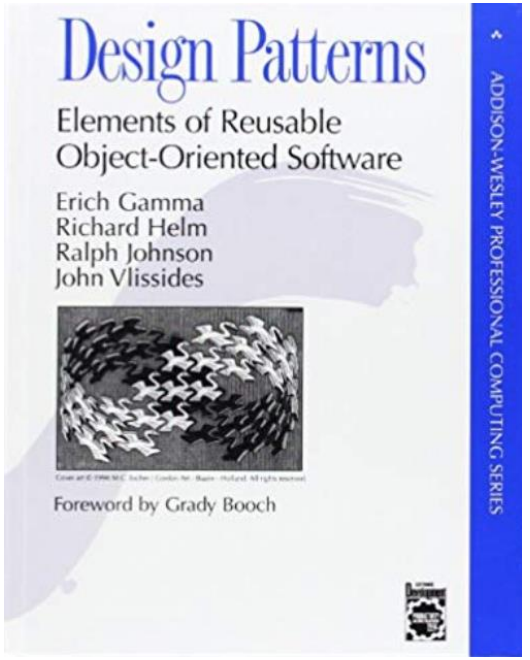
Design Pattern
Strategy Pattern

514770-1
Fall 2024
9/18/2024
Kyoung Shin Park
Computer Engineering
Dankook University

Design Pattern

- The design pattern is a structure of a standard solution by gathering the experiences of experts on common problems encountered in the development process, and if you are familiar with the design pattern, smooth communication between developers is possible.
- The 23 **Gang of Four (GoF)** patterns are generally considered the foundation for all other patterns.
- They are categorized in three groups:
  - **Creational** Pattern
  - **Structural** Pattern
  - **Behavioral** Pattern



Design Pattern

- Creational Pattern
  - Patterns related to object creation
  - Increases flexibility in the process of object creation and makes code maintenance easier
  - **Abstract Factory**
  - **Builder**
  - **Factory Method**
  - Prototype
  - **Singleton**

Design Pattern

- Structural Pattern
  - Patterns related to program structure
  - Patterns that can be used to design the structure of the program, such as data structure or interface structure
  - Patterns related to the composition/aggregation of classes or objects
  - **Adapter**
  - Bridge
  - **Composite**
  - **Decorator**
  - **Façade**
  - Flyweight
  - **Proxy**

Design Pattern

- Behavioral Pattern
  - Patterns that define how classes or objects interact and how responsibilities are distributed
  - Patterns related to the interaction of objects that are used repeatedly
  - Chain of Responsibility
  - **Command**
  - Interpreter
  - **Iterator**
  - Mediator
  - Memento
  - **Observer**
  - **State**
  - **Strategy**
  - **Template Method**
  - Visitor

Strategy Pattern

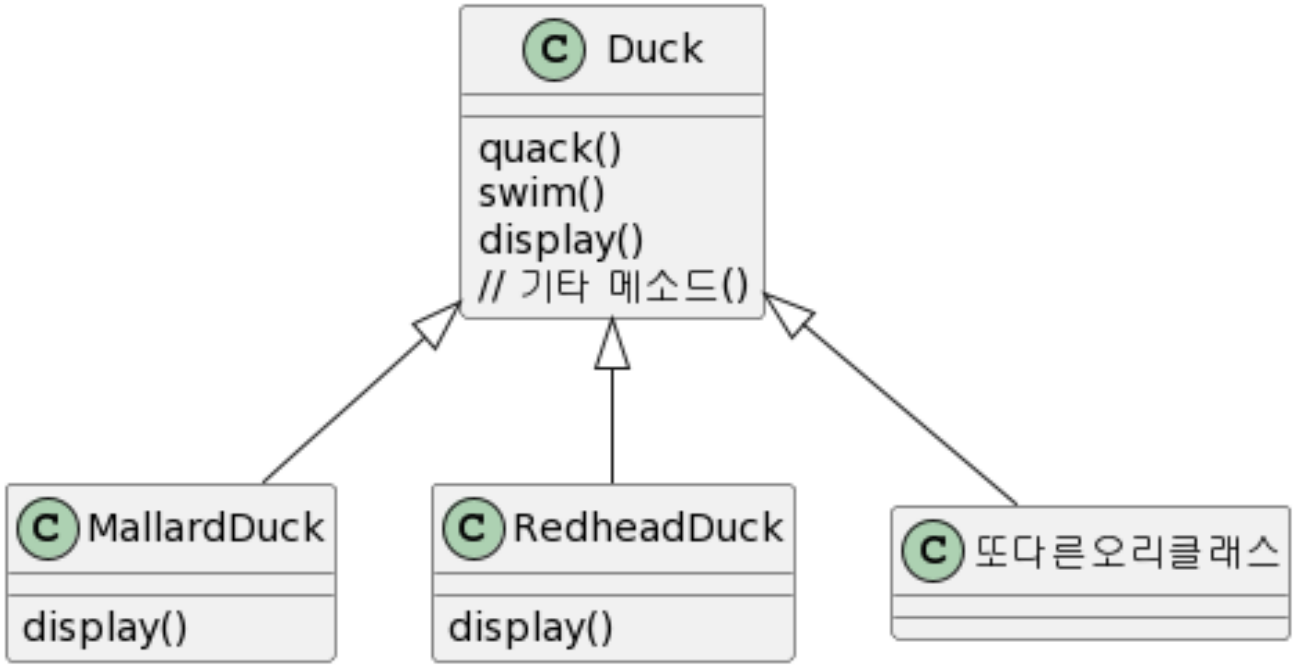
- "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."
- In Strategy pattern, a class behavior or its algorithm can be changed at run time.
- This pattern is also called as **Policy**.
- This pattern is based on **Open/Closed Principle**.
  - We don't need to modify the context [closed for modification], but can choose and add any implementation [open for extension].
- Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Strategy Pattern

- Where to use it
  - If the transportation to airport is different (city bus, personal car, taxi, metro, etc)
  - If the way you watch movies is different (regular ticket, invitation ticket, membership discount, etc)
  - Files being compressed as various formats (.zip, .gz, .7z, etc)
  - Documents being saved in various file formats (.docx, .pdf, .rtf, etc)
  - In Java, **Collections.sort()** method is one of the commonly used example of Strategy pattern.
    - Collections.sort() method will sort the objects based on the **comparator implementation** passed to it.

Duck (HFDP Ch. 1)

- Version 1
  - SimUDuck duck pond simulation game
    - The game can show a large variety of duck species swimming and making quacking sounds.
  - Create on Duck superclass from which all other duck types inherit



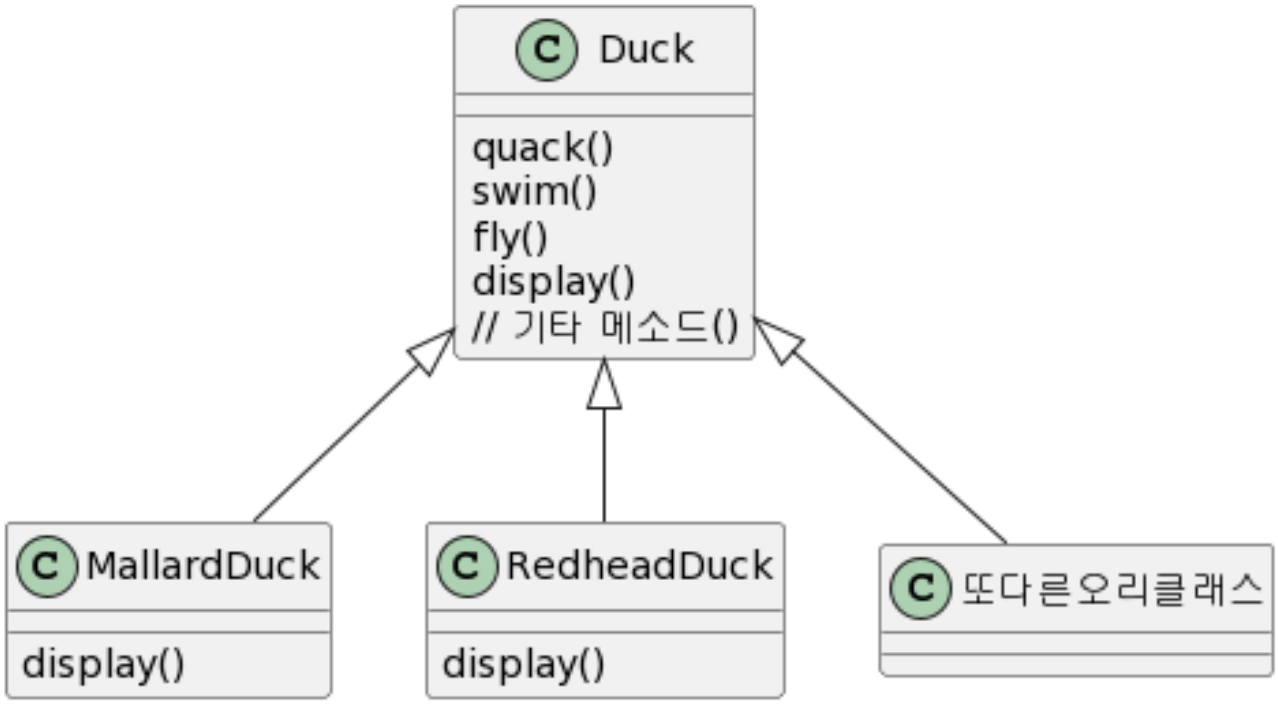
Duck (HFDP Ch. 1)

```
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();
    }
}
```

Duck (HFDP Ch. 1)

- Version 2
  - But now we need the ducks to **FLY**
    - Just need to add a **fly() method** in the Duck class



Duck (HFDP Ch. 1)

```
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");
    }
}
```

Duck (HFDP Ch. 1)

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

Duck (HFDP Ch. 1)

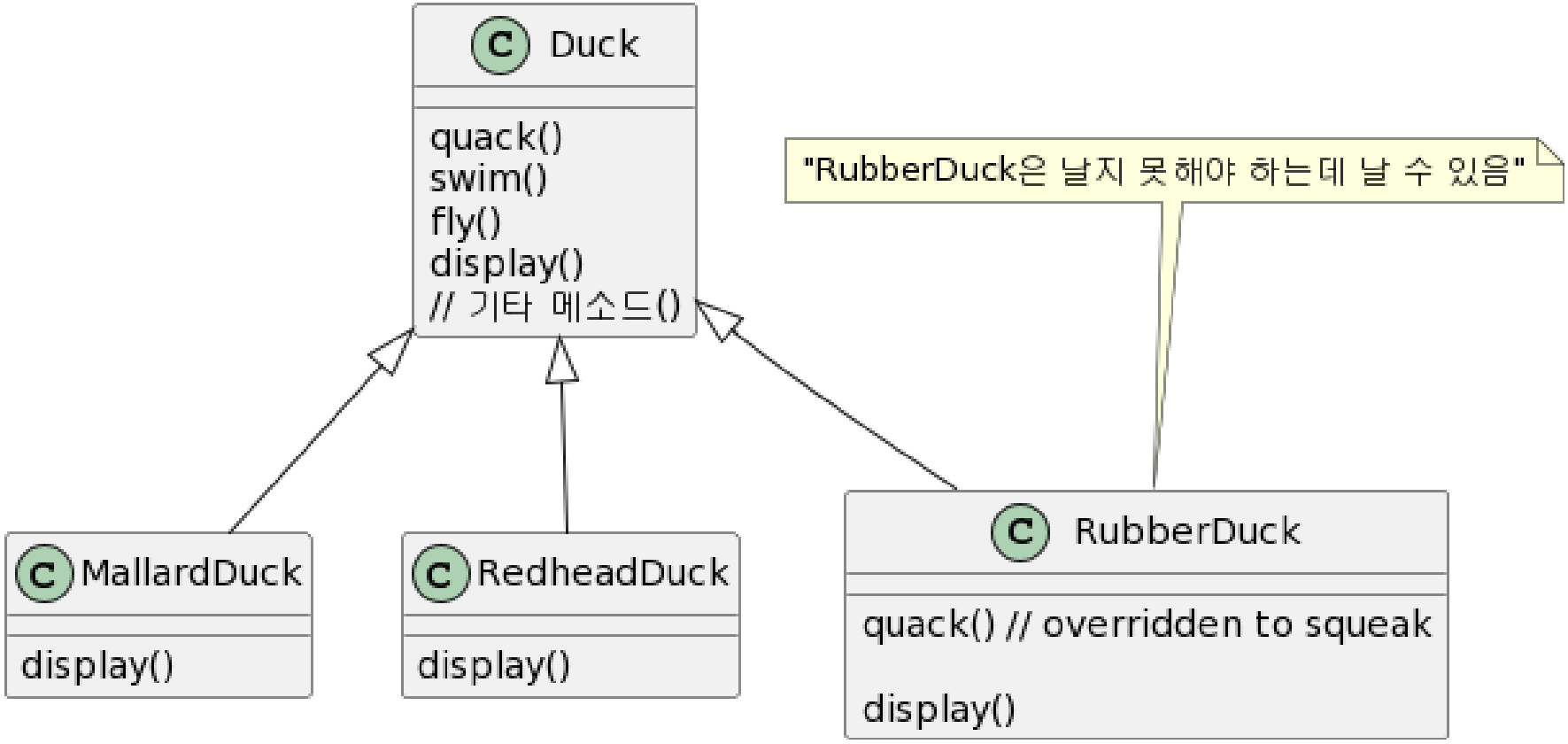
- Add a new class RubberDuck

```
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();
    }
}
```

Duck (HFDP Ch. 1)

- But something went horribly wrong
  - Notice that **not** all subclasses of Duck should **fly()**
  - **Solution: fly() method override**

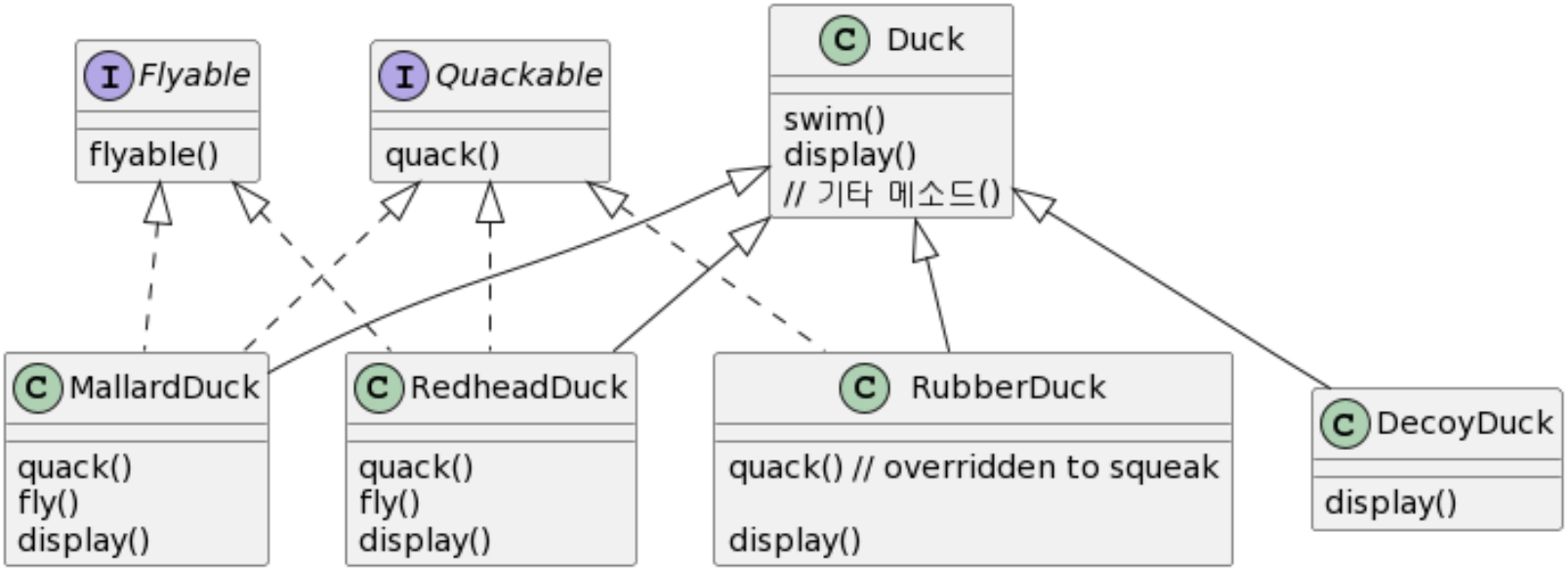


Duck (HFDP Ch. 1)

```
class RubberDuck extends Duck {
    @Override
    void quack() {
        System.out.println("squeak");
    }
    @Override
    void display() {
        System.out.println("RubberDuck");
    }
    @Override
    void fly() {
        System.out.println("cannot fly");
    }
}
```

Duck (HFDP Ch. 1)

- Version 3 & 4
  - How about an interface?



- MallardDuck, RedheadDuck, RubberDuck, etc implement the same code repeatedly. → In Java8, the possible solution is using Interface default method (Version 4)

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");
    }
}
```

Duck (HFDP Ch. 1)

```
class RubberDuck extends Duck
                implements Quackable {
    public 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();
        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();
    }
}
```

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 {
    public void quack() {
        System.out.println("squeak");
    }
    void display() {
        System.out.println("RubberDuck");
    }
}
```

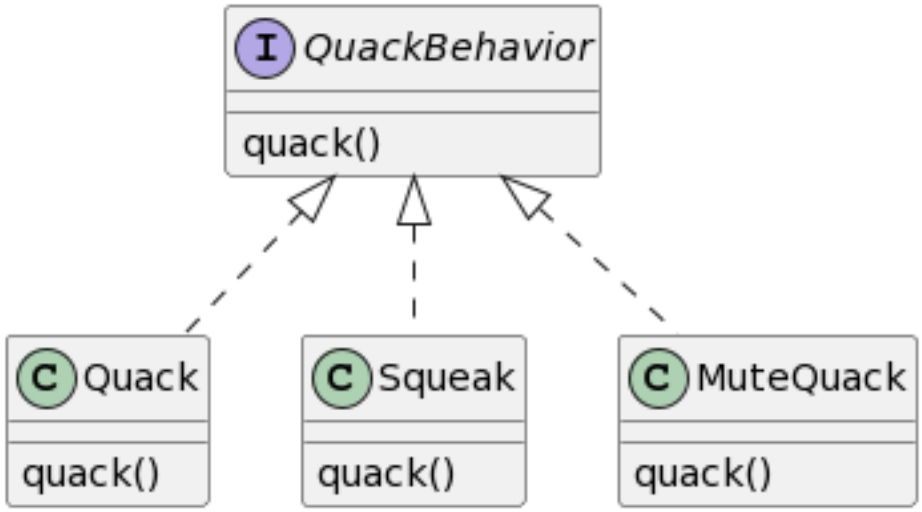
Duck (HFDP Ch. 1)

- Separating what changes from what stays the same
  - In Duck class, fly() and quack() change frequently
  - No other parts of it that appear to vary or change frequently
  - **To separate the "parts that change from those that stay the same"**, you need to create two sets of classes (totally apart from Duck), one for fly and one for quack.
    - Each set of classes should contain all the implementations of their respective behavior
      - Set related to fly()
      - Set related to quack()
    - Not implementing specific behavior in Duck class, but creating new class independently
    - In Duck or subclass, use the interface that actually implements behavior (FlyBehavior or QuackBehavior)
      - So it has nothing to do with Duck

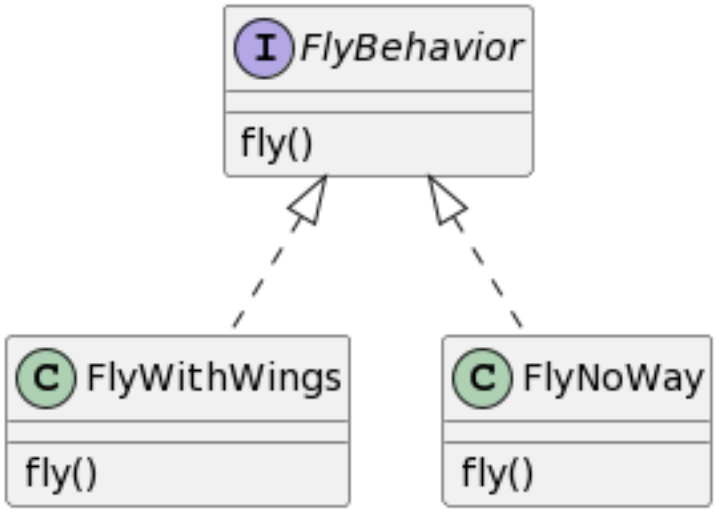
Duck (HFDP Ch. 1)

- Version 5

Encapsulated quack behavior



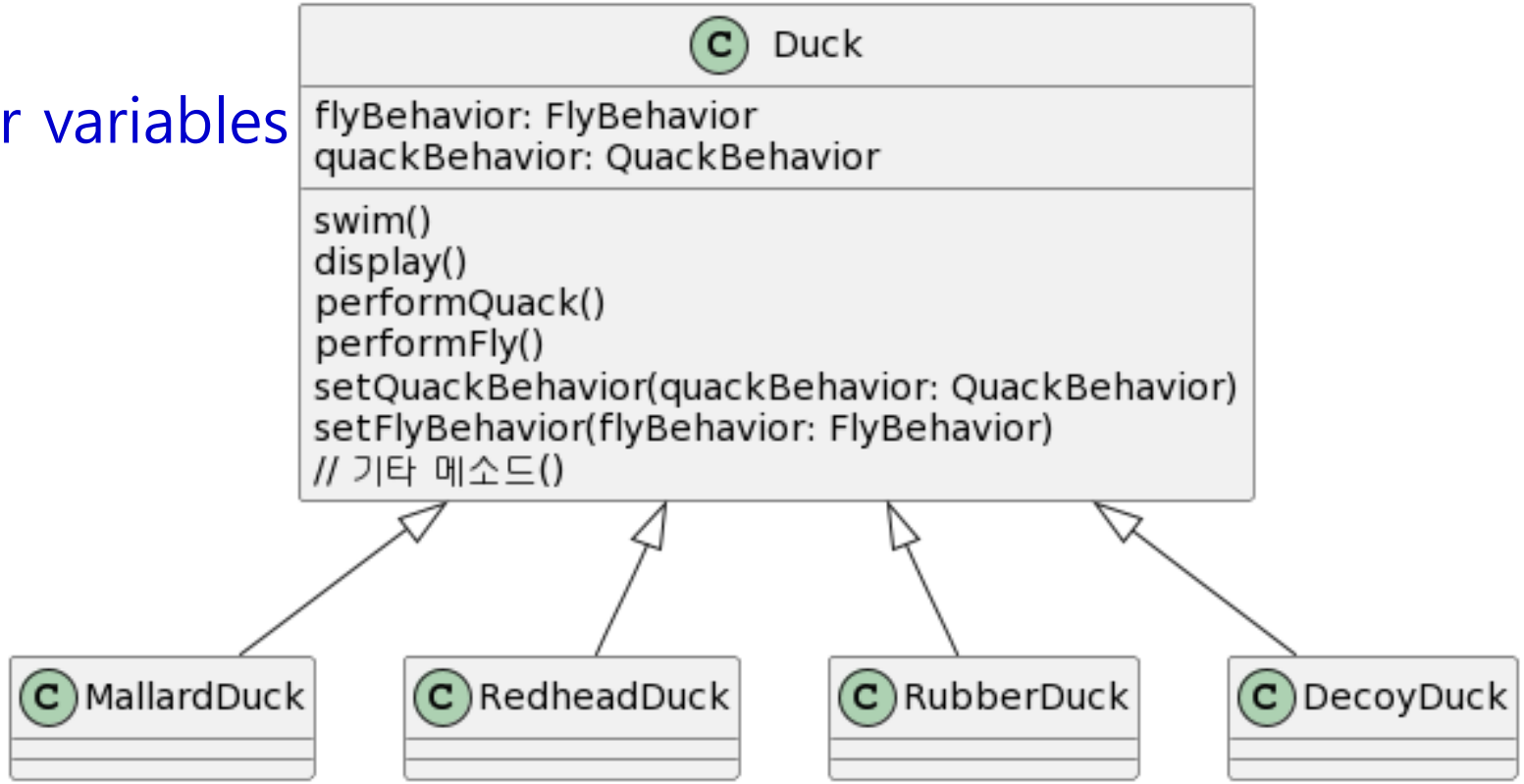
Encapsulated fly behavior



Duck (HFDP Ch. 1)

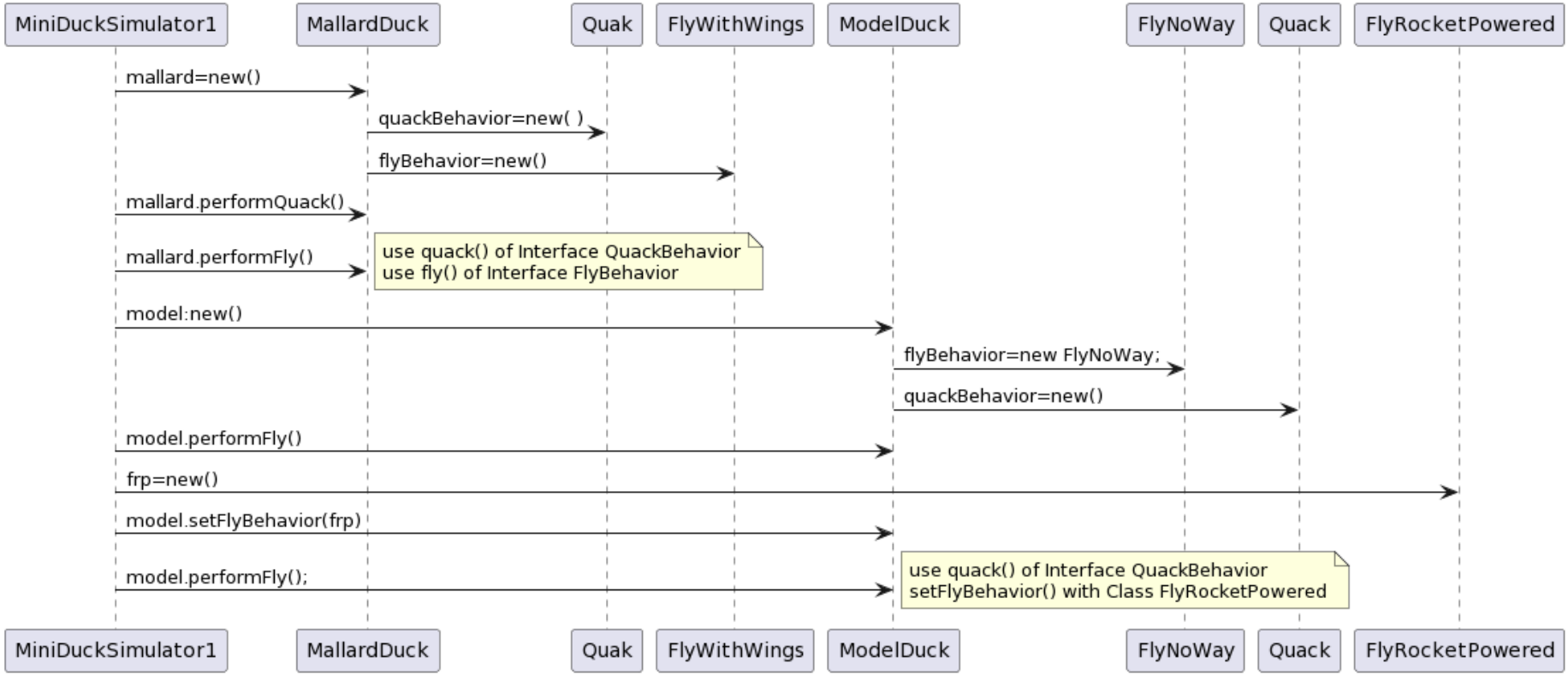
- Duck makes use of an encapsulated family of algorithms for both fly and quack.

Behavior variables



Duck (HFDP Ch. 1)

- Sequence Diagram



Duck (HFDP Ch. 1)

- Setting behavior dynamically
  - flyBehavior is set in ModelDuck's constructor

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

  - Also, the flyBehavior is modified using set function.

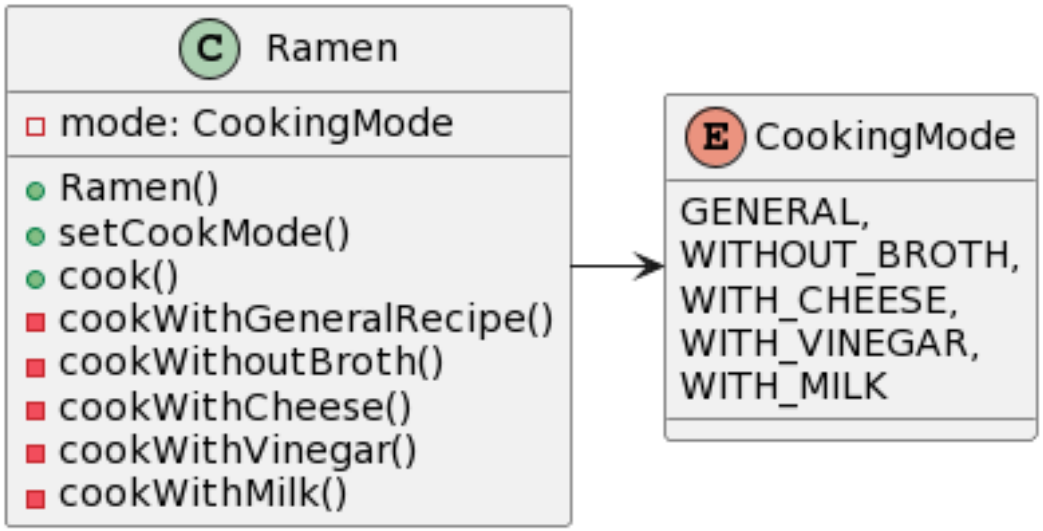
```
model.setFlyBehavior(new FlyRocketPowered());
```

Cooking Ramen

- In cooking ramen, different ramen classes use different recipes
  - Basic Ramen (cook)
  - Fried Ramen (cookWithoutBroth)
    - Boil the ramen with about a mug of water and stir-fly with 2/3 of the soup
  - Cheese Ramen (cookWithCheese)
    - After boiling, add sliced cheese
  - Vinegar Ramen (cookWithVinegar)
    - After boiling, add 1 piece with a small spoon of vinegar
  - Milk Ramen (cookWithMilk)
    - Boild noodles with milk instead of water

Cooking Ramen

- Version 1
  - Put all recipes in the client and select a recipe using it or switch statement



  - Problem
    - Difficult to add new recipes
    - Code gets too complex

Cooking Ramen

```
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;
    }
```

Cooking Ramen

```
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;
    }
}
```

Cooking Ramen

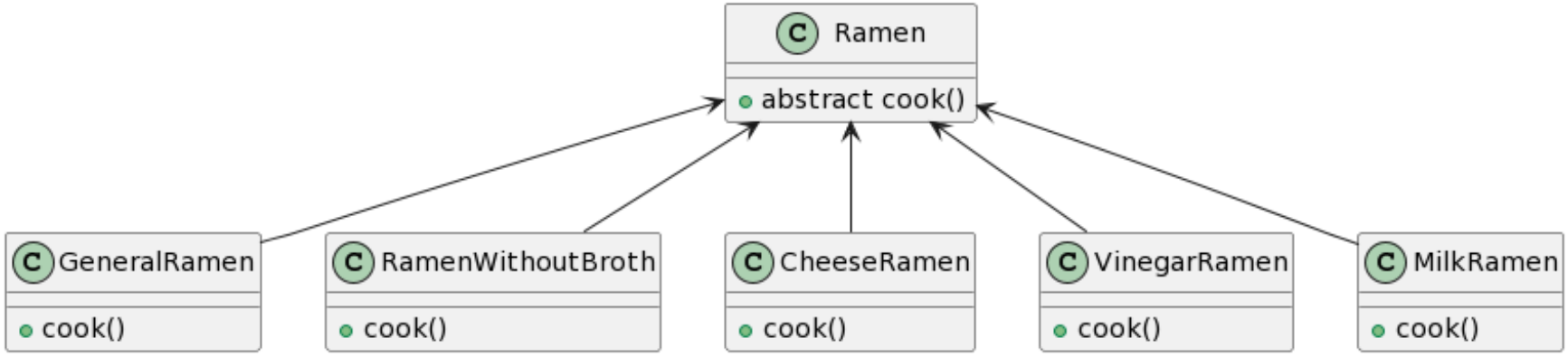
```
private void cookWithGeneralRecipe() {
        System.out.println("일반 조리법으로 끓이기");
    }
    private void cookWithoutBroth() {
        System.out.println("물을 적게 넣고 라면을 익힌
뒤에 라면 스프에 볶듯이 끓임");
    }
    private void cookWithCheese() {
        System.out.println("라면을 끓인 후에 치즈 넣기
");
    }
    private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

Cooking Ramen

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

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

Cooking Ramen

- Version 2
  - Use inheritance



  - Problem
    - What if you want to add a Fake Food Replica?
      - The overridden **cook()** should **not actually cook**.
    - Whenever a new class is added, the cook() method should be checked.

Cooking Ramen

```
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 익
힌 뒤에 라면 스프에 볶듯이 끓임");
    }
}
```

Cooking Ramen

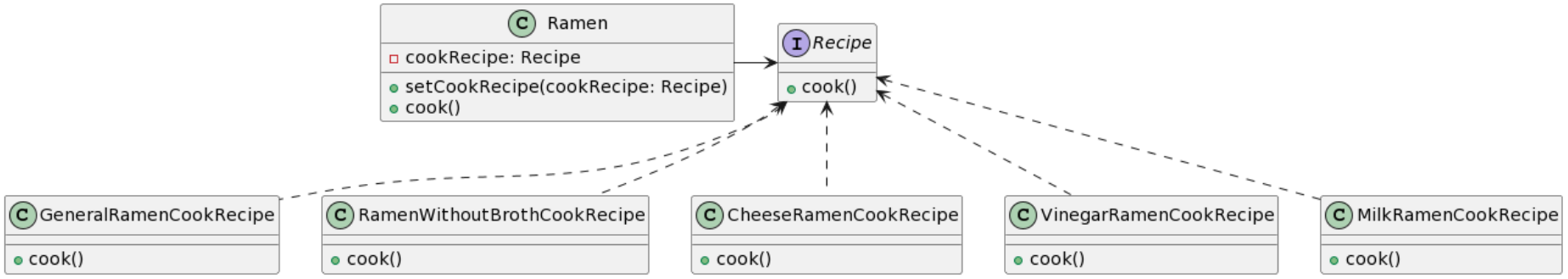
```
class CheeseRamen extends Ramen {
    public void cook() {
        System.out.println("라면을 끓인 후에 치즈 넣
기");
    }
}

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

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

Cooking Ramen

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

Cooking Ramen

- Version 3
  - Encapsulate the changing parts using interface
  - In the Ramen class, the changing parts can be interchangeable. (member variables and setter method)
  - Then, call the appropriate recipe's cook() method.



Cooking Ramen

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

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

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

Cooking Ramen

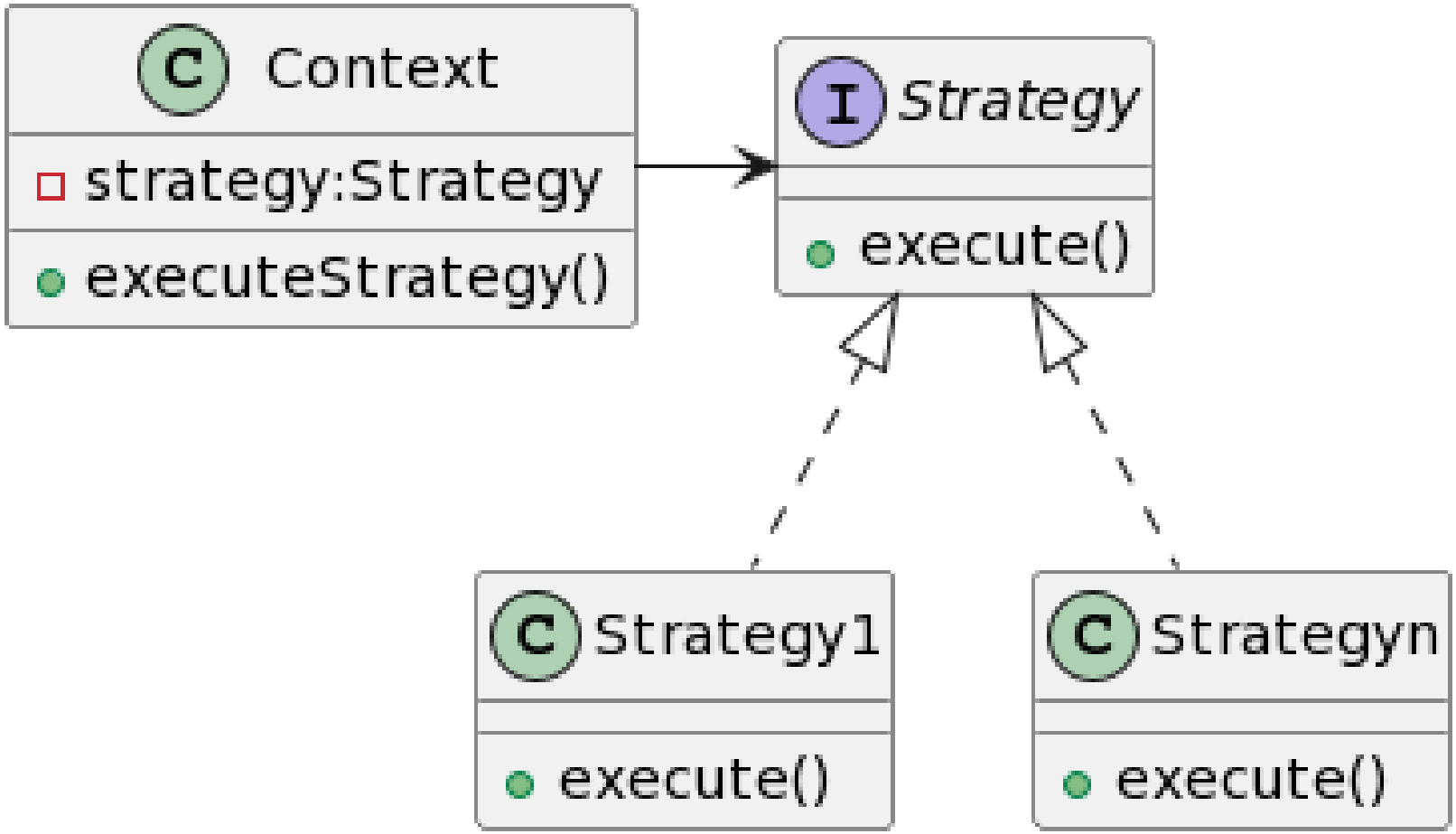
```
class MilkRamenRecipe implements Recipe {
    public void cook() {
        System.out.println("우유를 넣고 끓이기");
    }
}

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

Strategy Pattern

- Motivation
  - When there are several different algorithms exist and the algorithm is determined at the time of execution and another algorithm is selected using conditional statements, etc

	Description
Pattern	Strategy
Problem	Different algorithms exist, so they may be duplicated or have to selected using an if/switch-statement. OCP violation
Solution	Encapsulate redundancy and calls the appropriate algorithm at the time of execution (using inheritance or interface)
Result	Open Closed Principle. When a new strategy is added, others do not need to be changed.

Strategy Pattern



Define Strategy Pattern

- **Context** class
  - Includes **the encapsulated algorithm as member variables** where encapsulated algorithms can be exchanged and applied
- **Strategy interface**
  - Represents the encapsulated algorithm used at compile time
  - The actual implementation is delegated to the Strategy 1, 2,.., n subclasses
- **Strategy1, 2, .., n** class
  - Encapsulate the algorithm to be applied at run time
  - Implement the algorithm to be **executed** in the Context class