Introduction to Design Patterns

Helwan University
Faculty of Computers and Information

Supervisor
Dr. Mai Hamdallah
CS Dept., FCI, Helwan University
Presented by
Tamer Abd-Elaziz Yassen
Teaching Assistant CS Dept., FCI, Helwan University

History of Design Pattern

Civil Engineer

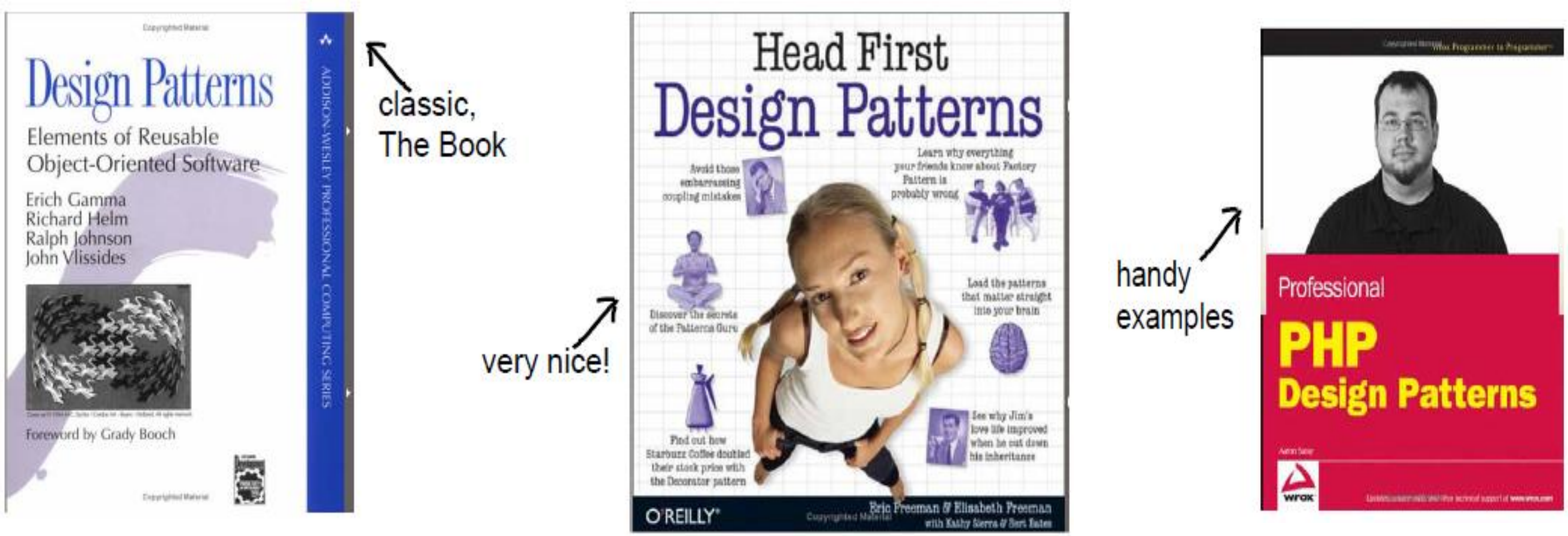
Christopher Alexander.

Gang of four : Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides


WHY
YOU SHOULD USE

Why Design Patterns

- A **software design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design.
- Recipes against common (OO-) programming problems
- Code reuse: no need to reinvent the wheel



Introduction to Design Patterns

Chapter 1

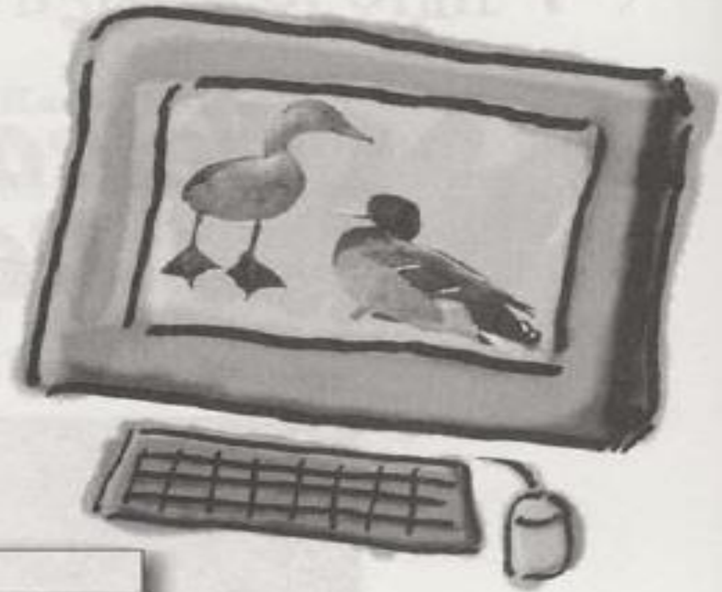
Strategy Pattern

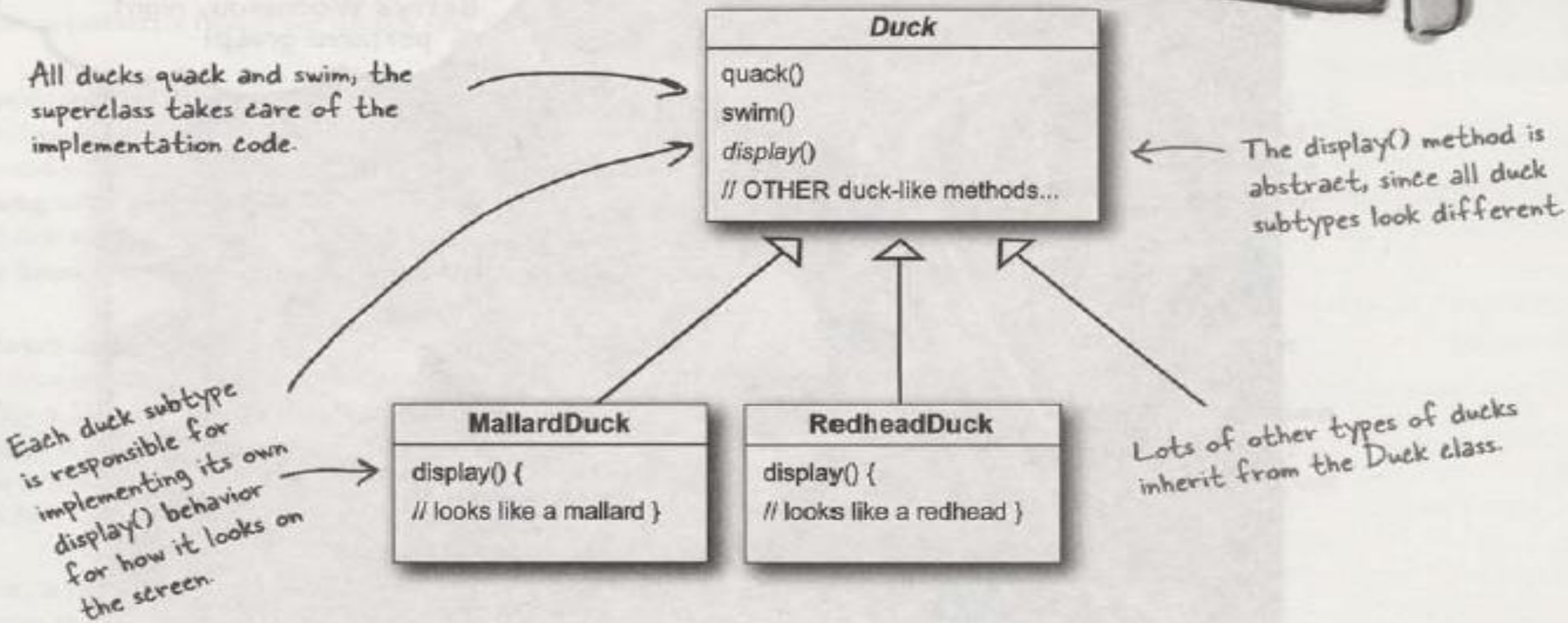
The one constant in software development:

CHANGE!

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.





Simple Simulation of Duck behavior

Duck

quack()
swim()
display()
// other duck methods

MallardDuck

display()
// looks like mallard

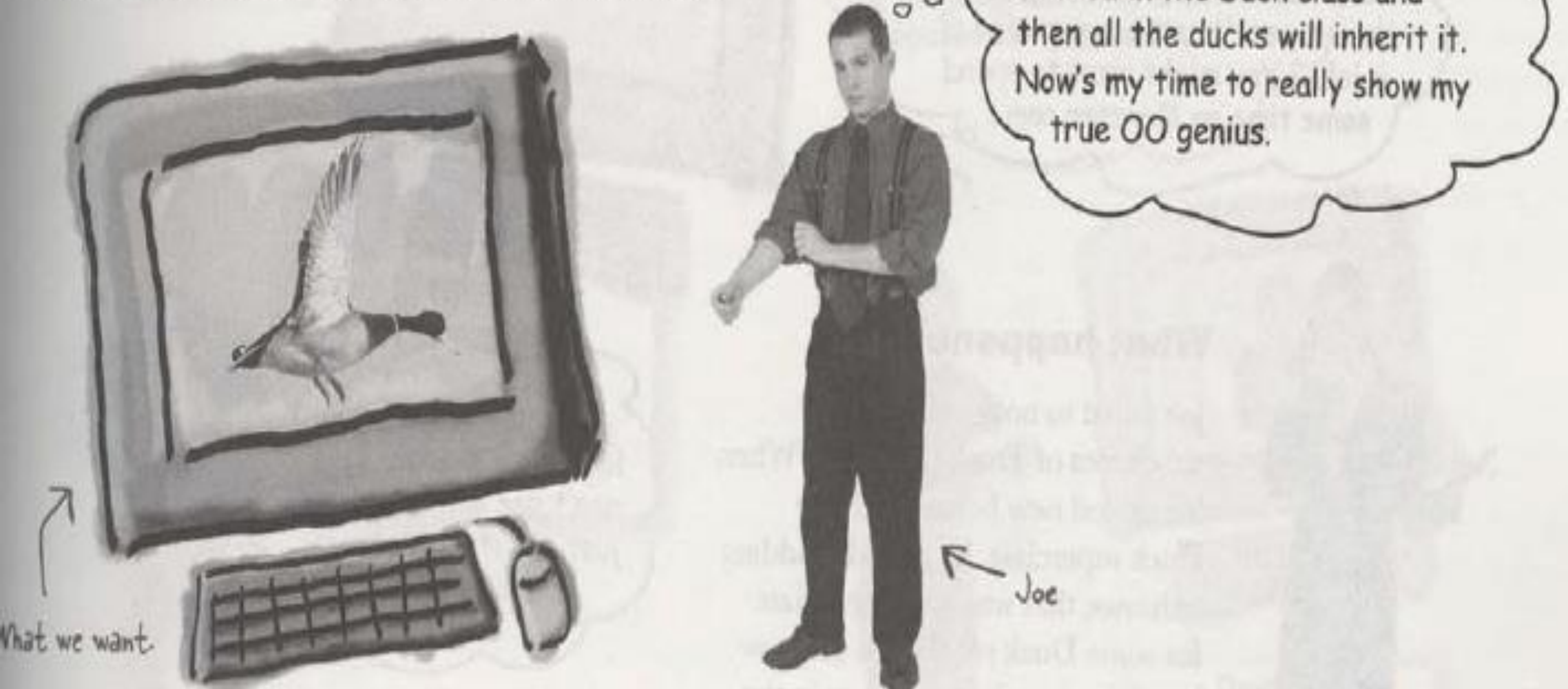
RedheadDuck

display()
// looks like redhead

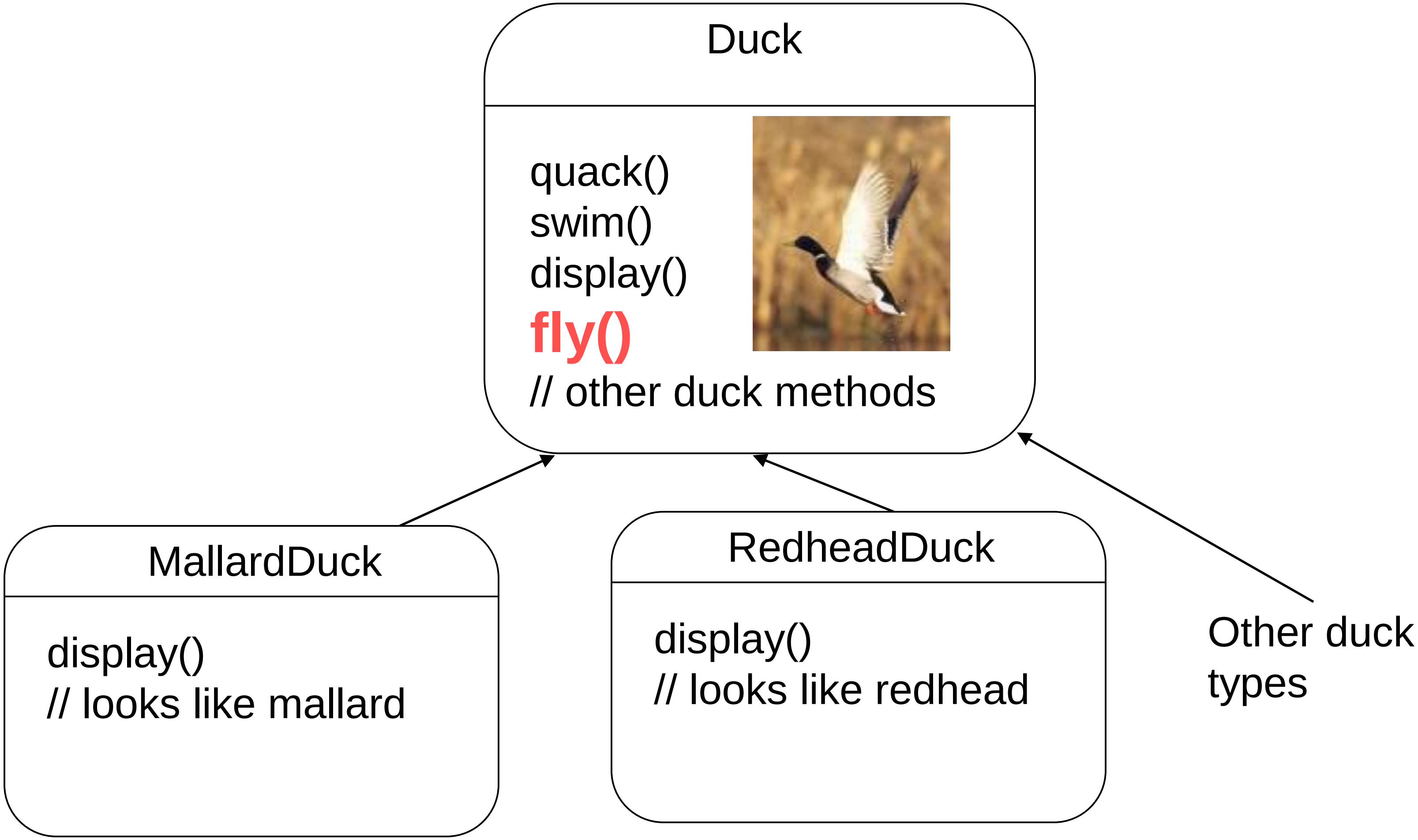
Other duck types

But now we need the ducks to FLY

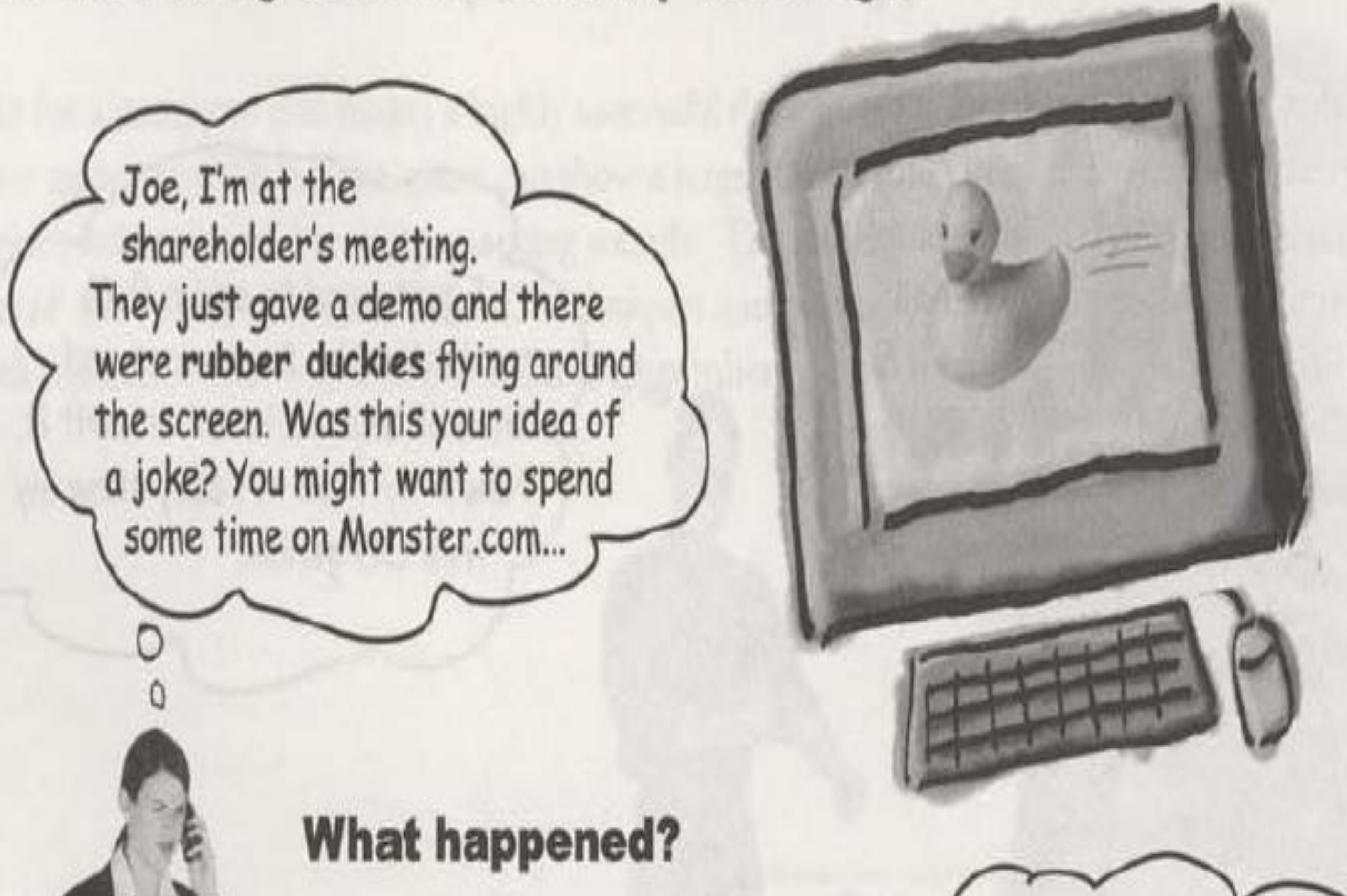
The executives decided that flying ducks is just what the simulator needs to blow away the other duck sim competitors. And of course Joe's manager told them it'll be no problem for Joe to just whip something up in a week. "After all", said Joe's boss, "he's an OO programmer... *how hard can it be?*"



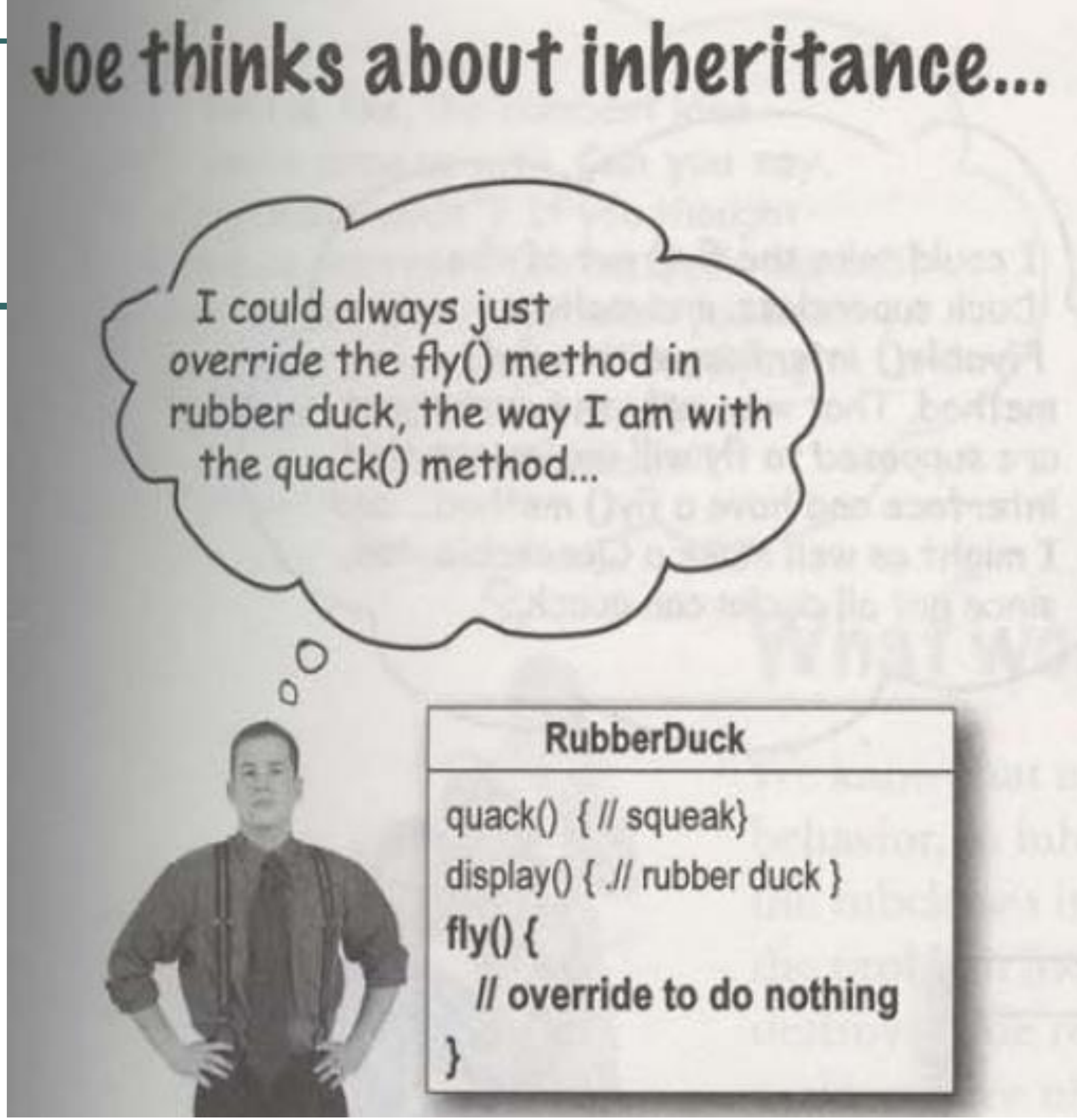
What if we want to simulate flying ducks?



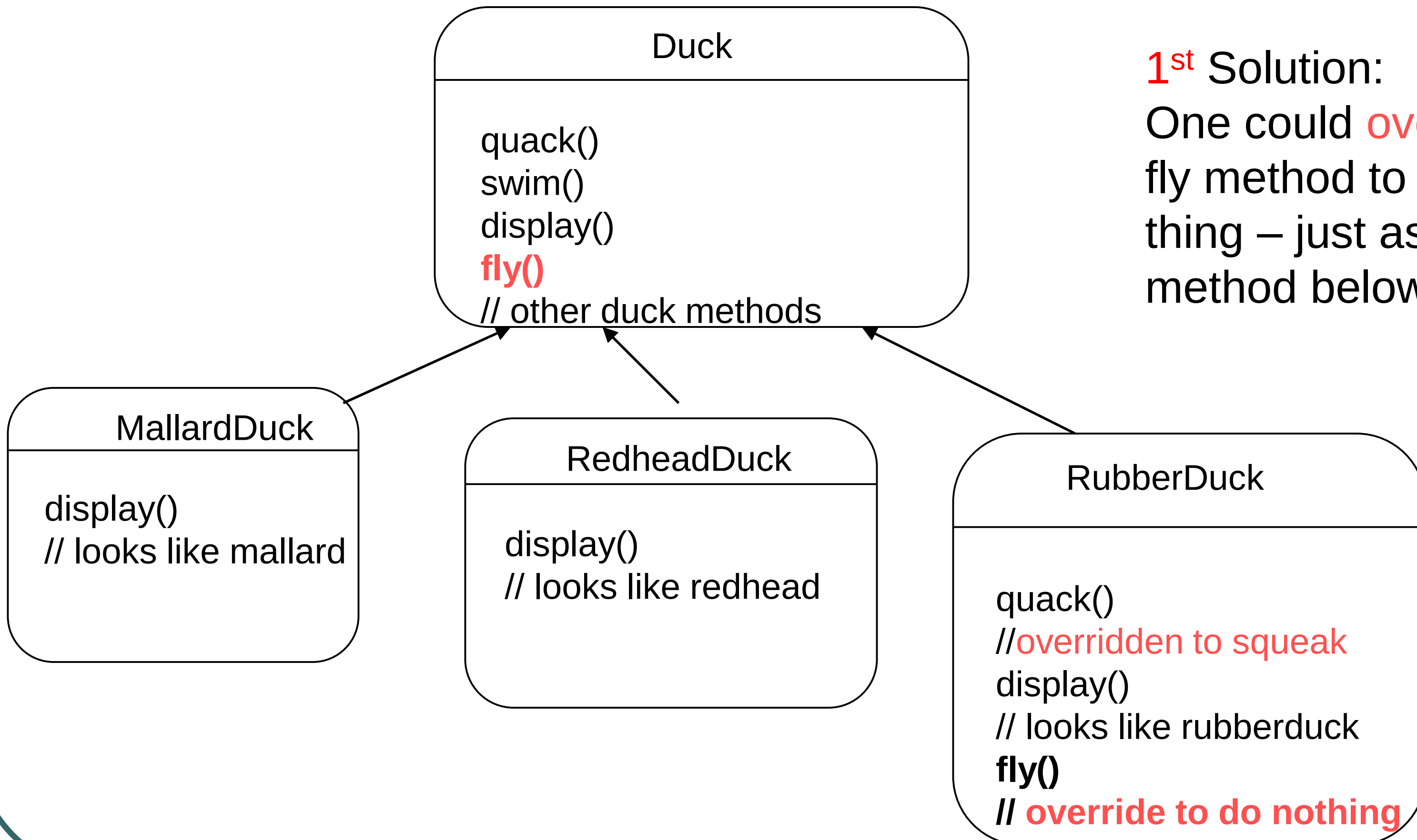
But something went horribly wrong...



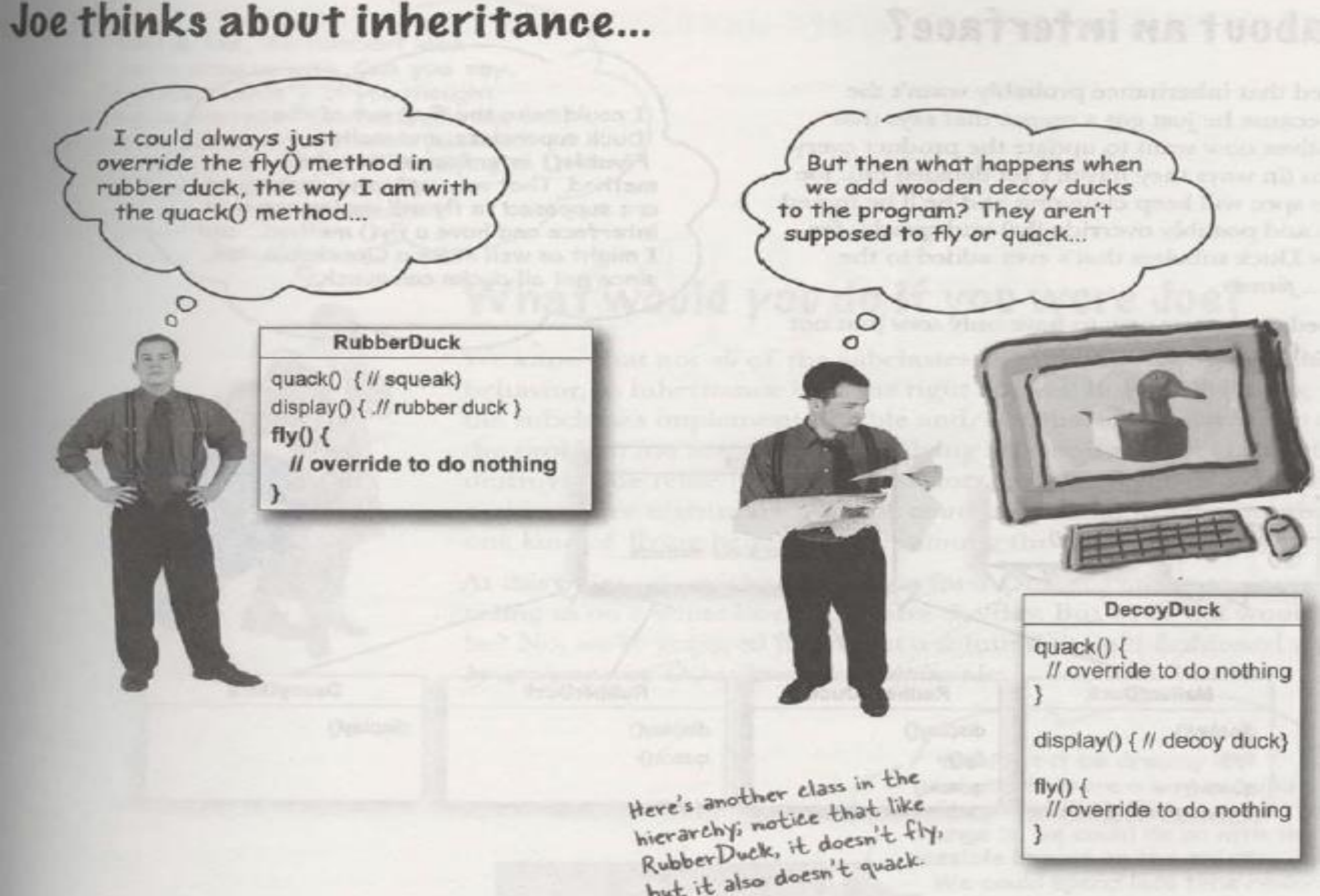

OK, so there's a slight flaw in my design. I don't see why they can't just call it a "feature". It's kind of cute...


Joe thinks about inheritance...
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
}

Tradeoffs in use of inheritance and maintenance



1st Solution:
One could override the fly method to the appropriate thing – just as the quack method below.


Joe thinks about inheritance...
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
}
But then what happens when we add wooden decoy ducks to the program? They aren't supposed to fly or quack...
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.

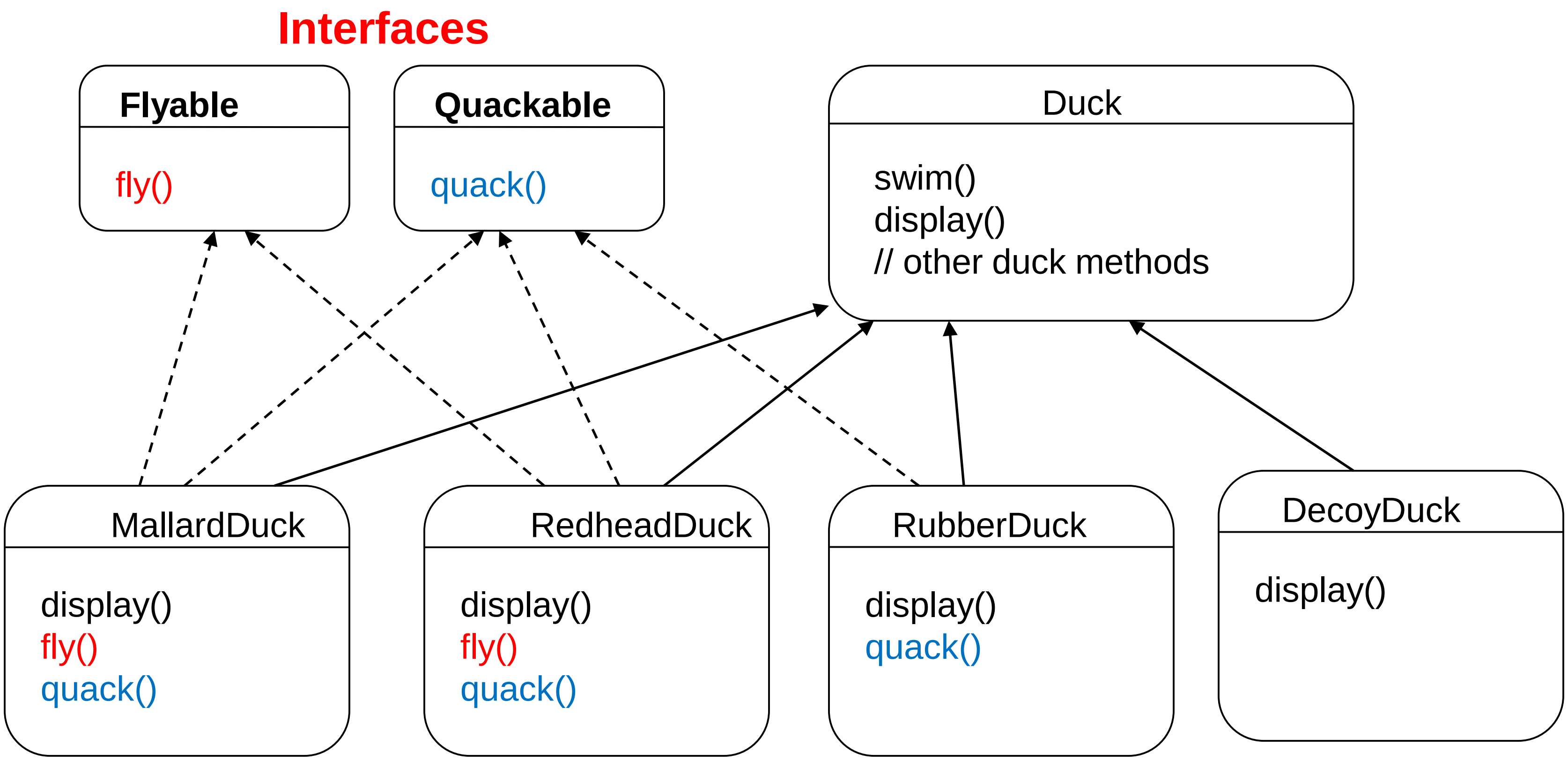
Override is not the right answer

Inheritance and override
is not always the right answer.

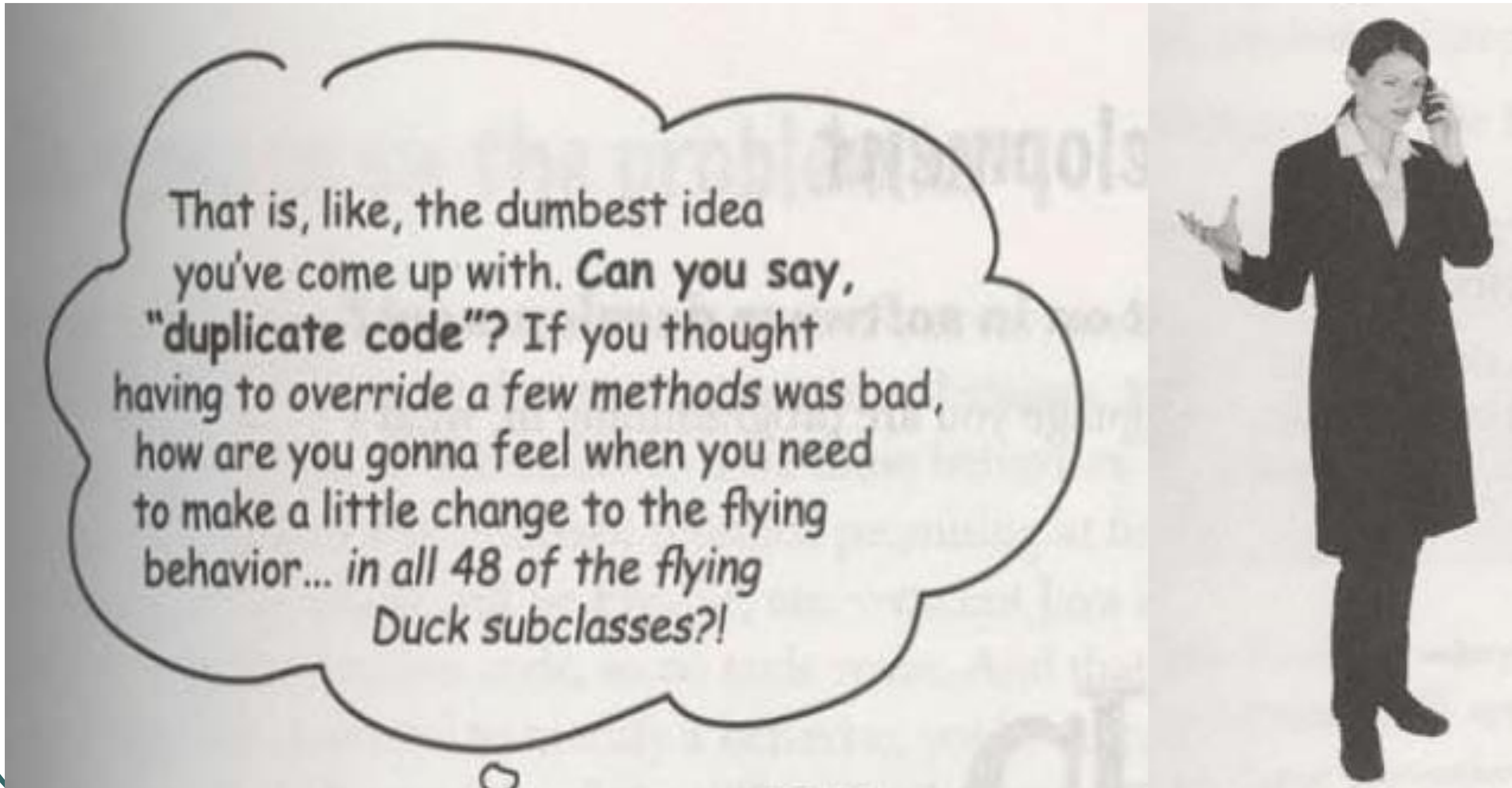
Every new class that inherits
unwanted behavior needs to be
overridden.

How about using interfaces instead?
May be 2nd Solution

Duck simulation recast using **interfaces.**



What do YOU think about this Design?



Interfaces is not the right answer

- By defining interfaces, every class that needs to support that interface needs to implement that functionality… destroys code reuse!

- So if you want to change the behavior defined by interfaces, every class that implements that behavior may potentially be impacted

And….

The one constant in software development

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?

CHANGE

(use a mirror to see the answer)

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

Design Principle

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

OR

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.

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.

Duck

fly()
quack()
swim()
display()
// other duck methods

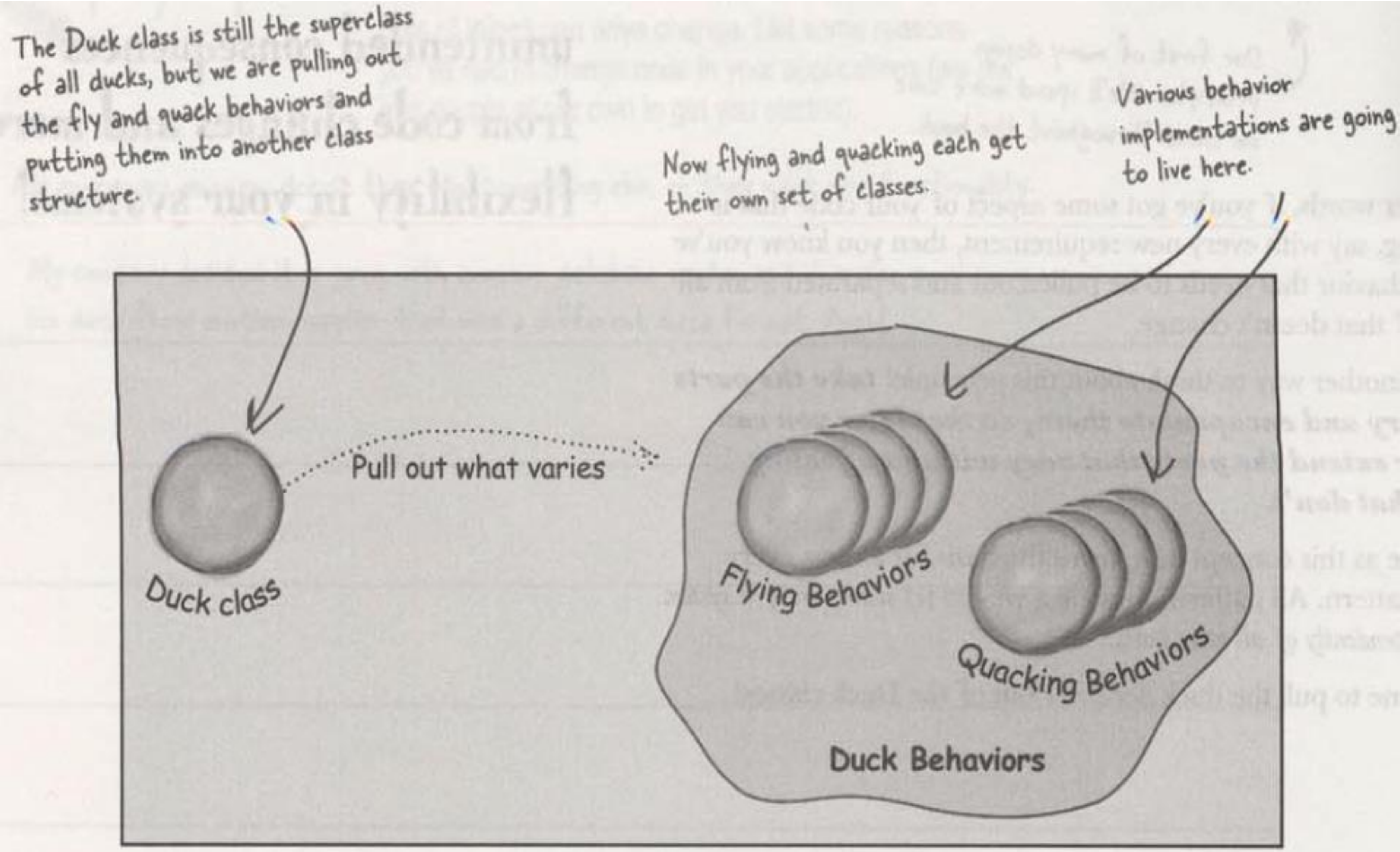
MallardDuck

display()
// looks like mallard

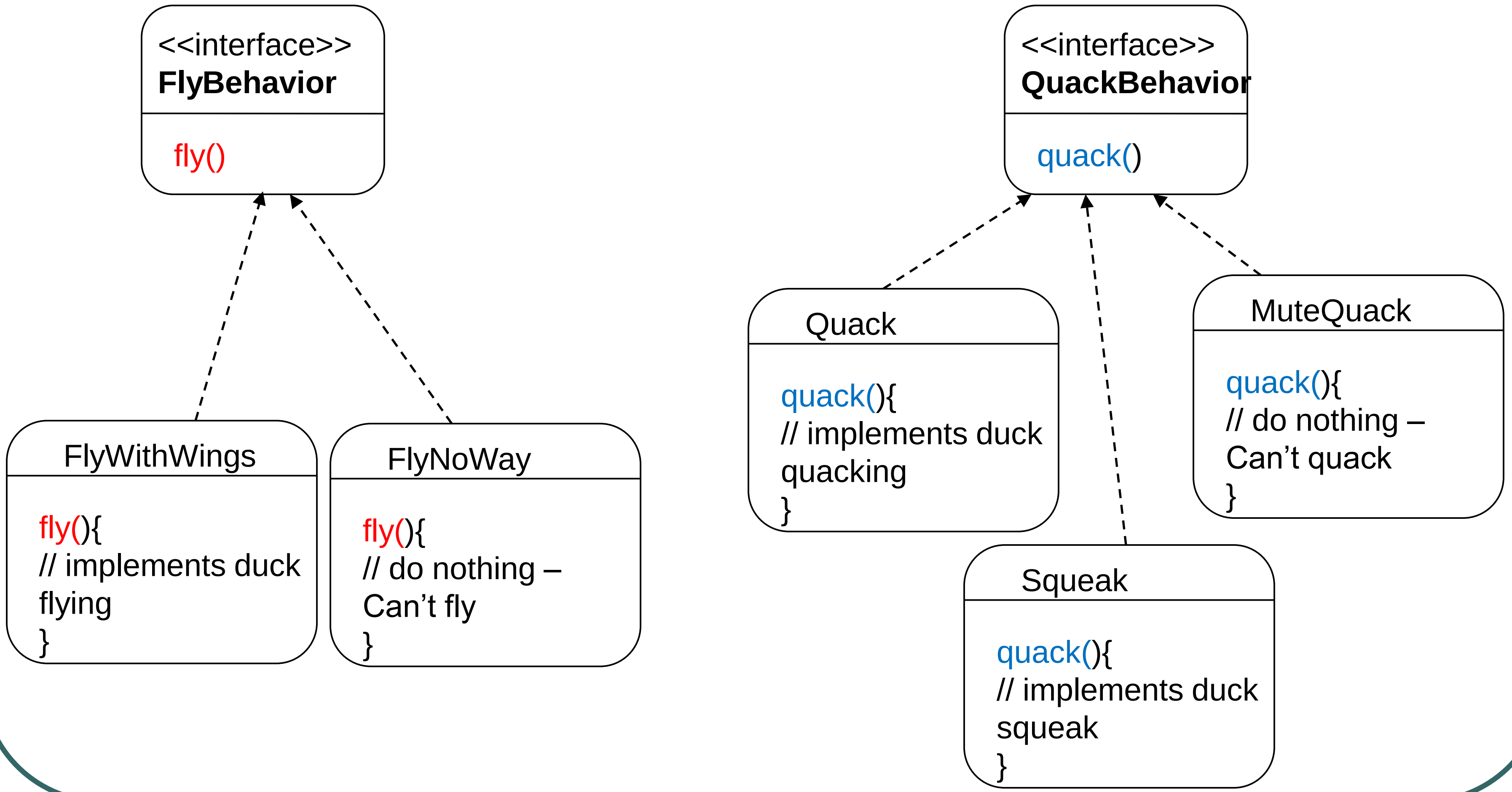
RedheadDuck

display()
// looks like redhead

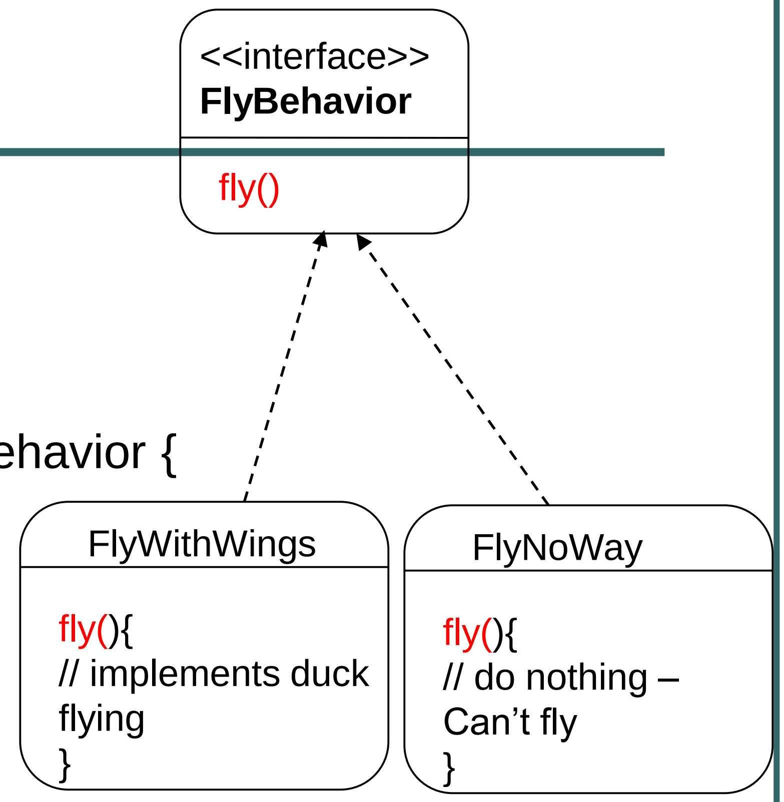
Other duck types


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

Implementing duck behaviors - revisited



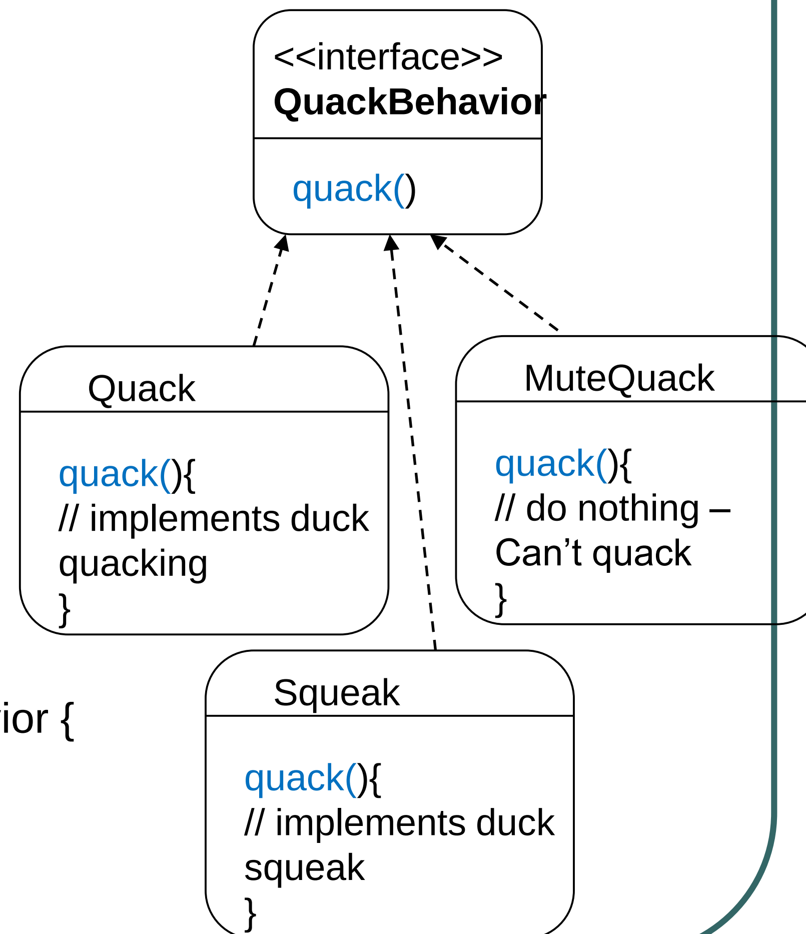
Specific behaviors by implementing interface FlyBehavior




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

------------------------------------------------------------
public class FlyNoWay implements FlyBehavior {
    public void fly() {
        System.out.println("I can't fly");
    }
}
```

Specific behaviors by implementing interface QuackBehavior

```
public interface QuackBehavior {
    public void quack();
}
public class Quack implements QuackBehavior {
    public void quack() {
        System.out.println("Quack");
    }
}
public class Squeak implements QuackBehavior {
    public void quack() {
        System.out.println("Squeak");
    }
}
public class MuteQuack implements QuackBehavior {
    public void quack() {
        System.out.println("<< Silence >>");
    }
}
```



Design Principle

- Program to an interface, not to an implementation.
- Really means program to a super type.

1. Integrating the Duck Behavior

Add 2 instance variables:

Behavior variables are declared as the behavior SUPERTYPE

Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
performQuack()
Swim()
Display()
performFly()
//OTHER duck-like methods

Instance variables hold a reference to a specific behavior at runtime

These general methods replace fly() and quack()

2. Implement Duck SuperClass

```
public abstract class Duck {

   // Declare two reference variables for the behavior interface types
   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior; // All duck subclasses inherit these

   public Duck() {
   }

  public void performFly() {
      flyBehavior.fly();             // Delegate to the behavior class
   }

   public void performQuack() {
      quackBehavior.quack();  // Delegate to the behavior class
   }
```

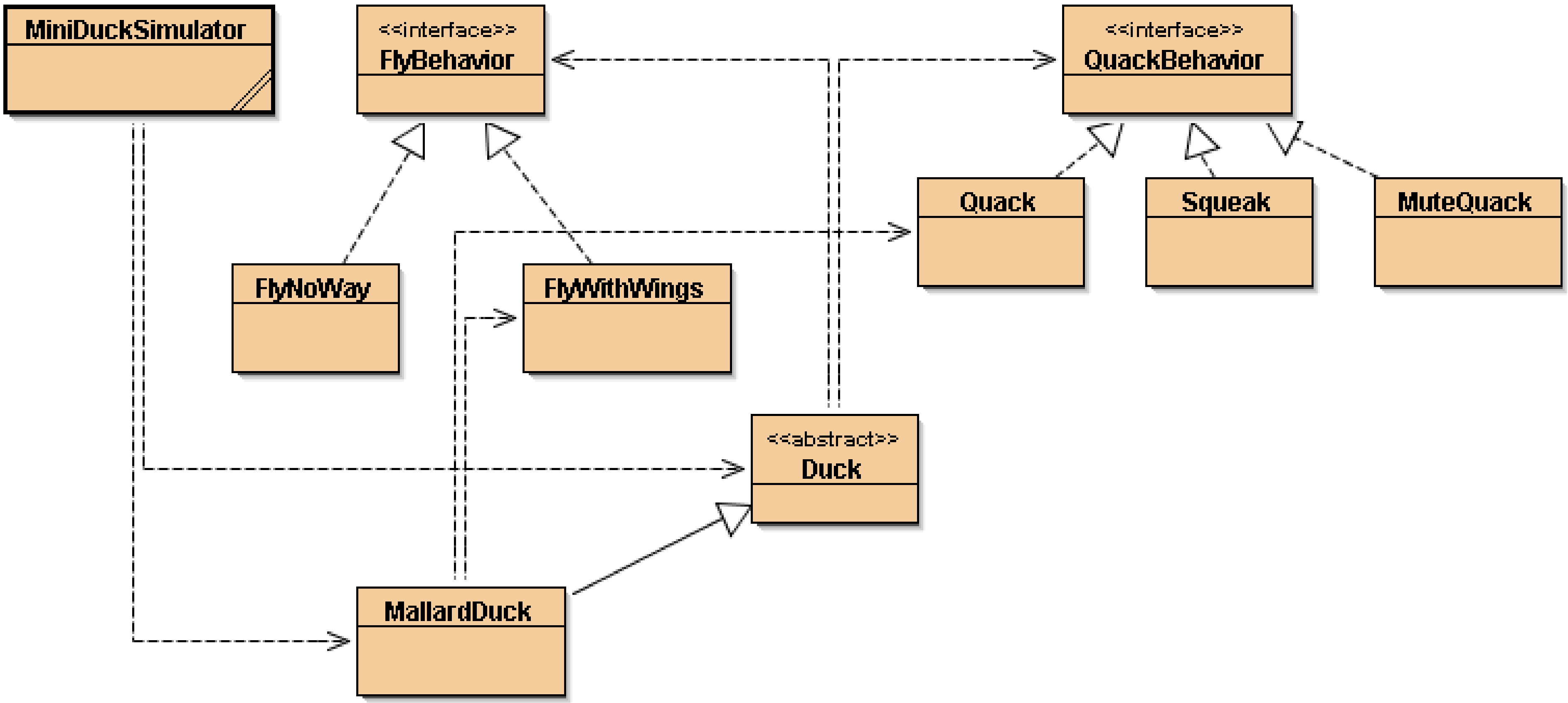
3. How to set the quackBehavior variable & flyBehavior variable

```
public class MallardDuck extends Duck {

    public MallardDuck() {

        quackBehavior = new Quack();
                        // 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

        flyBehavior = new FlyWithWings();
                        // And it uses flyWithWings as its flyBehavior type

    }

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

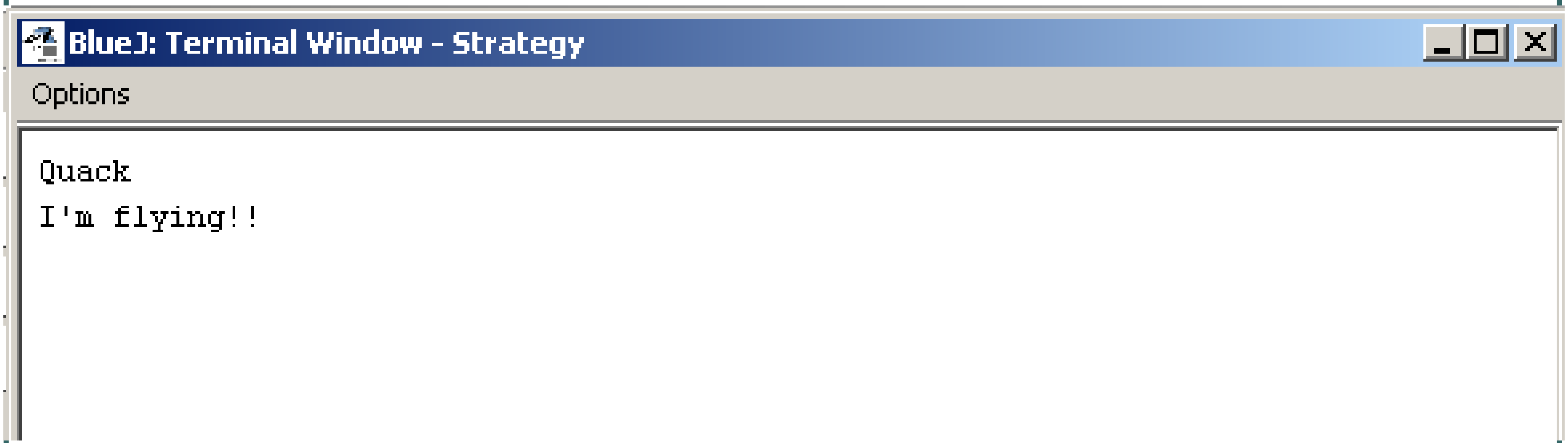
4. Type and compile the test class (MiniDuckSimulator.java)

```
public class MiniDuckSimulator {

    public static void main(String[] args) {

        Duck    mallard = new MallardDuck();
        mallard.performQuack();
          // 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)
        mallard.performFly();
           // Then we do the same thing with MallardDuck's inherited
           // performFly() method.
    }
}
```

Strategy project



Run MiniDuckSimulator

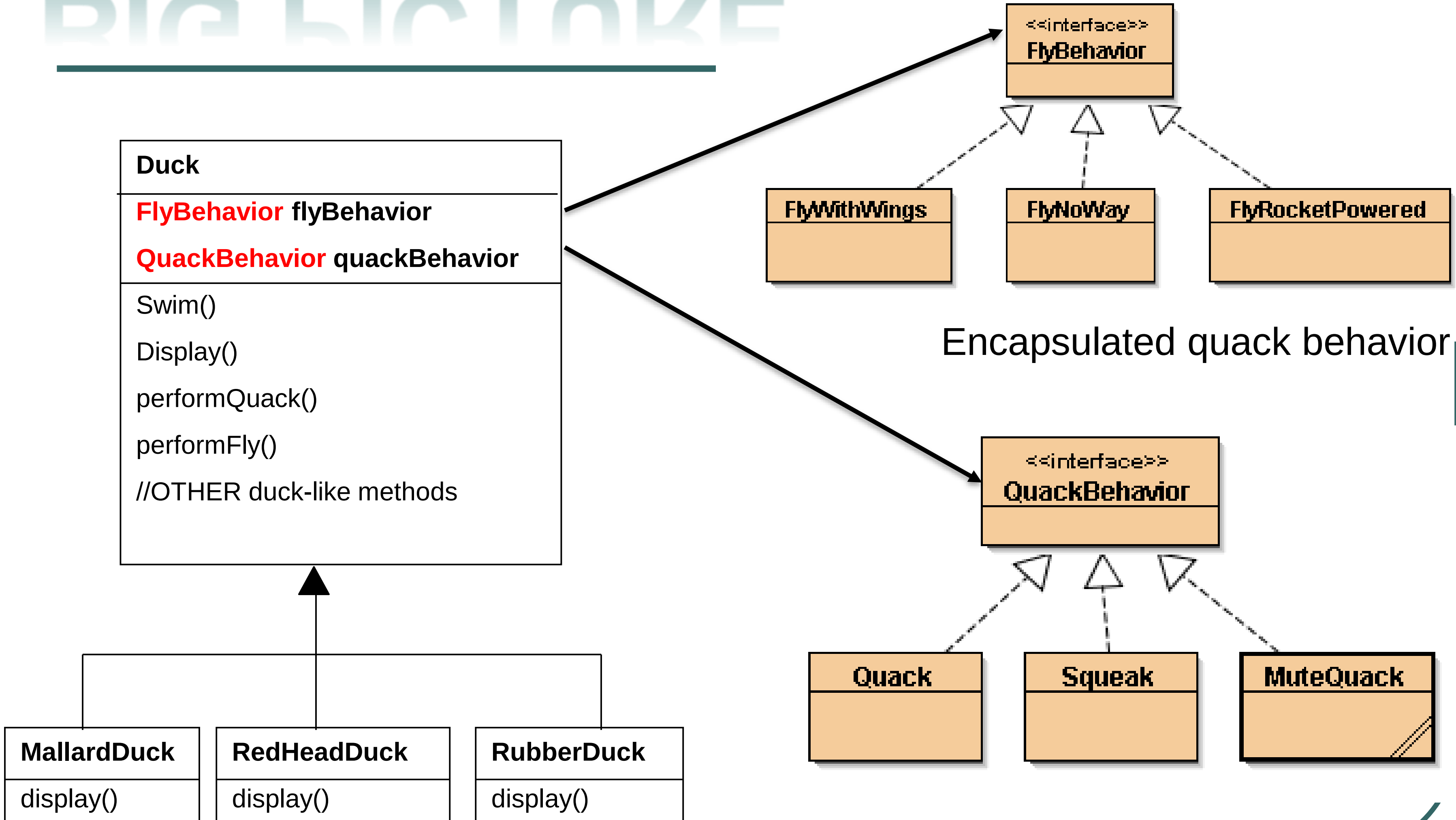


Big Picture on encapsulated behaviors

Reworked class structure

BIG PICTURE

Encapsulated fly behavior



Summary

- Reworked class structure
  - ducks extending Duck
  - fly behaviors implementing FlyBehavior
  - quack behaviors implementing QuackBehavior
- Think of each set of behaviors as a family of algorithms
- Relationships: IS-A, HAS-A, IMPELMENTS

Design Principle

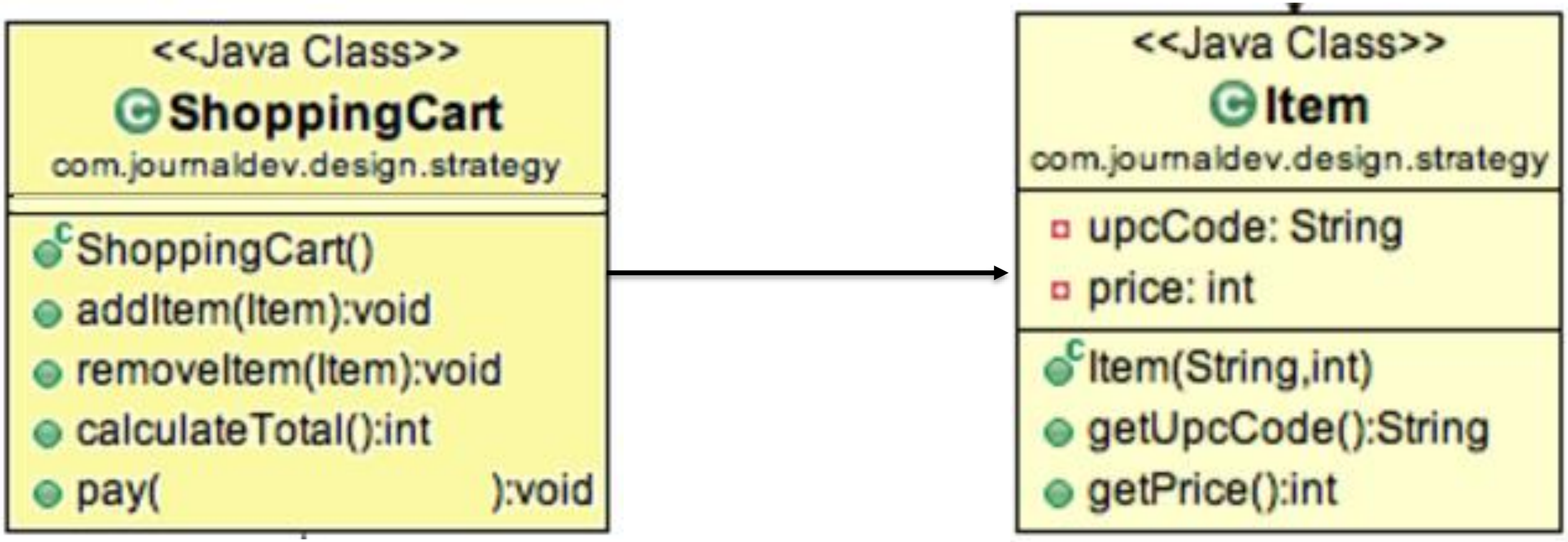
- Favor composition over inheritance

- HAS-A can be better than IS-A
- Allows changing behavior at run time

HAS-A can be better than IS-A

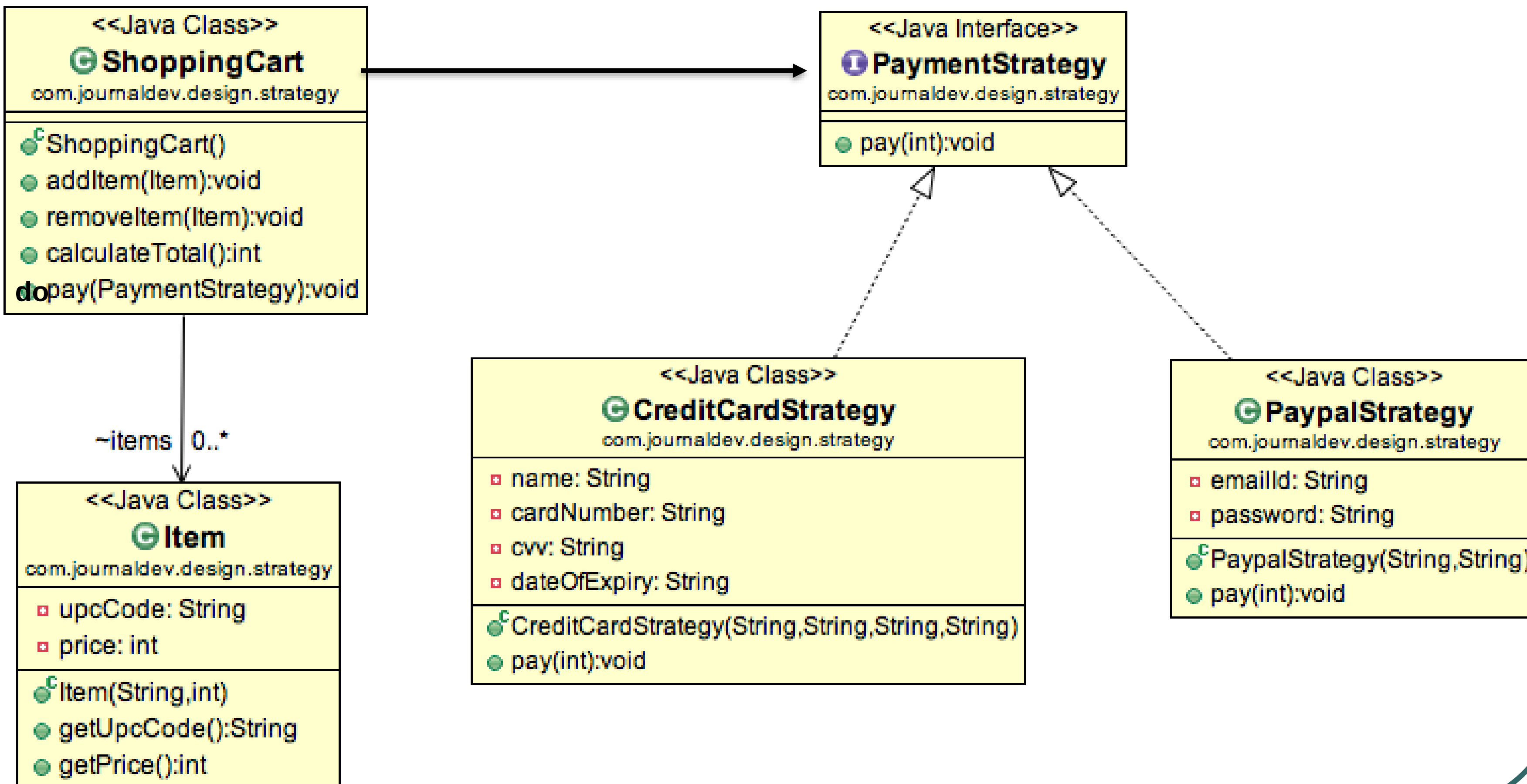
- Each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking
- **Composition** at work
  - Instead of inheriting behavior, ducks get their behavior by being *composed* with the right behavior object

A simple Shopping Cart

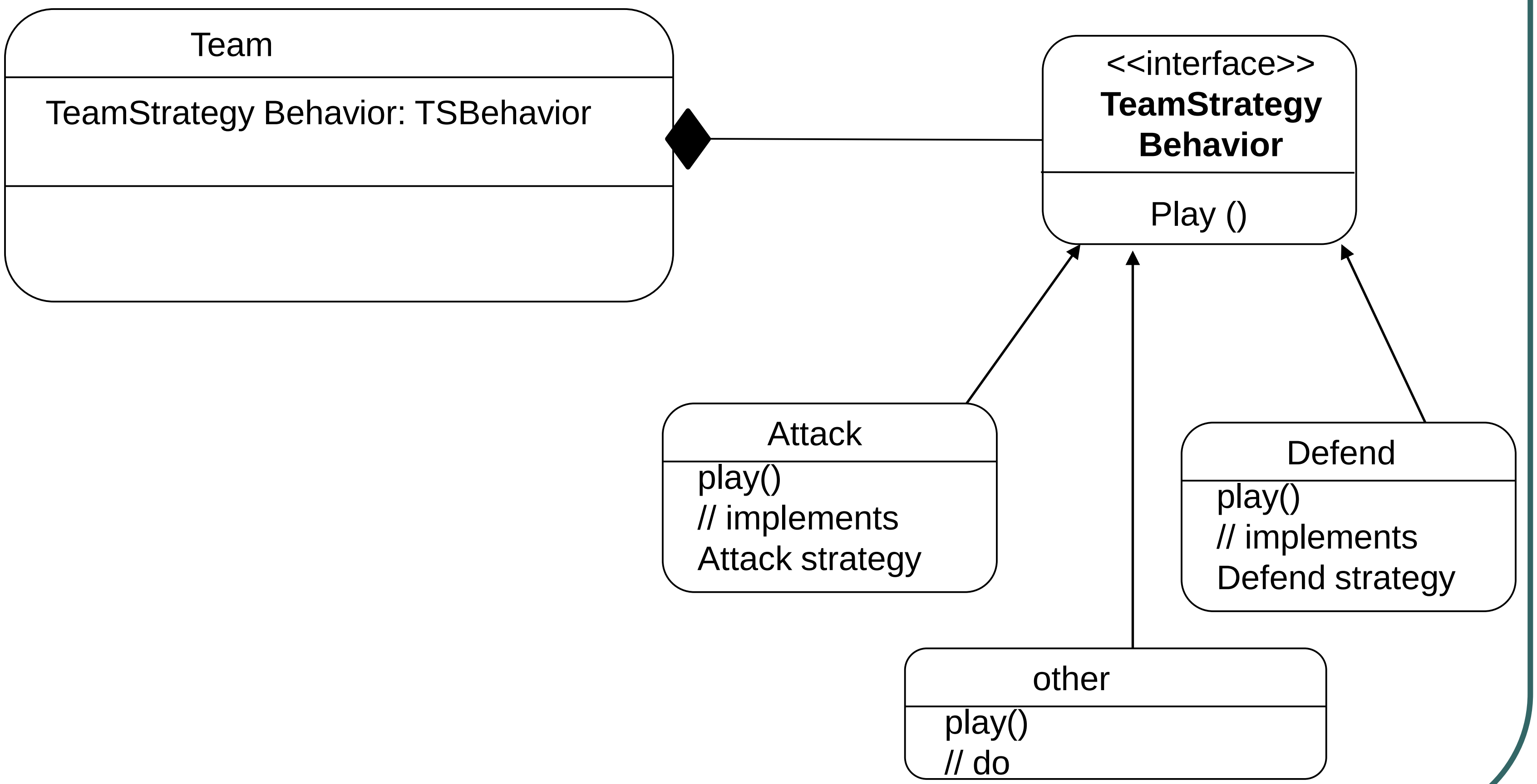


We have many payment strategies – using Credit Card or using PayPal, or Other types will added in the future.

Strategy Pattern Class Diagram



Team Strategy using the new approach



The 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.

Thanks

Any questions contact with me via e-mail : tamer.a.yassen@gmail.com