DESIGN PATTERNS

Young B. Park (ybpark@dankook.ac.kr)

POLYMORPHISM SAMPLE CODE

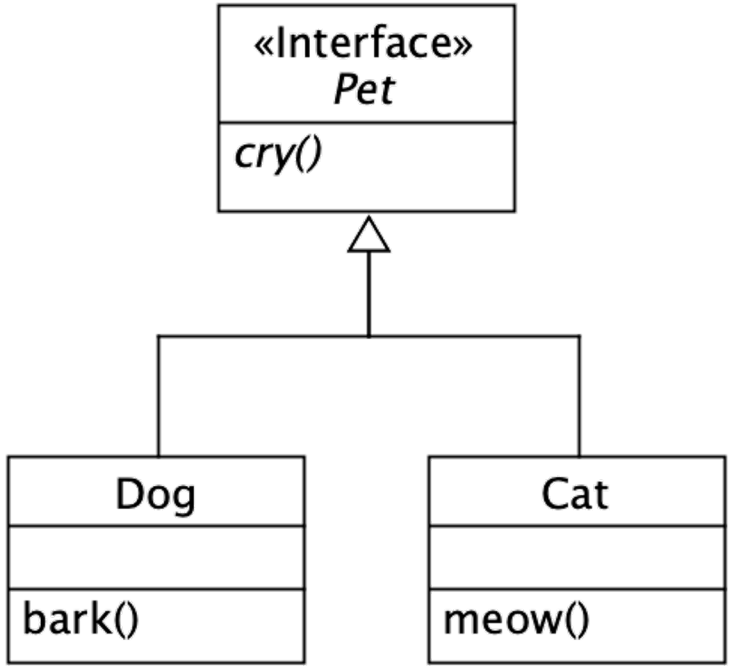
```
1.  public interface Pet {
2.      public void cry();
3.  }

4.  public class Dog implements Pet{
5.      public void cry(){
6.          bark();
7.      }
8.      private void bark() {
9.          System.out.println("멍멍!");
10.     }
11. }
```

```
12. public class Cat implements Pet{
13.     public void cry(){
14.         meow();
15.     }
16.     private void meow(){
17.         system.out.println("야옹!");
18.     }
19. }
```

POLYMORPHISM

```
1.  public class PetMain() {
2.      public static void main(String[] args) {
3.          Pet myPet;
4.          myPet = new Dog();
5.          myPet.cry();
6.          myPet = new Cat();
7.          myPet.cry();
8.      }
9.  }
```

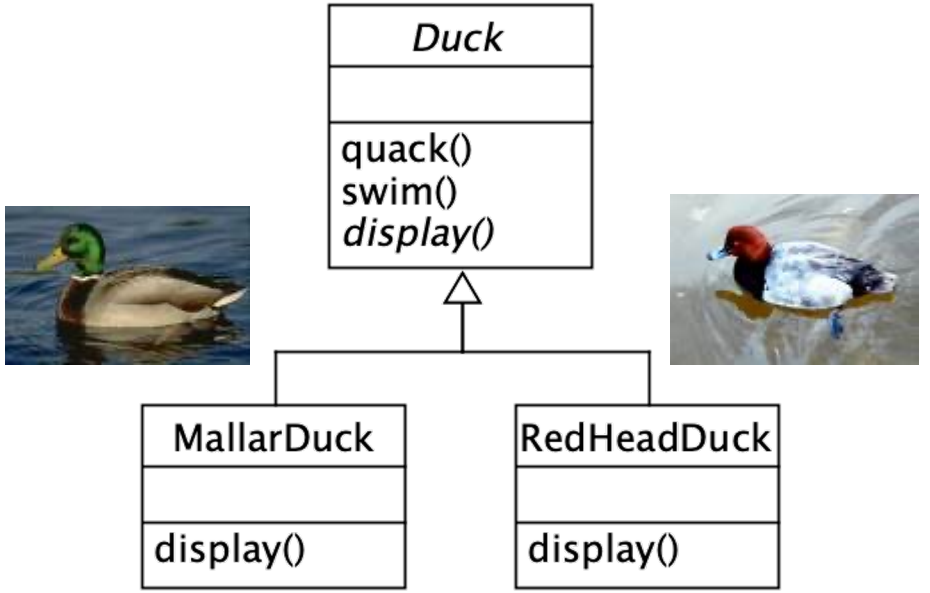


STRATEGY PATTERN

1. Goal: make code more understandable and/or more flexible.
2. Design Critiques 실습

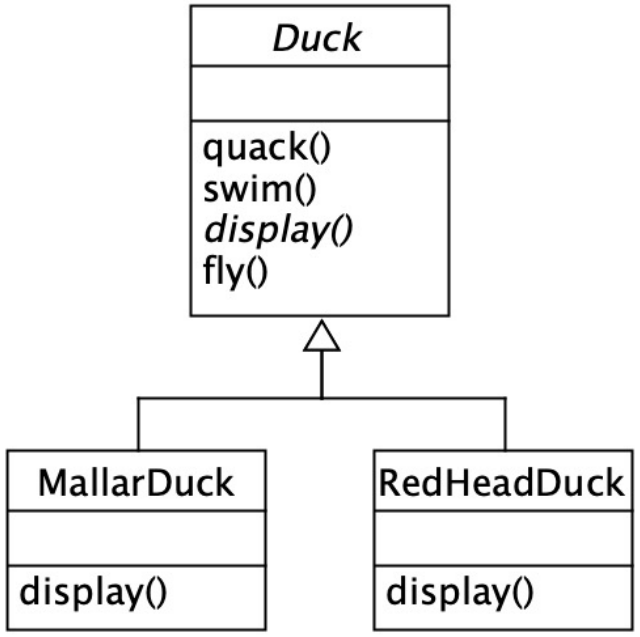
SIMPLE SIMULATION OF DUCK

- All duck quack & swim - Code reuse
  - quack()
  - swim()
- But they are look different
  - *display*()
- Inheritance – code reuse
- Realization – push to implements
- Override – refused bequest

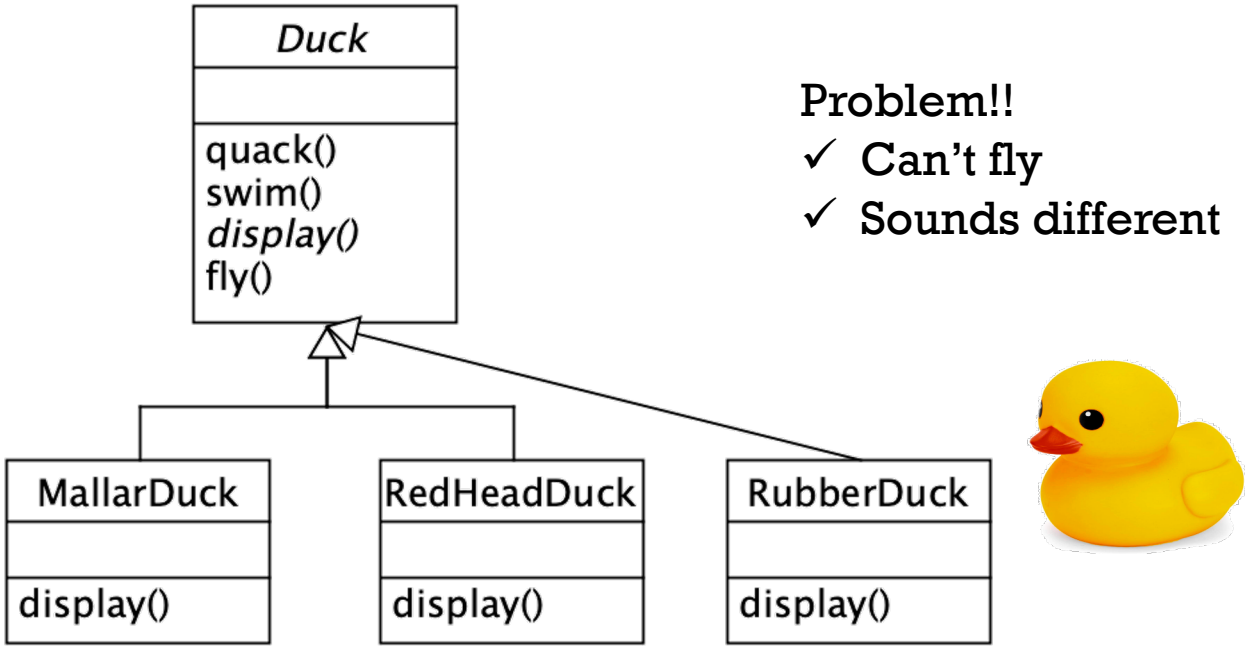


CHANGES...

- Adding function



- Extending model

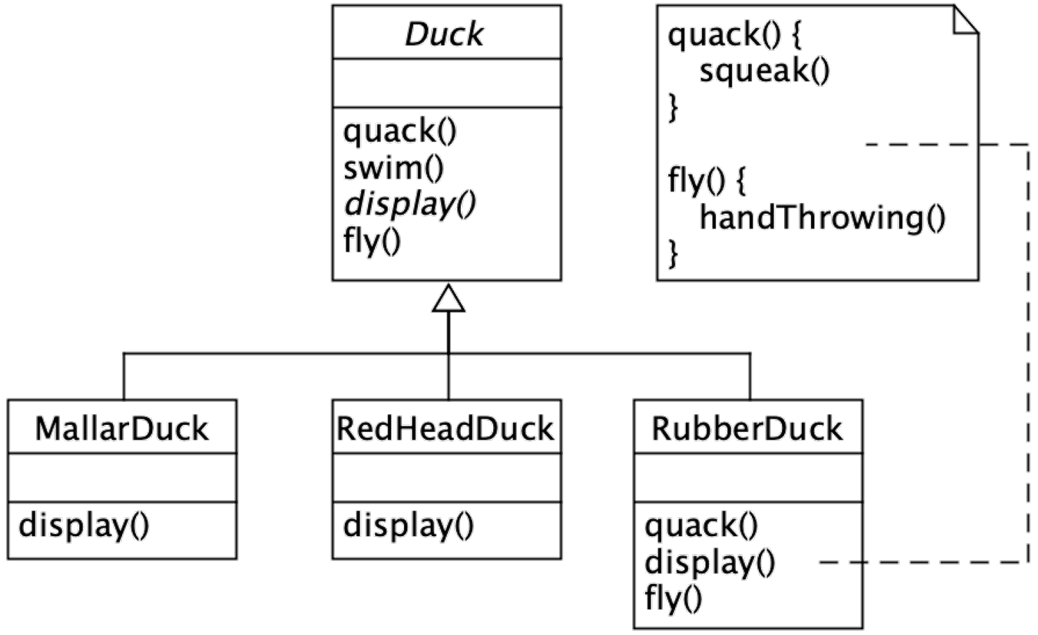


Problem!!

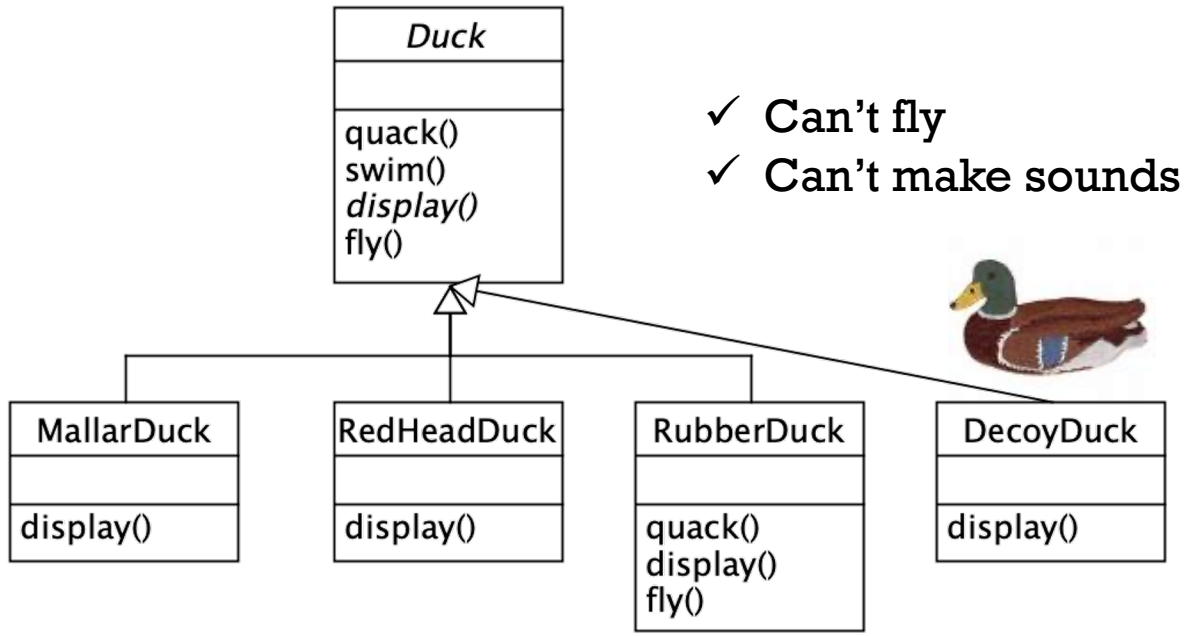
- ✓ Can't fly
- ✓ Sounds different

MORE CHANGES...

- Override can be a solution… But!



- Expecting more !!!

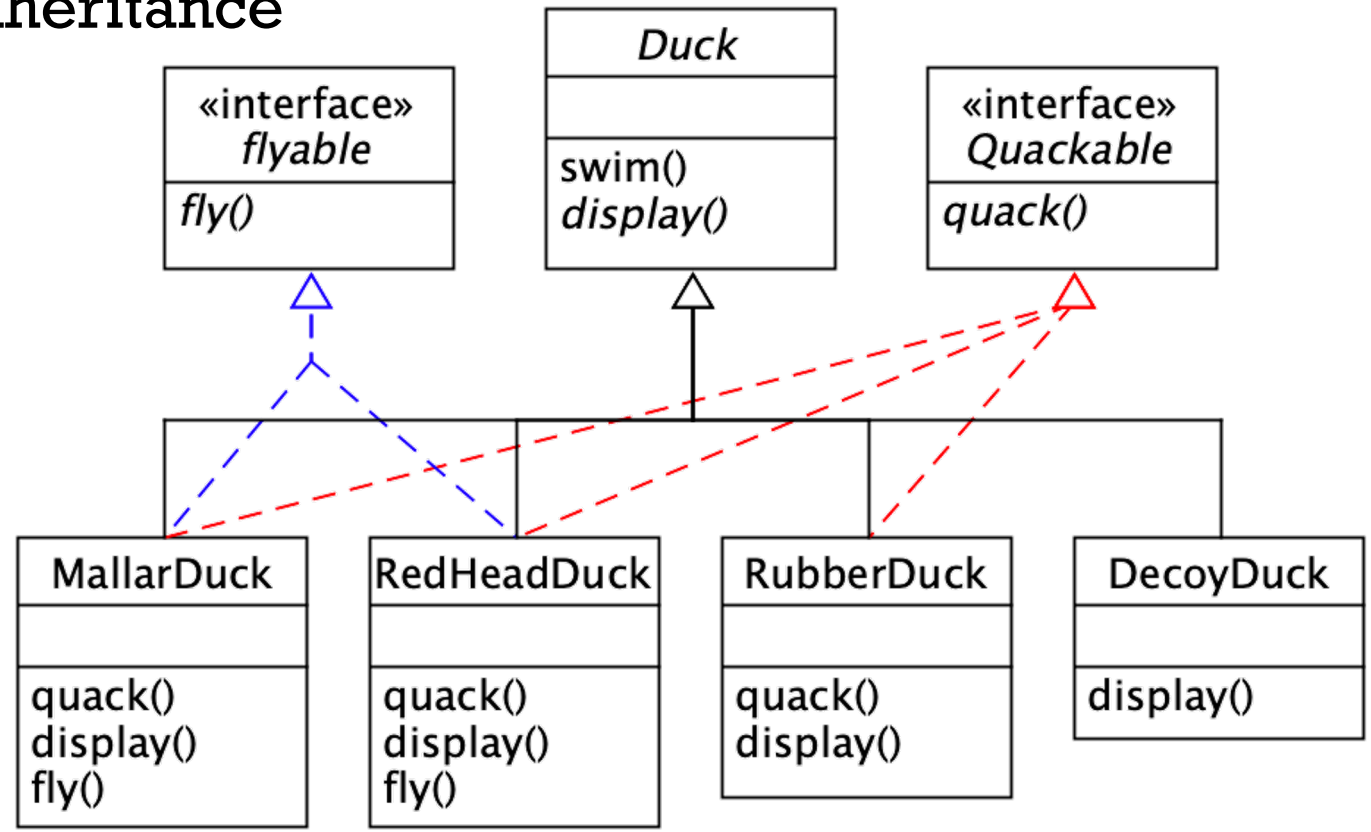


✓ Can't fly
✓ Can't make sounds

- cf. Interface-segregation Principal

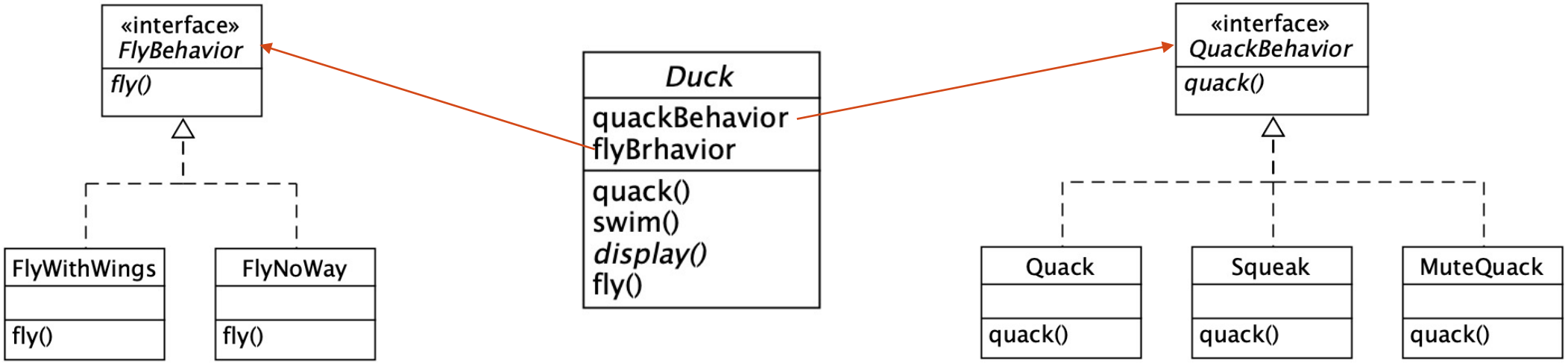
BAD CHOICE !

- Multiple Inheritance



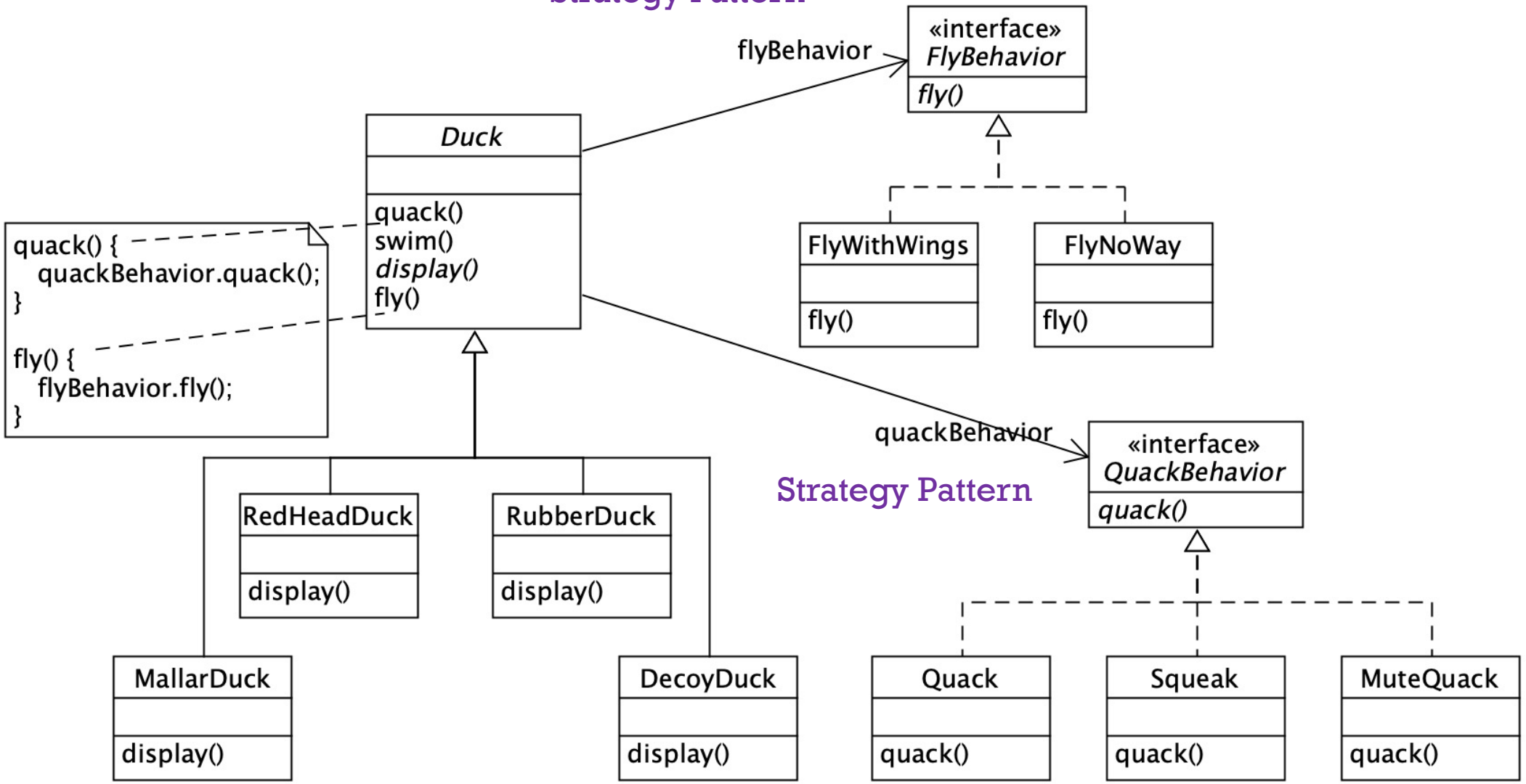
SEPARATING CHANGING BEHAVIORS

- Identify the aspects of your application that vary and separate them form what stays same (cf. Single Responsivity Principal)



- Program to an interface, not an implementation (cf. Dependency Inversion Principal)

Strategy Pattern



- Favor composition over inheritance

STRATEGY PATTERN

Name!

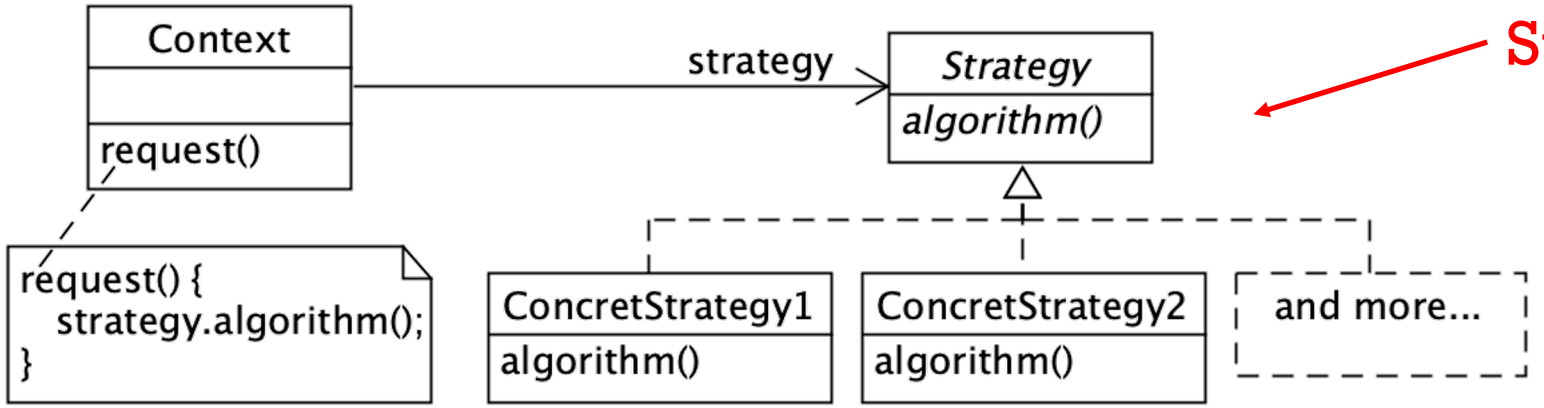
Intent!

+ Consequence!

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Different algorithms are implemented directly(hard-wired), it is impossible to add/change algorithms independently from the context. Wants more flexibility and want to exchange algorithm at run time

Motivation!

Structure!



UNDERSTANDING CONSEQUENCE

Pros (+)

- Avoids compile-time implementation dependencies.
- Provides a flexible alternative to subclassing.
- Avoids conditional statements for switching between algorithms.

Cons (–)

- Can make the common Strategy interface complex.
- Requires that clients understand how strategies differ.
- Introduces an additional level of indirection.