문제점 ① 모든 서브 클래스에 추가해야할 기능이 필요한 것은 아니다. 상속X

② 인터페이스는 코드 재사용성을 기대할 수 없다. Default 구현은 어떨까?

- 소프트웨어는 아무리 잘 디자인 해도 시간이 지나면서 점점 성장하고 변화한다.

디자인 원칙 애플리케이션에서 달라지는 부분을 찾아내고, 달라지지 않는 부분으로 부터 분리한다.

- 달라지는 부분을 찾아 나머지에 영향을 주지 않도록 캡슐화

=> 나중에 바뀌지 않는 부분에는 영향을 주지않고 수정·확장할 수 있다.

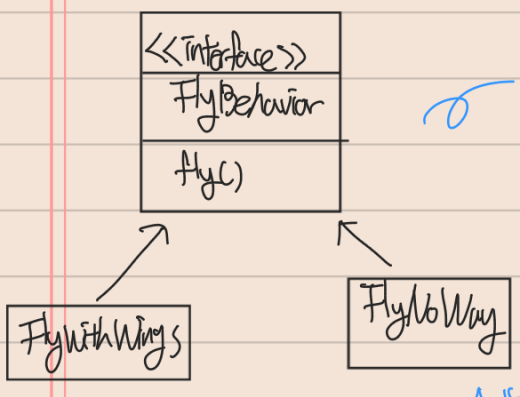
- fly()와 quack()은 오리마다 달라진다. → Duck class에서 분리하여 새로운 집합을 만들자
- Duck의 인스턴스에 행동을 할당할 수 있어야한다. (setter, 생성자 주입)

디자인 원칙 구현이 아닌 인터페이스에 맞춰서 프로그래밍한다. = 상위 형식에 맞추어 프로그래밍 한다.

ex) FlyBehavior, QuackBehavior의 인터페이스와 이를 구현하는 구현체

↳ Duck 클래스는 위의 행동을 "사용"

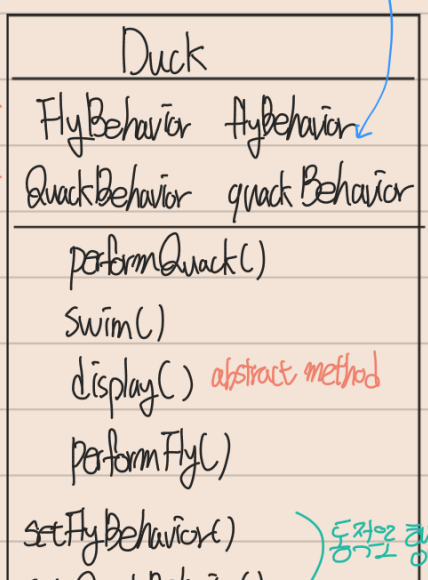
- Animal animal = new Dog(); < Animal animal = getAnimal(); (구체적으로 구현된 객체를 실행시 대입)



- 다른 객체에서도 나는 행동을 재사용 가능 (Duck 외부에 따로 존재)
- 이와 같은 방식으로 새로운 행동 추가 가능

Strategy Pattern

A에는 B가 있다 (구성 : composition)



상속시에 인스턴스 변수도 상속함

동적으로 행동을 지정

```
public class Duck {
  QuackBehavior quackBehavior;
  뿌요 클래스에 구현도 존재
  public void performQuack() {
    quackBehavior.quack();
  }
}
```

행동을 위임함

setQuackBehavior()

디자인 원칙

상속보다는 구성을 활용한다.

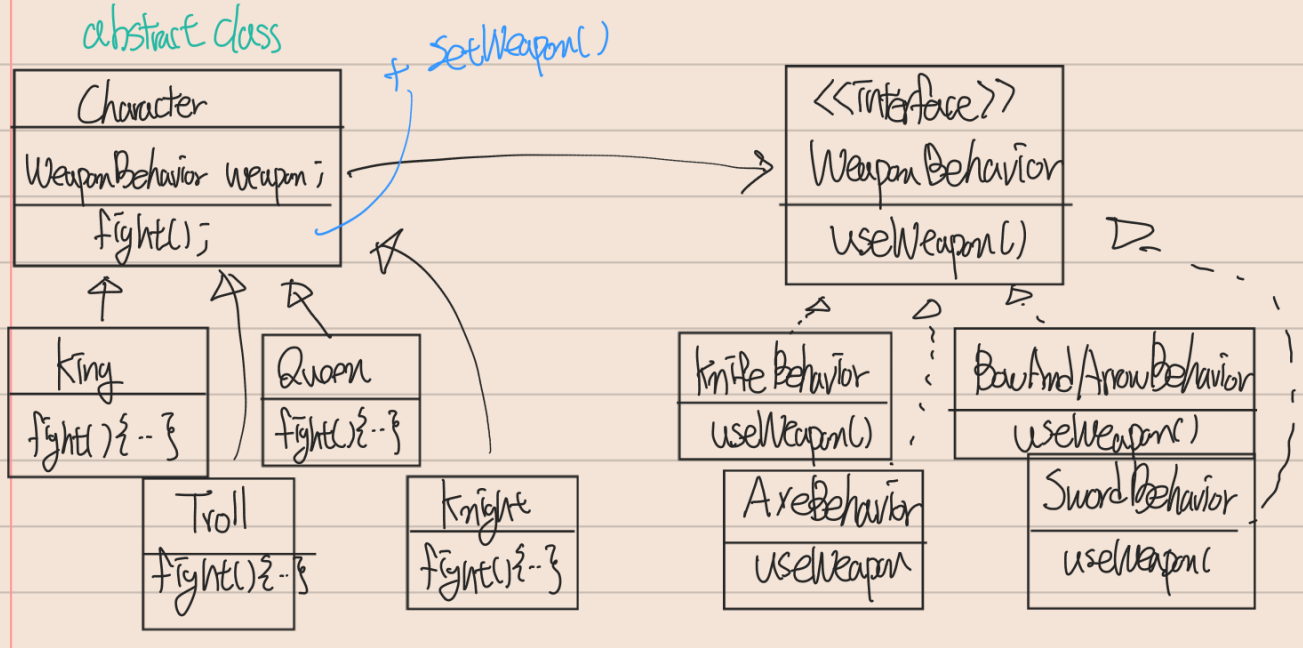
- 유연성을 크게 향상시킬수 있음. ex) 동적으로 변경가능
- 알고리즘군을 별도의 클래스로 캡슐화할 수 있음.

나중에 어떻게 바뀔까 고민하자

• 관리의 용이성과 확장성 보다 재사용성에 더 많은 노력을 기울여야할까? No!

스트래티지 패턴

알고리즘군을 정의하고 각각을 캡슐화하여 교환해서 사용할 수 있도록 만든다. 스트래티지를 활용하면 알고리즘을 사용하는 클라이언트와는 독립적으로 알고리즘을 변경할 수 있다



왜 배울까?

• 개발자들이 서로 공유할 수 있는 언어사용 => 특성/제약조건도 공유

• 자질구레한 객체수준에서 생각하는 것이아니라 패턴수준에서 생각.

↳ 아키텍처에 대해 생각하는 수준 상승

• 디자인 패턴은 라이브러리보다 높은 단계에 속함. - 클래스와 객체를 구성하여 어떤 문제를 해결하는 방법을 제공

• 간단하지만은 않은 객체지향 시스템 구축 방법들을 모아놓은 것!

• 대부분의 패턴과 원칙은 소프트웨어 변경 문제와 관련되어 있다.