디자인 패턴 소개

이관우

kwlee@hansung.ac.kr

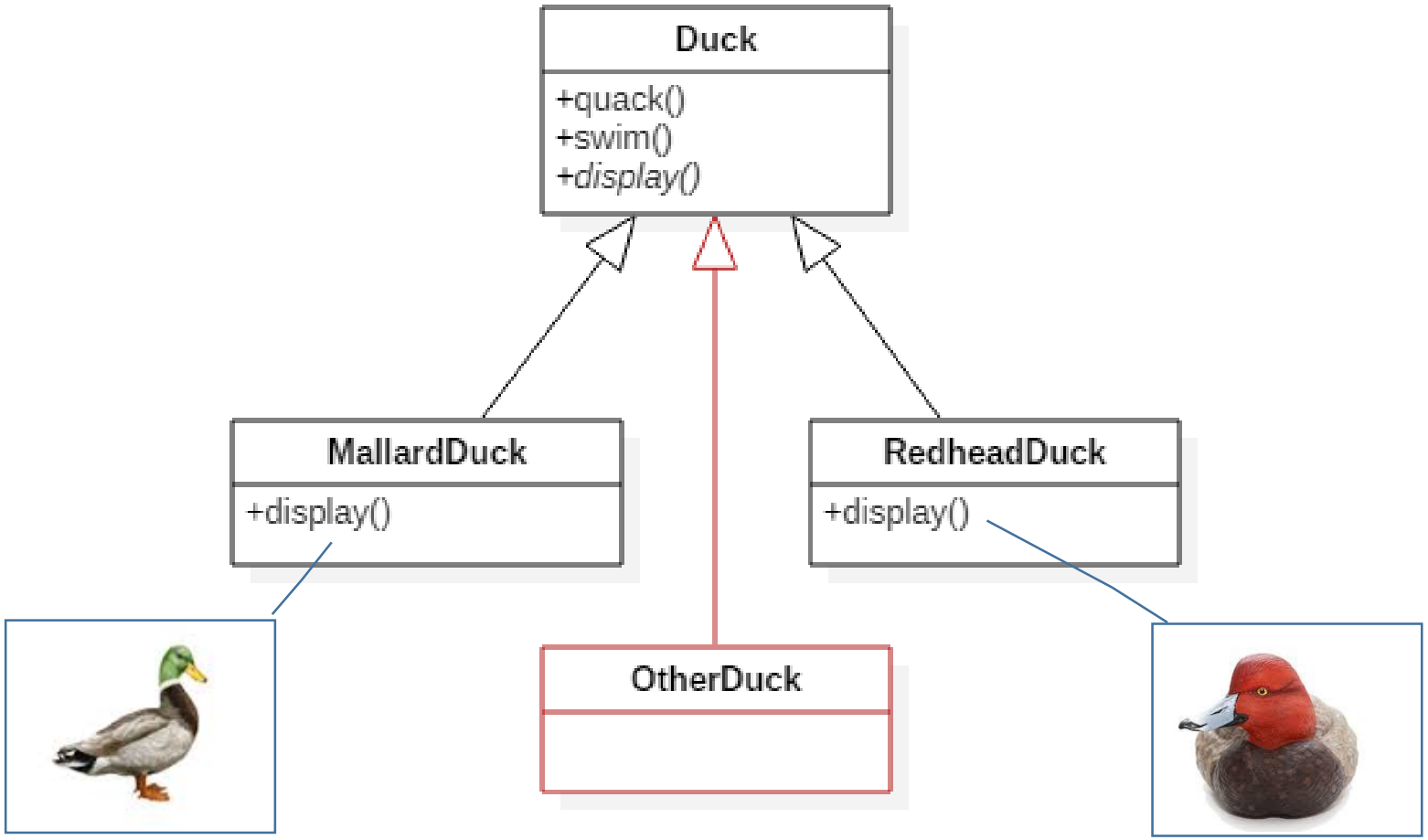
학습 목표

- 객체지향 기초 (상속, 인터페이스, 캡슐화, 다형성)의 기본 개념을 이해한다.
- 객체지향 디자인 문제를 분석한다.
- 객체지향 디자인 원리를 이해한다.
- 디자인 패턴의 역할을 이해한다.

SimUDuck 애플리케이션

- SimUDuck – 오리 연못 시뮬레이션 게임
  - 헤엄 치고, 소리 내는 다양한 오리를 보여줌

SimUDuck - 객체지향 설계



https://github.com/kwanulee/DesignPattern/tree/master/strategy/SimUDuckApp.v0

SimUDuck 예제 코드

1. Github 사이트에서 해당 예제코드 폴더를 찾아서 다운로드
   - 주소
     - https://github.com/kwanulee/DesignPattern
   - 다운로드 방법
     - https://github.com/kwanulee/DesignPattern/blob/master/Readme.md

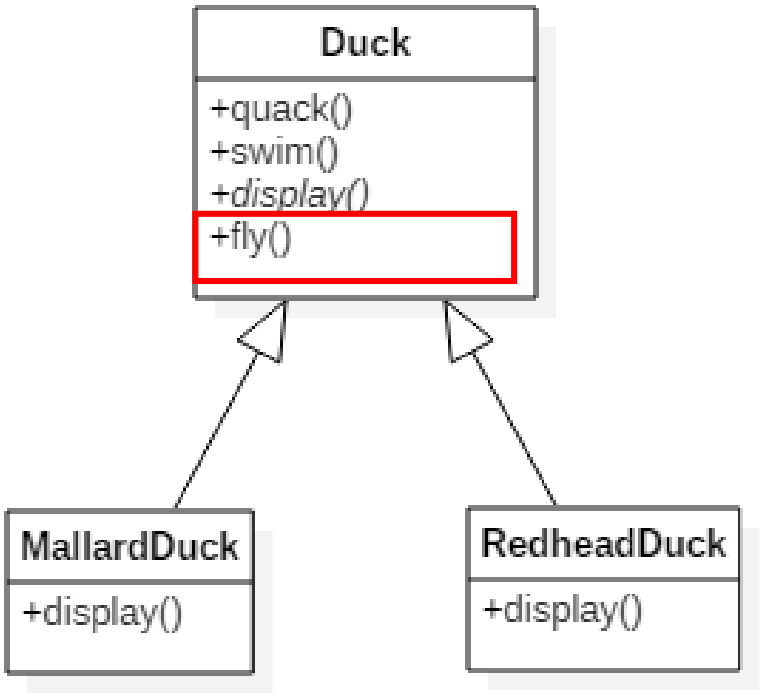
2. 다운로드한 zip파일의 압축을 품
3. Eclipse IDE 실행
   - [File]-[Open Project from File System..] 메뉴 선택
   - 압축을 푼 폴더를 선택

변경 요구 사항

오리 연못 시뮬레이션 게임 경쟁사들 간의 경쟁 심화

현재 시스템에서 오리들이 날 수 있도록 변경

간단한 해결책 (Revision 1)



https://github.com/kwanulee/DesignPattern/tree/master/strategy/SimUDuckApp.Revision1

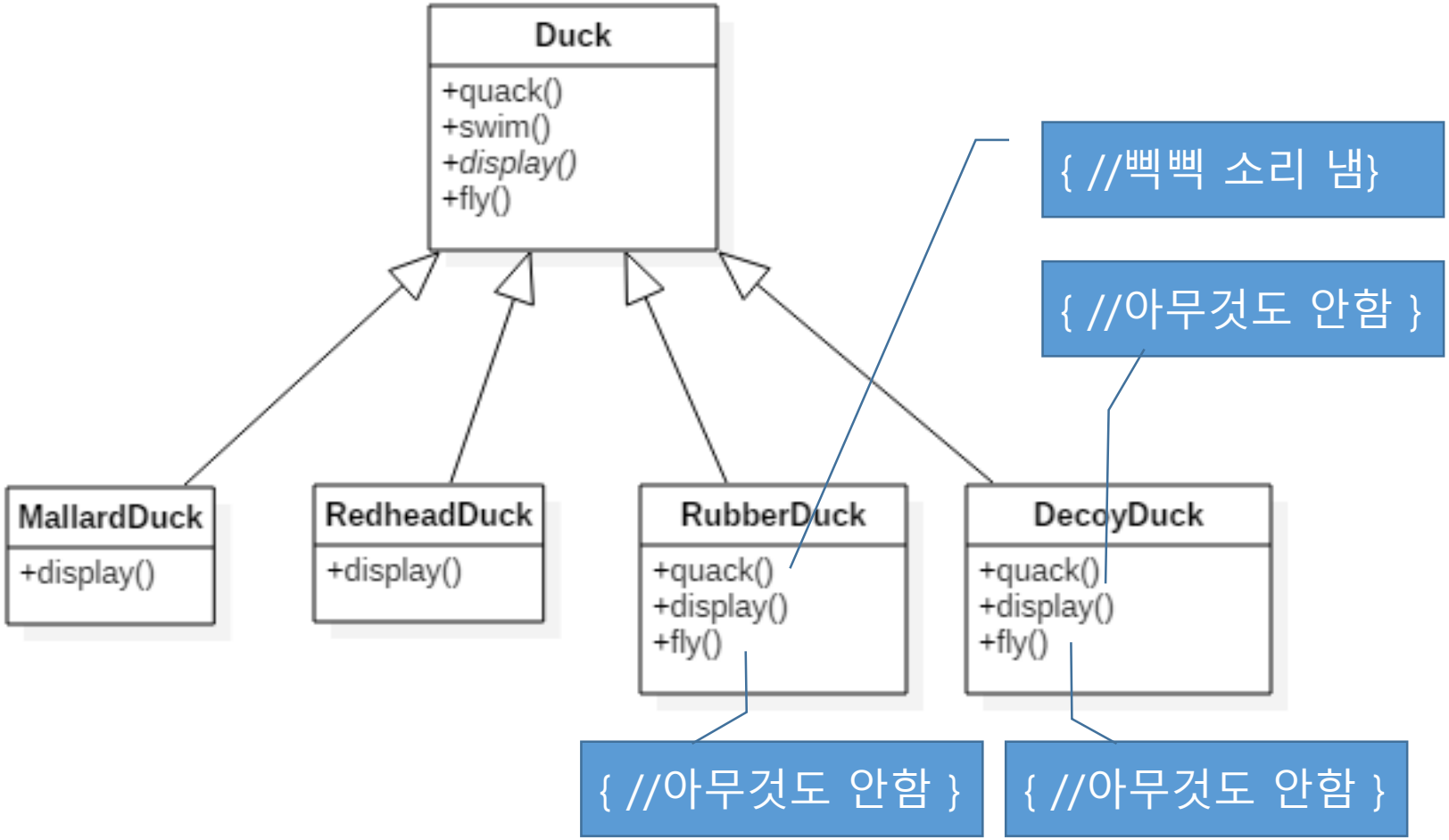
다음 경우에 무엇이 문제일까요?

Rubber Duck
(고무오리)

Decoy Duck
(모형 오리)

문제 해결 방안-상속 & 오버라이딩 사용 (Revision 2)

https://github.com/kwanulee/DesignPattern/tree/master/strategy/SimUDuckApp.Revision2



토의1

- Revision 2의 장단점에 대해서 나열 하시오.

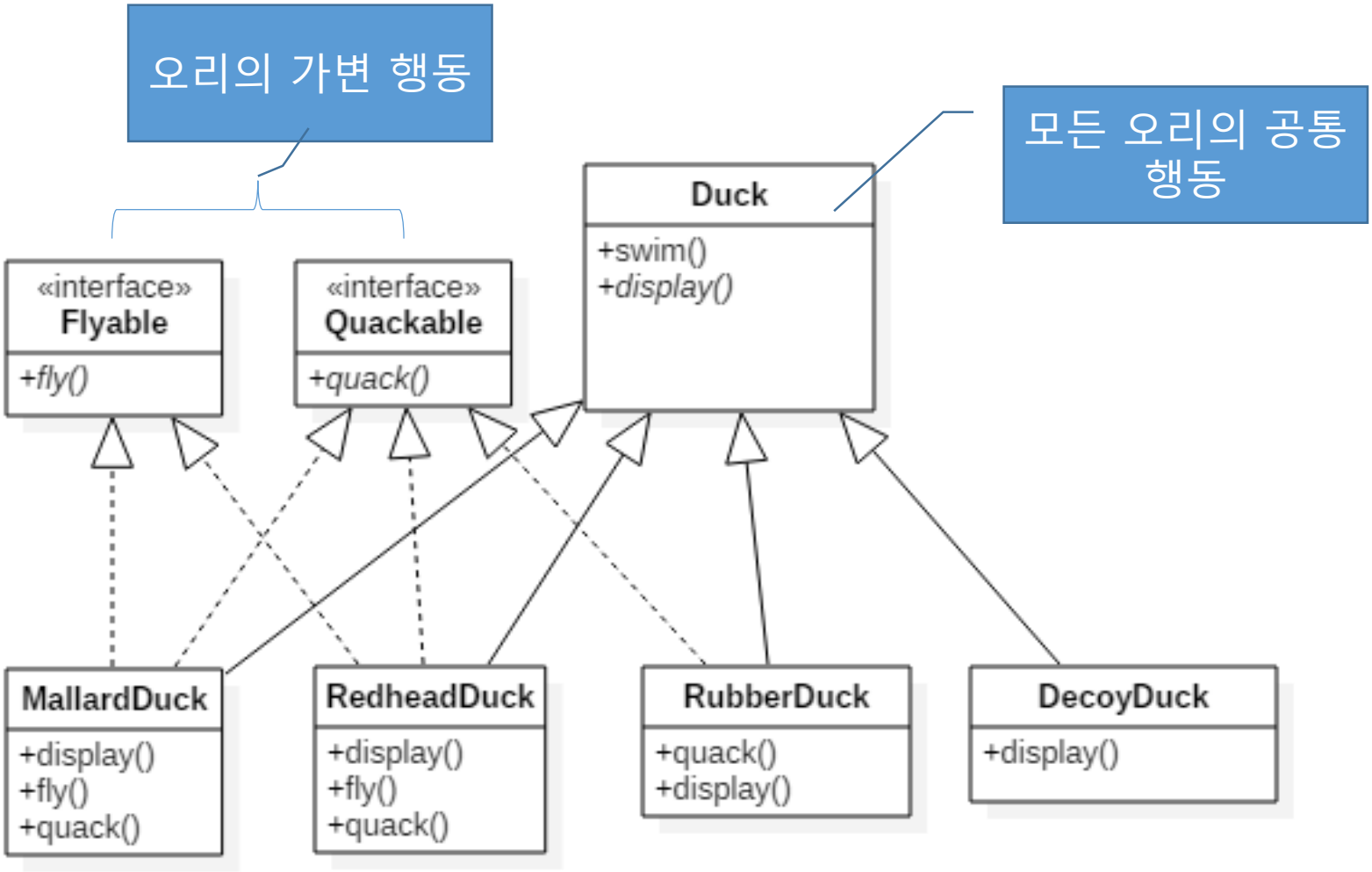
토의1

- Revision 2의 장단점에 대해서 나열 하시오.

- 장점
  1. 모든 오리의 공통 행동의 재사용이 용이하다
  2. 새로운 종류 오리의 추가가 용이하다.

- 단점
  1. Duck 서브클래스에서 코드가 중복될 수 있다.
  2. Duck 서브클래스의 정의만 보고는 오리들 사이의 행동 차이를 구분하기 힘든 경우가 있다.
  3. Duck 클래스의 변경이 Duck의 서브클래스에 원치 않는 영향을 끼칠 수 있다.

또 다른 문제해결 방안-인터페이스 사용 (Revision 3)



https://github.com/kwanulee/DesignPattern/tree/master/strategy/SimUDuckApp.Revision3

퀴즈 2

- Revision 3의 장단점에 대해서 나열 하시오.

퀴즈 2 (답)

- Revision 3의 장단점에 대해서 나열 하시오.

- 장점
  1. 모든 오리의 공통 행동의 재사용이 용이하다
  2. 새로운 종류 오리의 추가가 용이하다.
  3. Duck 서브클래스의 정의만 보고 오리들 사이의 행동 차이를 구분하기 용이하다
- 단점
  1. 인터페이스는 구현 코드가 없으므로, 코드의 재사용을 할 수 없다.
  2. Duck 서브클래스에서 코드가 중복된다.
  3. 오리 행위의 변경이 하나 이상의 Duck 서브클래스에 영향을 미칠 수 있다.

핵심 문제점 요약

- 상속 & 오버라이딩 사용 방안
  - 서브 클래스 마다 오리의 행동이 바뀔 수 있는데도, Duck 클래스에 오리의 행동을 정의하고, Duck의 서브클래스에서 재 정의
  - 문제점
    - Duck 클래스의 변경(나는 행동의 추가)이 여러 서브 클래스의 변경

- 인터페이스 사용 방안
  - 가변적인 오리의 행동을 인터페이스로 정의하고, 이를 Duck의 서브클래스에서 구현
  - 문제점
    - 오리의 한 행동(나는 동작)을 구현해야 할 경우에, Duck의 여러 서브클래스에서 중복적으로 구현

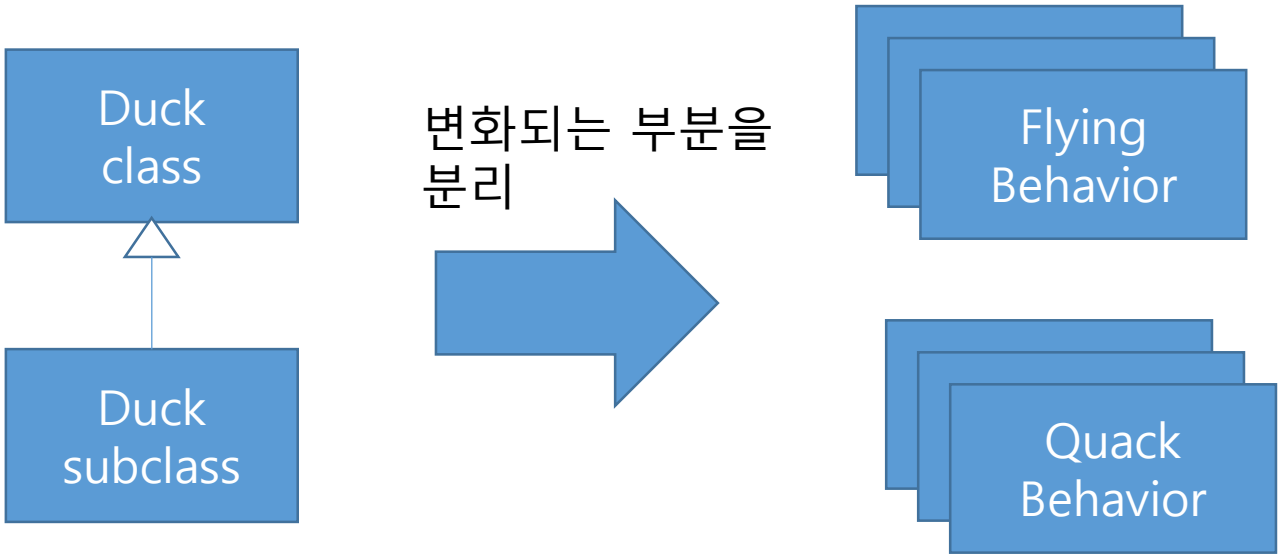
디자인 원칙

변화하는 부분과 변하지 않는 부분을 분리하라

바뀌는 부분은 캡슐화 시킴으로써,
변경의 영향을 한정시킴

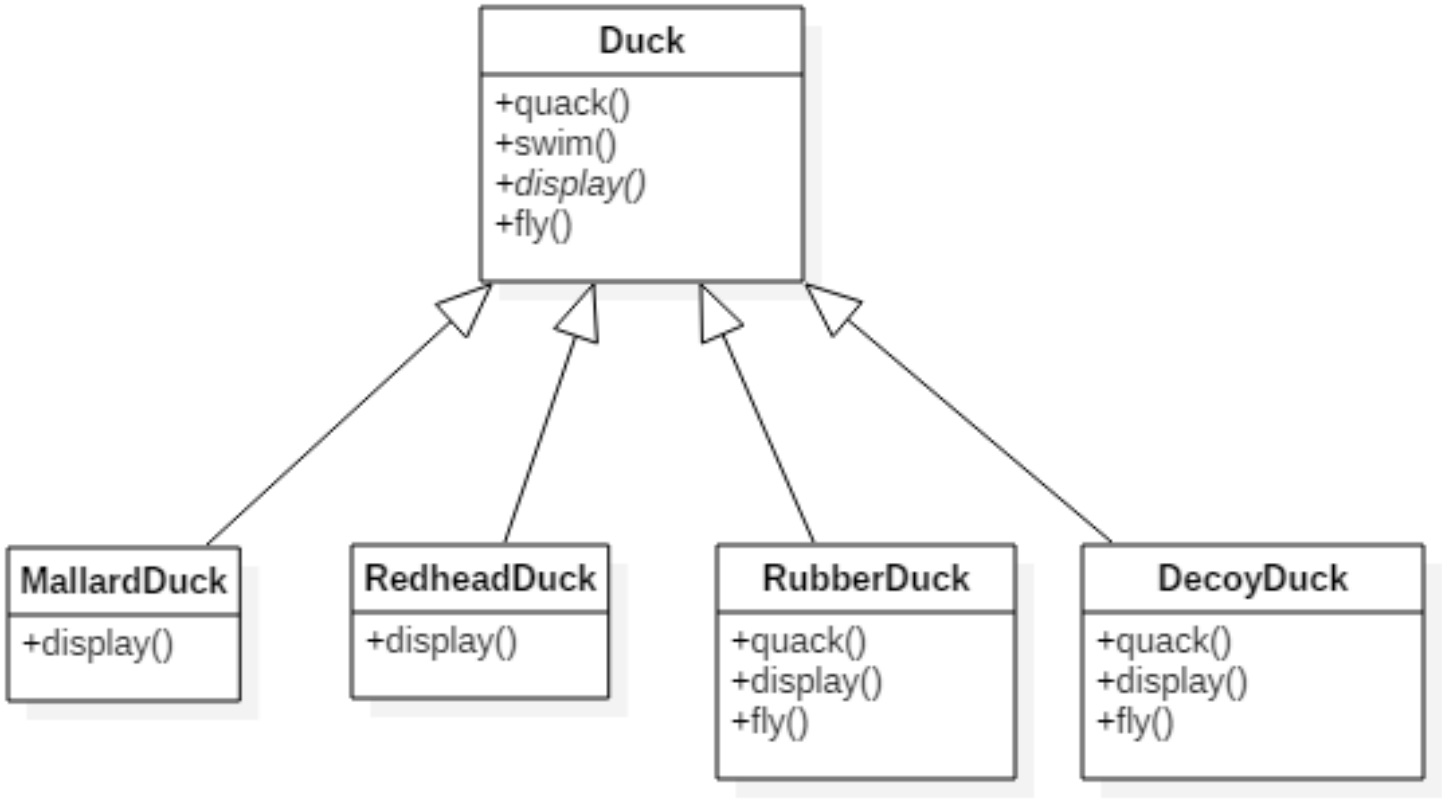
변화하는 부분의 분리

- fly()와 quack() 행위는 오리의 종류에 따라 변화되는 부분



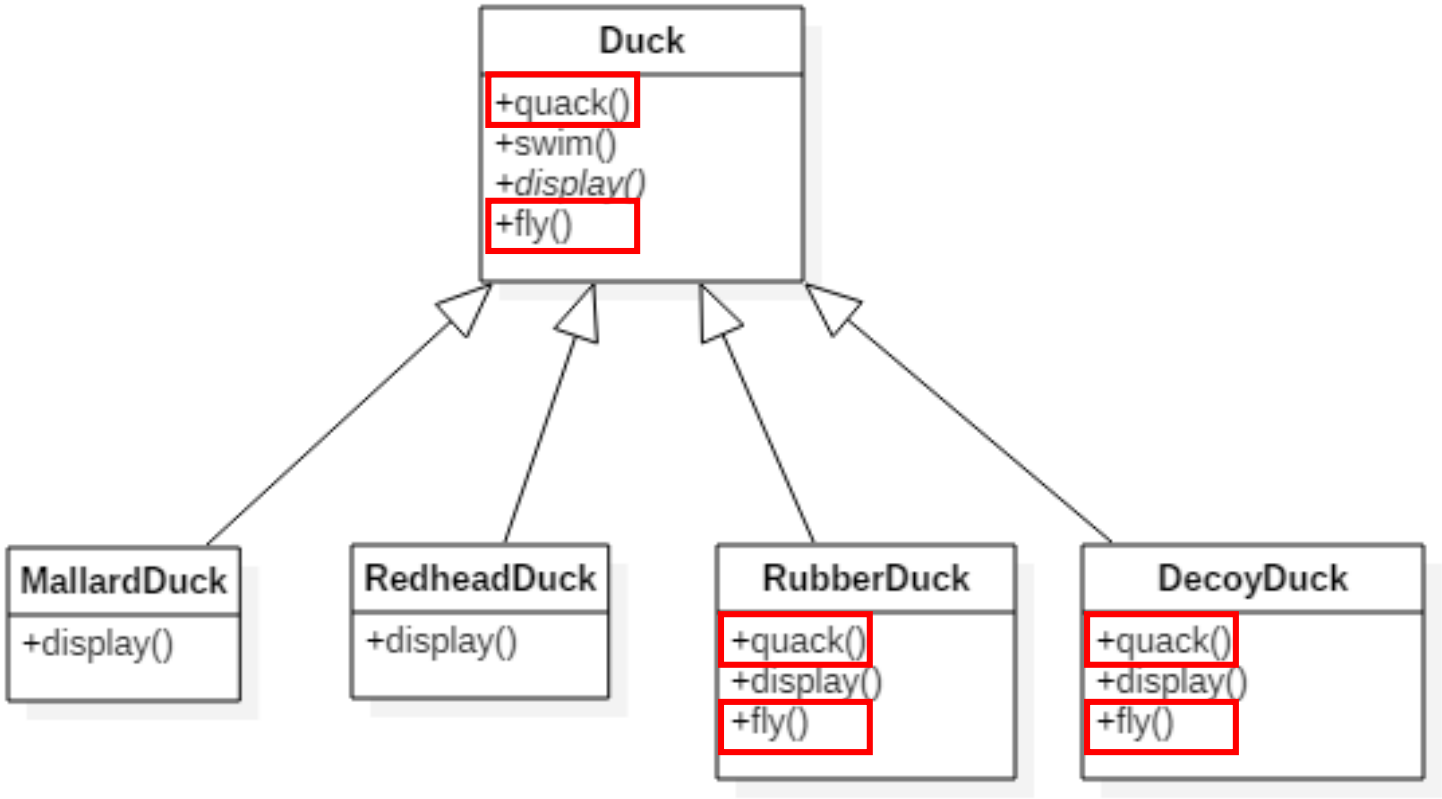
퀴즈 3

- 다음 디자인에서 오리의 종류에 따라 변화하는 행위로서 분리될 필요가 있는 부분은?



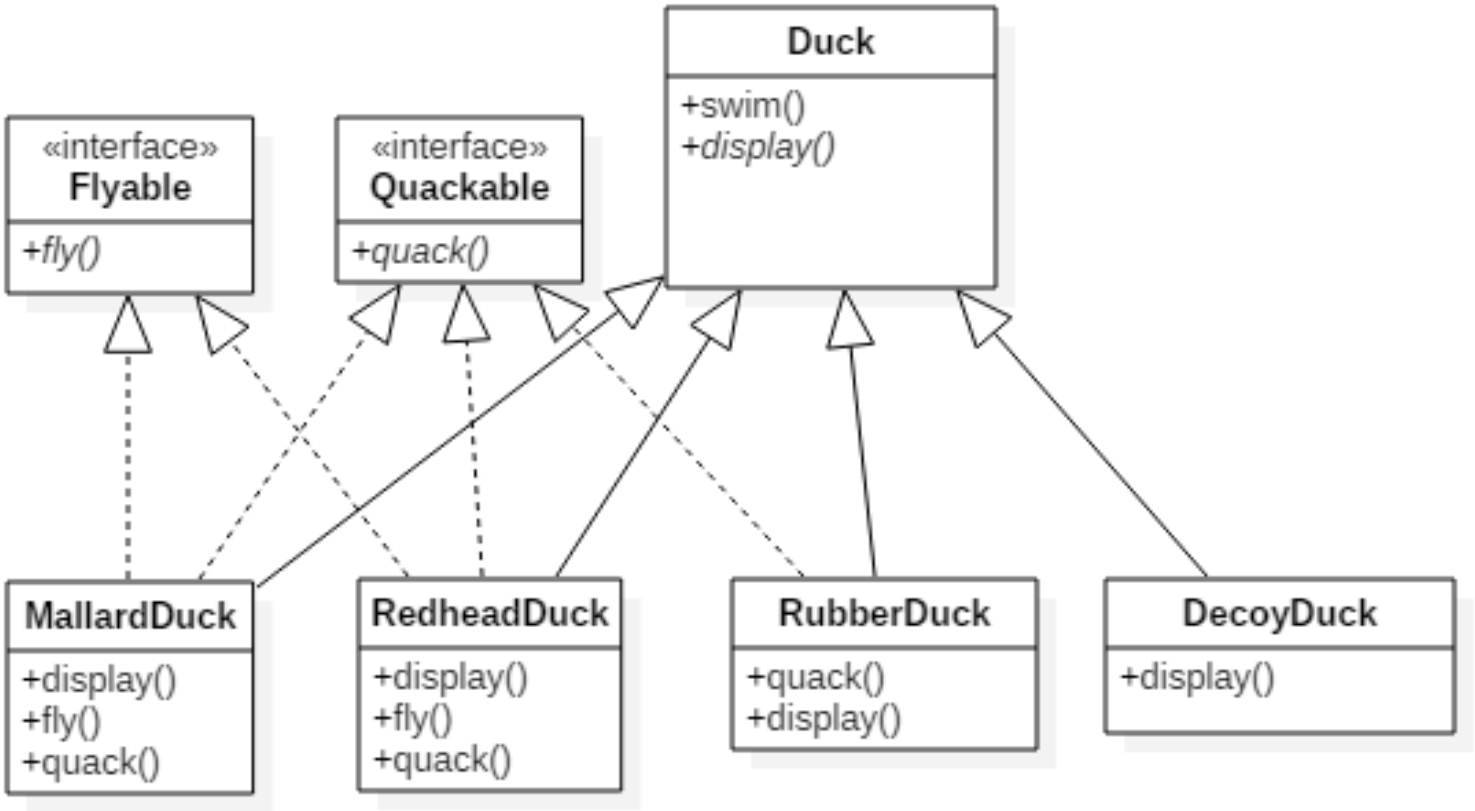
퀴즈 3 답

- 다음 디자인에서 오리의 종류에 따라 변화하는 행위로서 분리될 필요가 있는 부분은?



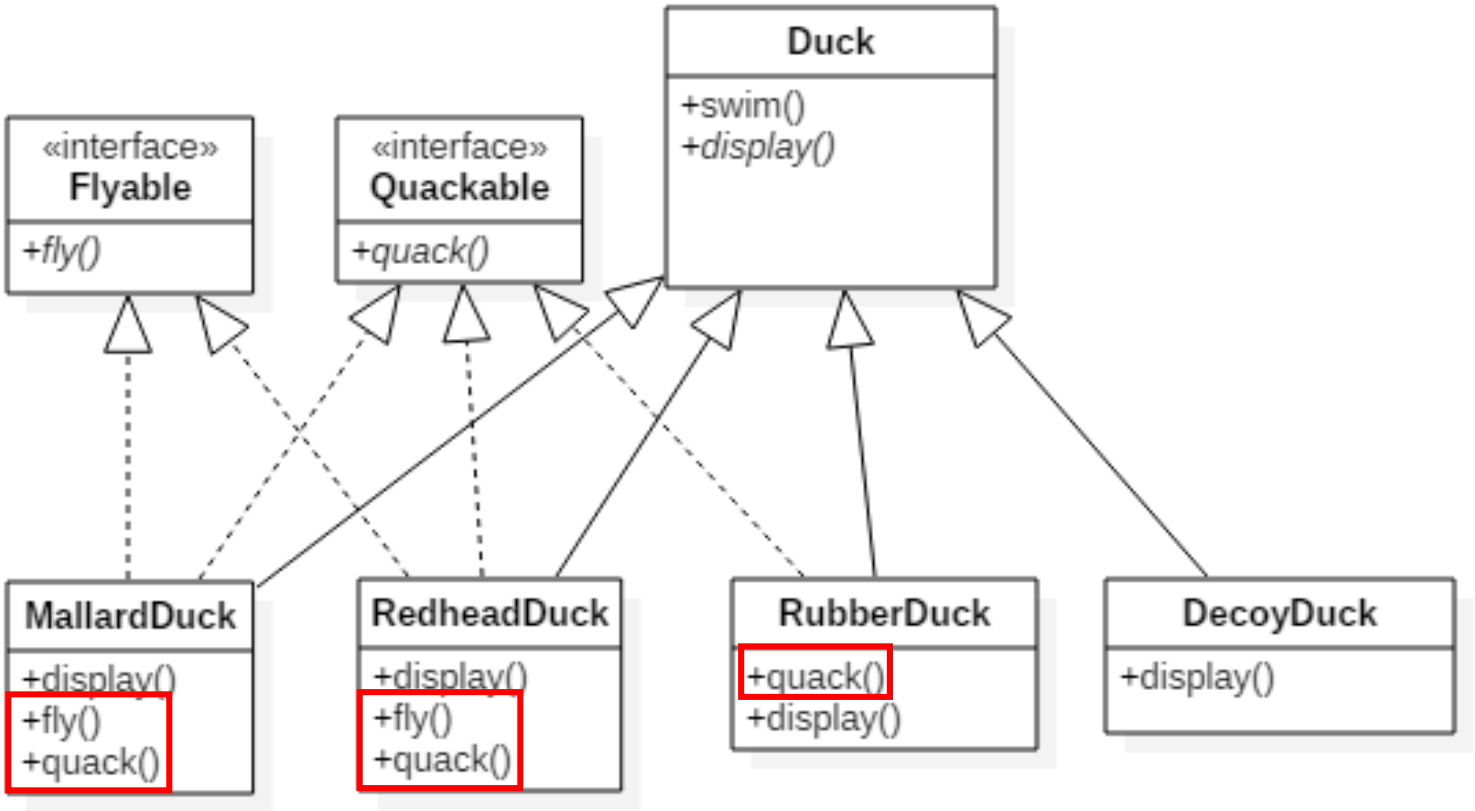
퀴즈 4

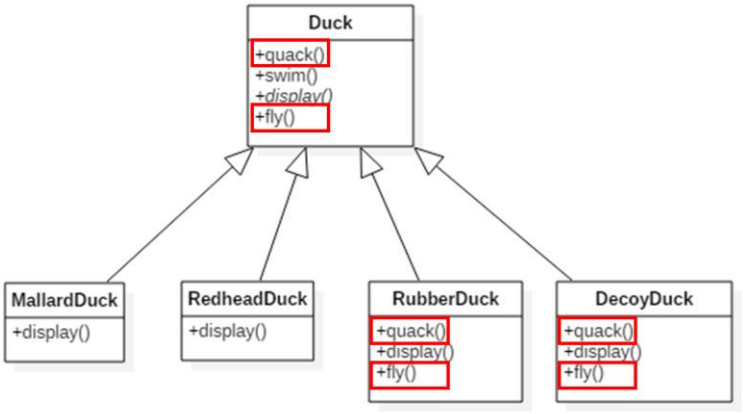
- 다음 디자인에서 오리의 종류에 따라 변화하는 행위로서 분리될 필요가 있는 부분은?

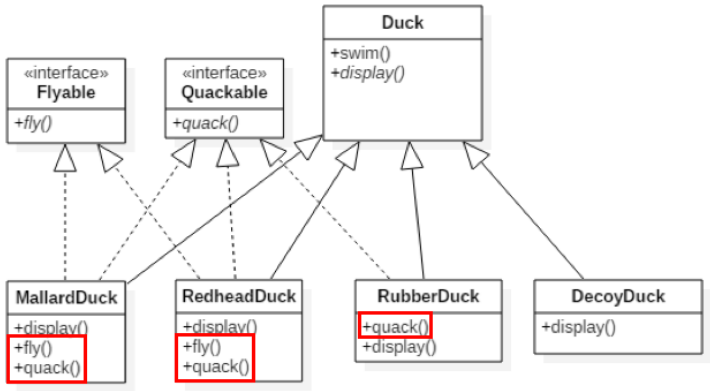


퀴즈 4 답

- 다음 디자인에서 오리의 종류에 따라 변화하는 행위로서 분리될 필요가 있는 부분은?







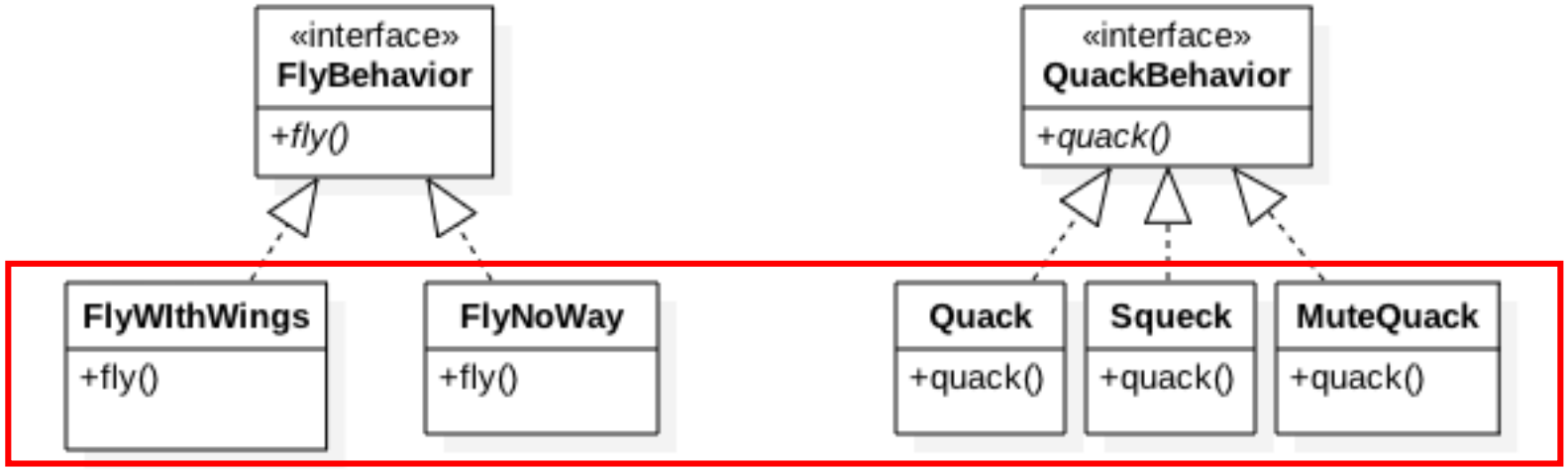
변경되는 부분(fly 혹은 quack 메소드)을 Duck 및 Duck의 서브클래스에서 어떻게 분리시킬 수 있을까요?

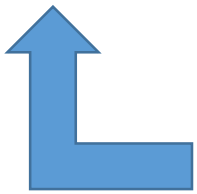
디자인 원칙

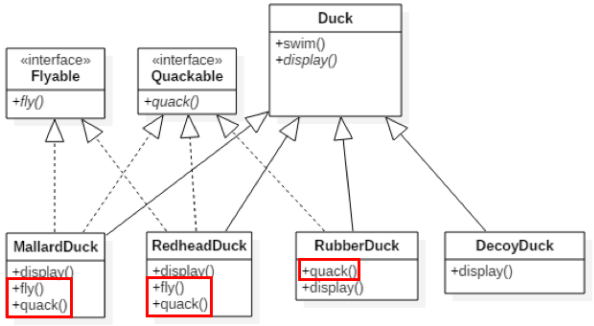
구현이 아닌 인터페이스에 맞춰서 프로그래밍하라

- **변경되는 부분 (fly 혹은 quack 메소드)**을 기존 구현 클래스(Duck 혹은 Duck 서브클래스)에서 프로그래밍하지 말고,**독립적인 인터페이스 (FlyBehavior 혹은 QuackBehavior)에서 맞춰서 프로그래밍** 한다.

인터페이스에 맞춘 다양한 Duck 행동 구현

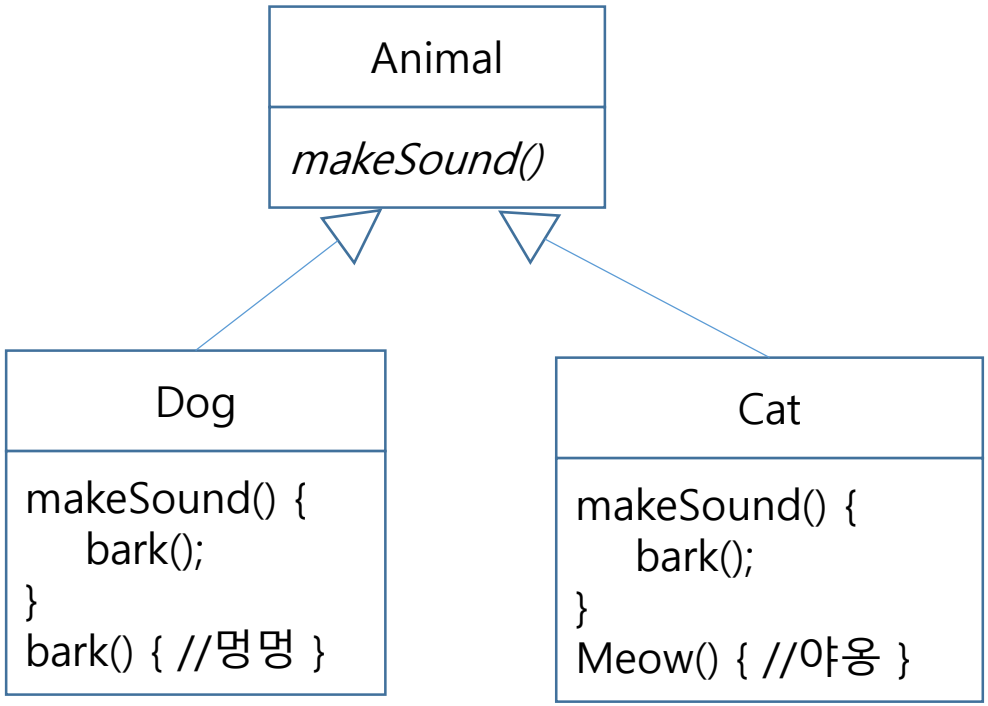






디자인 원칙

- "**인터페이스에 맞춰서 프로그래밍한다**"는 것은 "**상위 형식에 맞춰서 프로그래밍한다**"는 것을 의미하지, 자바의 인터페이스라는 구조를 지칭하는 것은 아닙니다.



구현에 맞춘 경우

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

상위 형식에 맞춘 경우

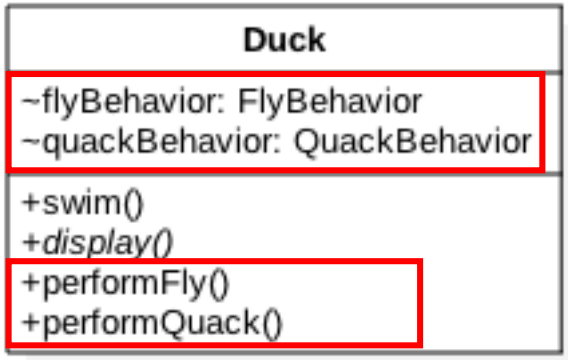
```
Animal a = new Dog();
a.makesound();

A = getAnimal();
a.makesound();
```

인터페이스에 맞춘 Duck 클래스 구현

1

인터페이스 변수

훅 메소드



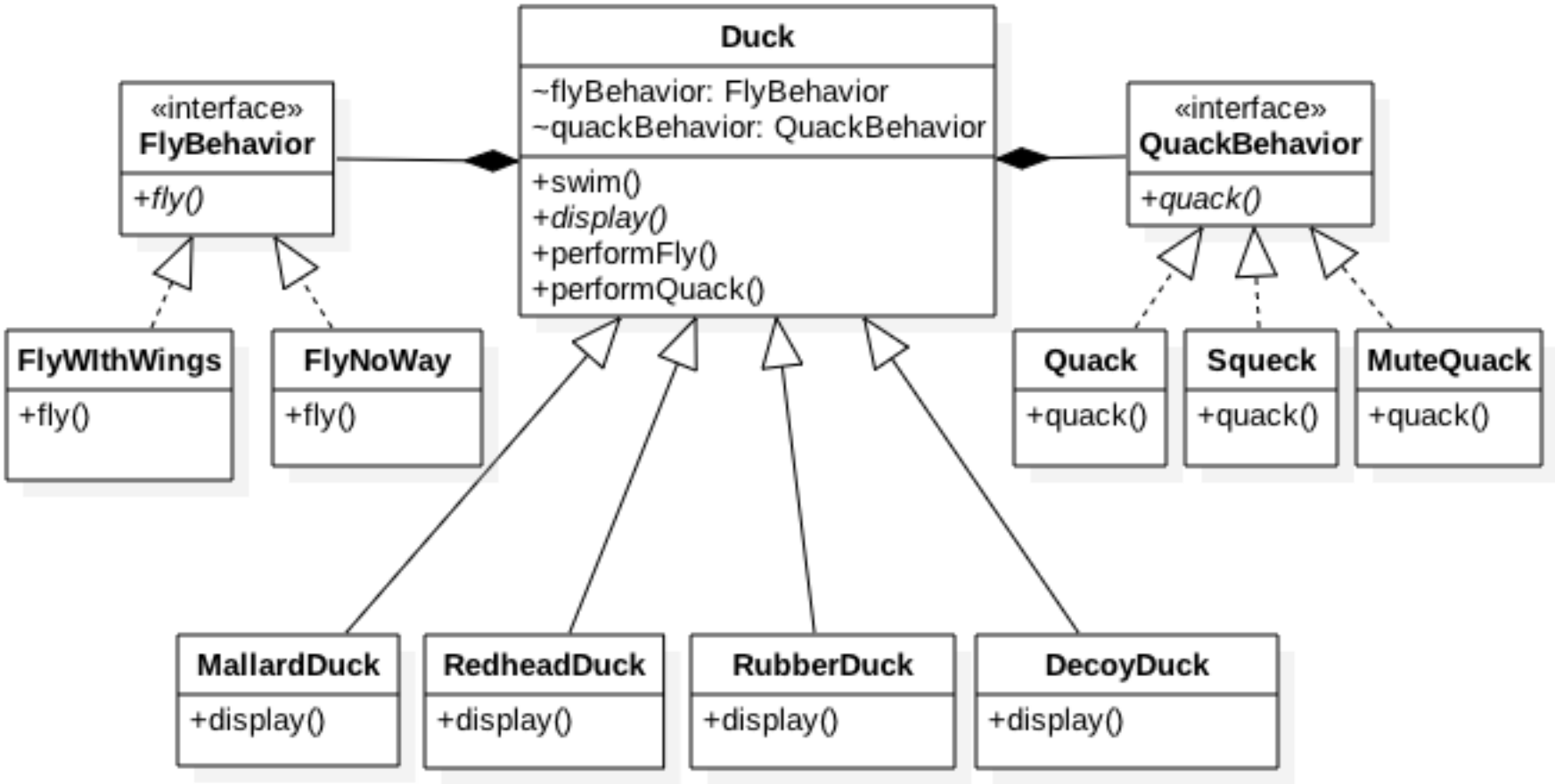
2 훅 메소드 구현

```
public class Duck {
    QuackBehavior quackBehavior;
    …

    public void performQuack() {
        quackBehavior.quack(); // 행동 위임
    }
}
```

3 인터페이스 변수 초기화

```
public class MallardDuck extends Duck{
    public MallardDuck() {
        quackBehavior = new Quack();   // 실질 행동 주체 설정
        flyBehavior = new FlyWithWings();
    }
    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

새로운 디자인 (Revision 4)

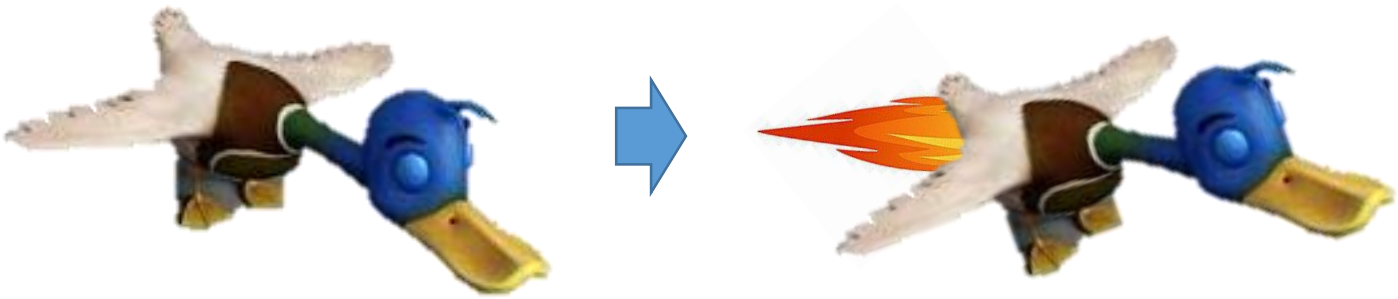


https://github.com/kwanulee/DesignPattern/tree/master/strategy/SimUDuckApp.Revision4

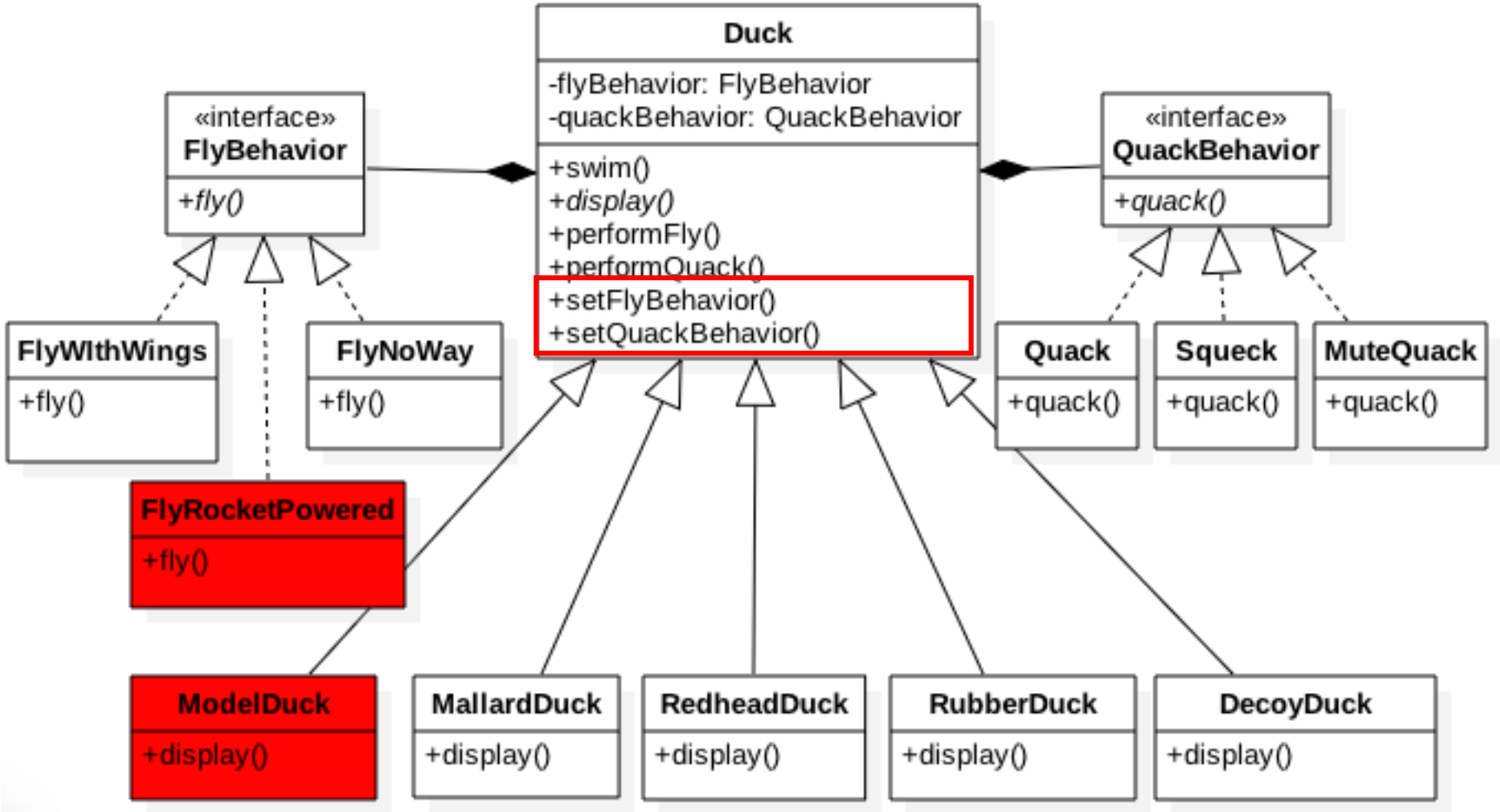
Revision 4의 장점

- 장점
  - 나는 행동과 꽥꽥거리는 행동의 재사용
  - 새로운 나는/꽥꽥거리는 행동의 추가가 용이

- 단점
  - 오리 행동의 동적 변경이 필요한 경우, 기존의 Duck 서브클래스의 변경이 필요함 (예, ModelDuck의 기존 나는 행동이 출시 후에 로켓추진 비행으로 변경되는 경우)



최종 디자인 (Revision 5)



https://github.com/kwanulee/DesignPattern/tree/master/strategy/SimUDuckApp.Revision5

디자인 패턴이란??

- Revision 5는 스트래티지 패턴 (Strategy Pattern)

> **스트래티지 패턴**에서는 알고리즘을 정의하고 각각을 캡슐화하여 교환해서 사용할 수 있도록 만든다. 스트래티지를 활용하면 알고리즘을 사용하는 클라이언트와는 독립적으로 알고리즘을 변경할 수 있다.

- 패턴의 이름은 패턴이 내포하고 있는 모든 내용을 함께 전달해 줌
- 패턴 수준에서 이야기를 하면 "디자인"에 더 오랫동안 집중할 수 있음
- 전문 용어의 사용은 개발 팀의 능력을 극대화 시키며, 신참 개발자들에게 훌륭한 자극제가 됨

핵심 정리

- 객체지향 기초 (추상화, 캡슐화, 다형성, 상속) 만 가지고는 훌륭한 객체지향 디자이너가 될 수 없슴
- 훌륭한 객체지향 디자인이라면, 재사용성, 확장성, 관리의 용이성을 갖춰야 함
- 패턴은 훌륭한 객체지향 디자인 품질을 갖춘 시스템을 만드는 방법을 제공
- 패턴은 검증받은 객체지향 경험의 산물로서 발명되는 것이 아니라 발견되는 것임

핵심 정리

- 패턴은 디자인 문제에 대한 일반적인 해법을 제공해 주는 것이고, 특정 애플리케이션에 패턴을 적용하는 것은 여러분의 몫
- 대부분의 패턴과 원칙은 소프트웨어의 변경 문제와 관련됨. 즉 시스템의 일부분을 나머지 부분과 무관하게 변경하는 방법을 제공
- 많은 경우에 시스템에서 바뀌는 부분을 골라내서 캡슐화시켜야 함
- 패턴은 다른 개발자들과 의사소통의 가치를 극대화 시킬 수 있는 전문용어 역할을 함