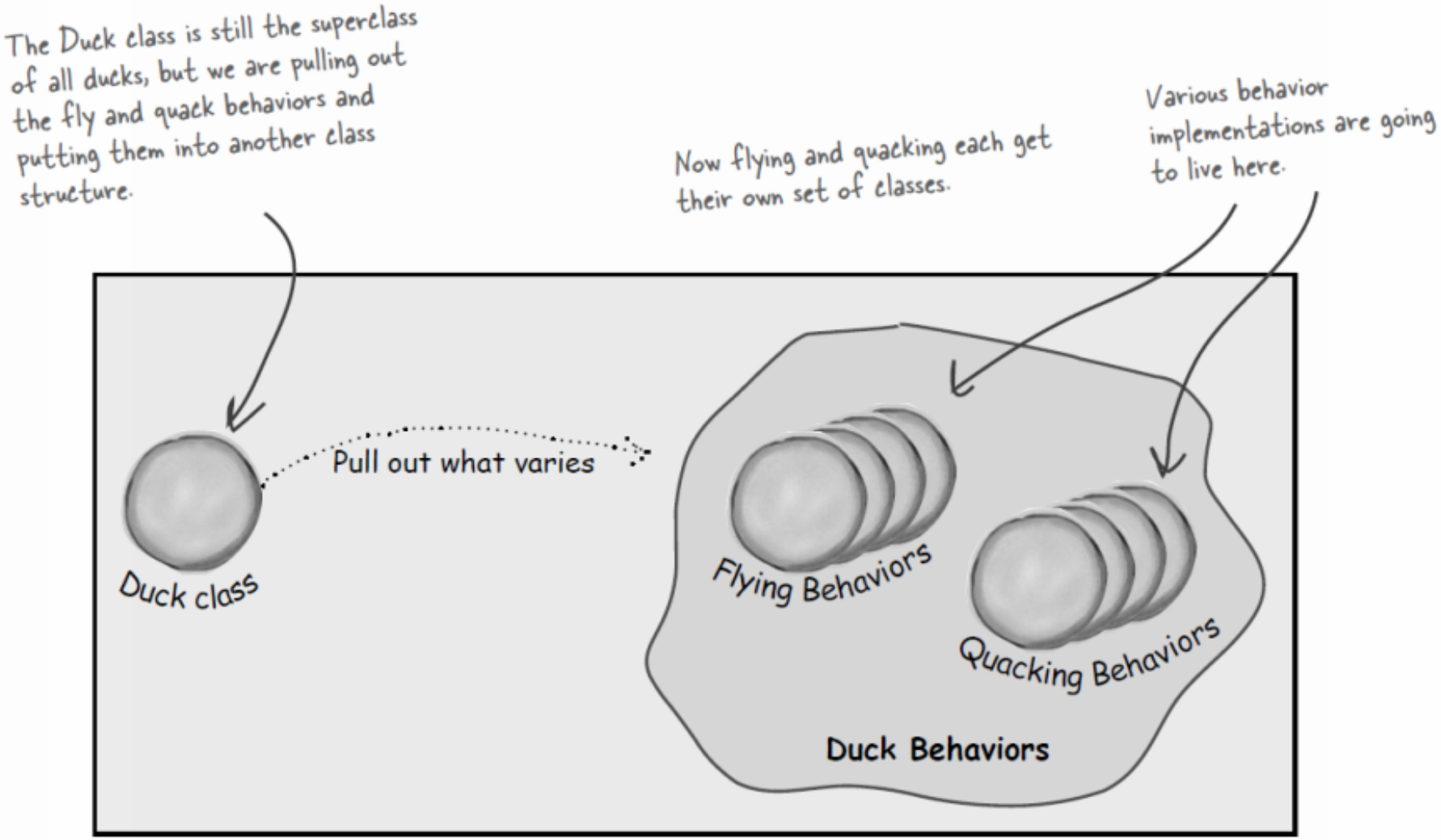
- The Flyable and Quackable [Java] interface sounded promising at first—only ducks that really do fly will be Flyable, etc.—except Java interfaces have no implementation code, so no code reuse.

- And 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, probably introducing new bugs along the way!

Design Principle

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

- Information Hiding: 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

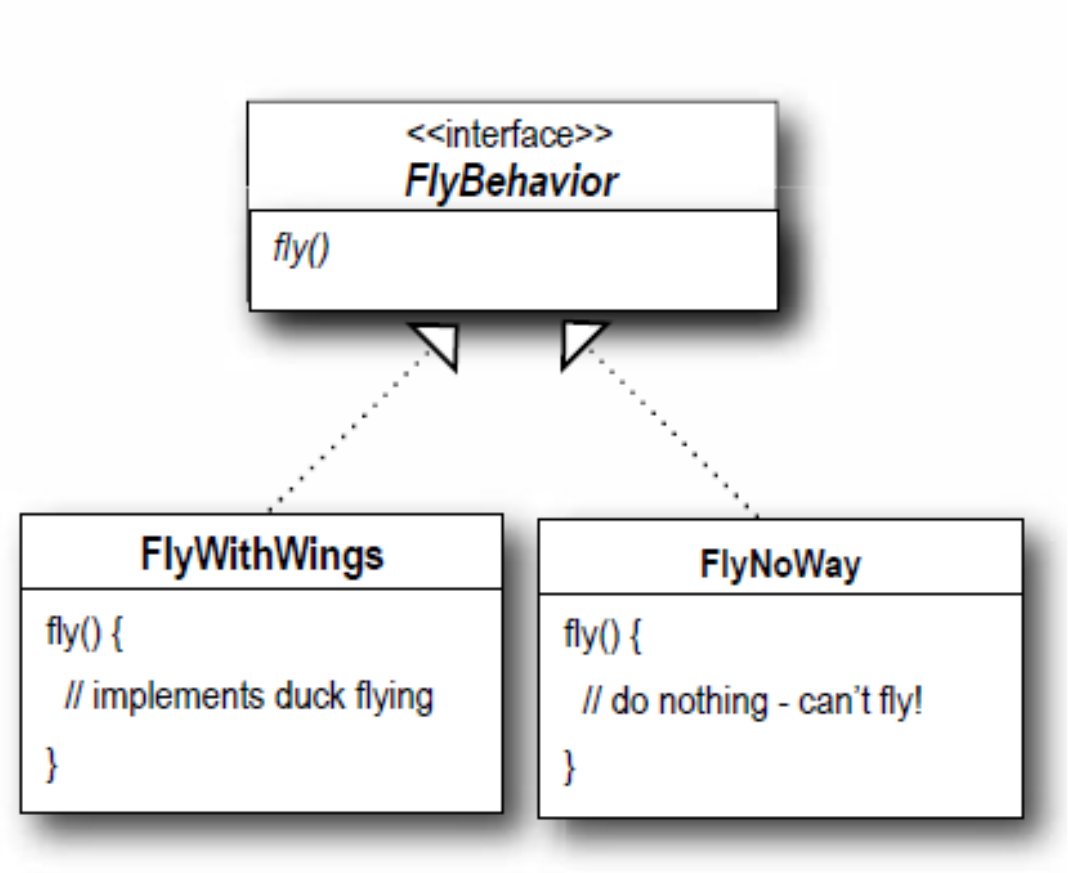

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.
Now flying and quacking each get their own set of classes.
Various behavior implementations are going to live here.
Pull out what varies
Duck class
Flying Behaviors
Quacking Behaviors
Duck Behaviors

- We'd like to keep things flexible. It was the inflexibility in the duck behaviors that got us into trouble in the first place.

- And we know that we want to assign behaviors to the instances of Duck.
  - For example, we might want to instantiate a new MallardDuck instance and initialize it with a specific type of flying behavior.
  - And while we're there, why not make sure that we can change the behavior of a duck dynamically?

Design Principle

Program to an interface not to an implementation



- Duck behaviours (e.g. flying, quacking) will live in a class that is separate from class DUCK.
- DUCK does not need to know how those behaviours are implemented

Not a Java "interface"

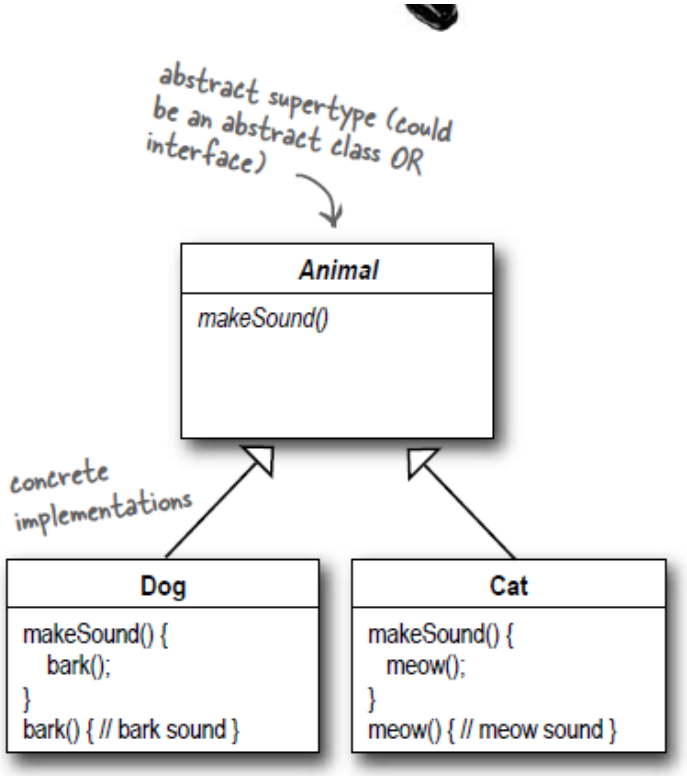
- The word "interface" is overloaded
- Here we do not mean a Java interface – rather we mean the concept of an interface
  - Features and their signatures
  - Meaningful comment
  - Pre/Post conditions to specify the features
  - Invariants to specify the safety and consistency of the data and business logic

Not a Java "interface"

- Programming to an implementation
  - list: LINKED_LIST[PERSON]

- Programming to an interface (or abstract type)
  - list: LIST[PERSON]
  - create {LINKED_LIST[PERSON]}list.make

In Java also
(programming to an interface gets us polymorphism)



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();
```

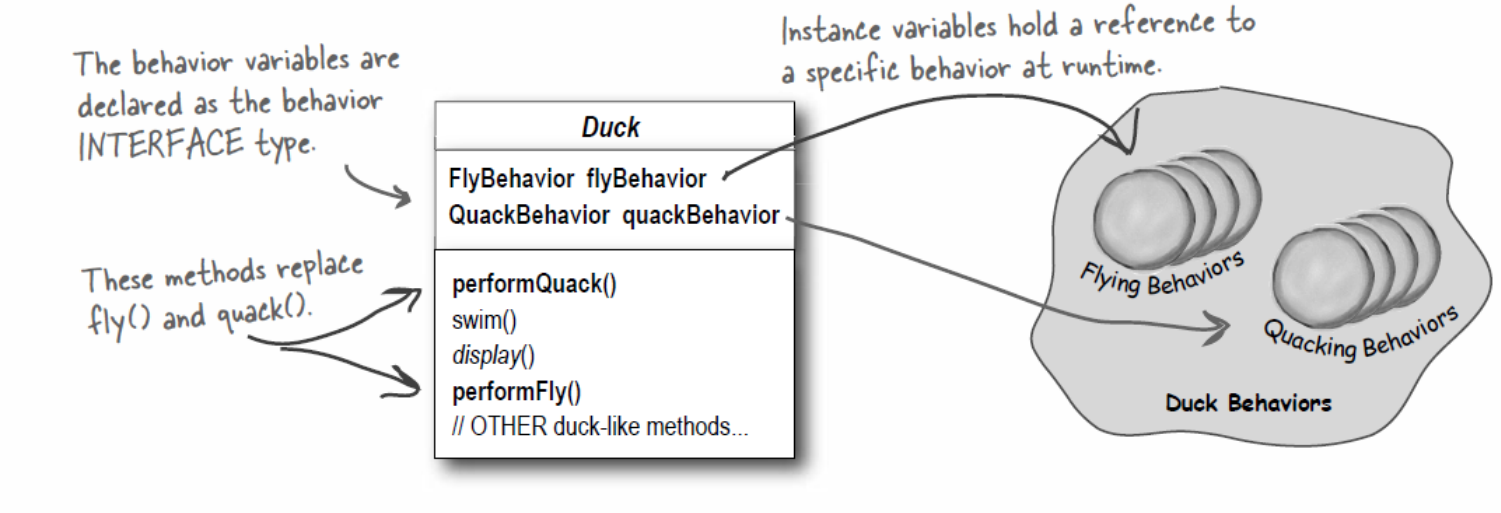
We know it's a Dog, but we can now use the animal reference polymorphically.

Even better, rather than 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().

Delegation

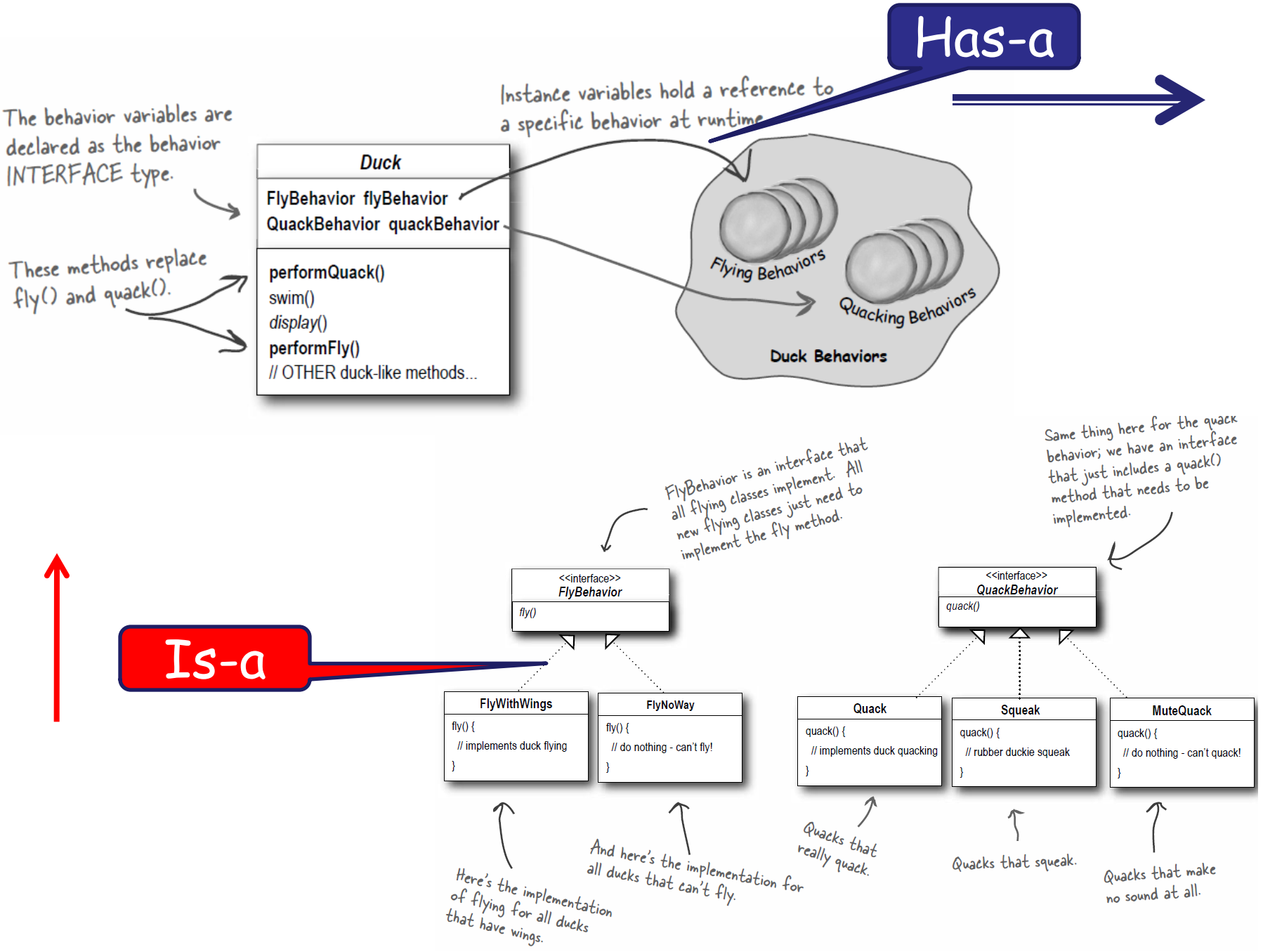


Now we implement performQuack():

```
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.


Has-a
The behavior variables are declared as the behavior INTERFACE type.
Instance variables hold a reference to a specific behavior at runtime.
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
performQuack()
swim()
display()
performFly()
// OTHER duck-like methods...
These methods replace fly() and quack().
Flying Behaviors
Quacking Behaviors
Duck Behaviors
Is-a
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.
<<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 for all ducks that can't fly.
Quacks that really quack.
Quacks that squeak.
Quacks that make no sound at all.

Question?

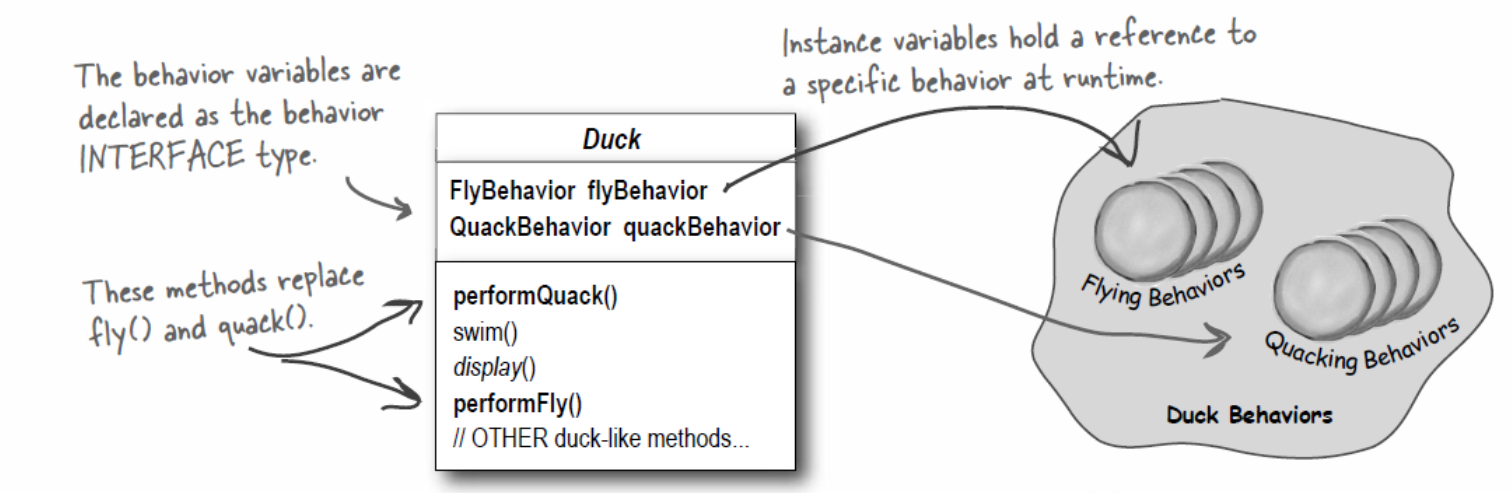
- Should we make DUCK a Java <<interface>> as well?
- Not in this case.
  - Duck can be abstract and some descendants like MALLARD_DUCK inherit common properties and methods like swim.
  - Now that we've removed what varies from the Duck inheritance, we get the benefits of this structure without the problems.

Question?

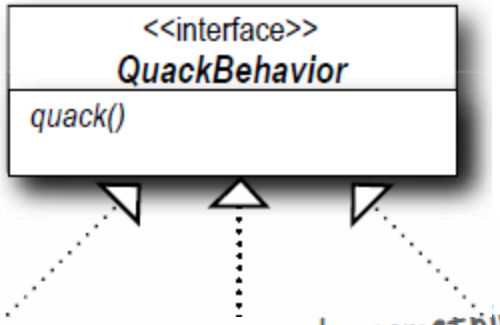
- Using our new design, what would you do if you needed to add rocket-powered flying to the SimUDuck app?

```
class
    FLY_ROCKET_POWERED
inherit
    FLY_BEHAVIOUR
```

Delegation



Program to an interface not to an implementation



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.

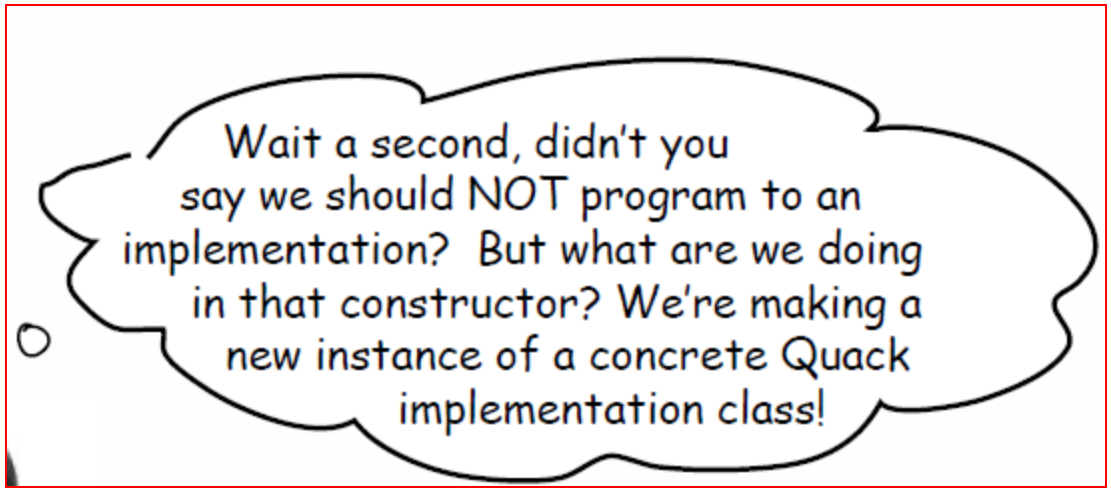
MALLARD_DUCK?

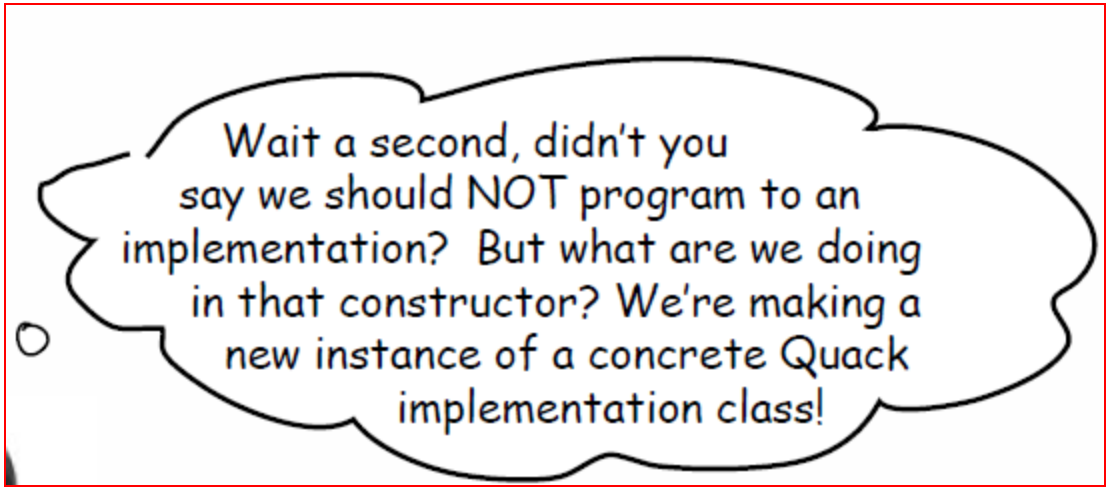
```
public class MallardDuck extends Duck {

    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.





- Good catch, that's exactly what we're doing... for now. [Other creator patterns can help]
- Still, notice that while we are setting the behaviors to concrete classes Quack or FlyWithWings, but we are assigning it to our behavior reference variable (which is abstract), and that could easily change at runtime.
- How? [see in a few slides]

class DUCK

```
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.

```
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).

```
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).

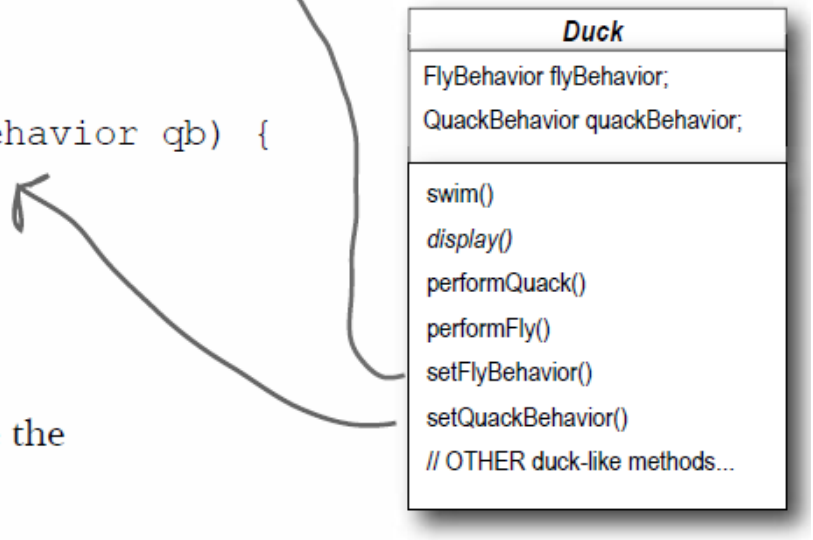
Then we do the same thing with MallardDuck's inherited performFly() method.

5 Run the code!

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

Setting behavior dynamically

```
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

class 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.

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();
    }
}
```

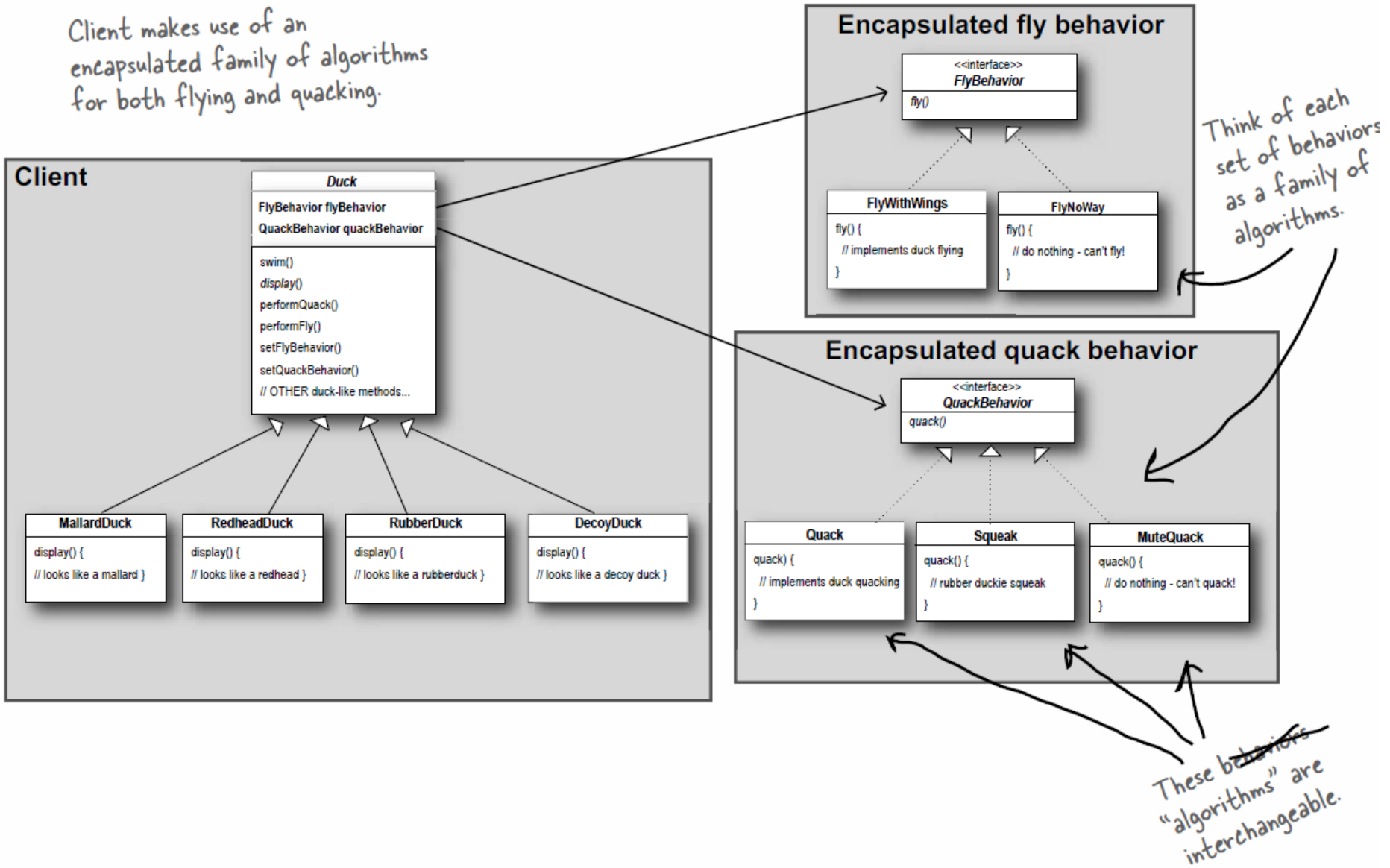
The first call to performFly() delegates to the flyBehavior object set in the ModelDuck's constructor, which is a FlyNoWay instance.

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
```

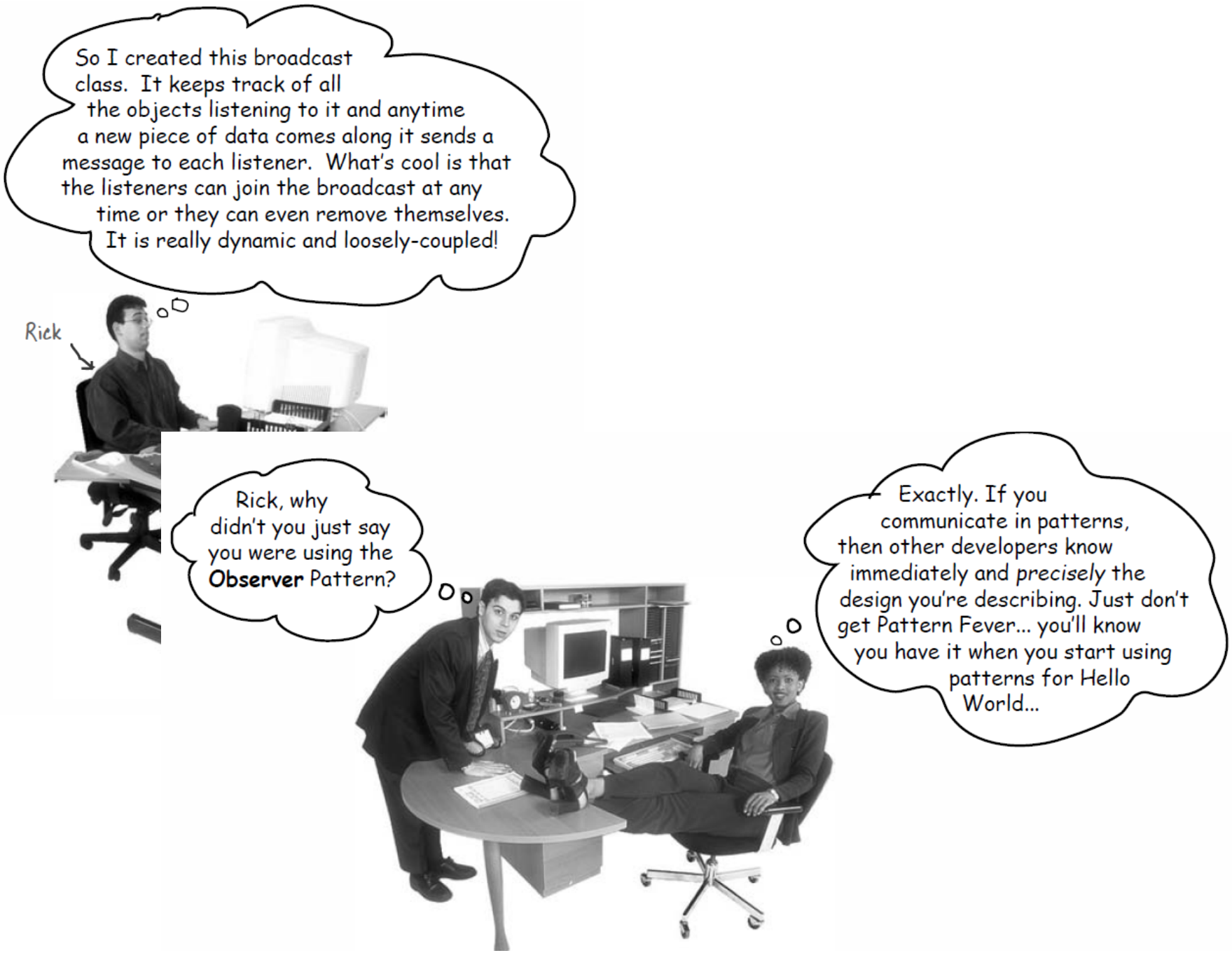

Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a 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.

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.

- use this definition when you want impress friends & influence key executives

- Patterns give developers a shared vocabulary as well as a shared code experience


So I created this broadcast class. It keeps track of all the objects listening to it and anytime a new piece of data comes along it sends a message to each listener. What's cool is that the listeners can join the broadcast at any time or they can even remove themselves. It is really dynamic and loosely-coupled!
Rick
Rick, why didn't you just say you were using the **Observer** Pattern?
Exactly. If you communicate in patterns, then other developers know immediately and *precisely* the design you're describing. Just don't get Pattern Fever... you'll know you have it when you start using patterns for Hello World...

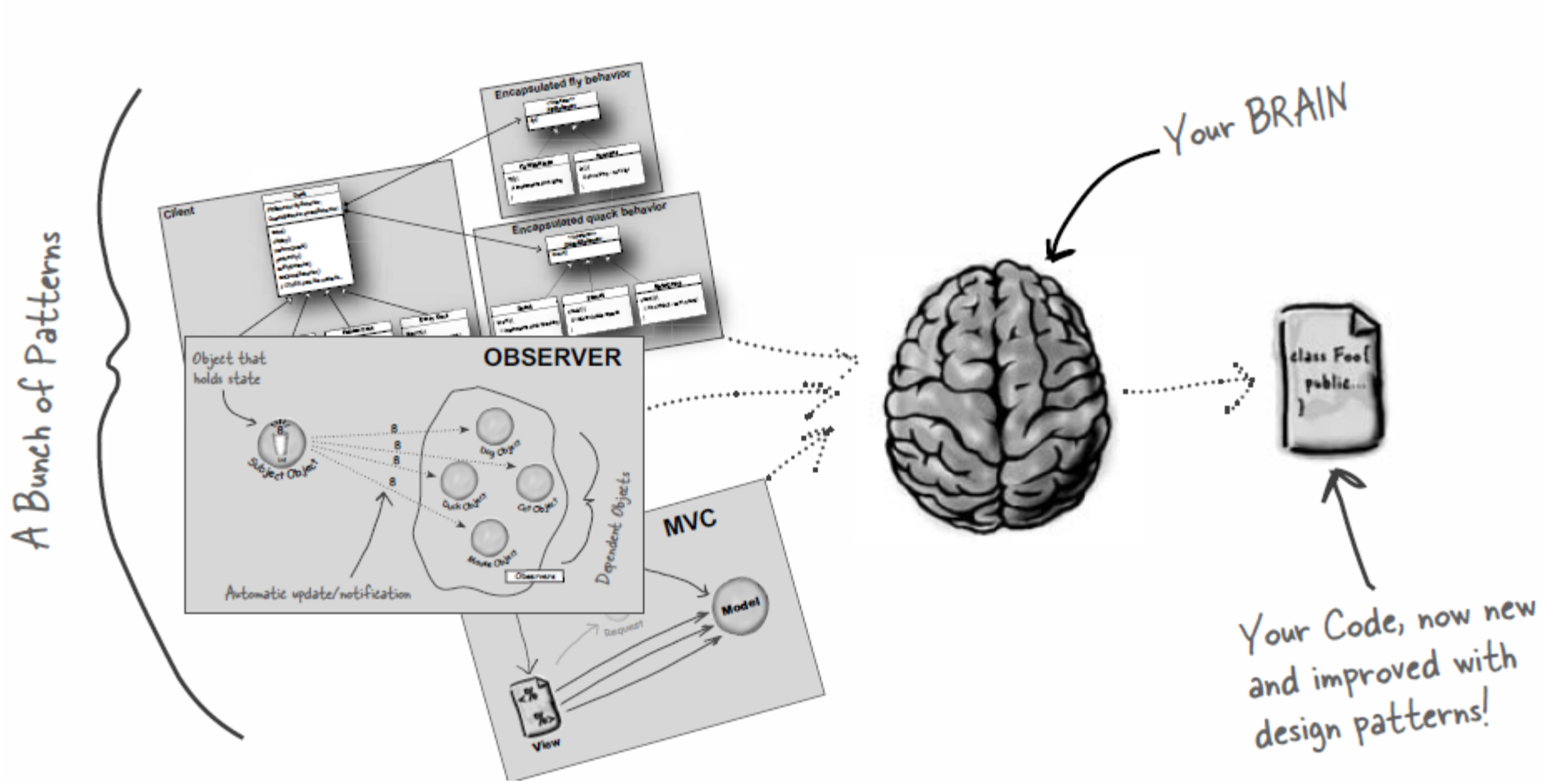
Shared pattern vocabularies are POWERFUL.
When you communicate with another developer or your team using patterns, you are communicating not just a pattern name but a whole set of qualities, characteristics and constraints that the pattern represents.

"We're using the strategy pattern to implement the various behaviors of our ducks." This tells you the duck behavior has been encapsulated into its own set of classes that can be easily expanded and changed, even at runtime if needed.

Patterns allow you to say more with less. When you use a pattern in a description, other developers quickly know precisely the design you have in mind.

Talking at the pattern level allows you to stay "in the design" longer. Talking about software systems using patterns allows you to keep the discussion at the design level, without having to dive down to the nitty gritty details of implementing objects and classes.

How many design meetings have you been in that quickly degrade into implementation details?


A Bunch of Patterns
Encapsulated fly behavior
Client
Encapsulated quack behavior
OBSERVER
Object that holds state
Automatic update/notification
Dependent Objects
MVC
Model
View
Your BRAIN
class Foo{
public...
}
Your Code, now new and improved with design patterns!

Developer: Okay, hmm, but isn't this all just good object-oriented design; I mean as long as I follow encapsulation and I know about abstraction, inheritance, and polymorphism, do I really need to think about Design Patterns? Isn't it pretty straightforward? Isn't this why I took all those OO courses? I think Design Patterns are useful for people who don't know good OO design.

Guru: Ah, this is one of the true misunderstandings of object-oriented development: that by knowing the OO basics we are automatically going to be good at building flexible, reusable, and maintainable systems.

Developer: No?

Guru: No. As it turns out, constructing OO systems that have these properties is not always obvious and has been discovered only through hard work.

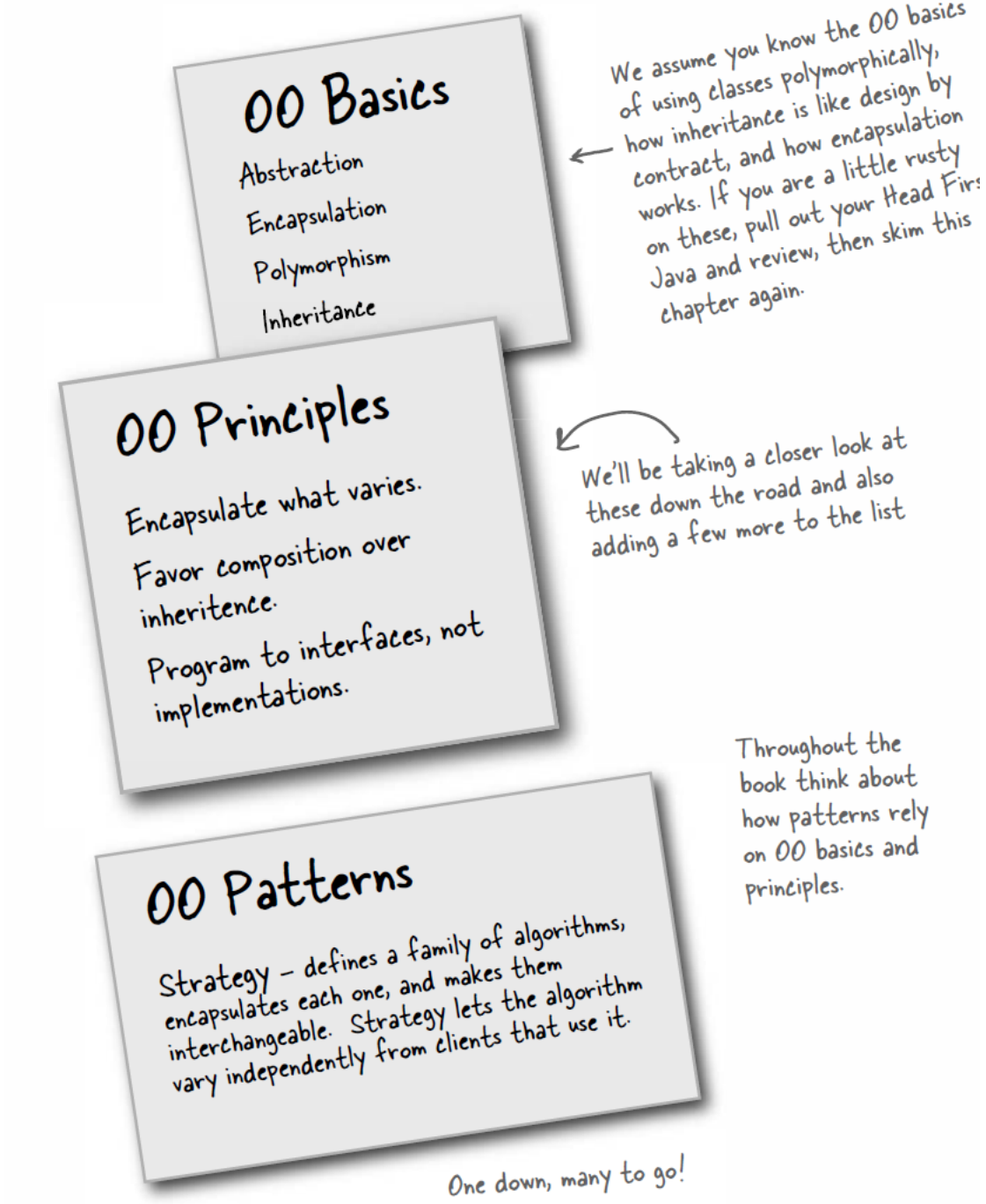
Developer: I think I'm starting to get it. These, sometimes non-obvious, ways of constructing object-oriented systems have been collected...

Guru: ...yes, into a set of patterns called Design Patterns.

Developer: So, by knowing patterns, I can skip the hard work and jump straight to designs that always work?

Guru: Yes, to an extent, but remember, design is an art. There will always be tradeoffs. But, if you follow well thought-out and time-tested design patterns, you'll be way ahead.

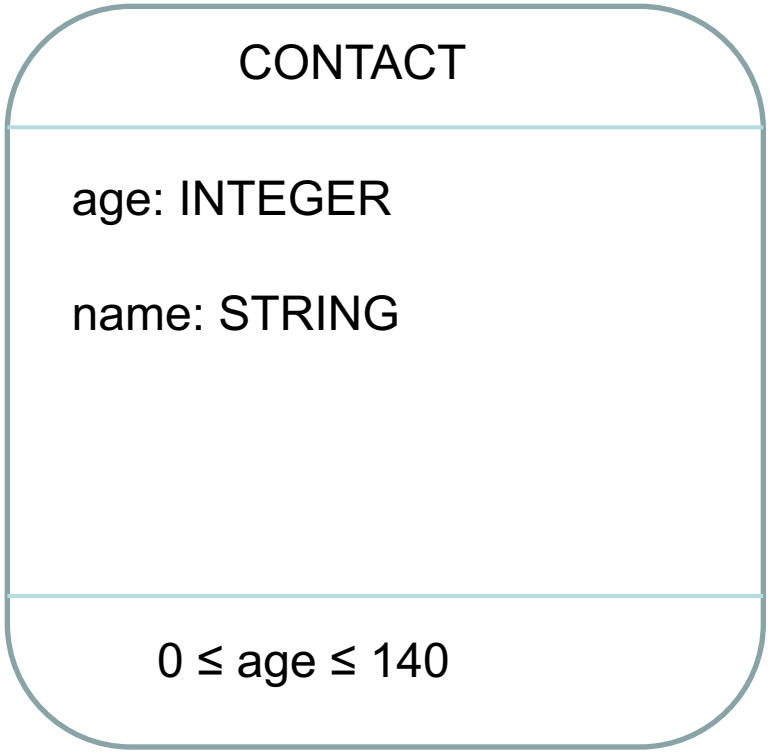
Developer: What do I do if I can't find a pattern?


OO Basics
Abstraction
Encapsulation
Polymorphism
Inheritance
We assume you know the OO basics of using classes polymorphically, how inheritance is like design by contract, and how encapsulation works. If you are a little rusty on these, pull out your Head First Java and review, then skim this chapter again.
OO Principles
Encapsulate what varies.
Favor composition over inheritence.
Program to interfaces, not implementations.
We'll be taking a closer look at these down the road and also adding a few more to the list
OO Patterns
Strategy – defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Throughout the book think about how patterns rely on OO basics and principles.
One down, many to go!

Patterns so far

- Singleton
- Iterator (you do it in Lab4)
- Multi-panel (multi-page webapp)
- Command Pattern (undo/redo)
- MVC (separate model from view/controller in ETF)
- Strategy Pattern
- Composite
- Visitor

Challenge Exercise



Challenge Exercise

- Consider a class CONTACT_MANAGER with an attribute

  array: ARRAY[CONTACT]

- Sort ARRAY[CONTACT] by age, name, etc
- Use different sorting algorithms e.g. quicksort, bubblesort etc.
- Change on the fly

Hint

- See Java comparator
- Eiffel (use regular expressions)

- DS_ARRAY_QUICKSORT
- DS_ARRAY_BUBBLE_SORT
- TWO_WAY_SORTED_LIST
- KL_PART_COMPARATOR

Not quite right?