DESIGN PATTERNS

Introduction

동의대학교 컴퓨터소프트웨어공학과

패턴 (Pattern)

- 경험 많은 사람들에게는 잘 알려진 설계를 개발자들은 모르고 있다?
- 패턴
  - 공통적으로 알려진 일하는 방식
  - 설계에서 같은 주제를 반복해온 사람들이 모아놓은 해결책
- 참고
  - http://www.hillside.net/patterns
  - E. Gamma, R. Helm, R. Johnson, and J. Vlissides [Gang of Four], *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
  - https://refactoring.guru/design-patterns
  - https://sourcemaking.com/design_patterns
  - https://www.dofactory.com/net/design-patterns
  - M. Fowler, *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

- 패턴의 중요성
  - 언어의 기본이나 모델링 기법을 이해하고 난 다음 단계에서 필요
  - 일련의 해답 제시 및 좋은 모델과 구축방법 제시
  - 문제를 명확하게 해서 문제 푸는 경위를 설명해 주고 어떤 상황에서 작동하고 또 작동하지 않는가도 설명
- 패턴은 언제 사용하는가?
  - 항상 사용가능
  - 분석, 설계, 코딩, 프로젝트 관리 모든 단계에서 가능한 패턴을 찾아 도움을 얻을 수 있어야 한다.

GRASP 패턴

- GRASP: General Responsibility Assignment Software Patterns
  - 객체에 책임을 할당하기 위한 기본적인 원리 설명
  - 종류
    - Information Expert 패턴
    - Creator 패턴
    - High Cohesion 패턴
    - Low Coupling 패턴
    - Controller 패턴
  - 이러한 설계 패턴을 이해하여 새로운 상호작용 다이어그램을 작성하는 동안에 적용할 수 있도록 경험을 축적하는 것이 필요함.
- 참고 URL
  - https://nesoy.github.io/articles/2019-05/GRASP-Pattern
  - http://www.kamilgrzybek.com/design/grasp-explained/
  - https://dzone.com/articles/solid-grasp-and-other-basic-principles-of-object-o

1. 디자인 패턴 소개

오리 모델링

(Strategy Pattern)

디자인 패턴의 종류

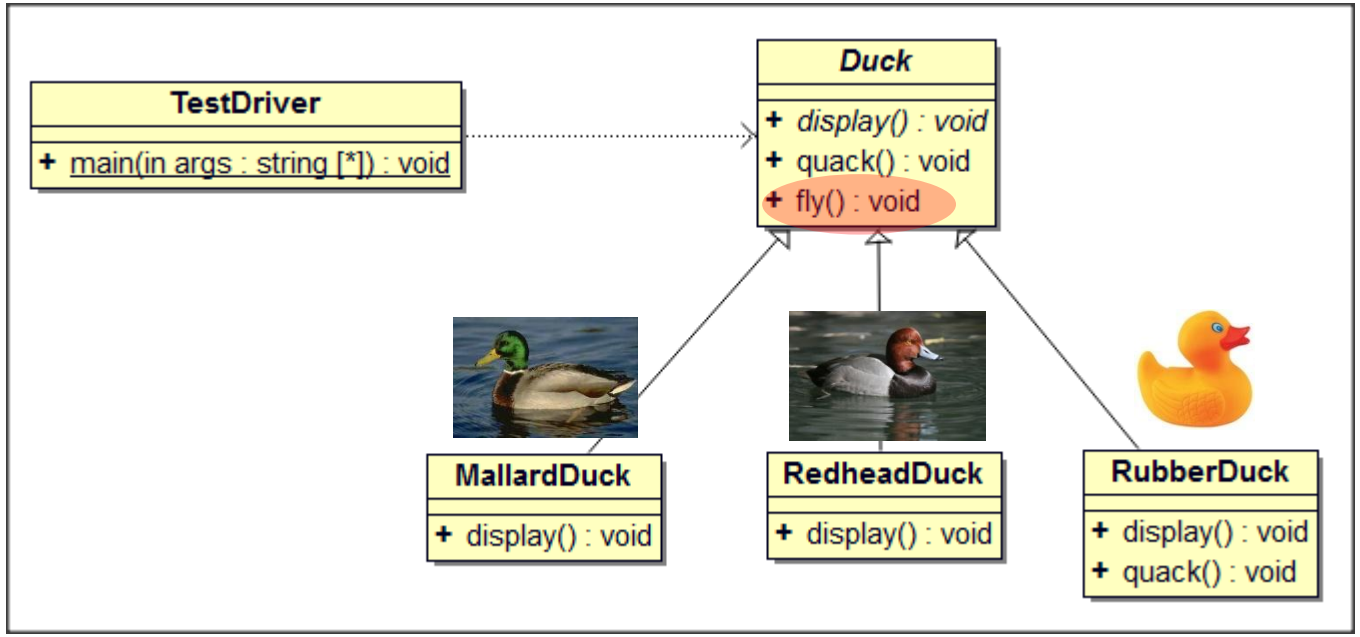
Creational Patterns	
Abstract Factory	Creates an instance of several families of classes
Builder	Separates object construction from its representation
Factory Method	Creates an instance of several derived classes
Prototype	A fully initialized instance to be copied or cloned
Singleton	A class of which only a single instance can exist

Structural Patterns	
Adapter	Match interfaces of different classes
Bridge	Separates an object's interface from its implementation
Composite	A tree structure of simple and composite objects
Decorator	Add responsibilities to objects dynamically
Facade	A single class that represents an entire subsystem
Flyweight	A fine-grained instance used for efficient sharing
Proxy	An object representing another object

Behavioral Patterns	
Chain of Resp.	A way of passing a request between a chain of objects
Command	Encapsulate a command request as an object
Interpreter	A way to include language elements in a program
Iterator	Sequentially access the elements of a collection
Mediator	Defines simplified communication between classes
Memento	Capture and restore an object's internal state
Observer	A way of notifying change to a number of classes
State	Alter an object's behavior when its state changes
Strategy	Encapsulates an algorithm inside a class
Template Method	Defer the exact steps of an algorithm to a subclass
Visitor	Defines a new operation to a class without change

오리 모델링 1

- 객체지향프로그램의 기본 원리인 **상속(inheritance)**을 사용하면 쉽게 모델링할 수 있을 것이다.
- Duck 클래스에 fly operation을 추가하면 RubberDuck도 날 수 있게 된다???
  - 고무오리는 날 수 없으므로 RubberDuck 클래스에 fly operation을 오버라이딩하여야 한다.

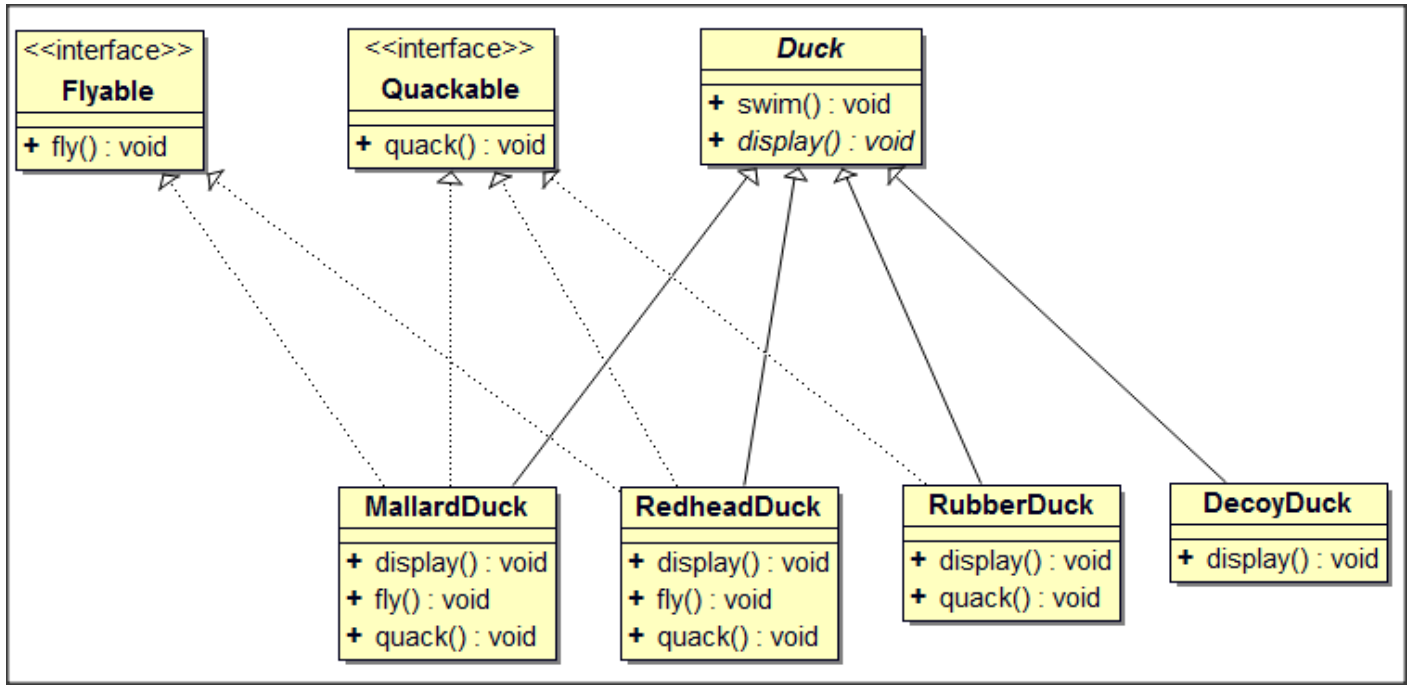


상속의 문제?

- Duck클래스에 fly operation을 추가함으로써 상속받는 클래스에서 경우에 따라 fly operation을 오버라이딩하여 새로 코드를 작성해 주어야 한다.
- 상속을 사용하는 가장 기본적인 이유 중 하나가 슈퍼클래스의 특징을 서브클래스에 상속해 주기 위한 것인데, 서브클래스에서 자주 오버라이딩하여 코딩해 주어야 한다면 코드 재사용성 면에서 효율적이지 못한다.
- 이 경우 인터페이스를 사용하여 Duck을 모델링하면?

오리 모델링 2

- 인터페이스를 이용
- 오리가 날 수 있거나 날 수 없다 → Flyable 인터페이스
- 오리가 여러 형태로 운다 → Quackable 인터페이스
- Duck클래스의 서브클래스에서 필요로 하는 인터페이스를 구현하면 오리를 날 수 있거나 없게 할 수 있고, 꽥꽥 울 수도 있고(MallardDuck, RedheadDuck) 삑삑 울 수도 있고(RubberDuck) 울지 않을 수도 있다.(DecoyDuck)



- 문제점
  - Duck 클래스의 각 서브클래스에서 Flyable, Quackable 인터페이스를 경우에 맞게 구현해 주어야 한다.
  - 유사한 코드가 많이 중복되어 코드 재사용성이 아주 안 좋은 예이다.
  - 자바 인터페이스에는 구현된 코드가 전혀 들어가지 않기 때문에 해당 인터페이스를 구현하는 클래스가 인터페이스의 오퍼레이션을 전부 구현해 주어야 한다.

디자인 원칙

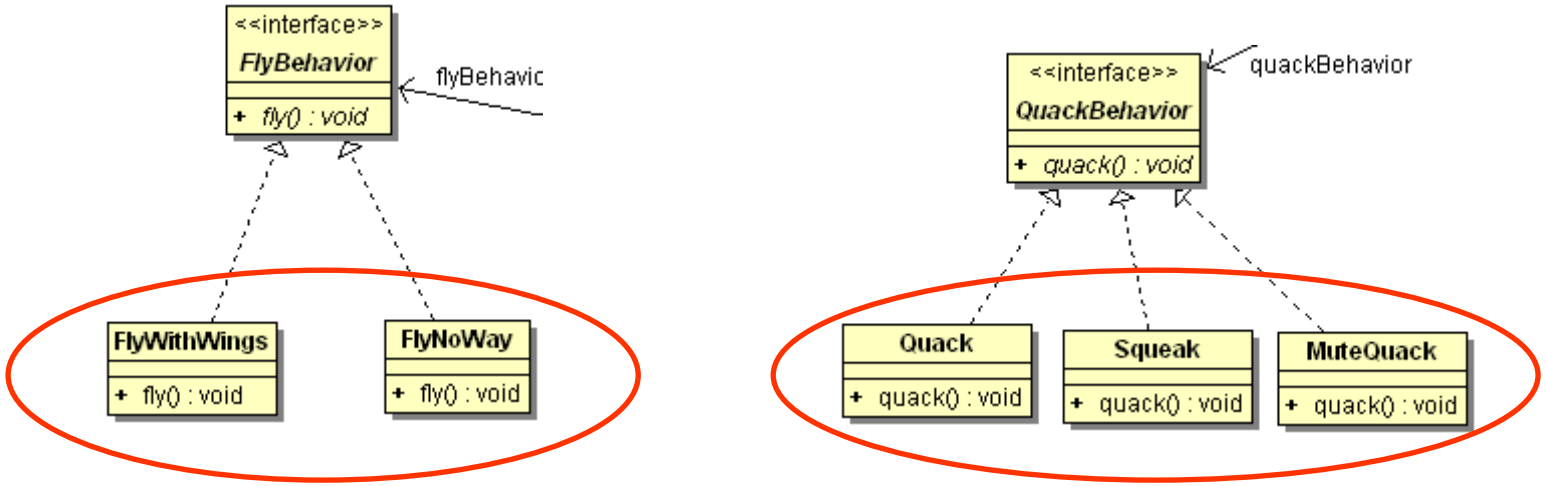
- 애플리케이션에서 달라지는 부분을 찾아내고, 달라지지 않는 부분으로부터 분리시킨다.
  - 달라지는 부분을 찾아서 나머지 코드에 영향을 주지 않도록 캡슐화한다.
  - 이렇게 함으로써 코드 변경 과정에서 의도하지 않은 일이 일어나는 것을 줄이고 시스템의 유연성은 향상시킬 수 있다.
- 구현이 아닌 인터페이스에 맞춰서 프로그래밍한다.
  - 각 행동은 인터페이스로 표현하고 행동을 구현할 때 해당 인터페이스를 구현한다.
  - 특정 행동을 원하는 클래스는 인터페이스를 구현하는 클래스를 선택하여 사용하면 된다. (코드 중복 제거)

오리 모델링 3

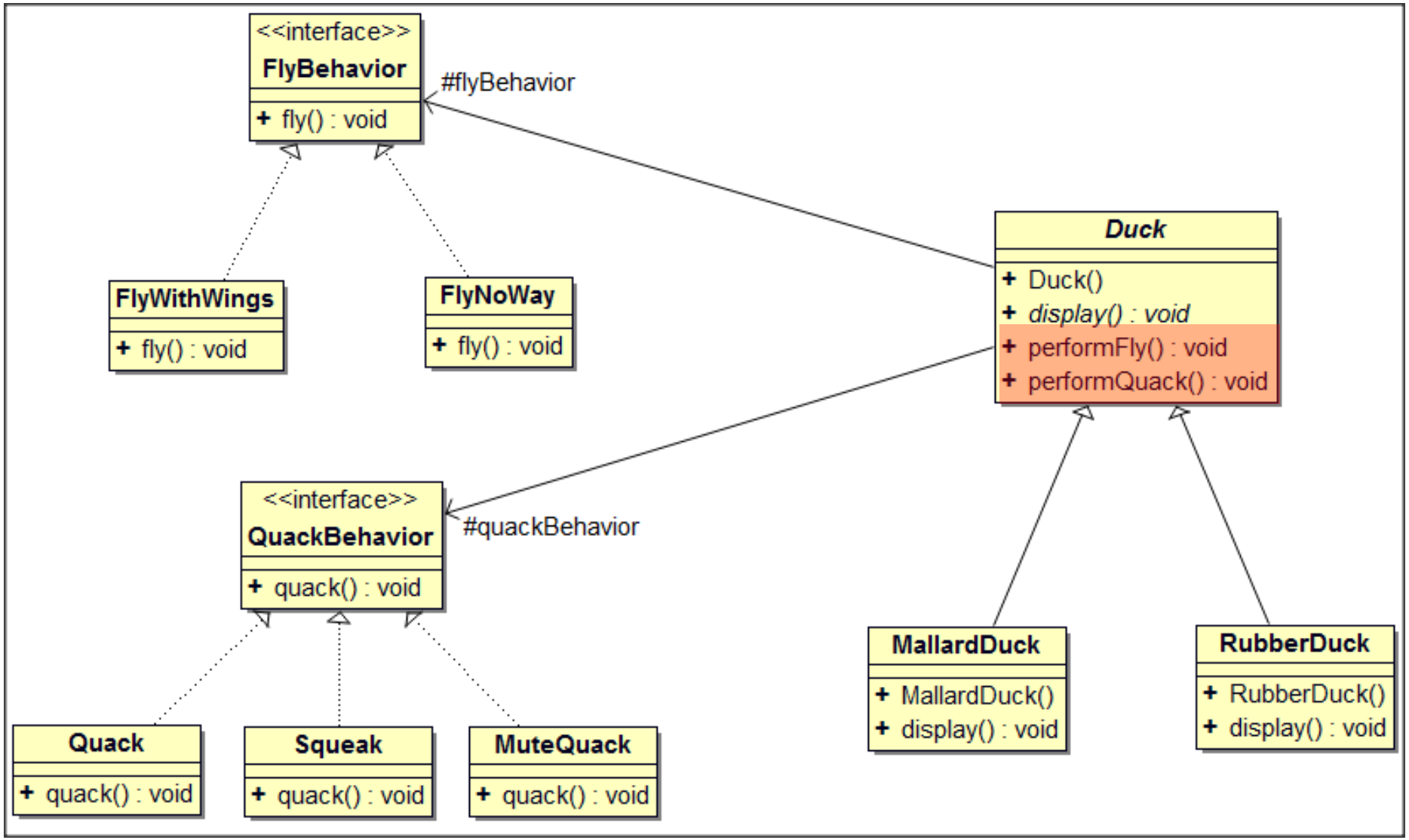
- 변경되는 부분을 분리하여 인터페이스로 구현
  - 오리의 나는 방법을 구현하는 fly() 오퍼레이션을 Duck 클래스로부터 분리한다.
    - 고무오리는 날 수 없다.
  - 오리는 우는 방법을 구현하는 quack() 오퍼레이션을 Duck 클래스로부터 분리한다.
    - 진짜 오리는 꽥꽥 울지만, 고무 오리는 삑삑 소리는 낸다. Decoy duck은 소리를 내지 못한다.
- 인터페이스를 다른 클래스에서 사용할 수 있도록 관련 클래스에 이를 구동시킬 수 있는 오퍼레이션을 추가한다.

인터페이스 만들기

- fly operation과 quack operation을 각각 FlyBehavior, QuackBehavior 인터페이스를 사용하여 표현하고 각 인터페이스를 구현하는 클래스를 경우에 맞게 준비해 둔다.



모델링 결과


<<interface>>
FlyBehavior
+ fly() : void
#flyBehavior
FlyWithWings
+ fly() : void
FlyNoWay
+ fly() : void
Duck
+ Duck()
+ display() : void
+ performFly() : void
+ performQuack() : void
<<interface>>
QuackBehavior
+ quack() : void
#quackBehavior
Quack
+ quack() : void
Squeak
+ quack() : void
MuteQuack
+ quack() : void
MallardDuck
+ MallardDuck()
+ display() : void
RubberDuck
+ RubberDuck()
+ display() : void

Strategy Pattern

- A special design pattern, whereby algorithms can be selected at runtime [from wikipedia]

이름	**Strategy**
문제	변하지만 서로 관련된 알고리즘이나 정책을 어떻게 설계할 것인가? 어떻게 이러한 알고리즘이나 정책을 변경할 수 있도록 설계하는가?
해결책 (조언)	각각의 알고리즘/정책/전략을 공통 인터페이스를 가진 독립된 클래스에 정의한다.

- 비슷한 유형의 알고리즘들 중에서 하나를 선택하여 사용할 경우 편리함.

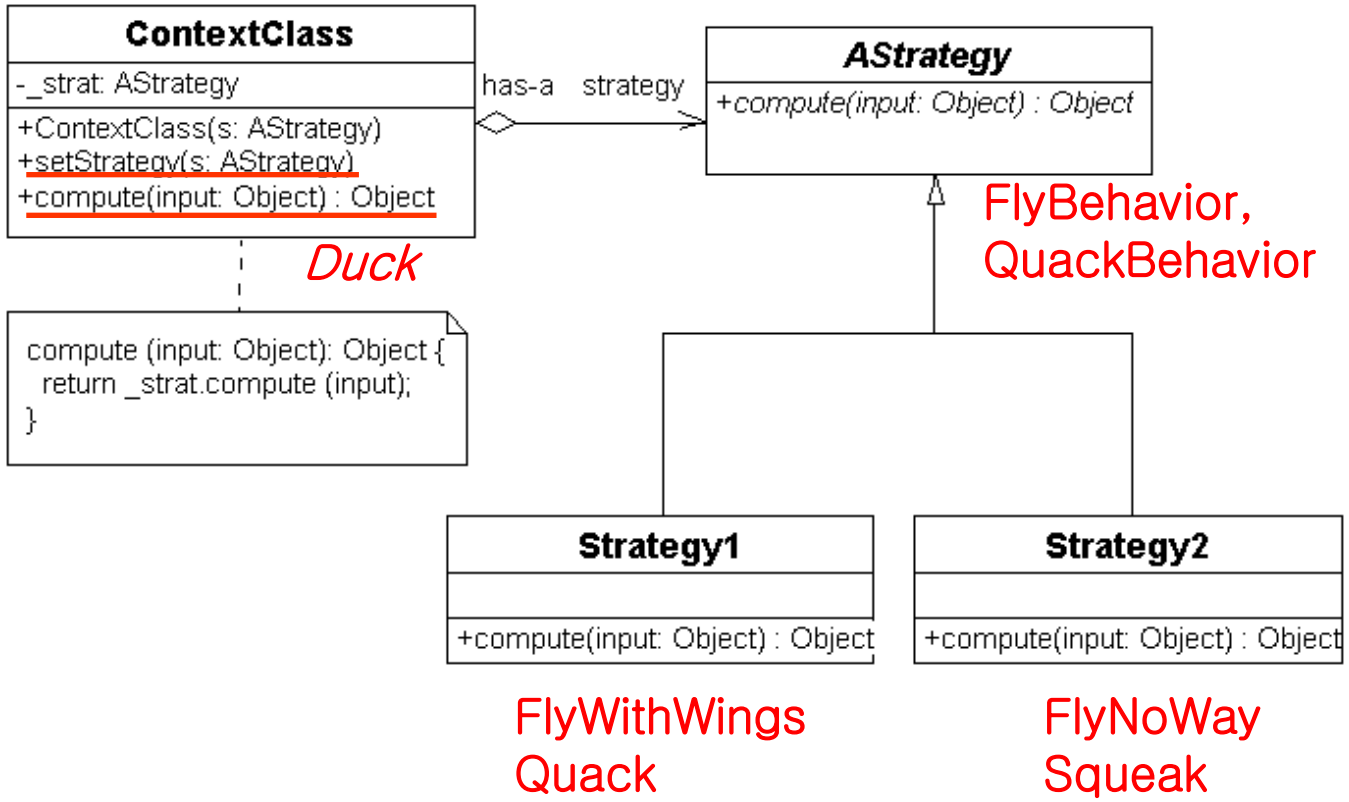


그림 출처: http://www.clear.rice.edu/comp212/00-fall/handouts/week04/designPatterns.htm

디자인 원칙

- 상속(inheritance)보다는 구성(composition)을 활용
  - "A에는 B가 있다" (has-a) 관계
  - 오리 클래스의 경우 행동을 상속받는 대신 올바른 행동 객체로 구성됨으로써 행동을 부여 받게 된다.
  - 구성을 이용하면 유연성을 크게 향상시킬 수 있다.
  - 실행 시에 행동을 바꿀 수 있다!!!


```
class A {
    private B itsB;
}
```

```
class B {

}
```

실습

오리 모델링 3의 코드 테스트

- 클래스와 인터페이스 코드를 생성한다.
- FlyBehavior, QuackBehavior 인터페이스는 다음과 같다.

```
1  package cse.design_pattern.ch01.model03;
2
3  interface FlyBehavior {
4
5    void fly();
6  }
```

```
1  package cse.design_pattern.ch01.model03;
2
3  interface QuackBehavior {
4
5    void quack();
6  }
```

- FlyBehavior 인터페이스의 구현 클래스

```java
package cse.design_pattern.ch01.model03;

public class FlyWithWings implements FlyBehavior {

    public void fly() {
        System.out.println("날 수 있어요!!");
    }

}
```

```java
package cse.design_pattern.ch01.model03;

public class FlyNoWay implements FlyBehavior {

    public void fly() {
        System.out.println("날지 못해요!!");
    }

}
```

- QuackBehavior 인터페이스의 구현 클래스

```java
package cse.design_pattern.ch01.model03;

public class Quack implements QuackBehavior {

    public void quack() {
        System.out.println("꽥");
    }

}
```

```java
package cse.design_pattern.ch01.model03;

public class Squeak implements QuackBehavior {

    public void quack() {
        System.out.println("삑");
    }

}
```

```java
package cse.design_pattern.ch01.model03;

public class MuteQuack implements QuackBehavior {

    public void quack() {
        System.out.println("조용");
    }

}
```

- Duck 클래스 구현

```
1   package cse.design_pattern.ch01.model03;
2
3   public abstract class Duck {
4
5       protected FlyBehavior flyBehavior;
6
7       protected QuackBehavior quackBehavior;
8
9       public void performFly() {
10          flyBehavior.fly();
11      }
12
13      public void performQuack() {
14          quackBehavior.quack();
15      }
16
17      public abstract void display();
18  }
```

- MallardDuck, RubberDuck 클래스 구현

```java
package cse.design_pattern.ch01.model03;

public class MallardDuck extends Duck {

    public MallardDuck() {
        flyBehavior = new FlyWithWings();
        quackBehavior = new Quack();
    }

    public void display() {
        System.out.println("저는 청둥 오리입니다.");
    }

}
```

```java
package cse.design_pattern.ch01.model03;

public class RubberDuck extends Duck {

    public RubberDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Squeak();
    }

    public void display() {
        System.out.println("저는 고무 오리입니다.");
    }

}
```

- 테스트 클래스 구현

```
5   package cse.design_pattern.ch01.model03;
6
7   /**...4 lines */
11  public class TestDriver {
12
13      /**
14       * @param args the command line arguments
15       */
16      public static void main(String[] args) {
17          Duck mallard = new MallardDuck();
18
19          mallard.display();
20          mallard.performFly();
21          mallard.performQuack();
22
23          System.out.println("----------------------");
24
25          Duck rubber = new RubberDuck();
26          rubber.display();
27          rubber.performFly();
28          rubber.performQuack();
29      }
30
31  }
```

- 실행 결과

```
--- exec-maven-plugin:1.2.
저는 청둥 오리입니다.
날 수 있어요!!
꽥
------------------------
저는 고무 오리입니다.
날지 못해요!!
삑
```