DESIGN PATTERNS

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

POLYMORPHISM SAMPLE CODE

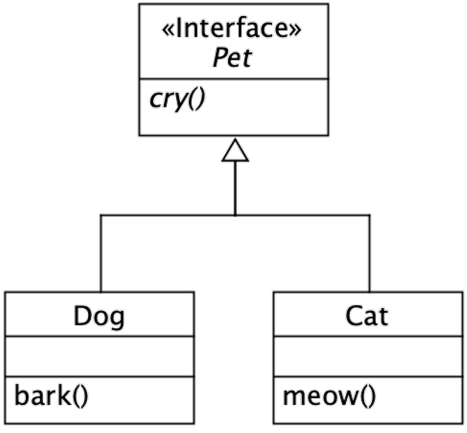
```
1.  public interface Pet {
2.      public void cry();
3.  }

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

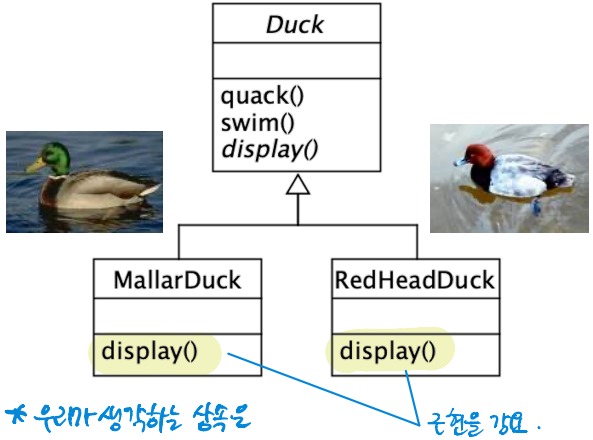
SIMPLE SIMULATION OF DUCK

- All duck quack & swim - Code reuse
  - quack()
  - swim()
- But they are look different
  - *display*()

- Inheritance – code reuse
- Realization – push to implements
- Override – refused bequest

상속을 거부한다.
내가 새로 구현할거야!!

interface/abstract class는 상속을 강요



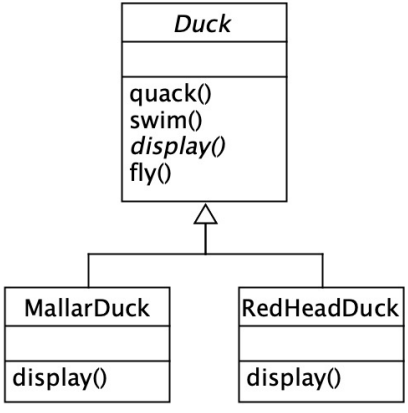
* 우리가 생각하는 상속은
Max 3단계.
대부분 2단계일 가능.

구현을 강요.

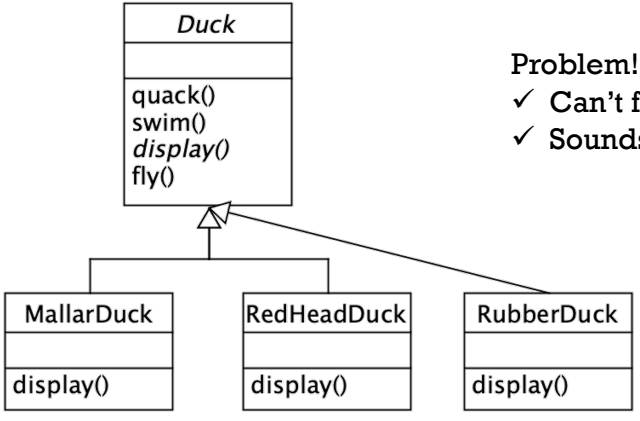
head first serize → Design Pattern 교재
강점 설명

CHANGES...

- Adding function



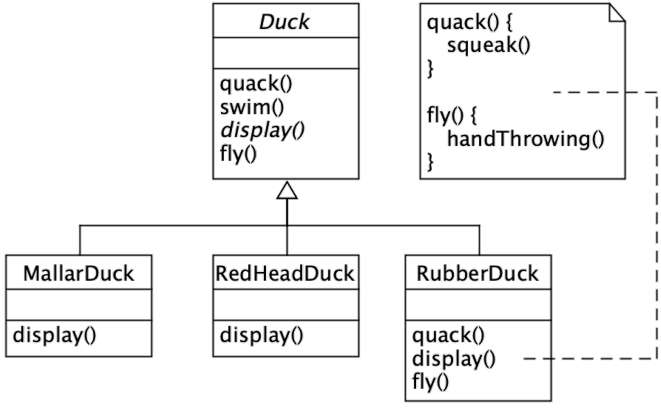
- Extending model



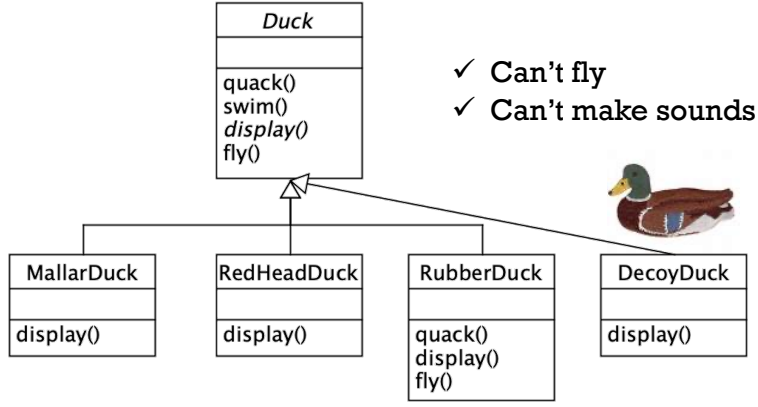
Problem!!
- ✓ Can't fly
- ✓ Sounds different

MORE CHANGES…

- Override can be a solution… But!



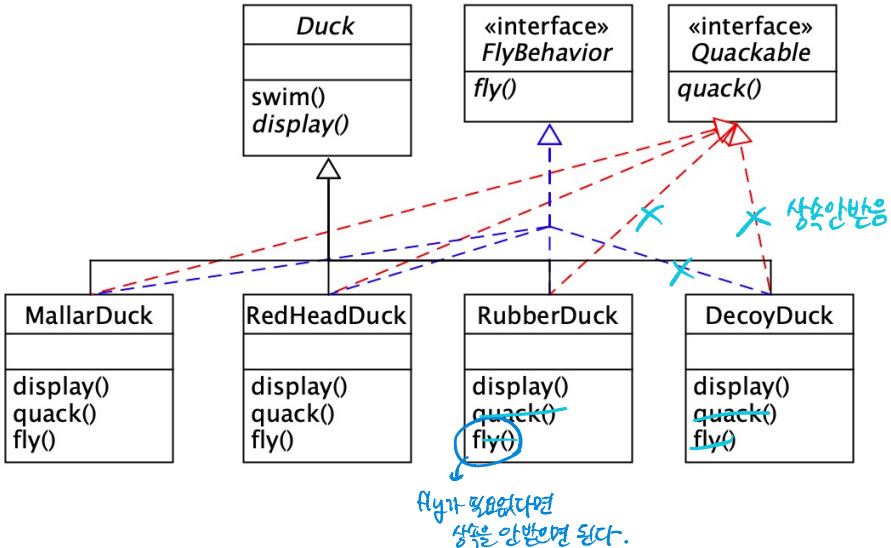
- Expecting more !!!



- cf. Interface-segregation Principal

BAD CHOICE !

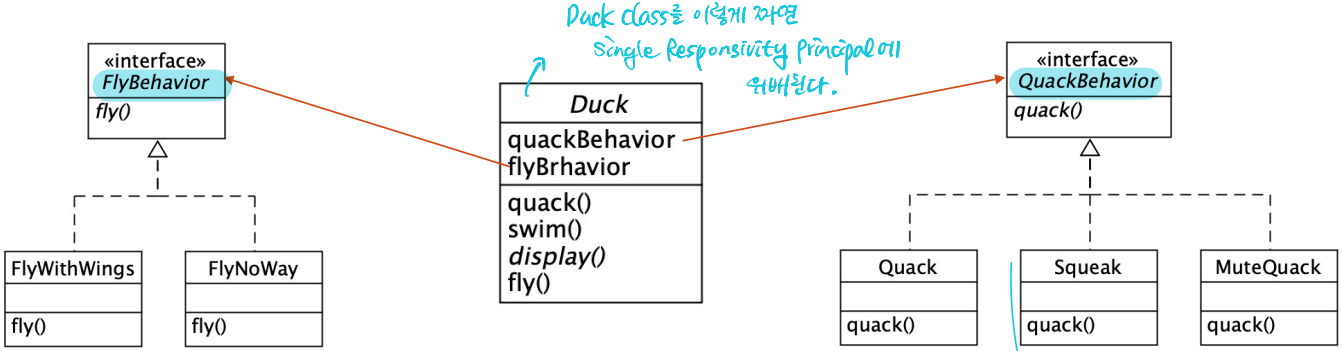
- Multiple Inheritance



SEPARATING CHANGING BEHAVIORS

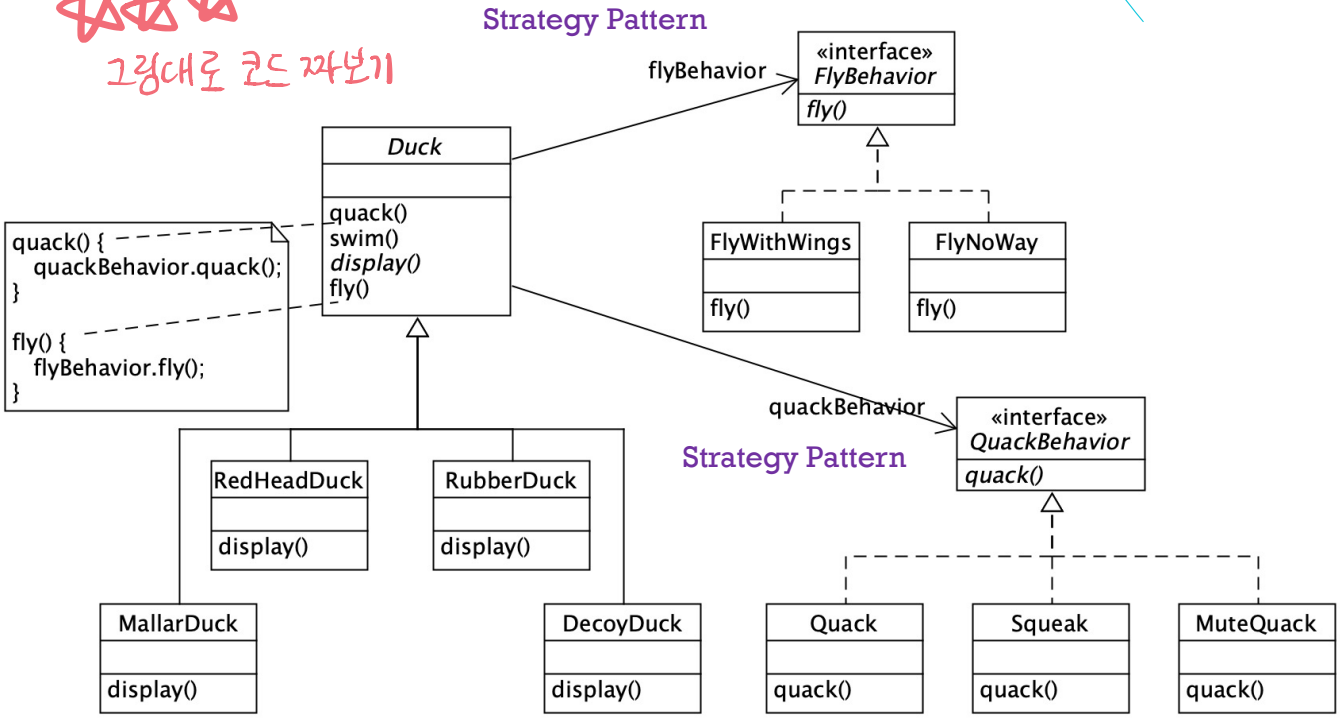
변하는부분 / 불변부분을 구분하였다. ⇒ 확장성에 최적화된 구조.

- 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)

그림대로 코드 짜보기



- Favor composition over inheritance

① 디자인패턴에는 이름이 있어야한다.

Name! 전문가들의 단어.

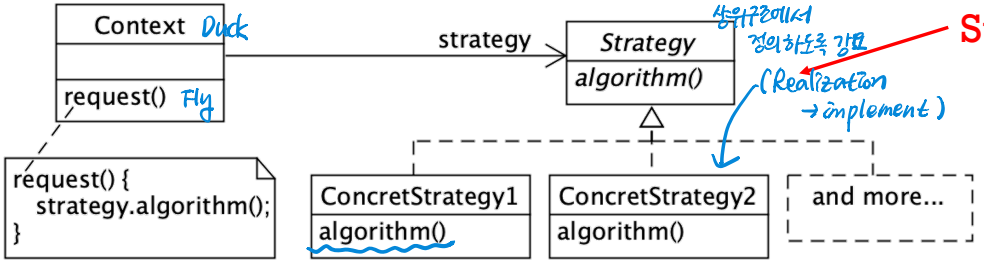
STRATEGY PATTERN

Intent! ② 정의

+ 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! ③ 움직



Structure! ④

다양한 구조를 선택할 수 있도록 하는 것 ⇒ Strategy

Pattern

UNDERSTANDING CONSEQUENCE

Pros (+)

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

상황에 따른 알고리즘 사용

Cons (–) ☆☆☆

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

Duck → Quack
→ Fly 이해되요.