# Zeroing in on the problem…

* Inheritance hasn’t worked out very well, since duck behavior keeps changing
* Not appropriate for all subclasses to share all behaviors
* Flyable and Quackable sounded promising, but no code reuse via Interfaces
* Luckily there is a design principle for just this situation:  
    
  **Design Principle**  
    
  Identify the aspects of your application that vary and separate them from what stays the same. (Composition, is one solution to this.)
* 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 result? Fewer unintended consequences from code changes and more flexibility in your systems!

# Separating what changes from what stays the same…

* Create two sets of classes, totally apart from Duck, one for fly and one for quack. Yes, I know, we are using objects for behaviors! Is this heresy? No!
* Each set of classes will hold all the implementations of their respective behaviors. For instance, we might have one class that implements quacking, another that implements squeaking, and another that implements silence.
* We know that fly() and quack() are the parts of the Duck class that vary across ducks. To separate those behaviors from the Duck class we’ll pull both methods out of the Duck class (or Interfaces) and create a new set of interfaces and concrete classes to represent each behavior.

# Goals for Designing the Duck Behaviors

* Keep things flexible
* Assign the behaviors to instances of Duck
* Change the behavior of a duck dynamically at runtime
* Include behavior setter methods in the ducks to do this
* Ducks won’t need to know any of the implementation   
  details for their own behaviors
* Use a design principle:  
    
  **Design Principle**  
    
  Program to an interface, not an implementation.  
  (Will DIP come to the rescue?)