Boring. Isn't it?

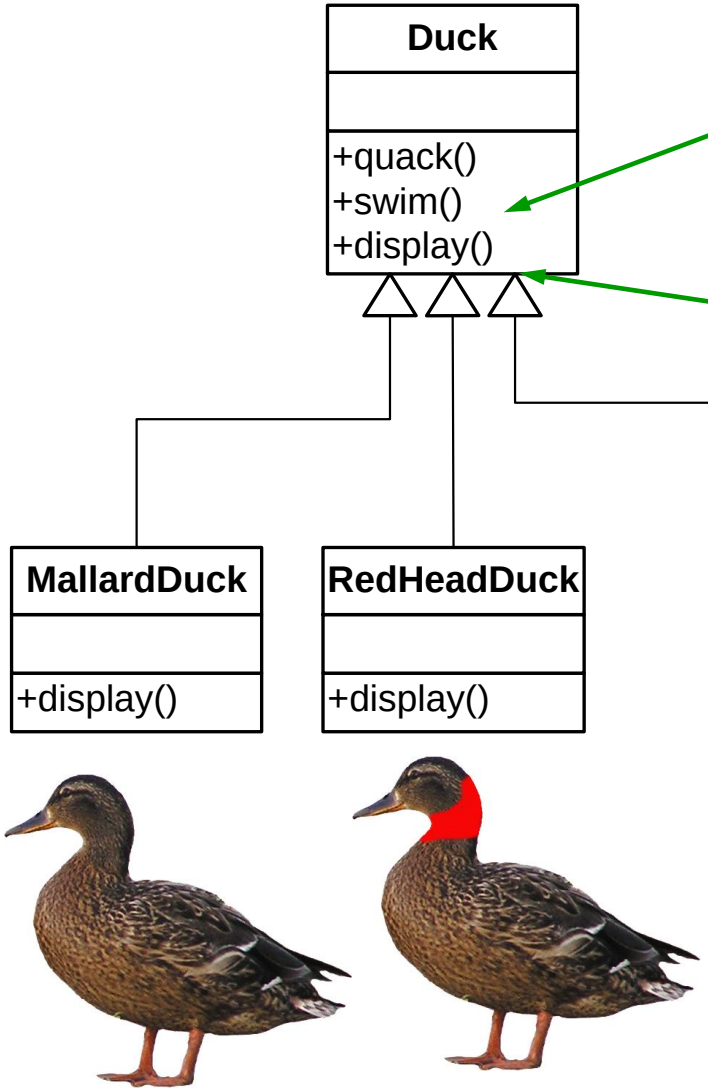
Let's do some programming !!!!!

The Specifications

- **Joe works at a company that produces a simulation game called *SimUDuck*. He is an OO Programmer and his duty is to implement the necessary functionality for the game**

- **The game should have the following specifications:**
  - A variety of different ducks should be integrated into the game
  - The ducks should swim
  - The duck should quake

A First Design of a Duck Simulator



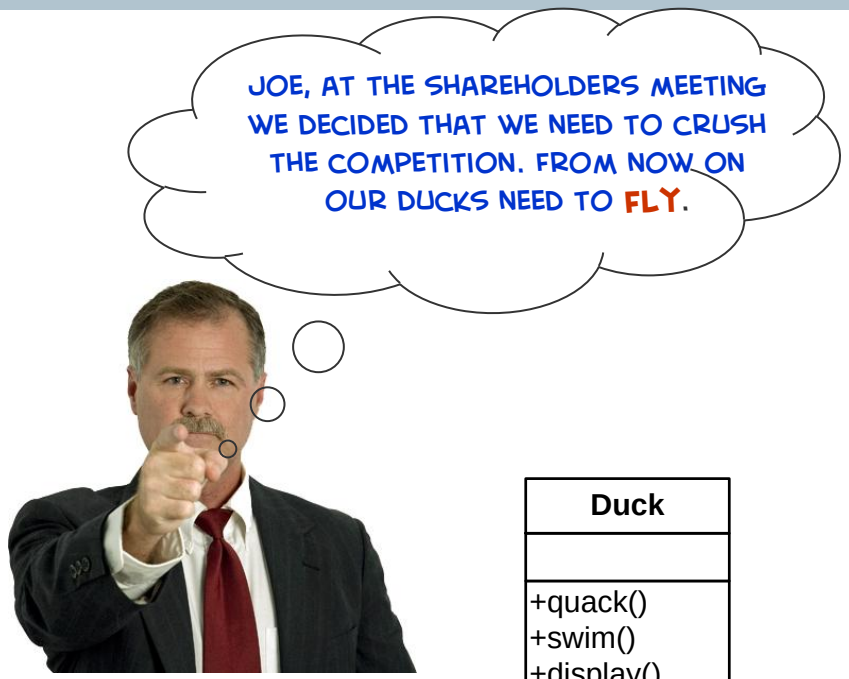
All ducks **`quack()`** and **`swim()`**. The superclass takes care of the implementation

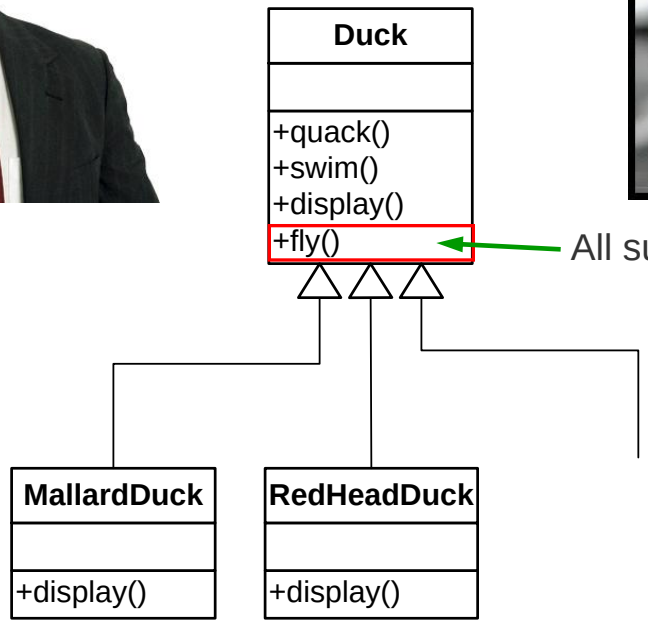
The **`display()`** method is abstract since all the duck subtypes look different

Each duck subtype is responsible for implementing its own **`display()`** behavior for how it looks on the screen

Lots of other types of ducks inherit from the Duck type

Ducks that Fly



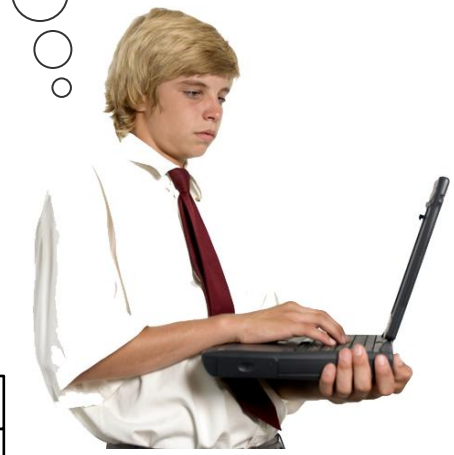


All subclasses inherit `fly()`

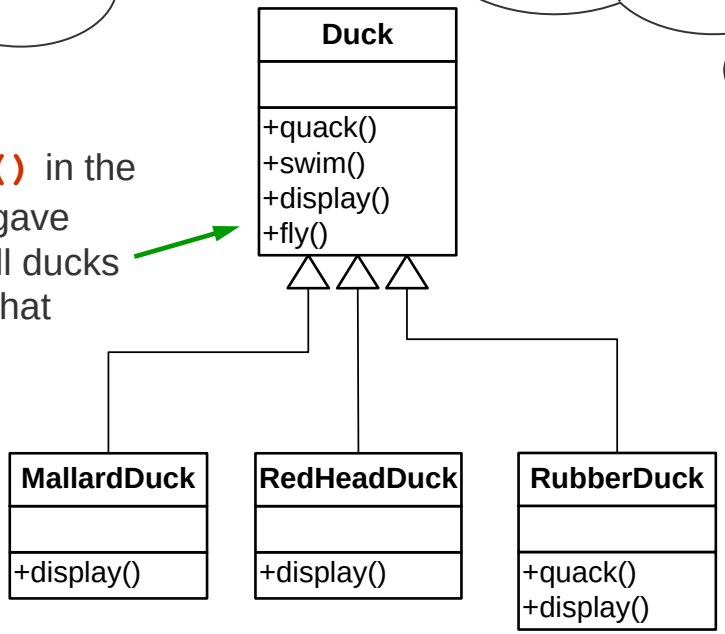
But Something Went Wrong

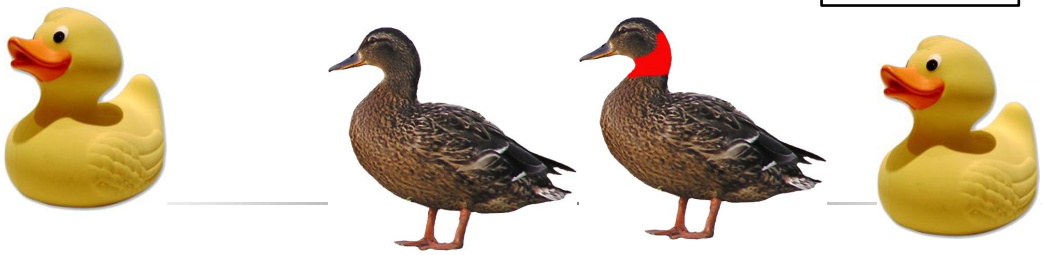
JOE, I'M AT THE SHAREHOLDER'S MEETING. THEY JUST GAVE A DEMO AND THERE WERE RUBBER DUCKIES FLYING AROUND THE SCREEN. IS THIS A JOKE OR WHAT?

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

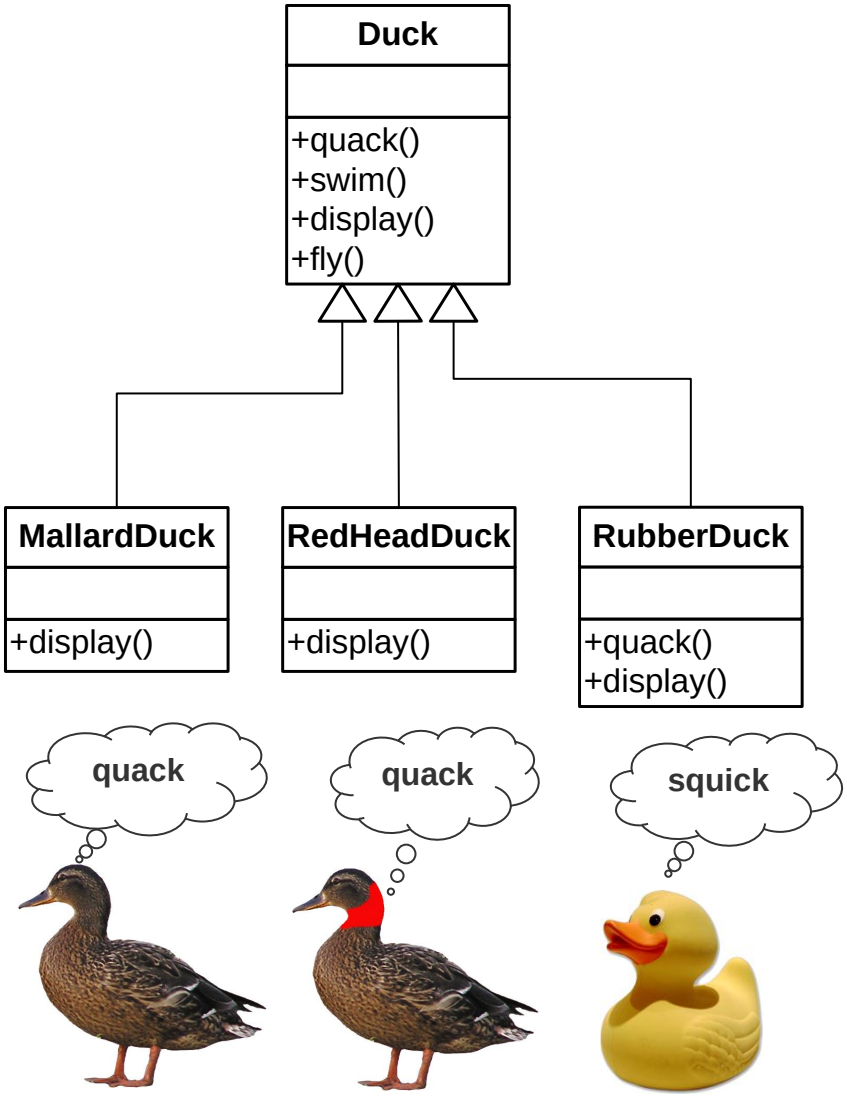


By putting **`fly()`** in the superclass Joe gave flying ability to all ducks including those that shouldn't





Inheritance at Work

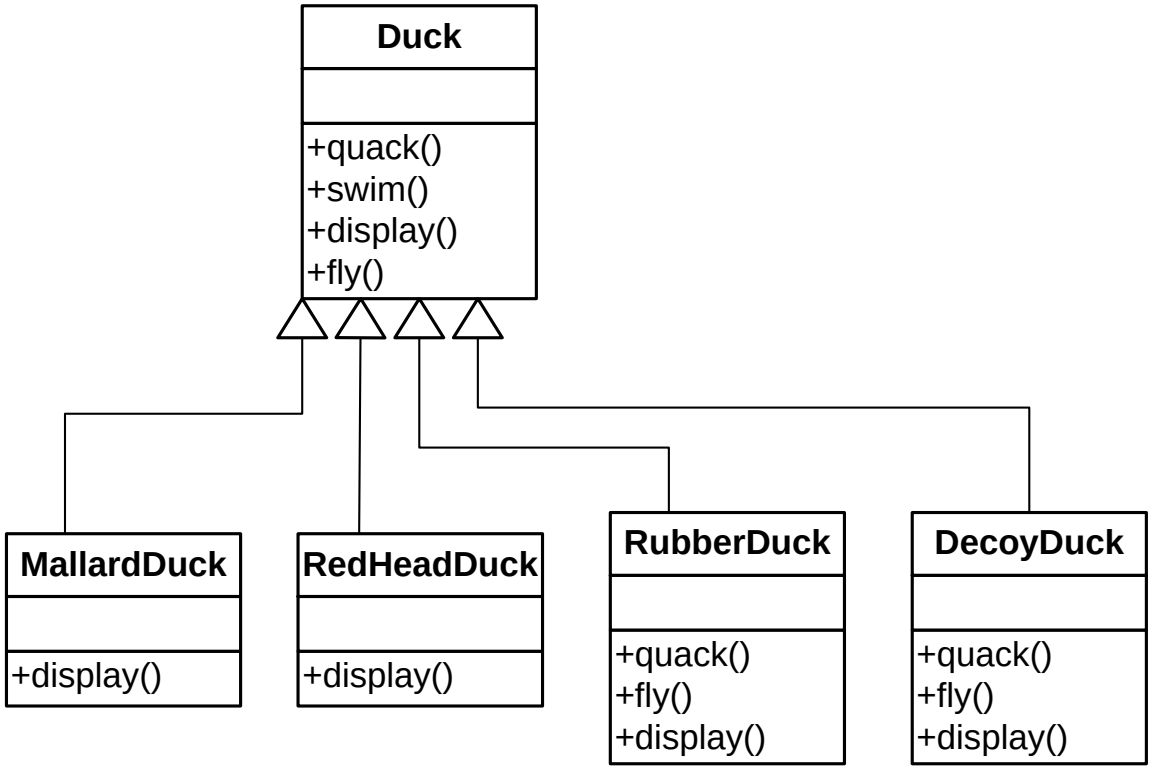



```
void Duck::quack(){
  cout << "quack, quack" << endl;
}

void RubberDuck::quack(){
  cout << "squick, squick" << endl;
}
```

We can override the **`fly()`** method in the rubber duck in a similar way that we override the **`quack()`** method

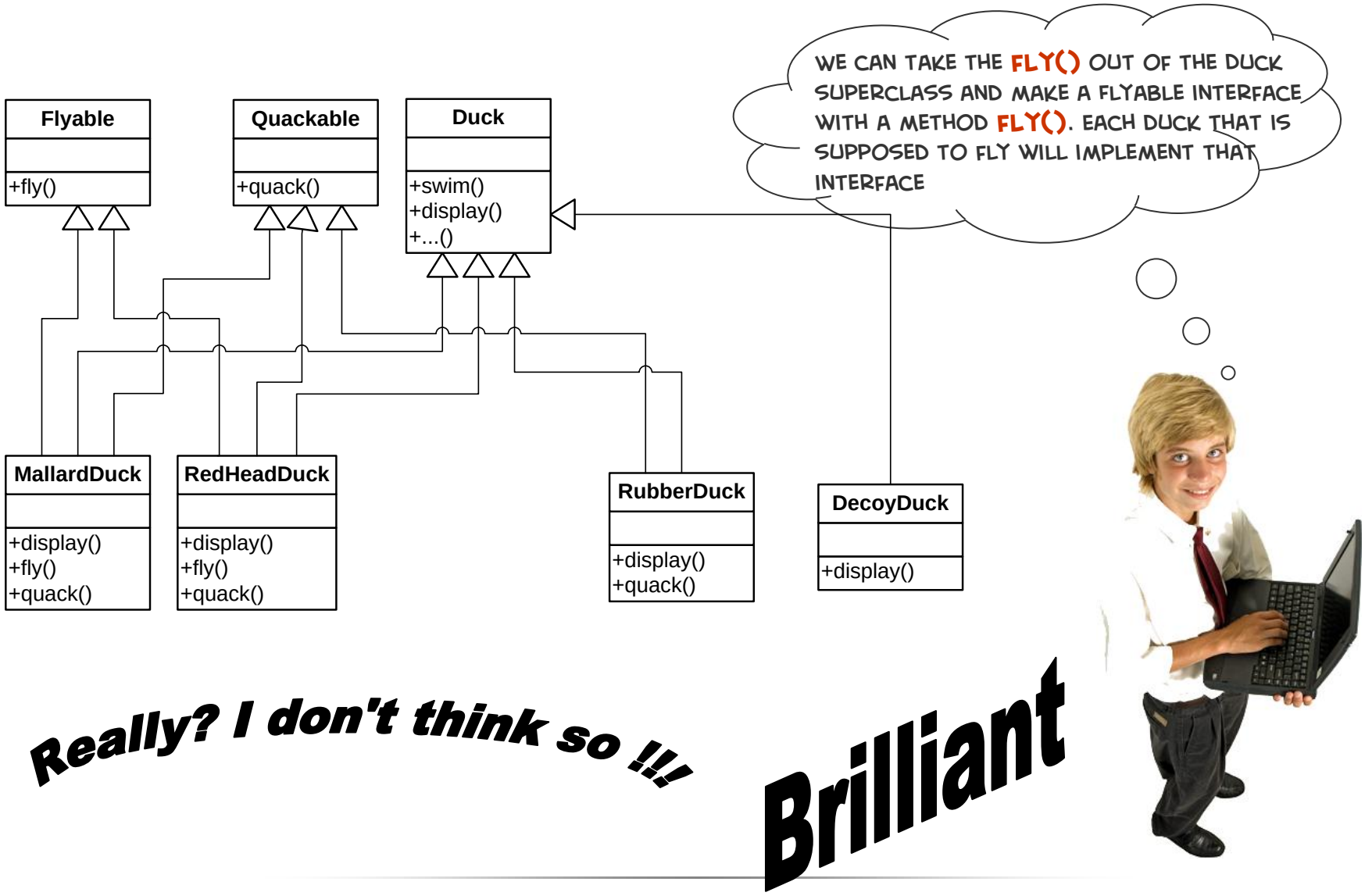
```
void Duck::fly(){
  // fly implementation
}

void RubberDuck::fly(){
  // do nothing
}
```

Yet Another Duck is Added to the Application



```
void DecoyDuck::quack(){
   // do nothing;
}

void DecoyDuck::fly(){
   // do nothing
}
```

How About an Interface?



Embracing Change

- **In SOFTWARE projects you can count on one thing that is constant:**

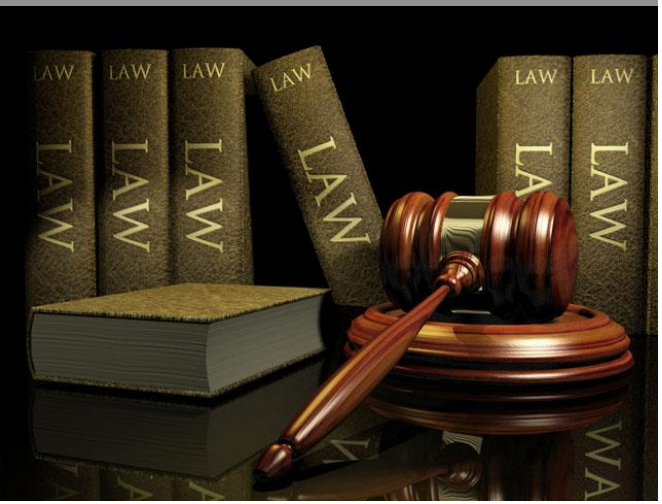
CHANGE

- Solution
  - Deal with it.
    - Make CHANGE part of your design.
    - Identify what vary and separate from the rest.

Change We Can Believe In

Encapsulate what varies

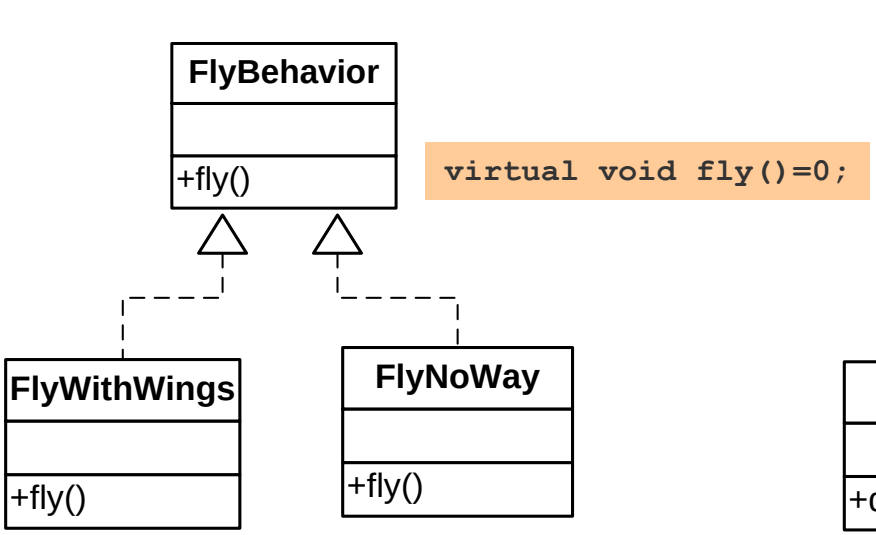
The Constitution of Software Architects



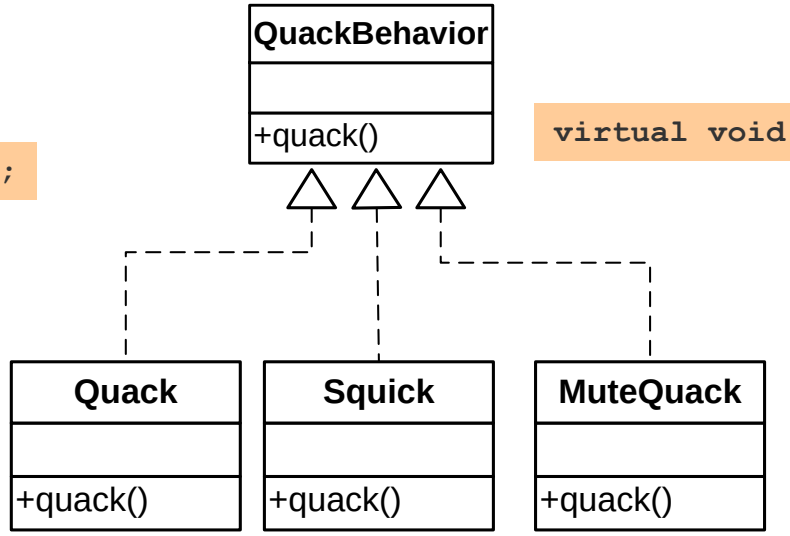
- **Encapsulate what varies.**
- **?????????**
- **?????????**
- **?????????**
- **?????????**
- **?????????**
- **?????????**
- **?????????**
- **?????????**

Embracing Change in Ducks

- `fly()` and `quack()` are the parts that vary
- We create a new set of classes to represent each behavior




```
virtual void fly()=0;
```




```
virtual void quack()=0;
```

```
void FlyWithWings::fly(){
  cout << "I'm flying!" << endl;
};
```

```
void Quack::quack(){
  cout << "Quack" << endl;
};
```

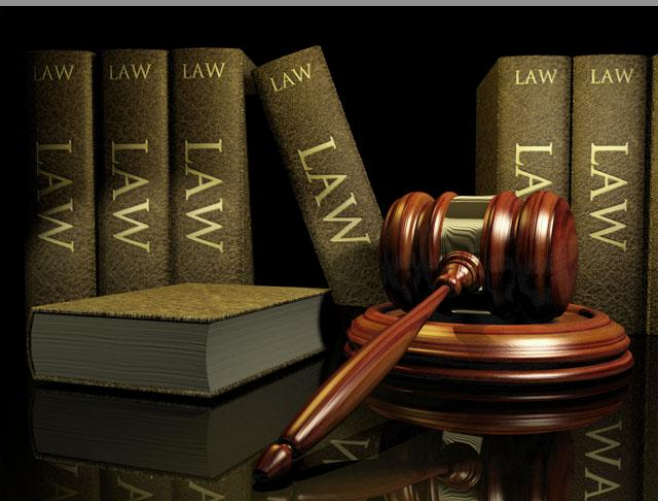
```
void Squeak::quack(){
  cout << "Squeak" << endl;
};
```

```
void FlyNoWay::fly(){
  cout << "I can't fly." << endl;
};
```

```
void MuteQuack::quack(){
   cout << "....." << endl;
};
```

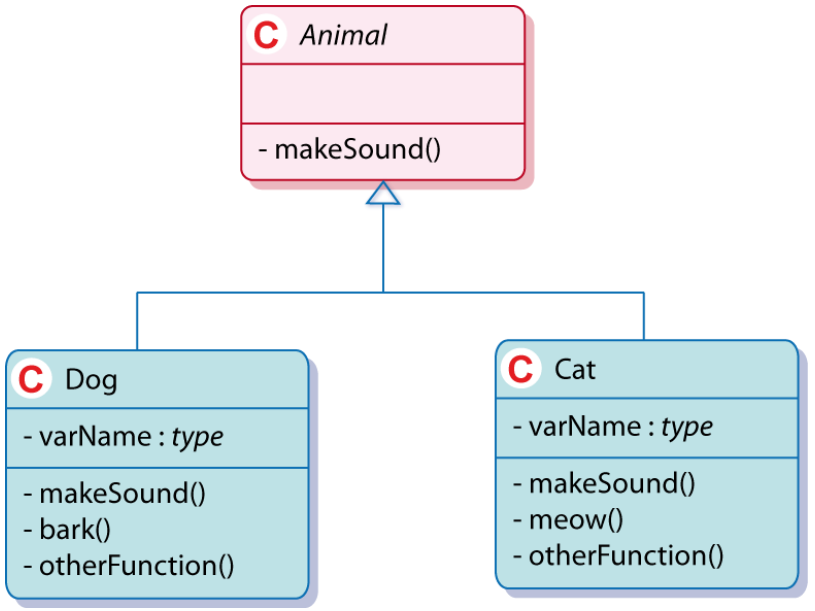
Program to an interface not to an implementation

The Constitution of Software Architects



- Encapsulate what varies.
- Program through an interface not to an implementation
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????

Design Principle Example




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

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

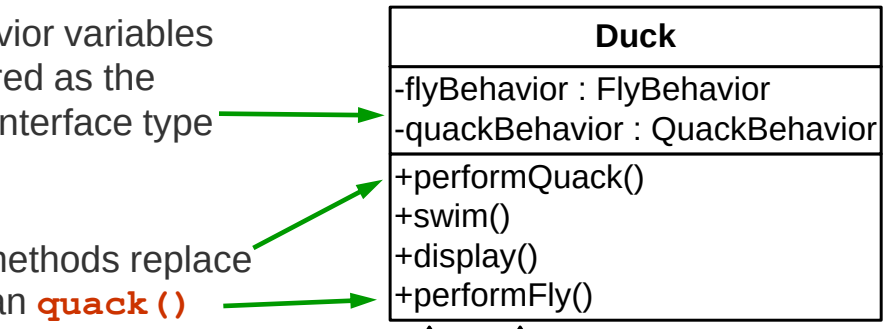
```
void Dog::makeSound(){
    bark();
}
```

```
void Cat::makeSound(){
    meow();
}
```

Integrating the Duck Behavior

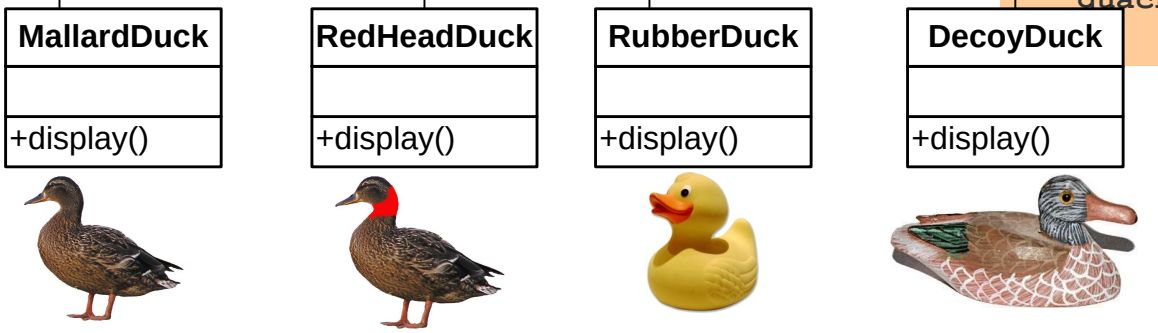
The behavior variables are declared as the behavior interface type

These methods replace `fly()` an `quack()`




```
class Duck{
public:
  FlyBehavior *flyBehavior;
  QuackBehavior *quackBehavior;
  ...
  void performFly();
  void performQuack();
  ...
};
```

```
void Duck::performFly(){
  flyBehavior->fly();
}
void Duck::performQuack(){
  quackBehavior->quack();
```



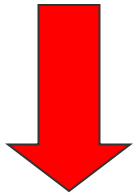

```
MallardDuck::MallardDuck(){
  flyBehavior = new FlyWithWings();
  quackBehavior = new Quack();
}
```

```
RubberDuck::RubberDuck(){
  flyBehavior = new FlyNoWay();
  quackBehavior = new Squick();
}
```

Design Principle Ahead

Duck
-flyBehavior : FlyBehavior -quackBehavior : QuackBehavior
+performQuack() +swim() +display() +performFly()

Each Duck **HAS A** FlyingBehavior and a QuackBehavior to which it delegates flying an quacking

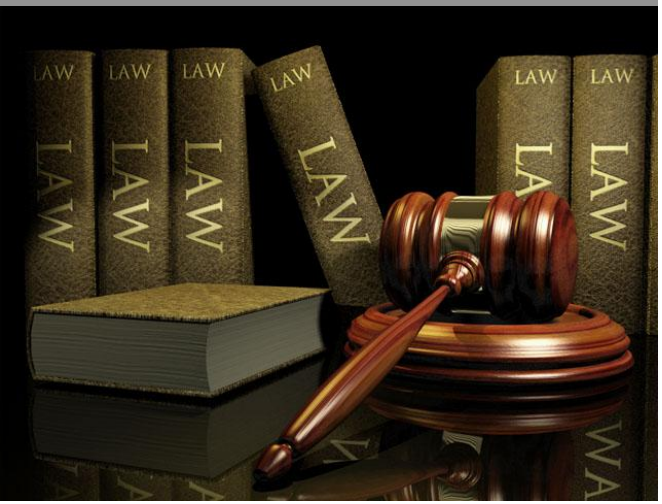


Composition

Instead of inheriting behavior, the duck get their behavior by being composed with the right behavior object

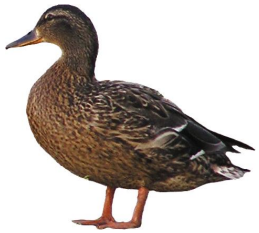
Favor Composition over Inheritance

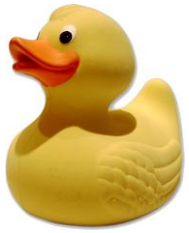
The Constitution of Software Architects



- Encapsulate what varies.
- Program through an interface not to an implementation
- Favor Composition over Inheritance
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????

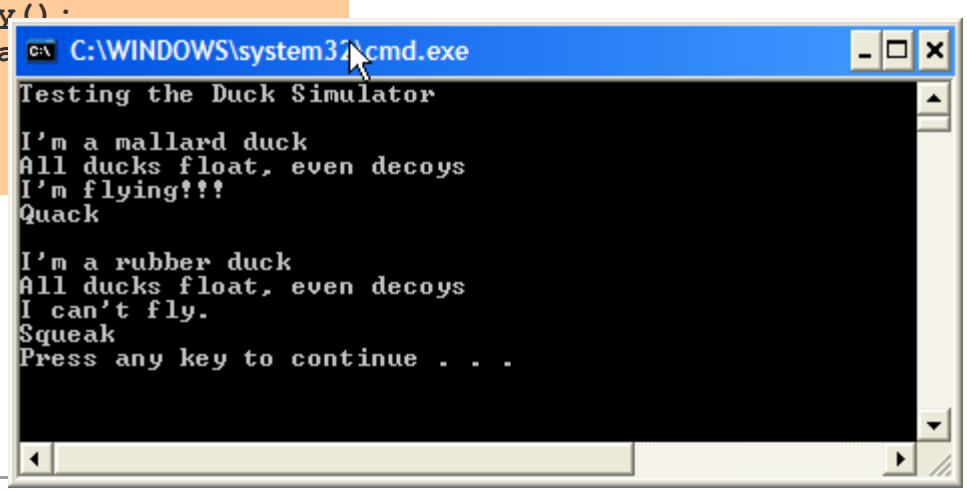
Testing the Duck Simulator





```
int main(){
  cout << "Testing the Duck Simulator"
  << endl << endl;

  Duck *mallard = new MallardDuck();
  mallard->display();
  mallard->swim();
  mallard->performFly();
  mallard->performQuack();

  cout << endl;

  Duck *rubberduck = new RubberDuck();
  rubberduck->display();
  rubberduck->swim();
  rubberduck->performFly();
  rubberduck->performQua

  return 0;
}
```

The mallard duck inherited **`performQuack()`** method which delegates to the object **`QuackBehavior`** (calls **`quack()`**) on the duck's inherited **`quackBehavior`** reference



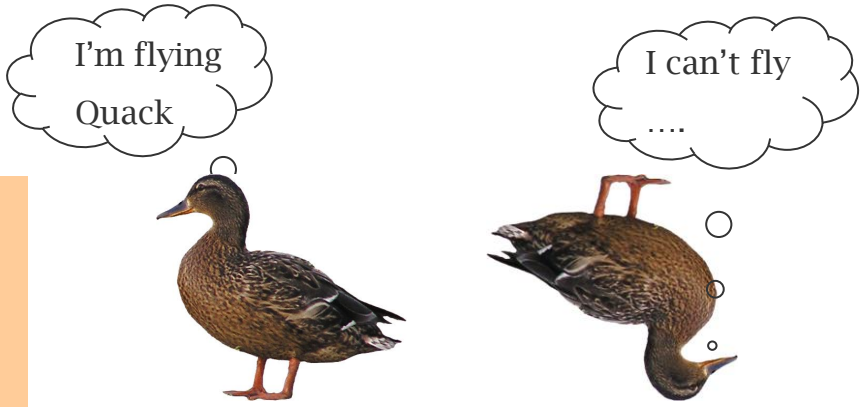
Shooting Duck Dynamically

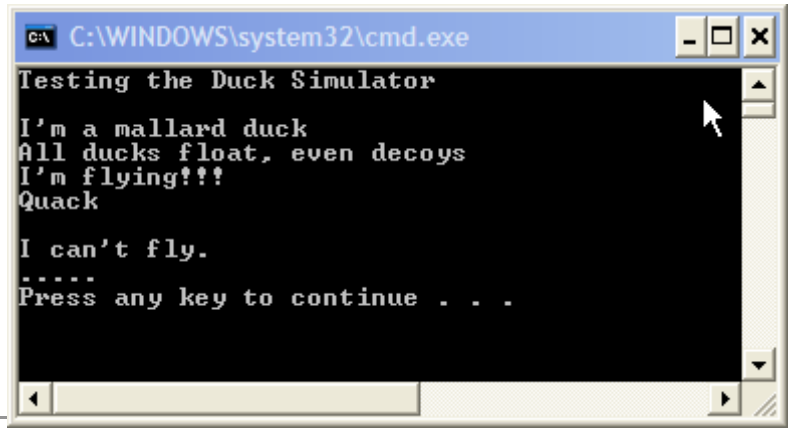
Shooting Duck Dynamically

Duck
-flyBehavior : FlyBehavior -quackBehavior : QuackBehavior
+performQuack() +swim() +display() +performFly() +setFlyBehavior() +setQuakBehavior()

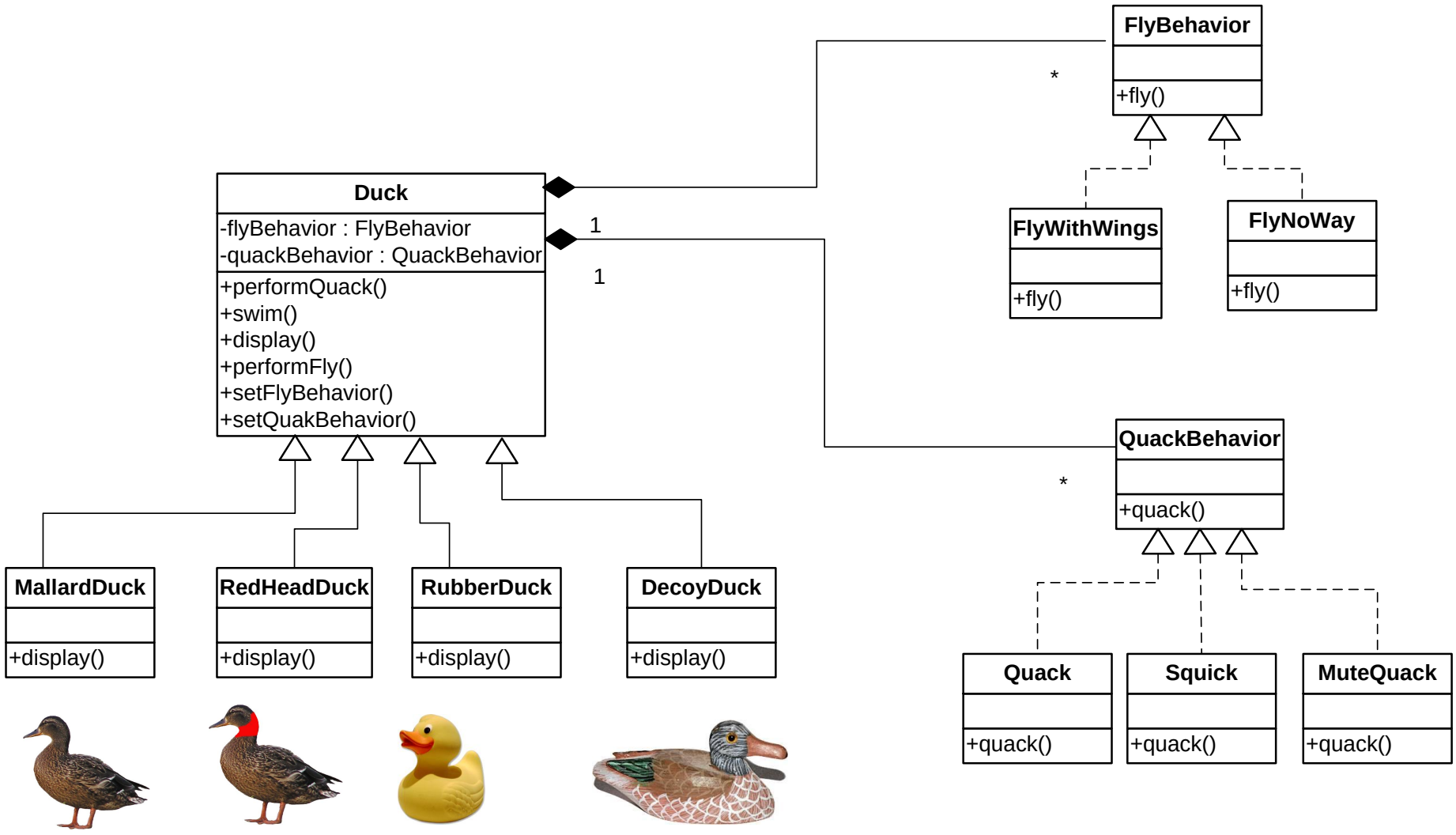
```
void Duck::setFlyBehavior(FlyBehavior *fb){
  flyBehavior = fb;
}
void Duck::setQuackBehavior(QuackBehavior *qb){
  quackBehavior = qb;
}
```

```
int main(){

  Duck *mallard = new MallardDuck();
  mallard->display();
  mallard->swim();
  mallard->performFly();
  mallard->performQuack();

  cout << endl;

  mallard->setFlyBehavior(new FlyNoWay());
  mallard->setQuackBehavior(new MuteQuack());
  mallard->performFly();
  mallard->performQuack();

  return 0;
}
```

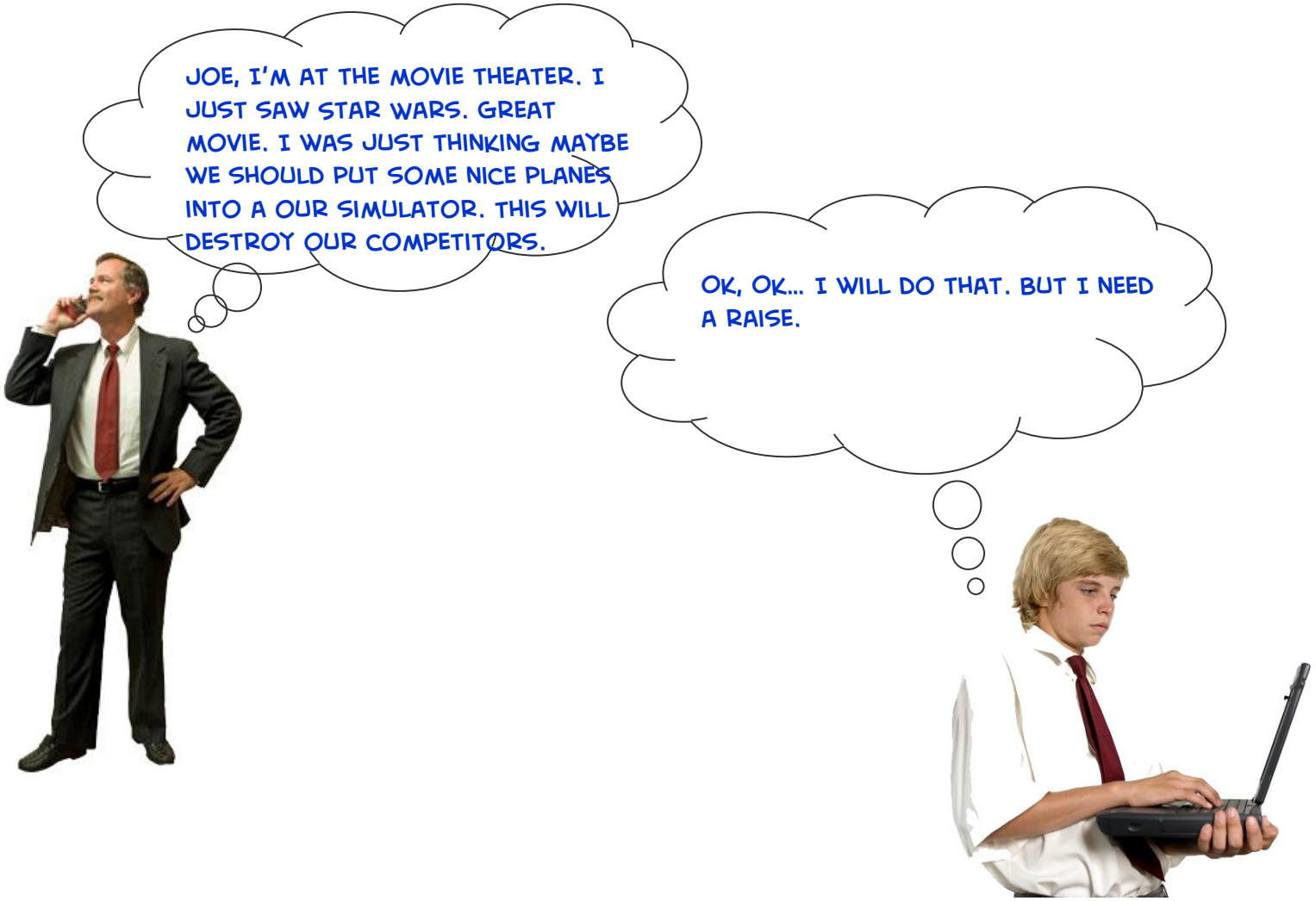




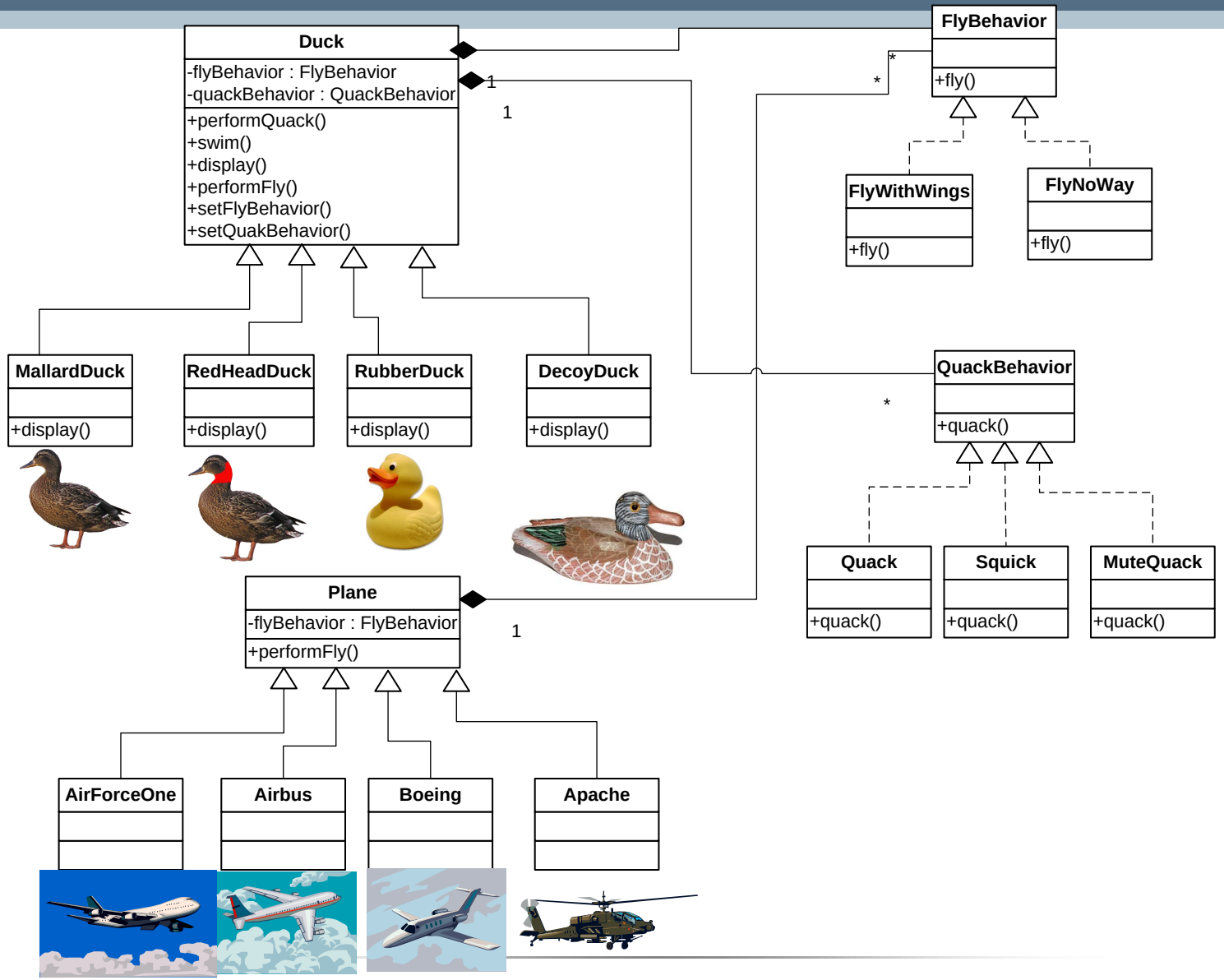
The Big Picture



Yet Another Change



Behavior Reuse




Congratulations !!!
This is your first design pattern called
STRATEGY