Design Principles

Encapsulate What Varies

Encapsulate What Varies

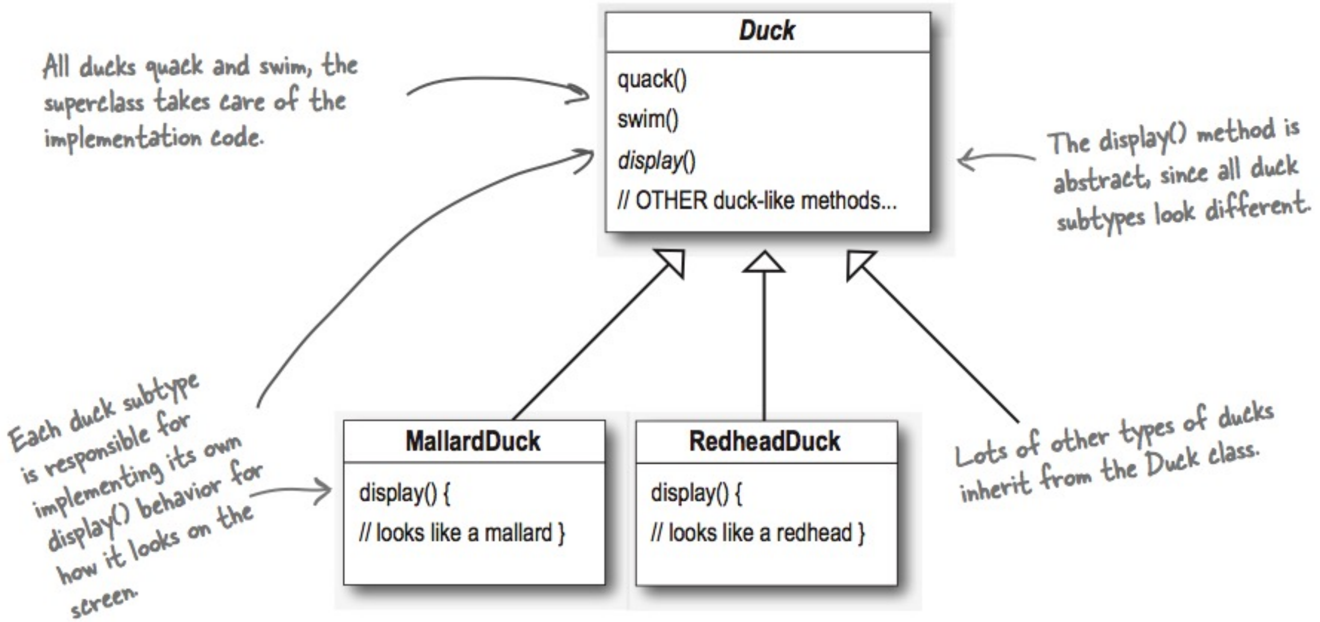
Design Principle:
Encapsulate what varies

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.

Case Study: Duck Hunt Simulation Game

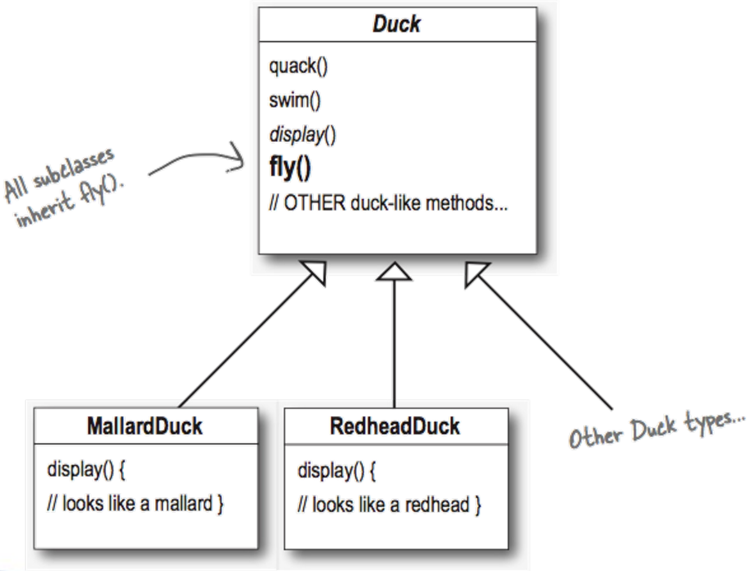
- Suppose a company is building a duck hunt simulation game …

Modeling Ducks



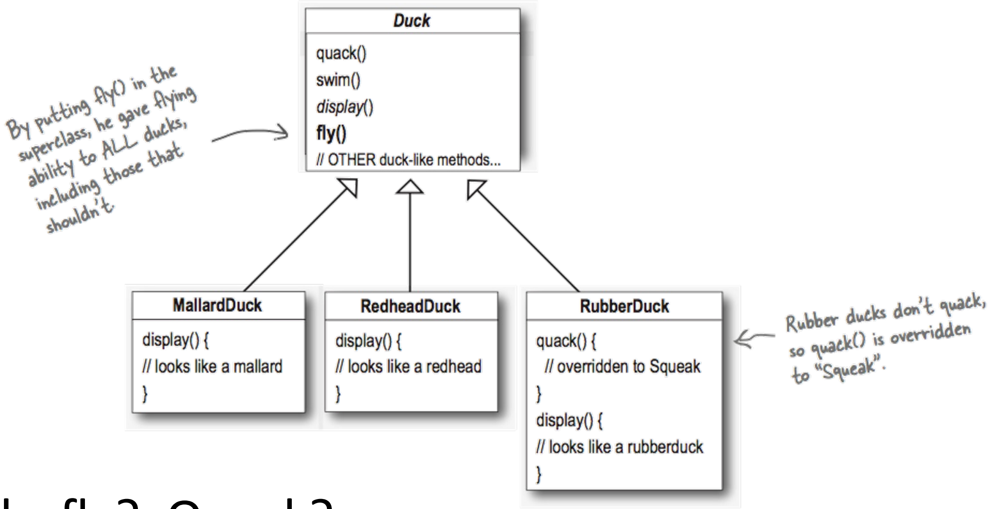
Making Ducks Fly

- Need to add code to make ducks fly in the simulation …



Making Ducks Fly: Problem

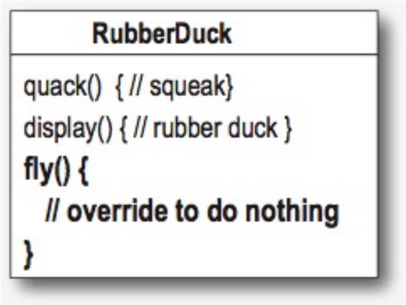
- The company decides to add a `RubberDuck` to the game as an Easter egg



- Should all ducks fly? Quack?

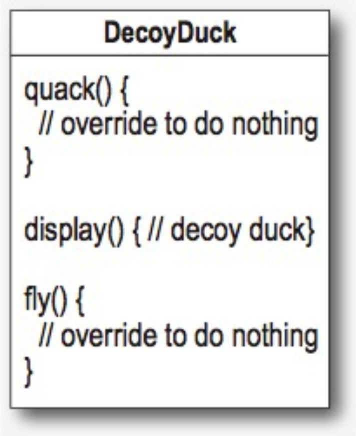
Making Ducks Fly: Solution

- Solutions?
  - We could override the `fly` method in `RubberDuck` to do nothing …



Making Ducks Fly: Solution

- This isn't a horrible solution, but what about when we add a `DecoyDuck` class for hunters in the game?
  - And decoys should neither quack nor fly …

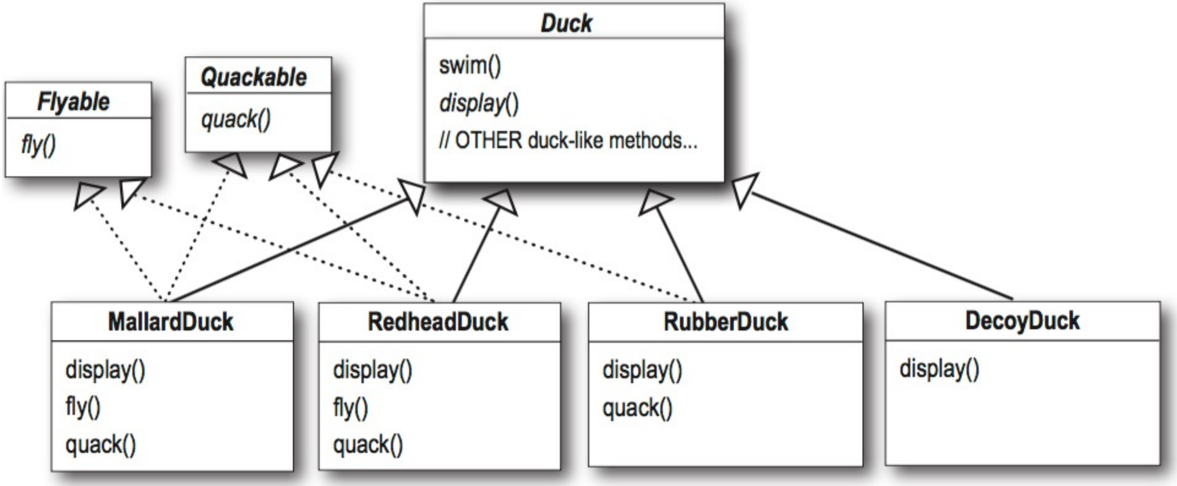


Making Ducks Fly: Solution

- Inheritance probably isn't the answer in this case
- As we add new types of ducks, we will have to examine and perhaps override fly and quack for each new class
- We will likely have a lot of duplicate code in the subclasses
- How many different ways can a duck really fly?

Making Ducks Fly: Solution

- Another option: use an abstract class (or in Java, an interface)



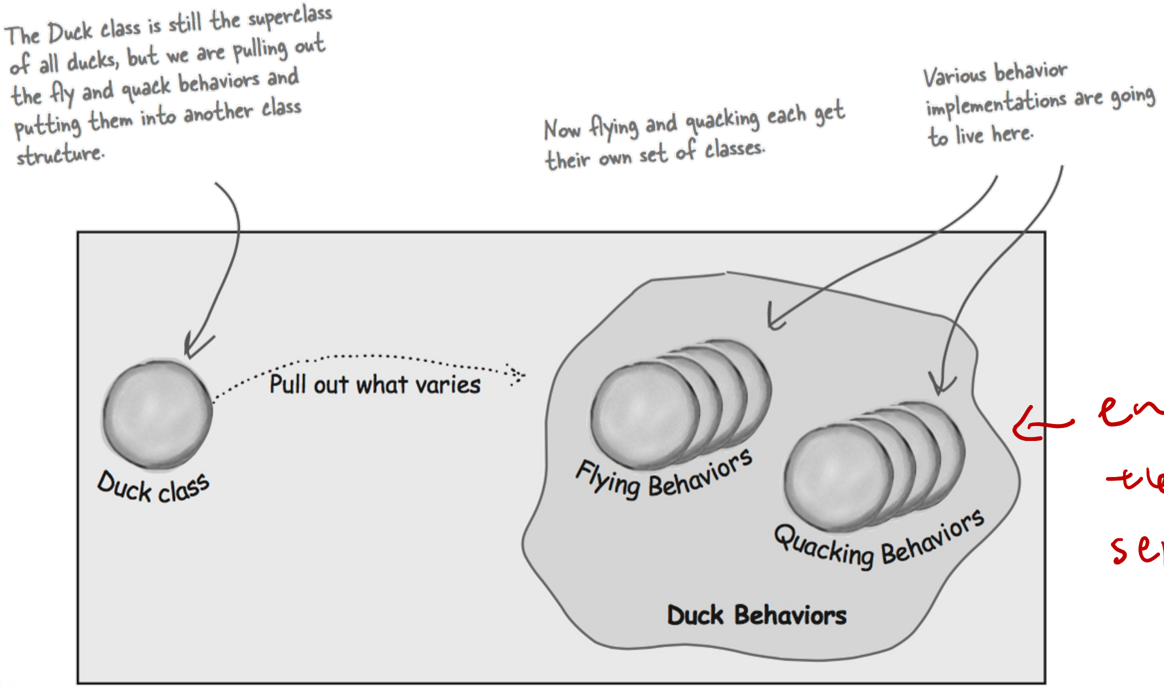
- Has this solved the problem?

Making Ducks Fly: Solution

- We know:
  - New ducks will be added to the system as time passes
  - Duck behaviours differ from duck type to duck type
  - Certain behaviours are not appropriate for all ducks

- We need to keep these considerations in mind when designing the system …

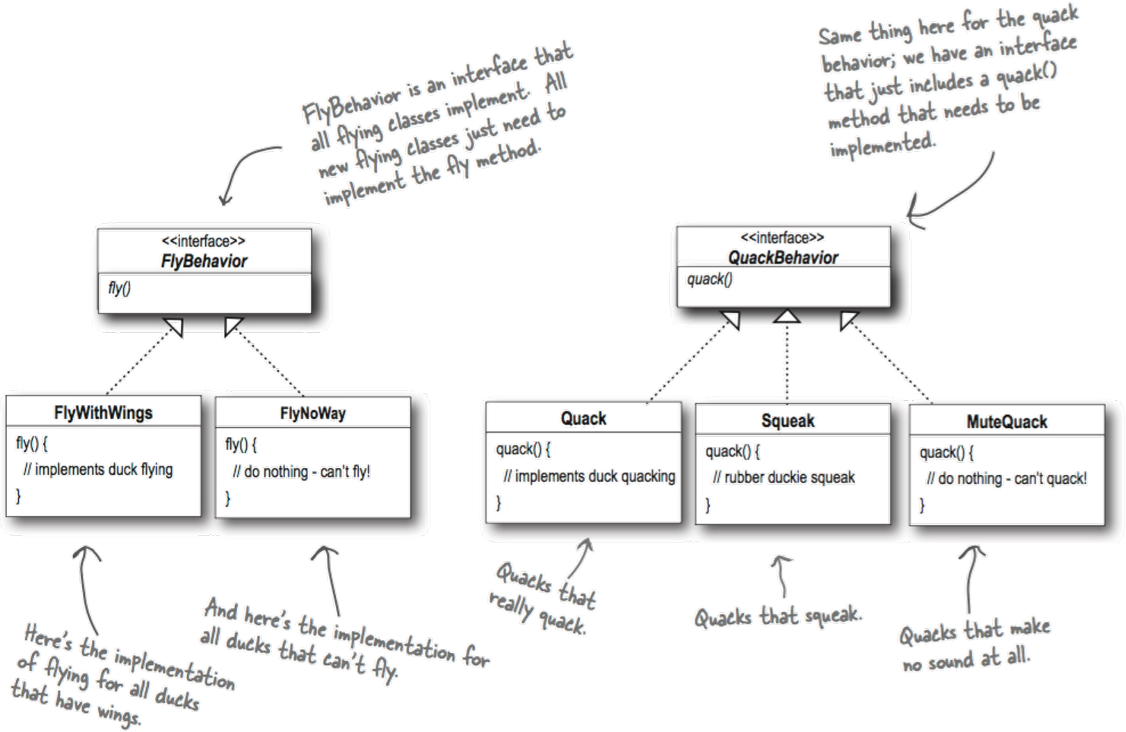
Encapsulate What Varies



← encapsulate them seperately.

Encapsulate What Varies

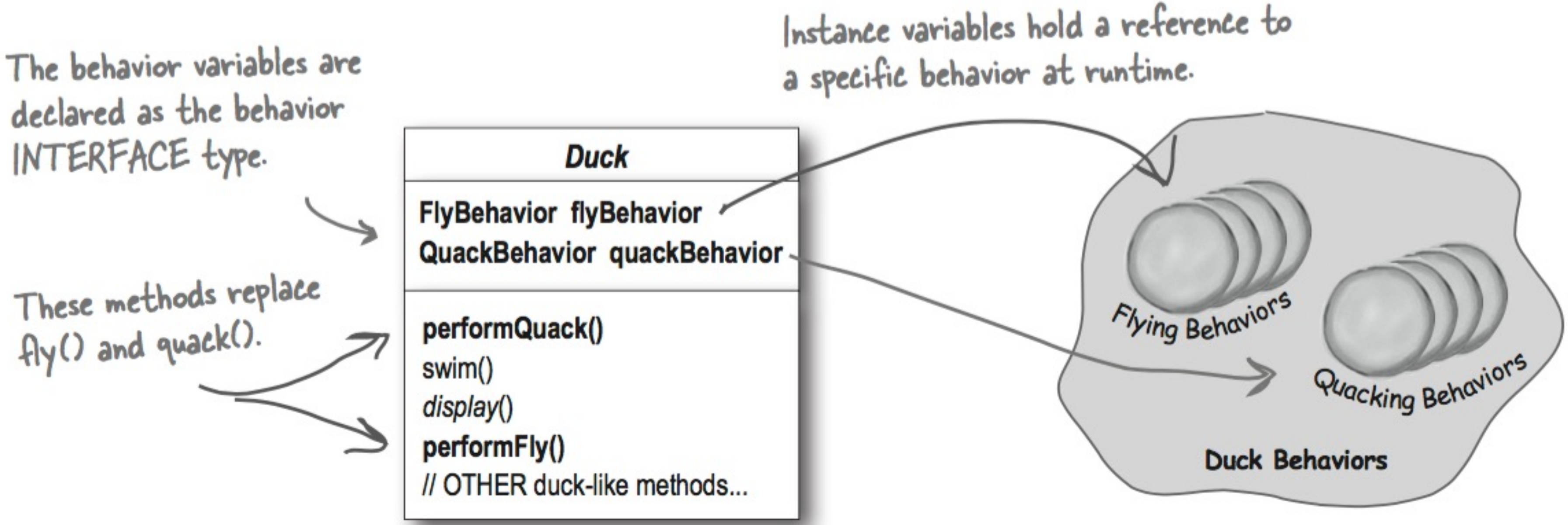
- We create an interface/abstract class for flying and quacking behaviour



Encapsulate What Varies

- Now the functionality that might change between subclasses has been encapsulated in its own set of classes
- We will store the flying and quacking behaviours of a duck as instance variables in the `Duck` class
- A duck will delegate its flying and quacking behaviours, rather than implementing them itself

Encapsulate What Varies



Encapsulate What Varies

```
class FlyBehaviour
{
   public:
      virtual void fly() = 0;
};
class FlyWithWings : public FlyBehaviour
{
   public:
      virtual void fly() {
        cout << "I’m flying with my wings!" << endl;
      }
};
class FlyNoWay : public FlyBehaviour
{
   public:
      virtual void fly() {
         cout << "I can’t actually fly!" << endl;
      }
};
```

just to make sure
fly() is the must implemented
thing.

Encapsulate What Varies

```
class Duck {
   public:
      Duck() { }
      ~Duck() {
         delete this->_flyBehaviour;
         delete this->_quackBehaviour;
      }
      void performFly() {
         this->_flyBehaviour->fly();
      }
      void performQuack() {
         this->_quackBehaviour->quack();
      }

   protected:
      FlyBehaviour* _flyBehaviour;
      QuackBehaviour* _quackBehaviour;
};
```

Encapsulate What Varies

```
class MallardDuck : public Duck
{
   public:
      MallardDuck() {
         this->_quackBehavior = new Quack();
         this->_flyBehavior = new FlyWithWings();
      }
};
```

Encapsulate What Varies

- Benefits:
  - Eliminated code duplication
  - Other types of objects can reuse the fly and quack behaviours
  - Can easily add / modify existing behaviours without necessarily (or heavily) modifying our duck classes
  - Can dynamically change behaviours at run-time