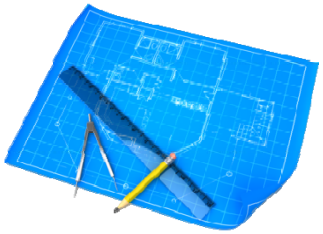


디자인 패턴 소개
Strategy 패턴

NOTE 03

한국기술교육대학교 컴퓨터공학부 김상진

sangjin@koreatech.ac.kr
www.facebook.com/sangjin.kim.koreatech

교육목표

- 디자인 패턴에 대한 기본적인 개념 이해
  - 디자인 패턴이 프로그램 개발 과정에서 어떤 역할을 하는지 이해
  - 핵심. 다른 사람들의 해결책을 활용하는 것
    - 공통적으로 많이 일어나는 문제에 대한 해결책
    - 고정된 코드 형태가 아니라 해결방법 또는 틀만 제시. 언어마다 상황마다 다르게 적용 가능
- 패턴의 종류 이해
- Strategy 패턴의 이해
  - 알고리즘의 군을 정의하고 캡슐화해주며, 서로 언제든지 바꿀 수 있도록 해줌

디자인 패턴이란

- 객체지향 기법을 활용하여 소프트웨어를 개발하면 해당 코드는 유연하고, 유지보수하기 쉽고, 재사용이 가능할 것으로 생각할 수 있음
- 객체지향 언어를 사용하였다고 하여 유연성, 유지보수 용이성, 재사용성이 그냥 제공되는 것은 아님
- 코드 설계는 예술적 요소가 있으며, 충분한 경험 없이는 효과적인 코드를 만들기는 쉽지 않음. 하지만 경험 많은 개발자가 유사한 문제에 대해 이미 만든 해결책들이 있으며, 설계 패턴이 이와 같은 해결책을 말함
- 디자인 패턴은 자주 발생하는 문제들에 대한 재사용이 가능한 객체지향적 해결책임
  - 패턴은 완전한 코드는 아니며, 보통 코드 골격처럼 사용할 수 있음
  - GoF는 총 23개의 패턴을 소개하였음

패턴의 분류 (1/2)

- 패턴 이름: 널리 사용되는 단일 이름이 있을 수 있고, 여러 개의 이름으로 불리는 경우도 있음
  - 개발자들간에 의사소통할 때 사용하는 용어가 됨
- 패턴 특성
  - 패턴의 종류
    - 생성(creational): 객체의 생성과 관련 (single or component)
      - 생성하는 객체의 구체적인 클래스와 이들의 생성방법을 숨김
    - 행위(behavioral): 객체와 객체 간 상호작용 관련 (component)
      - 알고리즘과 책임의 할당, 객체간의 상호작용과 관련된 패턴
      - 보통 상속보다는 포함관계를 활용
    - 구조(structural): 객체 간에 정적인 구조적 관계와 관련 (architectural)
      - 클래스와 객체들을 어떻게 구성하여 보다 큰 구조를 형성하는 방법과 관련된 패턴
  - 패턴의 수준: Single Class, Component, Architectural

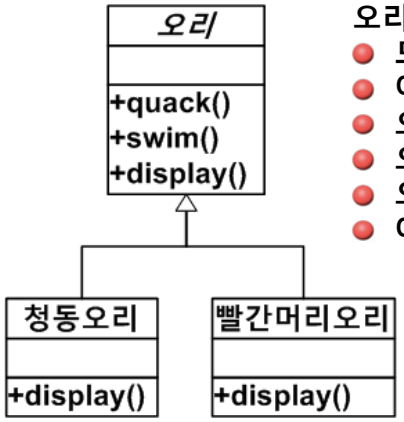
패턴의 분류 (2/2)

- 패턴 목적: 사용하는 목적
- 패턴 applicability: 적용할 수 있는 문제나 상황(context)
- 패턴의 장단점: 프로그래밍언어 관련 이슈와 구현 이슈 포함
- 패턴의 해결책
- 패턴의 변형
- 관련 패턴

패턴을 사용한 이유 및 선택방법

- 패턴을 사용하는 이유
  - 유연성: 코드의 유연성 확보
    - 올바른 수준의 추상화 제공, 객체간에 낮은 결합성 제공
  - 재사용성: 코드의 재사용 가능성을 높여줌
  - 개발자간 의사소통 강화
  - 가장 좋은 해결책의 사용
    - 설계 패턴들을 널리 검증된 해결책임
- 패턴을 선택하는 방법
  - 유사한 패턴이 많기 때문에 올바른 패턴을 선택하는 것이 쉽지 않음
  - 올바른 패턴을 선택하기 위해서는 각 패턴에 대한 이해가 충분히 있어야 함
  - 가장 먼저/쉽게 할 수 있는 일은 어떤 종류의 패턴을 적용해야 하는 것인지 부터 파악하는 것 (생성, 행위, 구조)

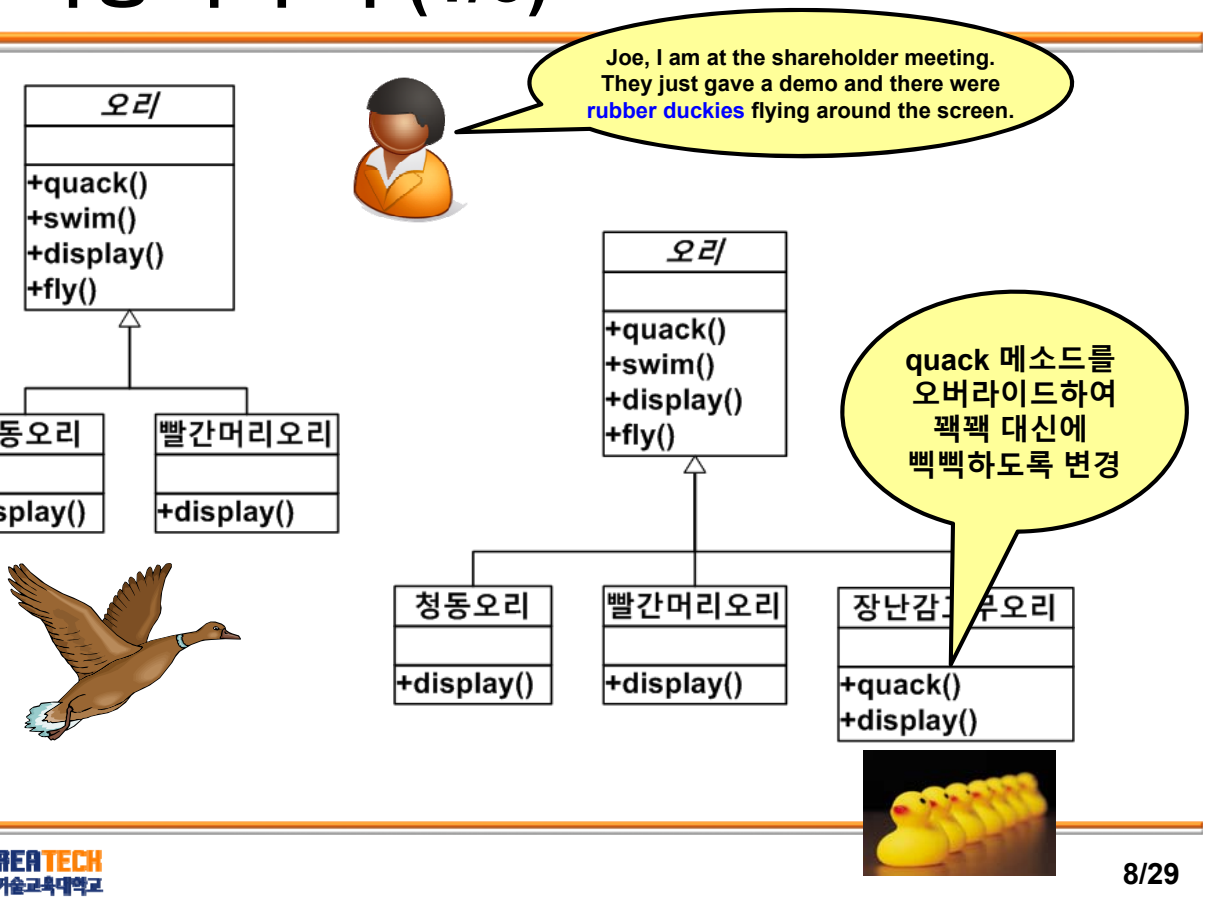
SimpleUDuck Application



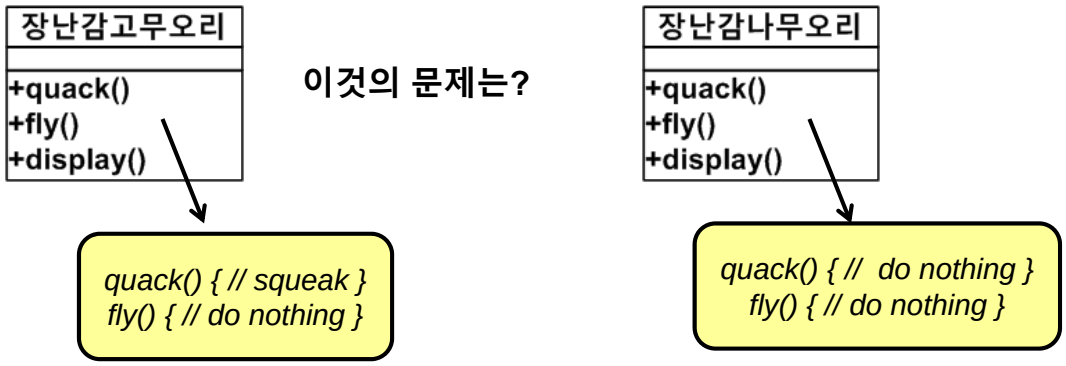
오리 시뮬레이션 프로그램
- 모든 오리는 꽥꽥 울 수 있고, 수영을 할 수 있음
- 이 두 종류의 행위는 오리 부모 클래스에서 구현됨
- 오리의 display 메소드는 추상메소드임
- 오리 클래스는 추상 클래스임
- 오리의 모든 자식클래스는 display 메소드를 구현해야 함
- 여기 두 클래스 외에 다양한 클래스가 상속되어 구현될 수 있음

오리 시뮬레이션 프로그램을 개선하기 위해
fly 기능을 추가하고 싶음
→ 어떻게? 오리 클래스에 fly 메소드 추가?

fly 기능의 추가 (1/3)



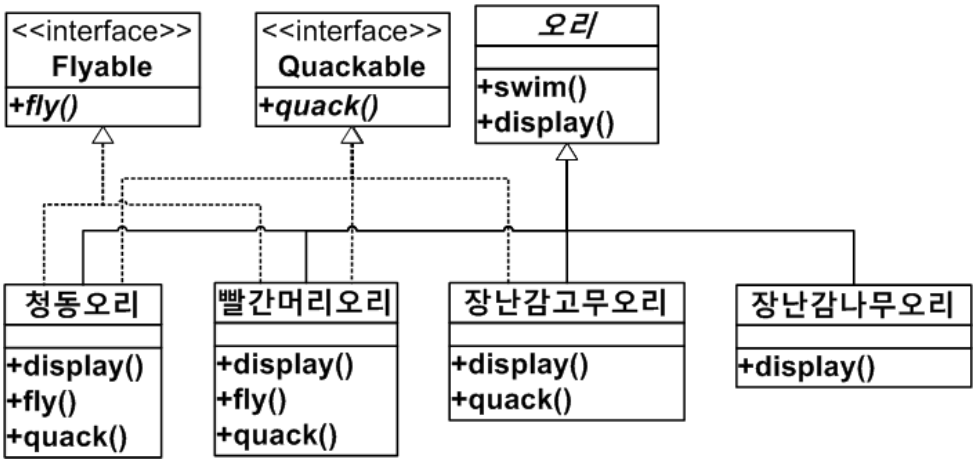
fly 기능의 추가 (2/3)

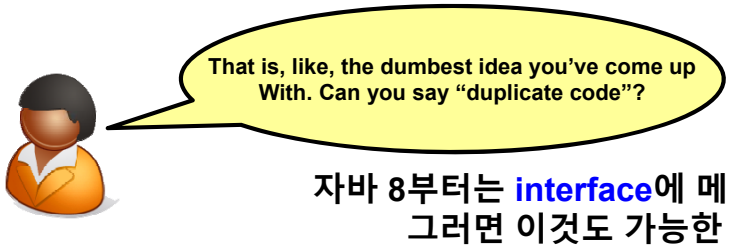


Which of the following are disadvantages of using inheritance to provide duck behavior?

A. *Code is duplicated across subclass*
B. *Runtime behavior changes are difficult*
C. *We can't make duck to dance*
D. *Hard to gain knowledge of all duck behavior*
E. *Ducks can't fly and quack at the same time*
F. *Changes can unintentionally affect other ducks.*

fly 기능의 추가 (3/3)





구현해야 하는 메소드의 수?

자바 8부터는 interface에 메소드 정의가 가능함
그러면 이것도 가능한 해결책인가?

관련 설계 원리 (1/3)

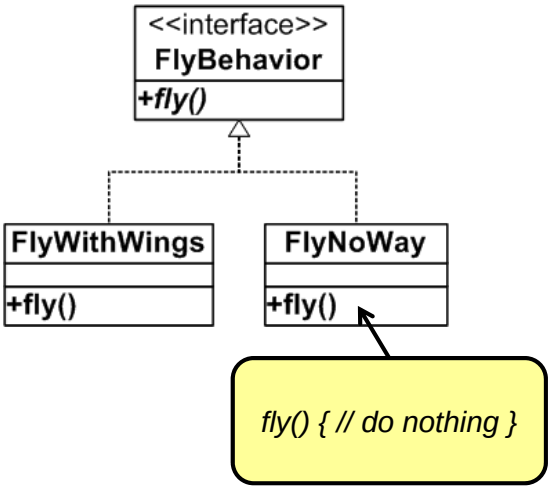
Design Principle 1
Identify the aspects of your application that vary and separate them from what stays the same

- 상속도 별로 좋은 해결책이 아님
- 자바 인터페이스 기능도 좋은 해결책이 아님
  - 자바 인터페이스는 구현을 포함하지 않음 → 재사용이 아님
  - 자바 8부터는 가능 하지만 한계가 있음
- 그러면 어떻게???
- 오리 클래스에서 fly 기능과 quack 기능은 특정 오리마다 다름
  - 이것을 오리 클래스로부터 분리해야 함
  - 어떻게? fly 기능과 quack 기능을 별도 클래스로
    - 이것은 또 무슨 소리? 행위를 클래스로? 그렇게 하지 말라고 했는데… 쏘리

관련 설계 원리 (2/3)

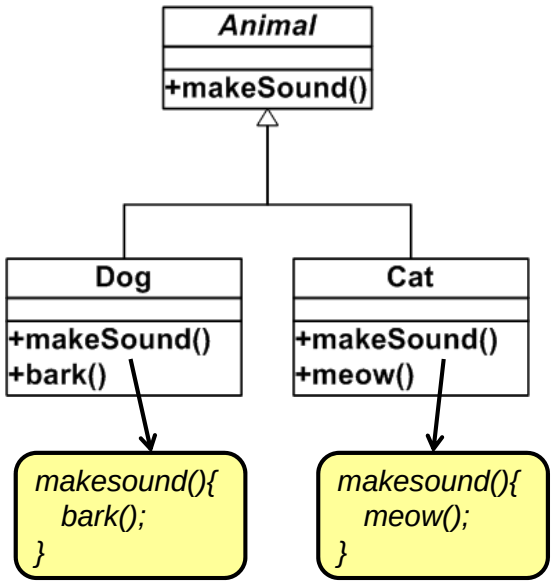
Design Principle 2
Program to an interface, not an implementation

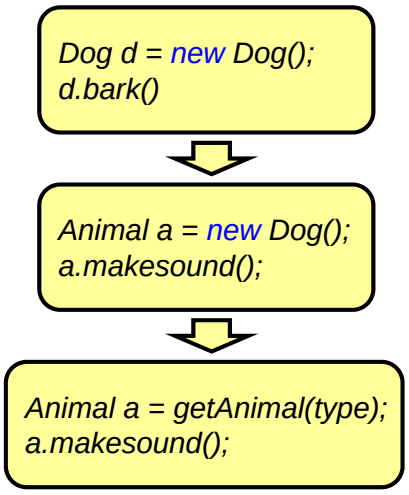
- 말이 너무 어려움
- FlyWithWings와 FlyNoWay는 오리 클래스와 어떻게 연결?
  - Has-a 관계로



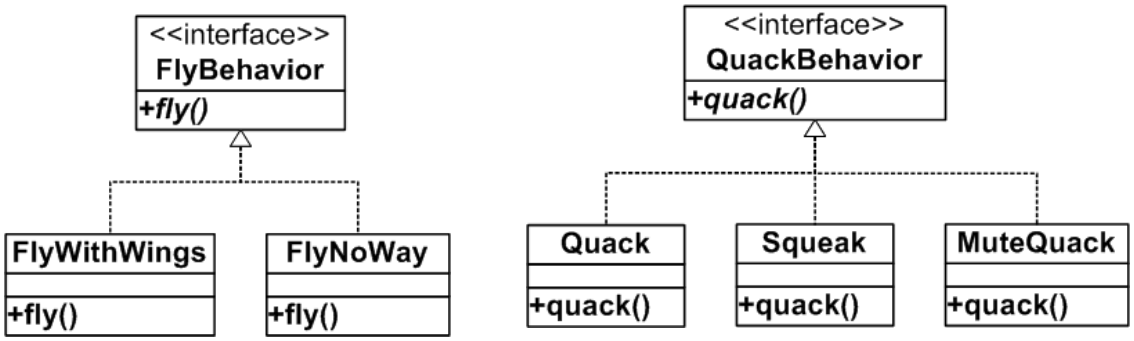
관련 설계 원리 (3/3)

- 예) 설계 원리2





fly 기능의 올바른 구현 (1/3)



- 이 방식의 장점
  - 코드 재사용이 가능
    - 날개로 나는 모든 오리에 대한 구현은 한 번
    - 오리의 상태를 활용한 구현은?
  - 기존 클래스를 변경하지 않고 새 행위를 추가하기 쉬움
    - 클래스의 복잡성을 줄일 수 있음
  - 조건문을 사용하지 않고 구현가능

fly 기능의 올바른 구현 (2/3)

오리
-flyBehavior : FlyBehavior -quackBehavior : QuackBehavior
+swim() +display() +performQuack() +performFly()

```
public abstract class Duck{
   protected FlyBehavior flyBehavior;
   protected QuacKBehavior quackBehavior;
   …
   public void performQuack(){
      quackBehavior.quack();
   }
}
```

```
public class MallardDuck extends Duck{
   public void MallardDuck(){
      flyBehavior = new FlyWithWings();
      quackBehavior = new Quack();
   }
   public void display(){
      System.out.println(" 난 청동오리야!");
   }
}
```

복합관계

fly 기능의 올바른 구현 (3/3)

- 장점의 예1) flyBehavior나 quackBehavior를 실행시간에 변경할 수 있음

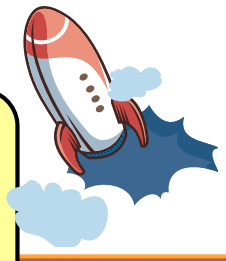
```
d.setFlyBehavior(new FlyNoWay());
d.fly();
```

- 장점의 예2) 새로운 오리 타입을 만들기 쉬움

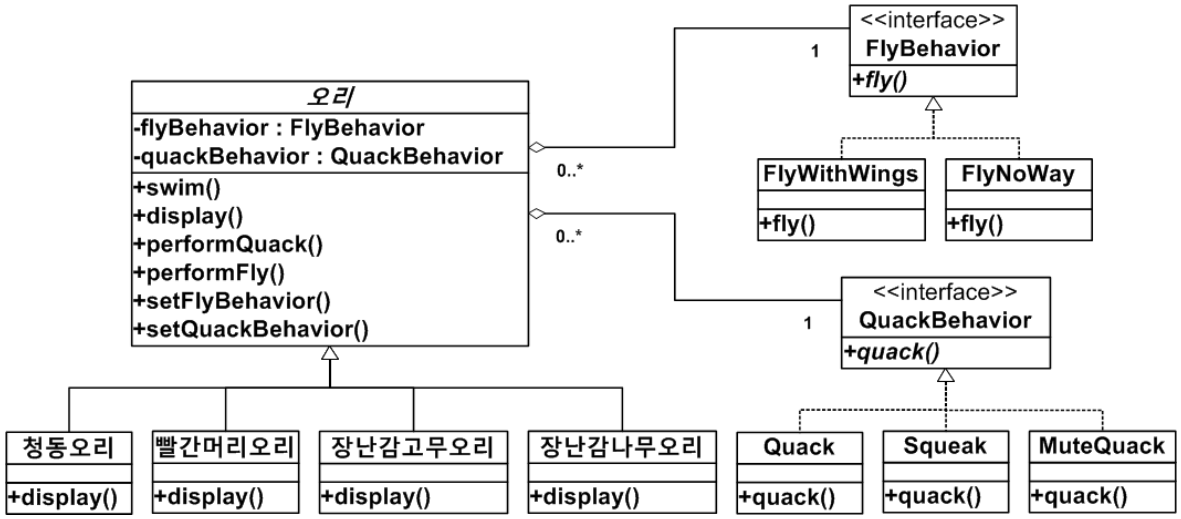
```
public class ModelDuck extends Duck{
   public ModelDuck(){
      flyBehavior = new FlyNoWay();
      quackBehavior = new Quack();
   }
   …
}
```

- 장점의 예3) 새로운 행위 추가가 쉬움

```
public class FlyRocketPowered implements FlyBehavior {
   public void fly(){
      System.out.println(" 난 로켓으로 난다!!!");
   }
}
```



최종결과



Has-a는 is-a보다 좋다

Design Principle 3
Favor composition over inheritance.

- is-a를 이용하는 것보다 has-a를 이용하는 것이 보다 유연한 해결책을 보통 가져다 줌
  - 특히, 실행시간 행위를 변경할 수 있음

Strategy Pattern (1/3)

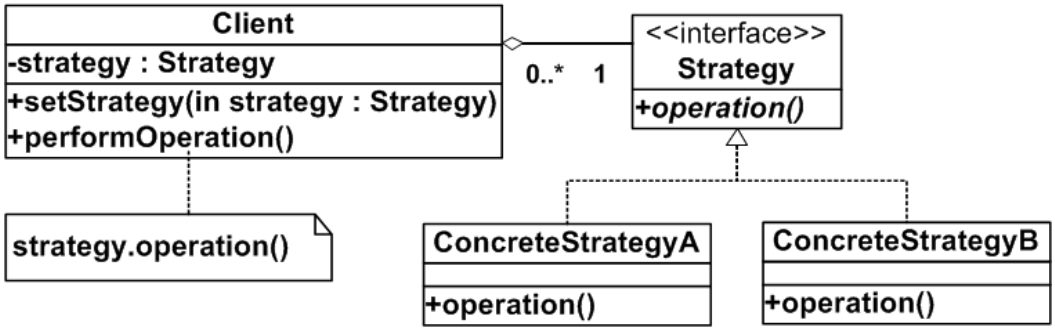
- 이번에 오리 시뮬레이션 프로그램 문제를 해결하기 위해 사용한 방법이 바로 전략 패턴(Strategy Pattern)임
- Strategy Pattern: 알고리즘의 군을 정의하고 캡슐화해주며, 서로 언제든지 바꿀 수 있도록 해줌
  - 객체의 상태에 따라 행위를 동적으로 바꿀 수 있도록 해줌
    - 당연한 것이지만 조건문을 사용하지 않음
- 등장 클래스
  - Strategy 인터페이스: 전략을 이용하기 위한 인터페이스 제공
  - Concrete Strategy 클래스: 다수가 제공되며, 구체적인 전략이 구현되어 있는 클래스
  - Client 클래스: Context 클래스라고도 하며, 전략을 실제 활용하는 클래스
- 관련 리펙토링: Replace Conditional Logic with Strategy

Strategy Pattern (2/3)

- 전략 패턴을 언제 사용하나?
  - 관련 있는 클래스들이 행위만 다를 경우
  - 한 알고리즘의 다양한 변형이 필요한 경우
  - 알고리즘이 클라이언트가 몰라야 하는 데이터를 사용할 경우
  - 한 클래스가 여러 행위를 조건문을 통해 정의할 경우

Strategy Pattern (3/3)

- 전략 패턴 관계도



- 장점
  - 위임이라는 느슨한 연결을 통해 전략을 쉽게 바꿀 수 있으며, 실행 중에도 변경 가능함
- 단점(고민)
  - 행위의 모델링(인터페이스, operation의 정의)이 쉽지 않을 수 있음
    - 클라이언트의 상태가 필요한 경우는 어떻게?

전략이 클라이언트 상태가 필요한 경우

- 클라이언트 자체를 전략에 전달하는 것은 결합성이 높아지는 단점이 있음
  - 특히, 특정 클라이언트에서만 해당 전략을 사용할 수 있음
    - 추가적으로 불필요한 getter들을 만들어야 할 수 있음
  - 꼭 필요한 데이터만 전달하는 것이 바람직함

```
public void performOperation(){
    strategy.doOperation(this);
  }
}
```

Strategy Pattern 예

- 예) 게임에서 유니트의 동작
- 예) 편집기의 포맷팅 기능
- 예) 연락처에 대한 다양한 뷰
- 예) 통계자료에 대한 다양한 뷰
- 예) 자동차의 브레이크 시스템 (ABS 유무)

Strategy Pattern 요약

- 종류: Behavioral
- 수준: Component
- 목적: 가능한 행위들을 정의하는 클래스의 집합을 정의하여 필요한 행위를 유연하게 사용할 수 있으며, 동적으로 변경 가능함
- Applicability
  - 한 행위를 다양하게 수행해야 하는 경우. 예) fly
  - 실행시간까지 객체의 행위 방법을 결정할 수 없는 경우
    - 동적으로 행위를 수행하는 방법을 결정할 수 있음
  - 행위를 수행하는 방법을 쉽게 추가하고 싶은 경우
    예) flyRocketPowered
  - 행위를 추가하면서 코드의 크기를 효과적으로 관리하고 싶을 경우
- 관련 패턴
  - Singleton: Strategy 클래스의 인스턴스는 보통 Singleton으로 표현됨

자바 enum

```
public enum Flyable{
   FlyNoWay{
      @Override
      public void fly(){}
   }
   FlyWithWings{
      @Override
      public void fly(){
         System.out.println(" 난 날개로 난다!!!");
      }
   }
   FlyWithRockets{
      @Override
      public void fly(){
         System.out.println(" 난 로켓으로 난다!!!");
      }
   }
   public abstract void fly();
}
```

```
public class MallardDuck extends Duck {
        public MallardDuck(){
          setFlyStrategy(Flyable.FlyWithWings);
        }
```


자바 Lambda

```
@FunctionalInterface
public interface Flyable {
      void fly();
}
```

```
Flyable flyWithBallons = ()->System.out.println("풍선으로 날고 있음");
Duck d = new MallardDuck();
d.display();
d.setFlyStrategy(flyWithBallons);
d.fly();
```

C++

```
class FlyBehavior {
public:
    FlyBehavior();
    virtual ~FlyBehavior();
    virtual void fly() = 0;
};

class FlyWithWings: public FlyBehavior{
public:
    void fly(){
        cout << "Fly with wings!" << endl;
    }
};

class FlyNoWay: public FlyBehavior{
public:
    void fly(){
        cout << "I can't fly!" << endl;
    }
};

class FlyWithRocket: public FlyBehavior{
public:
    void fly(){
        cout << "Fly with rocket!" << endl;
    }
};
```

```
class Duck {
private:
    unique_ptr<FlyBehavior> flyBehavior;
public:
    Duck(){}
    virtual ~Duck(){}
    void fly(){
        flyBehavior->fly();
    }
    void setFlyBehavior(
        unique_ptr<FlyBehavior> fb){
        flyBehavior = move(fb);
    }
    virtual void display() = 0;
};

class MallardDuck : public Duck
{
public:
    MallardDuck(){
        setFlyBehavior(
            unique_ptr<FlyBehavior>(
                new FlyWithWings()));
    }
    void display() override {
        cout << "I'm a mallard duck!"
            << endl;
    }
};
```

Python

```
class FlyBehavior(object):
    __metaclass__ = ABCMeta
    def fly(self):
        pass

class FlyWithRocket(FlyBehavior):
    def fly(self):
        print "Flying with rocket"

class FlyWithWings(FlyBehavior):
    def fly(self):
        print "Flying with wings"

class FlyNoWay(FlyBehavior):
    def fly(self):
        print "Can't fly"
```

```
class Duck(object):
    __metaclass__ = ABCMeta
    def setFlyingBehavior(self, flybehavior):
        if not isinstance(flybehavior, FlyBehavior):
            raise TypeError("must use FlyBehavior")
        self.flybehavior = flybehavior
    def fly(self):
        self.flybehavior.fly()
    @abstractmethod
    def display(self):
        pass

class MallardDuck(Duck):
    def __init__(self):
        self.flybehavior = FlyWithWings()
    def display(self):
        print "I'am a MallardDuck"
```

Javascript

```
var inherits = function(child, parent) {
    child.prototype =
            Object.create(parent.prototype);
    child.prototype.constructor = child;
};

var Duck = function(){
    this.flyable = null;
};
Duck.prototype = {
    fly: function(){
        this.flyable.fly();
    },
    setFlyBehavior: function(flyable){
        this.flyable = flyable;
    },
    display: function(){}
};

var MallardDuck = function(){
    Duck.call(this);
    this.flyable = new FlyWithWings();
};
inherits(MallardDuck, Duck);
MallardDuck.prototype.display
    = function(){
    console.log("I am a Mallard Duck!!!");
};
```

```
var Flyable = function(){};

Flyable.prototype.fly = function(){}

var FlyWithWings = function(){
    Flyable.call(this);
};
var FlyWithRocket = function(){
    Flyable.call(this);
};
var FlyNoWay = function(){
    Flyable.call(this);
};
inherits(FlyWithWings, Flyable);
inherits(FlyWithRocket, Flyable);
inherits(FlyNoWay, Flyable);

FlyWithWings.prototype.fly
    = function(){
    console.log("flying with wings");
};
FlyWithRocket.prototype.fly
    = function(){
    console.log("flying with rocket");
};
FlyNoWay.prototype.fly = function(){
    console.log("can't fly");
};
```