고급객체지향 프로그래밍
강의노트 #03

Strategy Pattern

조용주
ycho@smu.ac.kr

Design Patterns 분류

- GoF 가 디자인 패턴을 23 가지로 정리하고 세 가지로 크게 분류
- 생성 패턴 (Creation Patterns)
  - 객체의 생성 과정과 연관된 패턴
    - **추상 팩토리 (Abstract Factory)**
    - 빌더 (Builder)
    - **팩토리 메소드 (Factory Method)**
    - 프로토타입 (Prototype)
    - **싱글턴 (Singleton)**

Design Patterns 분류

- 구조 패턴 (Structural Patterns)
  - 클래스나 객체의 합성 / 집약에 관련된 패턴
    - **어댑터 (Adapter)**
    - 브리지 (Bridge)
    - **컴포지트 (Composite)**
    - **데코레이터 (Decorator)**
    - 퍼사드 (Façade)
    - 플라이웨이트 (Flyweight)
    - **프록시 (Proxy)**

Design Patterns 분류

- 행위 패턴 (Behavioral Patterns)
  - 클래스나 객체들이 상호작용하는 방법과 책임을 분산시키는 방법을 정의하는 패턴
    - 책임 연쇄 (Chain of Responsibility)
    - **커맨드 (Command)**
    - 인터프리터 (Interpreter)
    - **반복자 (Iterator)**
    - 미디에이터 (Mediator)
    - 메멘토 (Memento)
    - **옵서버 (Observer)**
    - **스테이트 (State)**
    - **스트래티지 (Strategy)**
    - **템플릿 메소드 (Template Method)**
    - 비지터 (Visitor)

스트래티지 패턴 (Strategy Pattern)

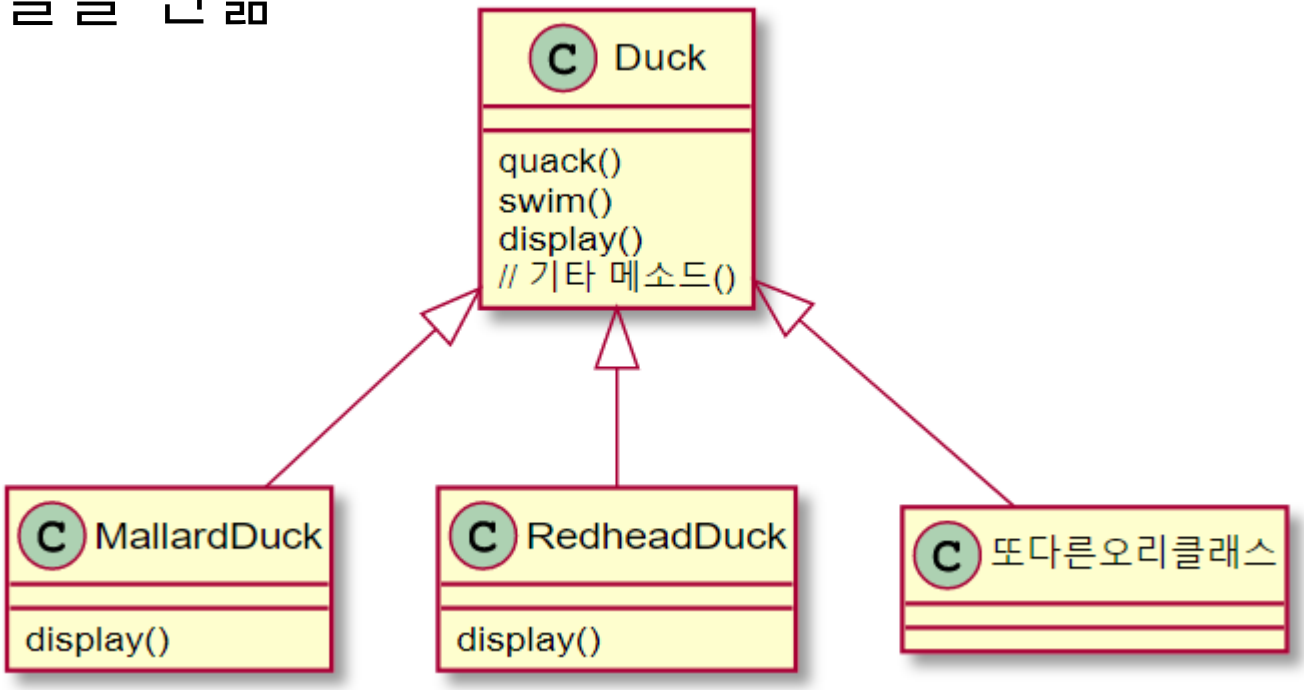
- 목적
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it
- 폴리시 패턴 (Policy Pattern) 이라고 부르기도 함
  - 여러 정책 (policy) 이 존재하고, 상황에 따라 적합한 정책을 적용시킴
- 서로 다른 알고리즘들이 존재하고, 실행 중 적합한 알고리즘을 선택해서 적용
  - 클라이언트에 모든 알고리즘을 포함시키는 것은 클라이언트 코드의 양이 늘어나고 복잡해짐 ◊ 유지 보수 어려움
  - 모든 알고리즘이 동시에 사용되는 것이 아니면 굳이 함께 넣어야 할 필요 없음
  - 새로운 알고리즘 추가가 어려움. 기존 코드를 수정해야 함

스트래티지 패턴 (Strategy Pattern)

- 예
  - 조리법이 다른 경우
  - 파일의 압축 방법이 다른 경우
  - 영화를 보는 방식이 다른 경우 (초대권, 멤버쉽 할인 등)
  - 자바의 정렬
    - Comparator 인터페이스를 이용하는 경우, 서로 다른 비교 방법을 구현하고 실행 시점에 적절한 방법을 선택

사례 1 – Duck (HFDP Ch. 1)

- Version 1
  - SimUDuck 이라는 오리 연못 시뮬레이션 게임 개발
    - 헤엄 치고 꽥꽥거리는 소리를 내는 다양한 오리가 있음
  - Duck 클래스를 구성하고 이로부터 상속 받아 다른 클래스들을 만듦



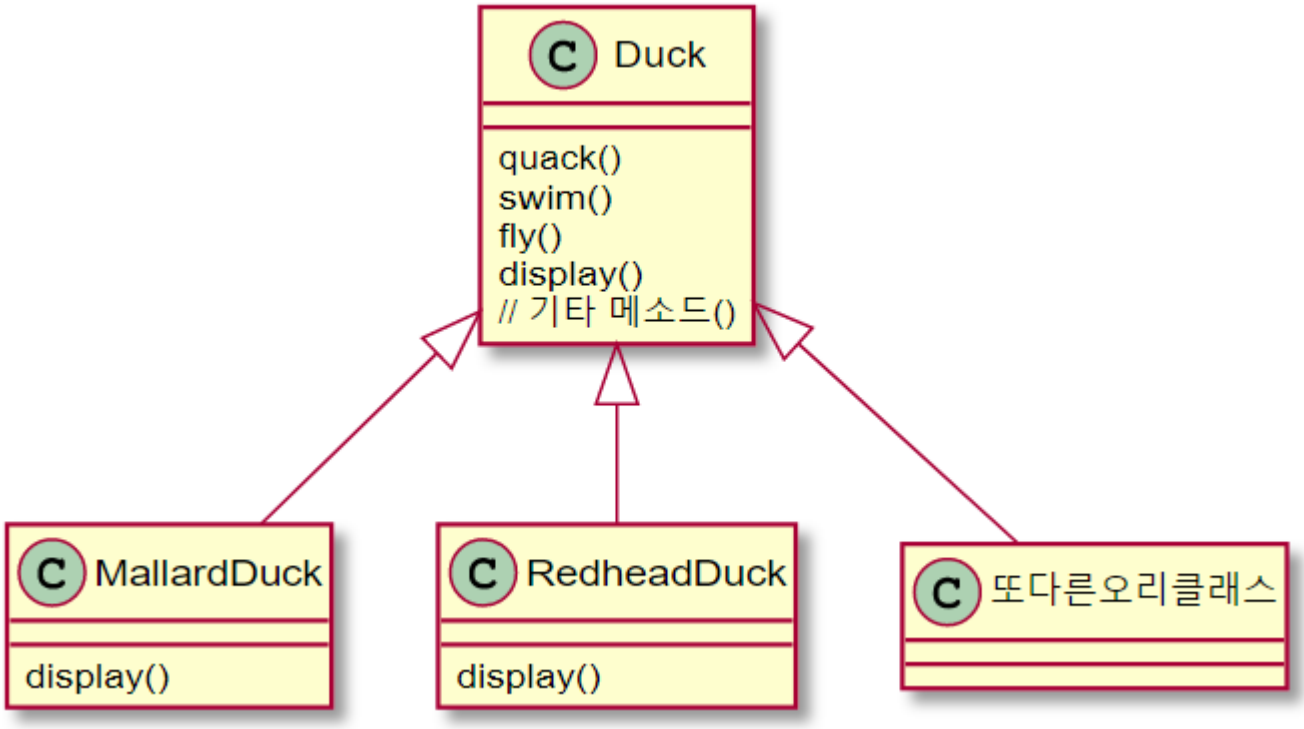
사례 1 – Duck (HFDP Ch. 1)

```
class Duck {
    void quack() {
        System.out.println("quack");
    }
    void swim() {
        System.out.println("swimming");
    }
    void display() {
        System.out.println("Duck");
    }
}

class MallardDuck extends Duck {
    void display() {
        System.out.println("MallardDuck");
    }
}
```

```
class RedheadDuck extends Duck {
    void display() {
        System.out.println("RedheadDuck");
    }
}

public class Main {
    public static void main(String[] args) {
   // write your code here
        Duck d1 = new Duck();
        Duck d2 = new MallardDuck();
        Duck d3 = new RedheadDuck();
        d1.display();
        d2.display();
        d3.display();
        d1.quack();
        d2.quack();
        d3.quack();
    }
}
```

사례 1 – Duck (HFDP Ch. 1)

- Version 2
  - 오리를 날게 하고 싶음
    - 상위 클래스인 Duck 에 fly() 기능 추가

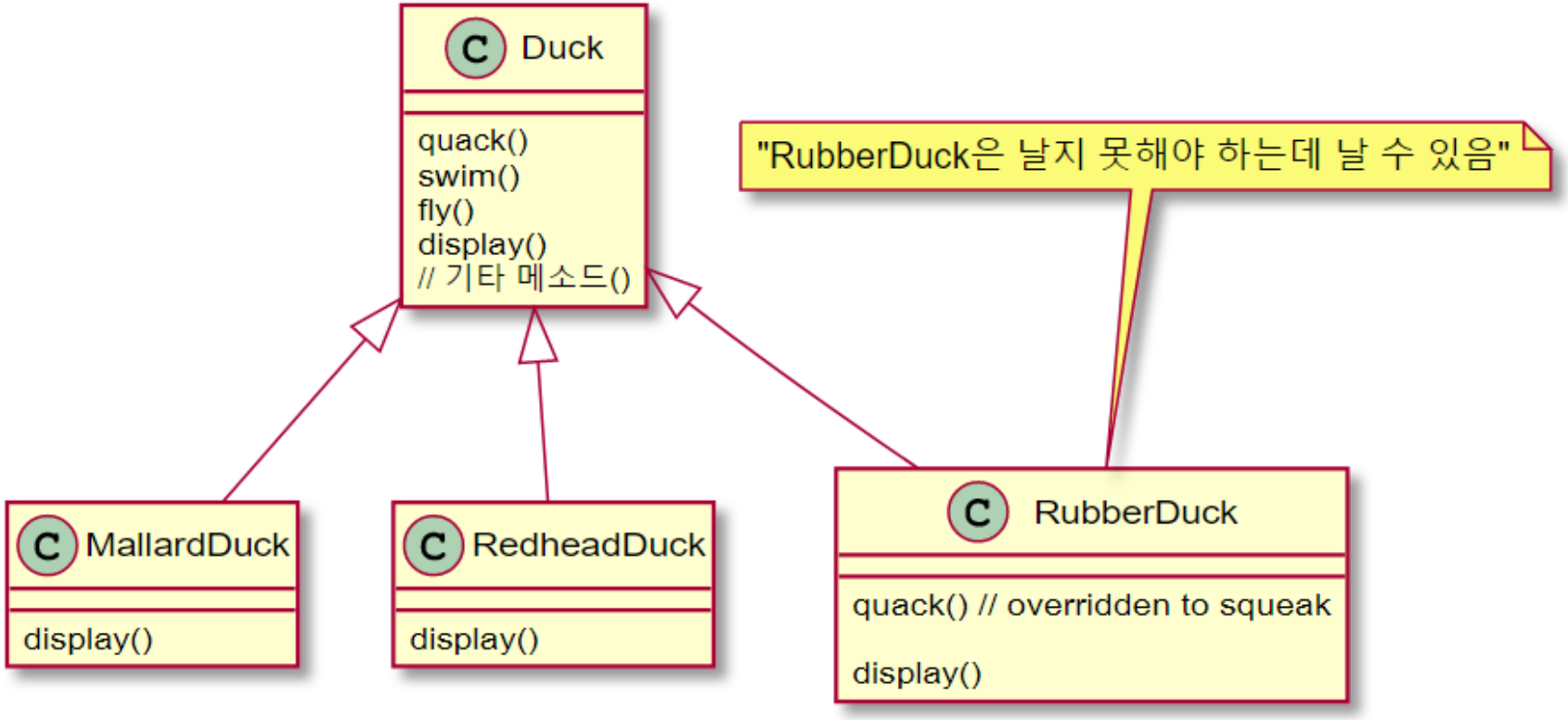


```
class Duck {
    void quack() {
        System.out.println("quack");
    }
    void swim() {
        System.out.println("swimming");
    }
    void fly() {
        System.out.println("flying");
    }
    void display() {
        System.out.println("Duck");
    }
}
class RubberDuck extends Duck {
    void quack() {
        System.out.println("squeak");
    }
    void display() {
        System.out.println("RubberDuck");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // write your code here
        Duck d1 = new Duck();
        Duck d2 = new MallardDuck();
        Duck d3 = new RedheadDuck();
        Duck d4 = new RubberDuck();
        d1.display();
        d2.display();
        d3.display();
        d4.display();
        d1.quack();
        d2.quack();
        d3.quack();
        d4.quack();
        d1.fly();
        d2.fly();
        d3.fly();
        d4.fly();
    }
}
```

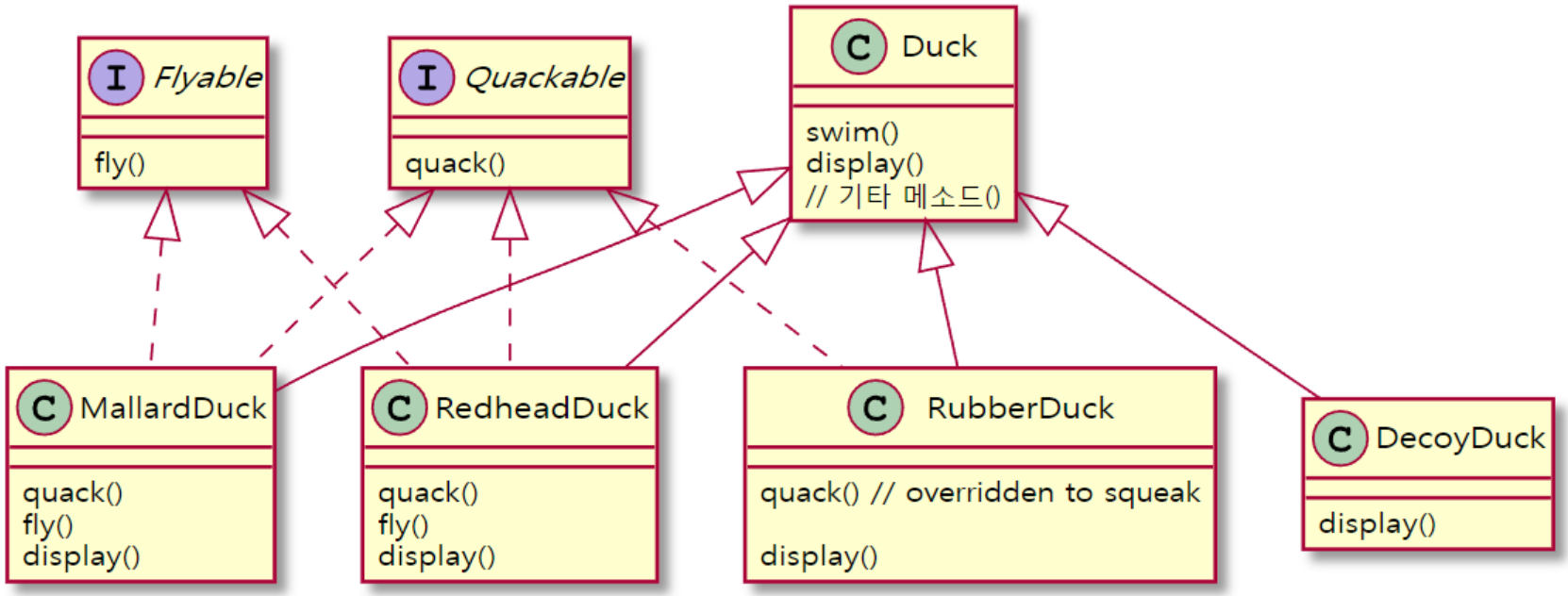
사례 1 – Duck (HFDP Ch. 1)

- 문제
  - 원하지 않는 자식 클래스에 fly() 기능이 추가됨



사례 1 – Duck (HFDP Ch. 1)

- Version 3 & 4
  - 인터페이스를 이용한다면 ?



- 인터페이스에 코드를 넣을 수 없으므로 MallardDuck, RedheadDuck, RubberDuck 등이 같은 코드를 반복해서 구현하는 경우가 발생할 수 있음 ◊ Java8 에서는 디폴트 메소드를 이용하면 일부 해결 가능 (Version 4)

사례 1 – Duck (HFDP Ch. 1)

- Version 3

```
class Duck {
    void swim() {
        System.out.println("swimming");
    }
    void display() {
        System.out.println("Duck");
    }
}

interface Flyable {
    void fly();
}

interface Quackable {
    void quack();
}
```

```
class MallardDuck extends Duck
              implements Flyable, Quackable {
    void display() {
        System.out.println("MallardDuck");
    }
    public void quack() {
        System.out.println("quack");
    }
    public void fly() {
        System.out.println("flying");
    }
}
class RedheadDuck extends Duck
            implements Flyable, Quackable  {
    void display() {
        System.out.println("RedheadDuck");
    }
    public void quack() {
        System.out.println("quack");
    }
    public void fly() {
        System.out.println("flying");
    }
}
```

```
class RubberDuck extends Duck
              implements Quackable, Flyable {
    public void quack() {
        System.out.println("squeak");
    }
    void display() {
        System.out.println("RubberDuck");
    }
    public void fly() {
        System.out.println("cannot fly");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // write your code here
        Duck d1 = new Duck();
        MallardDuck d2 = new MallardDuck();
        RedheadDuck d3 = new RedheadDuck();
        RubberDuck d4 = new RubberDuck();
        d1.display();
        d2.display();
        d3.display();
        d4.display();
        //d1.quack();
        d2.quack();
        d3.quack();
        d4.quack();
        //d1.fly();
        d2.fly();
        d3.fly();
        d4.fly();
    }
}
```

사례 1 – Duck (HFDP Ch. 1)

- Version 4

```
class Duck {
    void swim() {
        System.out.println("swimming");
    }
    void display() {
        System.out.println("Duck");
    }
}

interface Flyable {
    default void fly() {
        System.out.println("flying");
    }
}

interface Quackable {
    default void quack() {
        System.out.println("quack");
    }
}
```

```
class MallardDuck extends Duck
              implements Flyable, Quackable {
    void display() {
        System.out.println("MallardDuck");
    }
}
class RedheadDuck extends Duck
            implements Flyable, Quackable  {
    void display() {
        System.out.println("RedheadDuck");
    }
}
class RubberDuck extends Duck
      implements Quackable, Flyable {
    public void quack() {
        System.out.println("squeak");
    }
    void display() {
        System.out.println("RubberDuck");
    }
    public void fly() {
        System.out.println("cannot fly");
    }
}
```

사례 1 – Duck (HFDP Ch. 1)

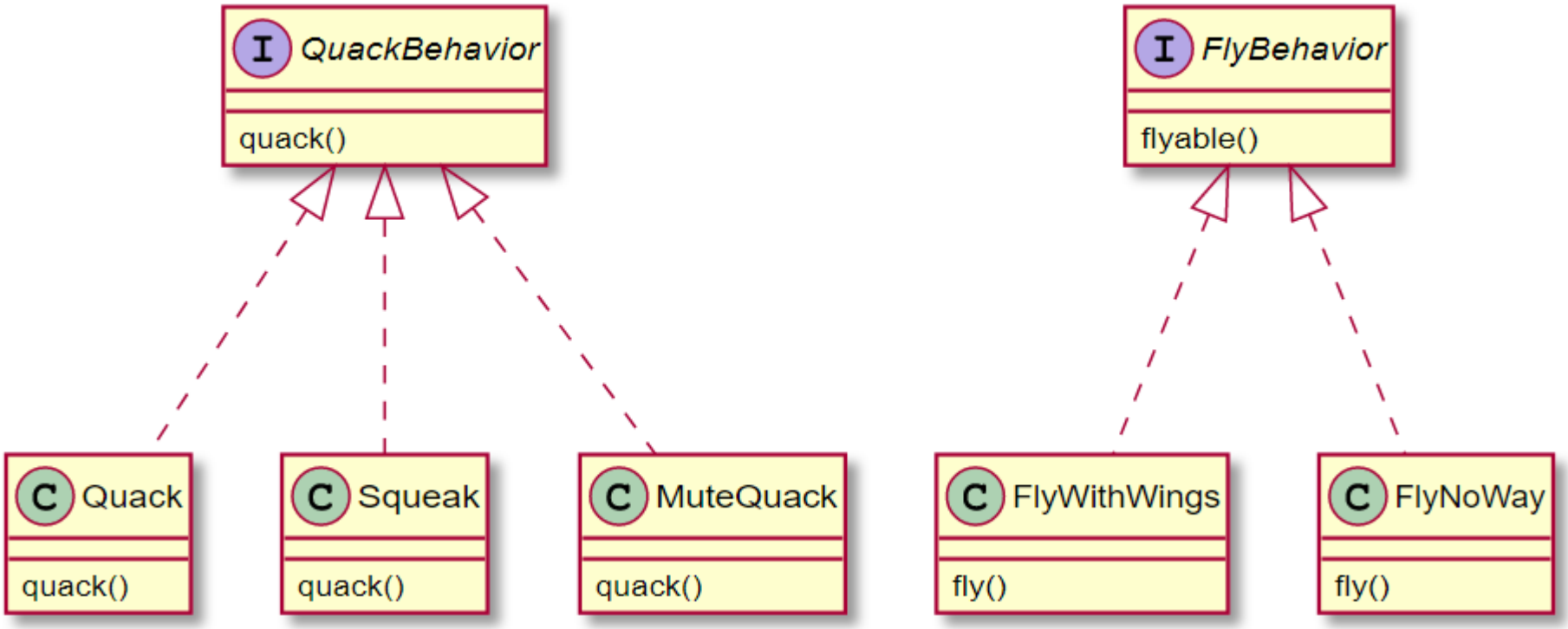
- 바뀌는 부분과 그렇지 않은 부분 분리하기
  - Duck 클래스에서 fly() 와 quack() 부분이 자주 바뀜
  - 나머지 코드는 변함없음
  - " 변화하는 부분과 그대로 있는 부분 " 을 분리하려면 두 개의 클래스 집합을 만들어야 함
    - 각 클래스 집합에는 각각의 행동을 구현한 것을 넣을 것
      - 나는 것과 관련된 집합
      - 꽥꽥거리는 것과 같은 집합
    - 특정 행동을 Duck 클래스에서 구현하는 것이 아니라, 독립적으로 새로운 클래스를 만들어서 구현
    - Duck 또는 서브 클래스에서는 행동을 실제로 구현한 인터페이스를 사용 (FlyBehavior 또는 QuackBehavior)
      - 따라서 Duck 과는 무관해짐

사례 1 – Duck (HFDP Ch. 1)

- Version 5

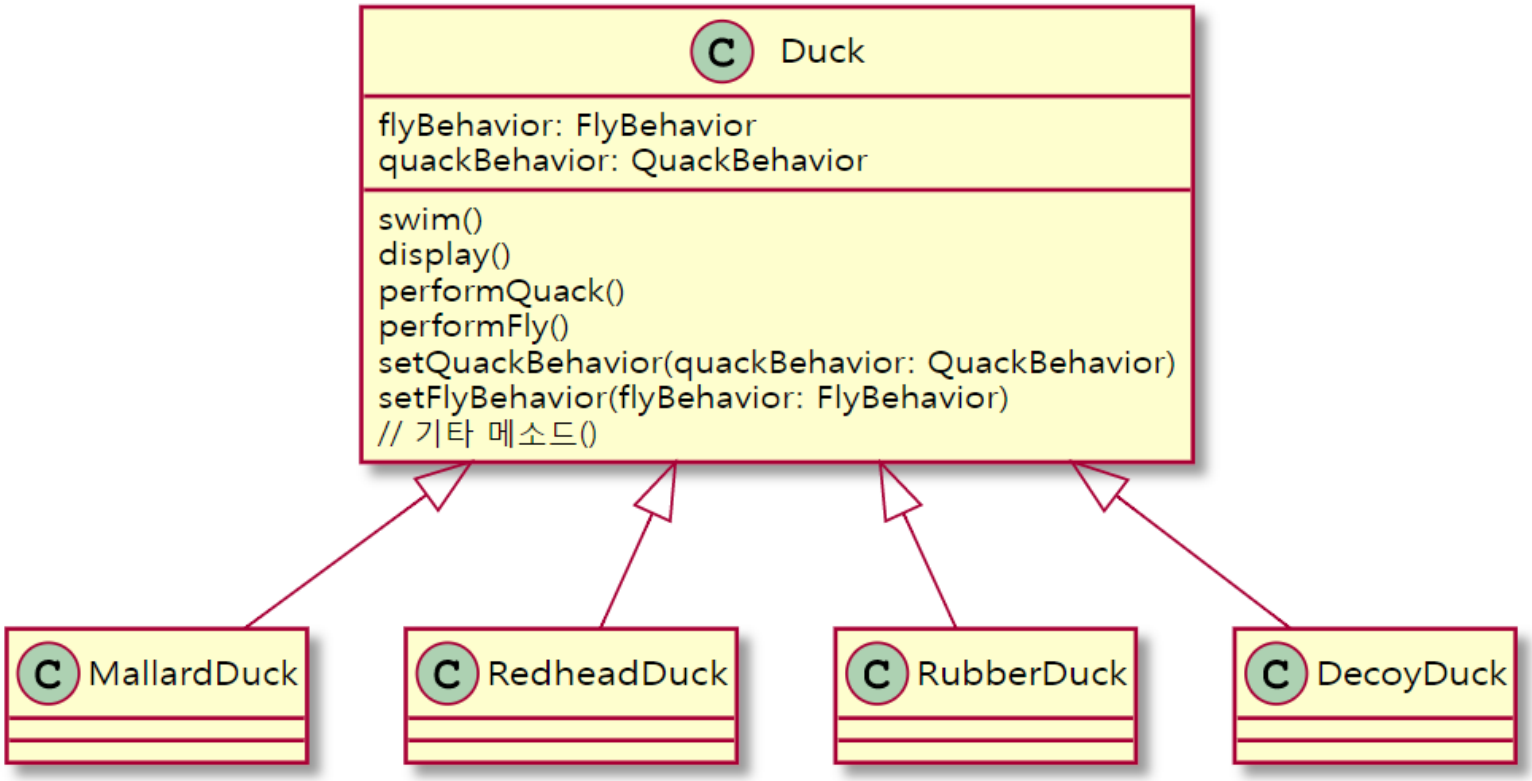
캡슐화된
꽥꽥거리는 행동

캡슐화된
나는 행동



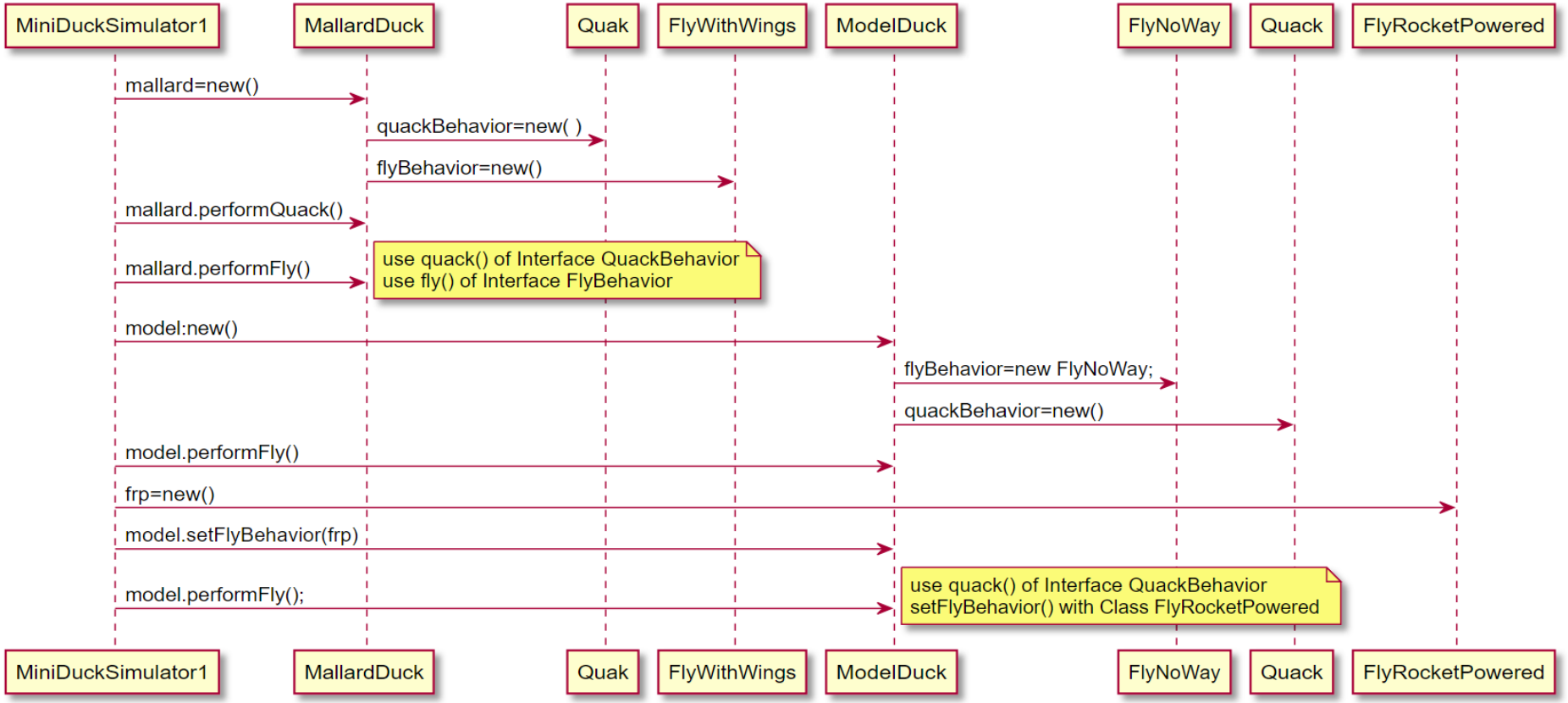
사례 1 – Duck (HFDP Ch. 1)

클라이언트



사례 1 – Duck (HFDP Ch. 1)

- 시퀀스 다이어그램



사례 1 – Duck (HFDP Ch. 1)

- ModelDuck 의 생성자에서 생성한 FlyBehavior 를 사용하고 있다 .

```
flyBehavior = new FlyNoWay();
```

- 또한 set 함수를 통해 FlyBehavior 를 수정하고 있다 .

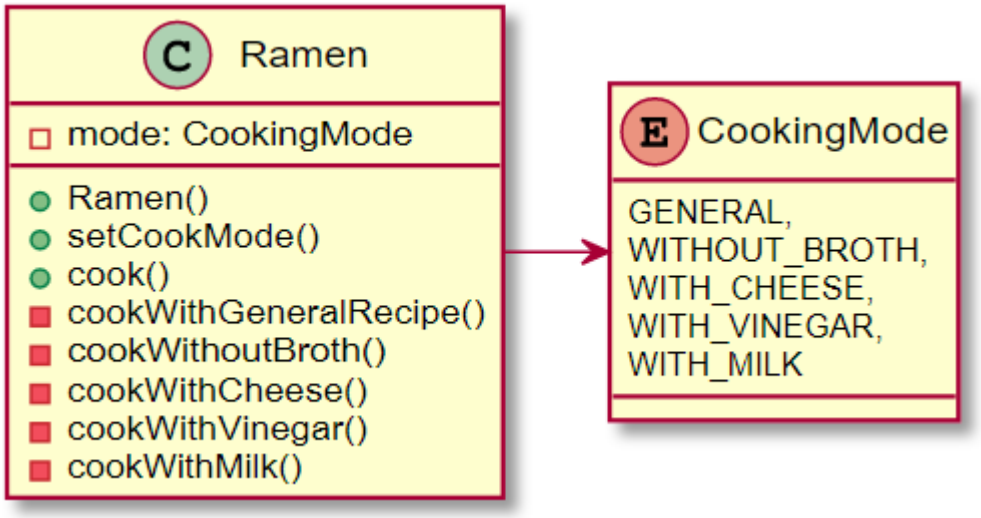
```
model.setFlyBehavior(new FlyRocketPowered());
```

사례 2 – 라면 조리

- 조리 방법
  - 기본 조리 (cook)
  - 볶음 라면 (cookWithoutBroth)
    - 머그 컵 한 잔 정도의 물로 라면 끓이고 스프 2/3 정도 넣고 볶기
  - 치즈 라면 (cookWithCheese)
    - 끓인 후 슬라이스 치즈 얹기
  - 식초 라면 (cookWithVinegar)
    - 끓인 후 작은 숟가락으로 1 개 정도 추가
  - 우유 라면 (cookWithMilk)
    - 물 대신 우유로 라면 끓이기

사례 2 – 라면 조리

- Version 1
  - 클라이언트에 모든 조리법을 넣고 if 문 또는 switch 문으로 조리법 선택



  - 문제점 :
    - 새로운 조리 방법 추가 어려움

사례 2 – 라면 조리

```
class Ramen {
    public static enum CookingMode {
        GENERAL,
        WITHOUT_BROTH,
        WITH_CHEESE,
        WITH_VINEGAR,
        WITH_MILK
    }

    private CookingMode mode;

    Ramen() {
        mode = CookingMode.GENERAL;
    }

    public void setCookMode(CookingMode mode) {
        this.mode = mode;
    }
```

사례 2 – 라면 조리

```
public void cook() {
    switch (mode) {
        case GENERAL:
            cookWithGeneralRecipe();
            break;
        case WITHOUT_BROTH:
            cookWithoutBroth();
            break;
        case WITH_CHEESE:
            cookWithCheese();
            break;
        case WITH_VINEGAR:
            cookWithVinegar();
            break;
        case WITH_MILK:
            cookWithMilk();
            break;
    }
}
```

사례 2 – 라면 조리

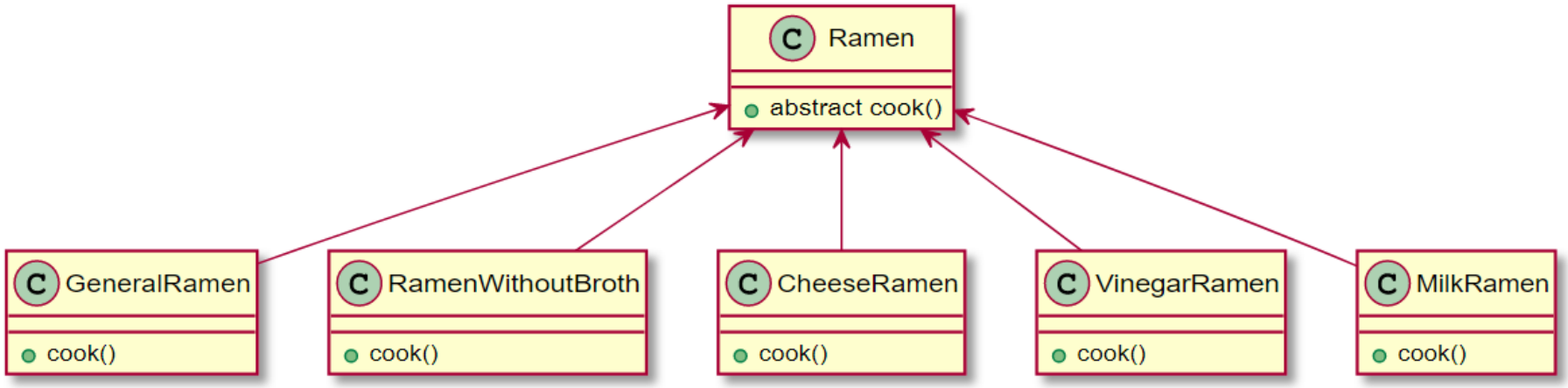
```
private void cookWithGeneralRecipe() {
        System.out.println(" 일반 조리법으로 끓이기 ");
    }
    private void cookWithoutBroth() {
        System.out.println(" 물을 적게 넣고 라면을 익힌
뒤에 라면 스프에 볶듯이 끓임 ");
    }
    private void cookWithCheese() {
        System.out.println(" 라면을 끓인 후에 치즈 넣
기 ");
    }
    private void cookWithVinegar() {
        System.out.println(" 라면을 끓인 후에 식초 약간
넣기 ");
    }
    private void cookWithMilk() {
        System.out.println(" 우유를 넣고 끓이기 ");
    }
}
```

사례 2 – 라면 조리

```
public class Main {
    public static void main(String[] args) {
        Ramen cook = new Ramen();
        cook.cook();

        cook.setCookMode(
                  Ramen.CookingMode.WITH_CHEESE);
        cook.cook();
    }
}
```

사례 2 – 라면 조리

- Version 2
  - 상속 사용



  - 문제점
    - 음식 모형 (FoodModel) 을 추가한다면 ?
      - cook() 을 오버라이드 해서 실제 요리하지 않도록 함
    - 새로운 클래스가 추가될 때마다 cook() 함수를 확인해야 함

사례 2 – 라면 조리

```
abstract class Ramen {
    public abstract void cook();
}

class GeneralRamen extends Ramen {
    public void cook() {
        System.out.println(" 일반 조리법으로 끓이기 ");
    }
}

class RamenWithoutBroth extends Ramen {
    public void cook() {
        System.out.println(" 물을 적게 넣고 라면을 익힌
뒤에 라면 스프에 볶듯이 끓임 ");
    }
}
```

사례 2 – 라면 조리

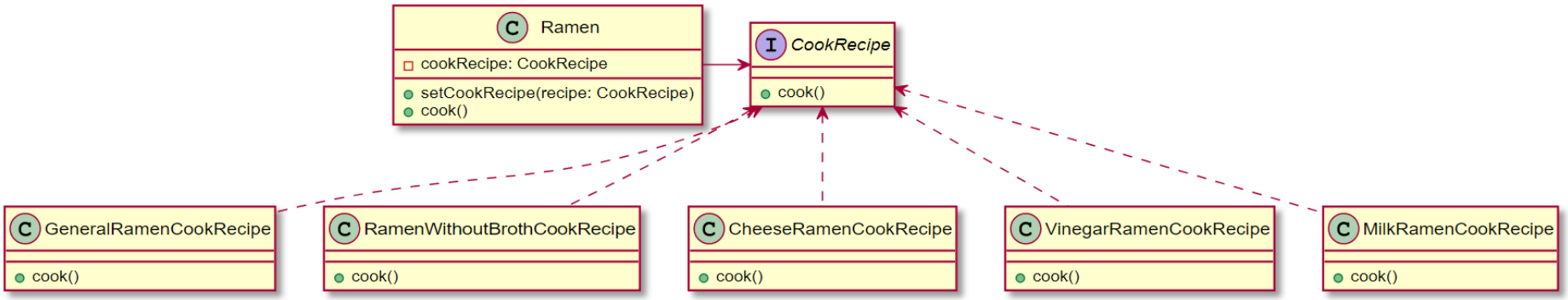
```
class CheeseRamen extends Ramen {
    public void cook() {
        System.out.println("라면을 끓인 후에 치즈 넣기");
    }
}

class VinegarRamen extends Ramen {
    public void cook() {
        System.out.println("라면을 끓인 후에 식초 약간 넣기");
    }
}

class MilkRamen extends Ramen {
    public void cook() {
        System.out.println("우유를 넣고 끓이기");
    }
}
```

사례 2 – 라면 조리

```
public class Main {
    public static void main(String[] args) {
        Ramen cook = new GeneralRamen();
        cook.cook();
        cook = new CheeseRamen();
        cook.cook();
    }
}
```

사례 2 – 라면 조리

- Version 3
  - 인터페이스를 이용해서 변화하는 부분을 캡슐화시킴
  - Ramen 클래스에서는 변화하는 부분을 바꿔서 사용할 수 있도록 처리 (멤버 변수와 설정 메소드 (setter method))
  - cook() 멤베 함수에서 Recipe 의 cook() 함수 호출



사례 2 – 라면 조리

```
interface Recipe {
    public void cook();
}

class Ramen {
    Recipe recipe = new GeneralRamenRecipe();
    public void setRecipe(Recipe recipe) {
        this.recipe = recipe;
    }
    public void cook() {
        recipe.cook();
    }
}

class GeneralRamenRecipe implements Recipe {
    public void cook() {
        System.out.println(" 일반 조리법으로 끓이기 ");
    }
}
```

```
class RamenWithoutBrothRecipe implements Recipe {
    public void cook() {
        System.out.println(" 물을 적게 넣고 라면을 익힌
뒤에 라면 스프에 볶듯이 끓임 ");
    }
}

class CheeseRamenRecipe implements Recipe {
    public void cook() {
        System.out.println(" 라면을 끓인 후에 치즈 넣
기 ");
    }
}

class VinegarRamenRecipe implements Recipe {
    public void cook() {
        System.out.println(" 라면을 끓인 후에 식초 약간
넣기 ");
    }
}
```

사례 2 – 라면 조리

```
class MilkRamenRecipe implements Recipe {
    public void cook() {
        System.out.println("우유를 넣고 끓이기");
    }
}

public class Main {
    public static void main(String[] args) {
        Ramen cook = new Ramen();
        cook.cook();
        cook.setRecipe(new CheeseRamenRecipe());
        cook.cook();
    }
}
```

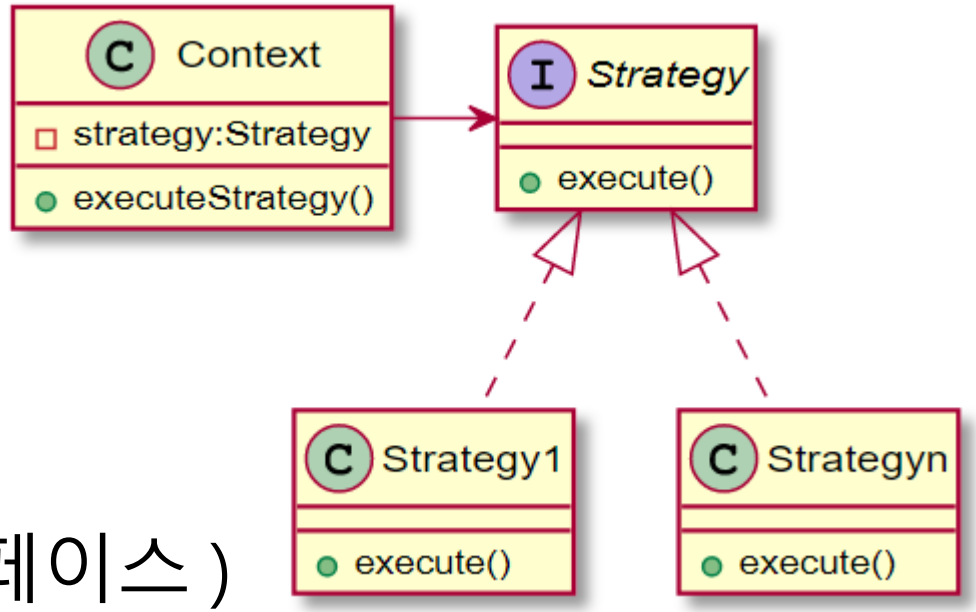
디자인 패턴 요소

- 패턴이 필요한 경우
  - 경우에 따라 서로 다른 여러 알고리즘이 존재
  - 알고리즘이 실행 시점에 결정되어져서 조건문 등을 이용해서 다른 알고리즘을 선택하는 경우

요소	설명
이름	스트래티지 (Strategy)
문제	알고리즘의 다른 버전이 존재해서, 중복으로 존재하거나 if 문을 이용해서 선택해야 함 . OCP 위반
해결방안	중복을 공통화시키고, 실행 시점에 맞는 알고리즘을 호출하도록 함 (상속 또는 인터페이스 활용)
결과	OCP. 수정할 경우 Strategy 를 추가하고, 나머지는 변경하지 않아도 됨

스트래티지 패턴 (Strategy Pattern)

- Context 클래스
  - 캡슐화된 알고리즘을 멤버 변수로 포함
  - 캡슐화된 알고리즘을 교환해서 적용시킬 수 있음



- Strategy 클래스 (인터페이스)
  - 컴파일 시점에서 사용하는 캡슐화된 알고리즘을 나타냄
  - 실제 구현은 하위 $Strategy_n$ 클래스에 위임
  - 인터페이스 또는 클래스로 구현 가능
- $Strategy_n$
  - 실행 시점에 적용될 알고리즘을 캡슐화
  - Context 에서 실행될 알고리즘을 구현