

04/09 스터디 노트

날짜	@2025년 4월 9일
태그	디자인패턴

1. 행위 패턴

생성 (Creational) 패턴	구조 (Structural) 패턴	행위 (Behavioral) 패턴
<ul style="list-style-type: none">추상 팩토리 (Abstract Factory)빌더 (Builder)팩토리 메서드 (Factory Method)프로토타입 (Prototype)싱글톤 (Singleton)	<ul style="list-style-type: none">어댑터 (Adapter)브리지 (Bridge)컴퍼지트 (Composite)데코레이터 (Decorator)퍼사드 (Facade)플라이웨이트 (Flyweight)프록시 (Proxy)	<ul style="list-style-type: none">책임 연쇄 (Chain of Responsibility)커맨드 (Command)인터프리터 (Interpreter)이터레이터 (Iterator)미디에이터 (Mediator)메멘토 (Memento)옵저버 (Observer)테이트 (State)스트래티지 (Strategy)템플릿 메서드 (Template Method)비지터 (Visitor)

1-1. 개념

- 행위(Behavioral) 패턴은 객체나 클래스 간의 책임 분산과 커뮤니케이션 방법에 관련된 패턴.
- 어떤 객체가 어떤 방식으로 다른 객체와 상호작용하는가? 에 초점을 맞춤.
- 알고리즘을 캡슐화하거나, 실행 시점을 지연시키거나, 이벤트 흐름을 제어하는 등의 역할을 한다.

1-2. 종류

- 전략 패턴
- 옵저버 패턴
- 상태 패턴

<https://refactoring.guru/ko/design-patterns/behavioral-patterns>

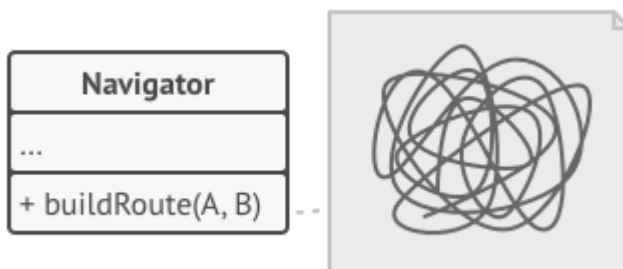
전략 패턴 (Strategy Pattern)



1. 개념

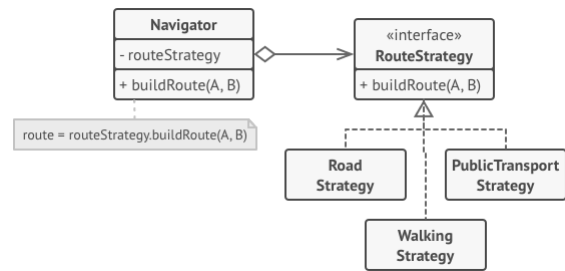
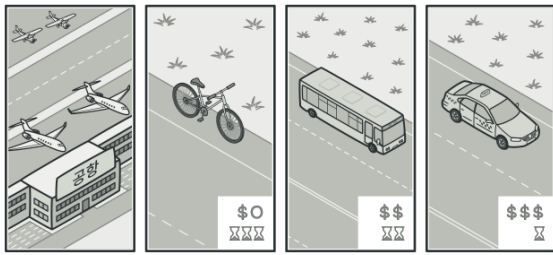
- 전략 패턴은 **행위(알고리즘)**을 캡슐화하여, 동적으로 교체 가능하게 하는 패턴이다.
- 클라이언트는 구체적인 알고리즘에 의존하지 않고, **공통 인터페이스**를 통해 전략을 사용한다.
- 조건문을 제거하고, **유지보수가 쉬운 코드 구조**를 만든다.

2. 문제 제기 (왜 필요할까?)



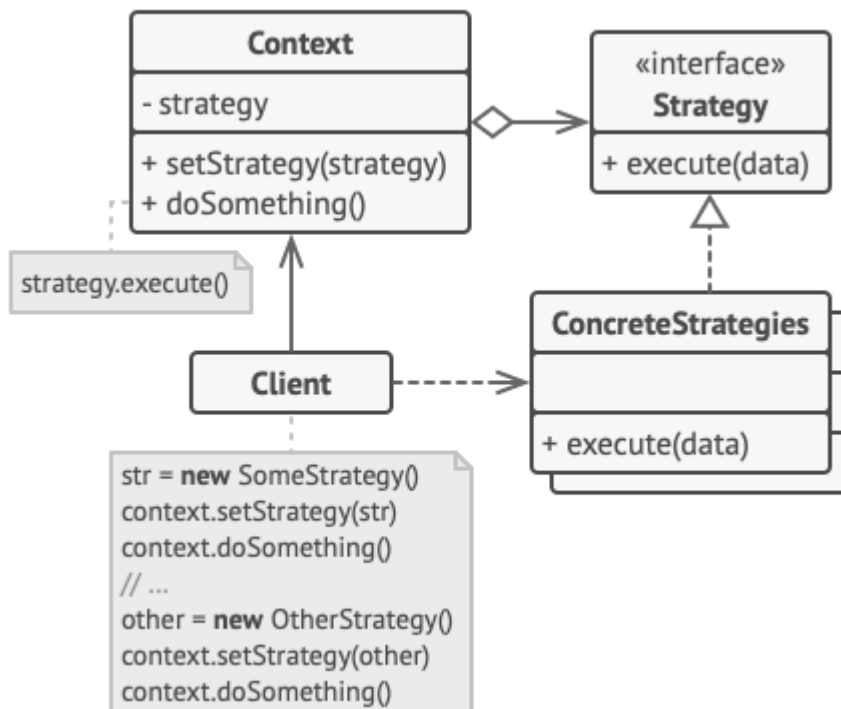
"네비게이션 앱에서 자동 경로 계획 기능을 만들고 있다. 처음에는 자동차 경로만 보여주다가, 점차 도보, 자전거, 대중교통의 기능을 추가하였다. 그런데, if-else문으로 분기하자니 클래스의 크기도 너무 커졌고, 유지보수하기 어려워졌다."

3. 예시



- **실생활:** 네비게이션 앱에서 공항에 도착하기 위한 경로 안내로 도보, 자전거, 대중교통, 자동차를 각기 이용할 때.
- **프로그래밍:** 쇼핑몰에서 할인 정책을 '정률 할인', '정액 할인', '쿠폰 할인' 등으로 나누어 구현할 때.

4. 구현 코드



```
// 전략 인터페이스
interface DiscountStrategy {
    int getDiscountPrice(int price);
}
```

```
// 전략 구현체 1 - 정률 할인
```

```

class RateDiscount implements DiscountStrategy {
    public int getDiscountPrice(int price) {
        return (int)(price * 0.9); // 10% 할인
    }
}

// 전략 구현체 2 - 정액 할인
class AmountDiscount implements DiscountStrategy {
    public int getDiscountPrice(int price) {
        return price - 1000;
    }
}

// Context 클래스
class Product {
    private DiscountStrategy strategy;

    public void setStrategy(DiscountStrategy strategy) {
        this.strategy = strategy;
    }

    public int getFinalPrice(int price) {
        return strategy.getDiscountPrice(price);
    }
}

// 사용 예
public class Main {
    public static void main(String[] args) {
        Product p = new Product();

        p.setStrategy(new RateDiscount());
        System.out.println("정률 할인: " + p.getFinalPrice(10000));

        p.setStrategy(new AmountDiscount());
        System.out.println("정액 할인: " + p.getFinalPrice(10000));
    }
}

```

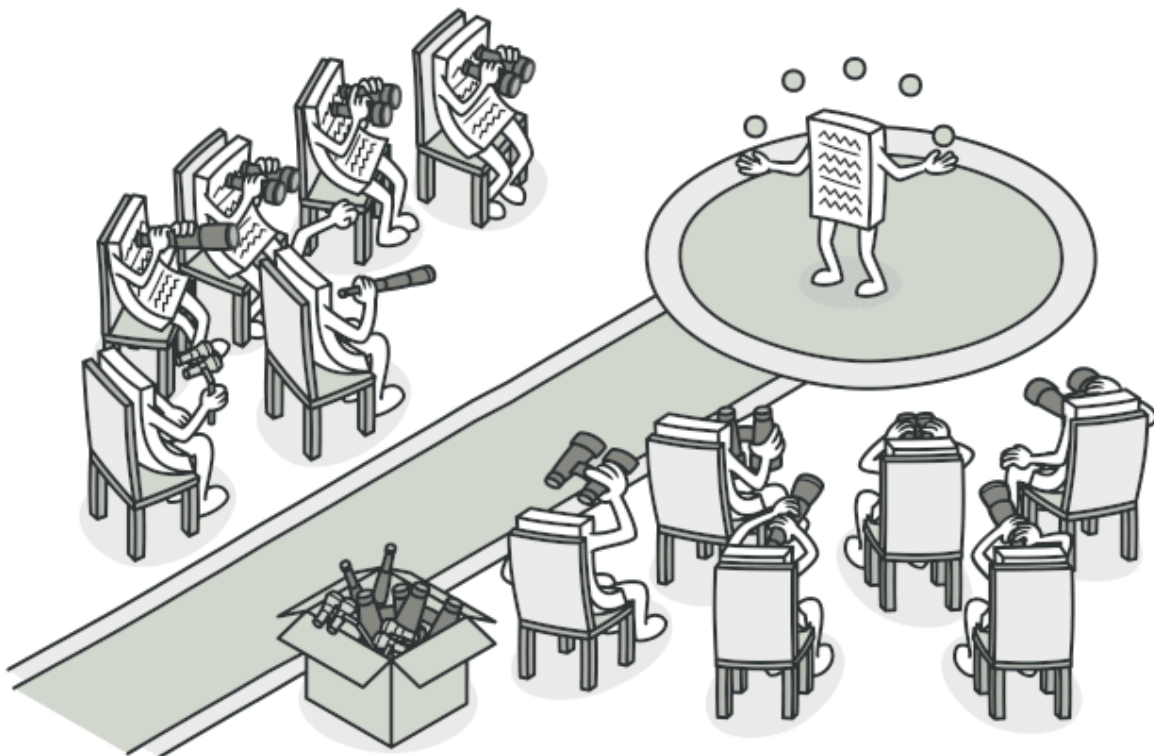
장점

- 조건문 제거 → 코드 간결하고 명확해짐
- 새로운 전략 추가가 **OCP(Open-Closed Principle)**를 지킴
- 전략을 런타임에 유연하게 변경 가능

단점

- 프로그램이 복잡해짐
- 전략 클래스가 많아질 수 있음
 - 요즘은 익명 함수의 존재로 동일하게 구현할 수 있음
- 클라이언트가 전략을 직접 선택해야 할 수도 있음
 - 부담을 떠넘김

옵저버 패턴 (Observer Pattern)



개념

- 옵저버 패턴은 객체의 상태 변화를 관찰하는 다른 객체들에게 **자동으로 알림**을 보내는 패턴이다.

- 주체(Subject)와 관찰자(Observer) 간의 **1:N 관계**를 유지하며, 느슨한 결합을 가능하게 한다.
- 이벤트 기반 시스템이나 데이터 흐름에 적합하다.

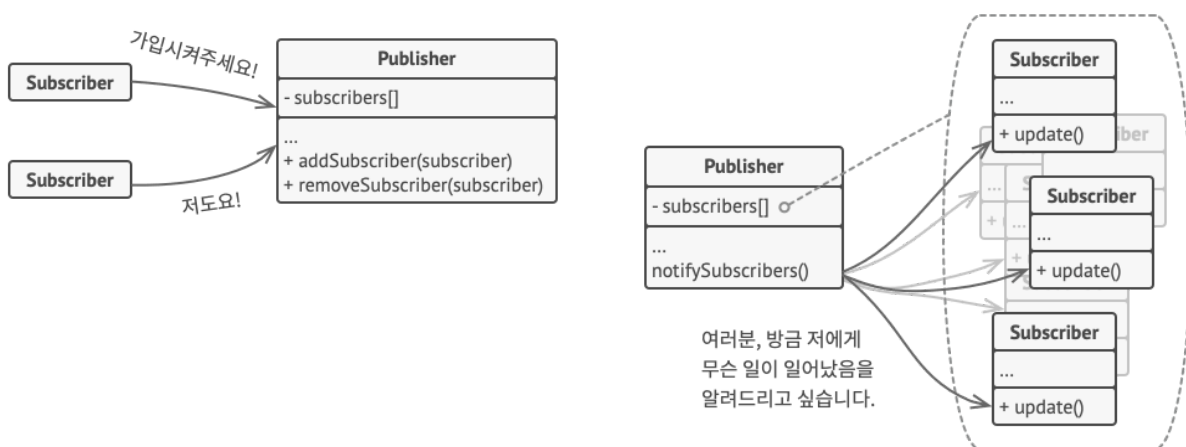
문제 제기



"스마트폰 매장에서 새로운 제품이 출시될 때마다 모든 고객들에게 스팸처럼 이메일을 보낼 수 있다. 그러나, 제품 출시 소식을 듣고 싶지 않은 고객이 존재할 수 있다."

"그렇다고, 새로운 제품이 출시되었는지를 고객 입장에서 일일이 확인하는 것도 시간 낭비이다."

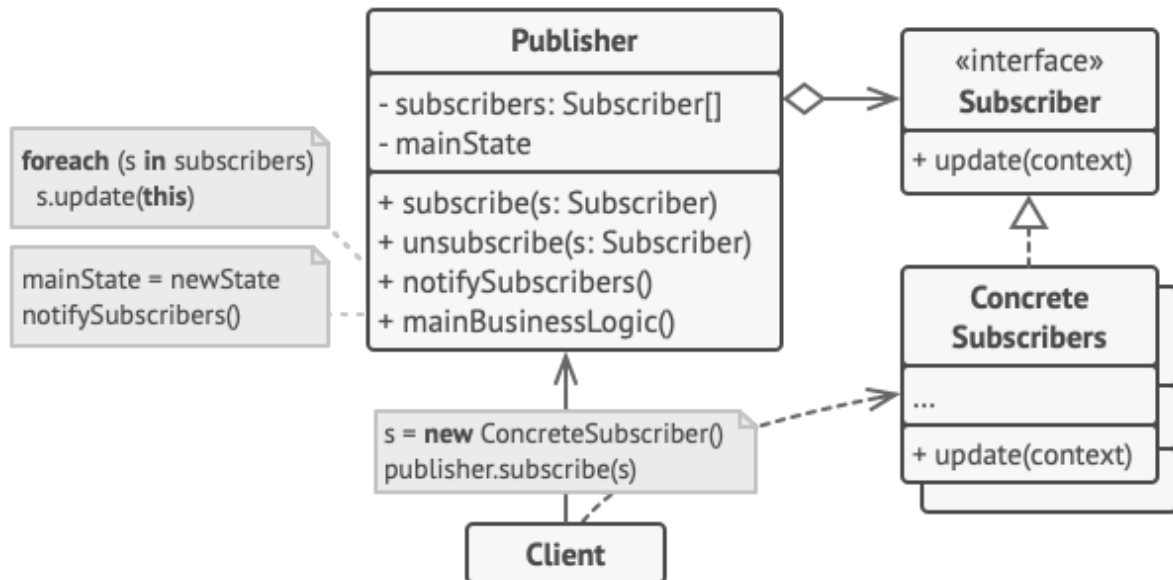
예시



- **실생활:** 유튜브 구독 – 새 영상 업로드 시 구독자에게 알림

- **프로그래밍:** 이벤트 리스너, 모델-뷰 간 데이터 동기화 (MVC 구조)

구현 코드



```

// 옵저버 인터페이스
interface Observer {
    void update(String message);
}

// 주체 인터페이스
interface Subject {
    void attach(Observer o);
    void detach(Observer o);
    void notifyObservers();
}

// Concrete Subject
class Board implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String post;

    public void newPost(String content) {
        this.post = content;
        notifyObservers();
    }
}
  
```

```

    }

    public void attach(Observer o) {
        observers.add(o);
    }

    public void detach(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(post);
        }
    }
}

// Concrete Observer
class Subscriber implements Observer {
    private String name;

    public Subscriber(String name) {
        this.name = name;
    }

    public void update(String message) {
        System.out.println(name + "님, 새 게시글: " + message);
    }
}

```

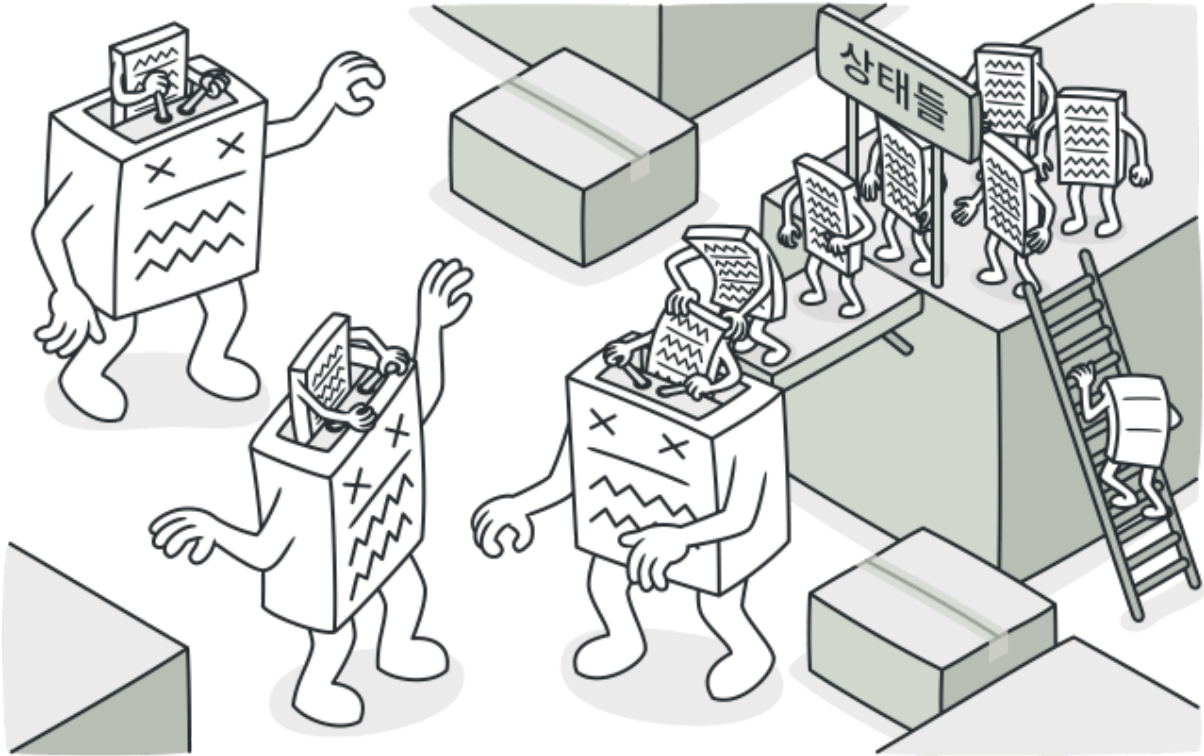
장점

- 느슨한 결합으로 객체 간 독립성 유지
- 다수의 구독자에게 자동 동기화
- 새로운 Observer 추가가 용이

단점

- 순환 참조, 메모리 누수 가능성 (detach 안 할 경우)
- 디버깅이 어려울 수 있음

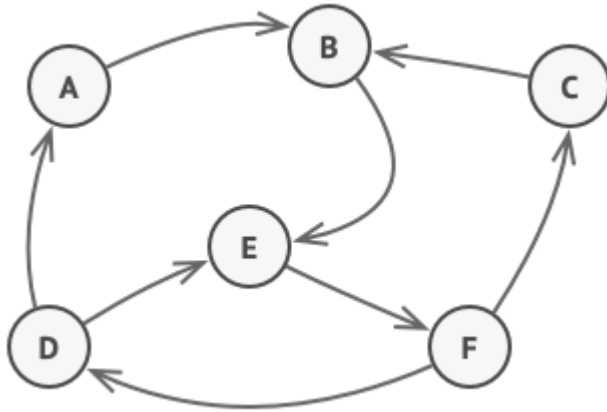
상태 패턴 (State Pattern)



개념

- 상태 패턴은 객체의 **내부 상태에 따라 행위가 달라지는** 구조를 구현한다.
- 상태 전이 로직을 조건문이 아닌 **상태 객체로 분리**하여 관리한다.
- 상태가 많고 자주 바뀌는 상황에 유용하다.

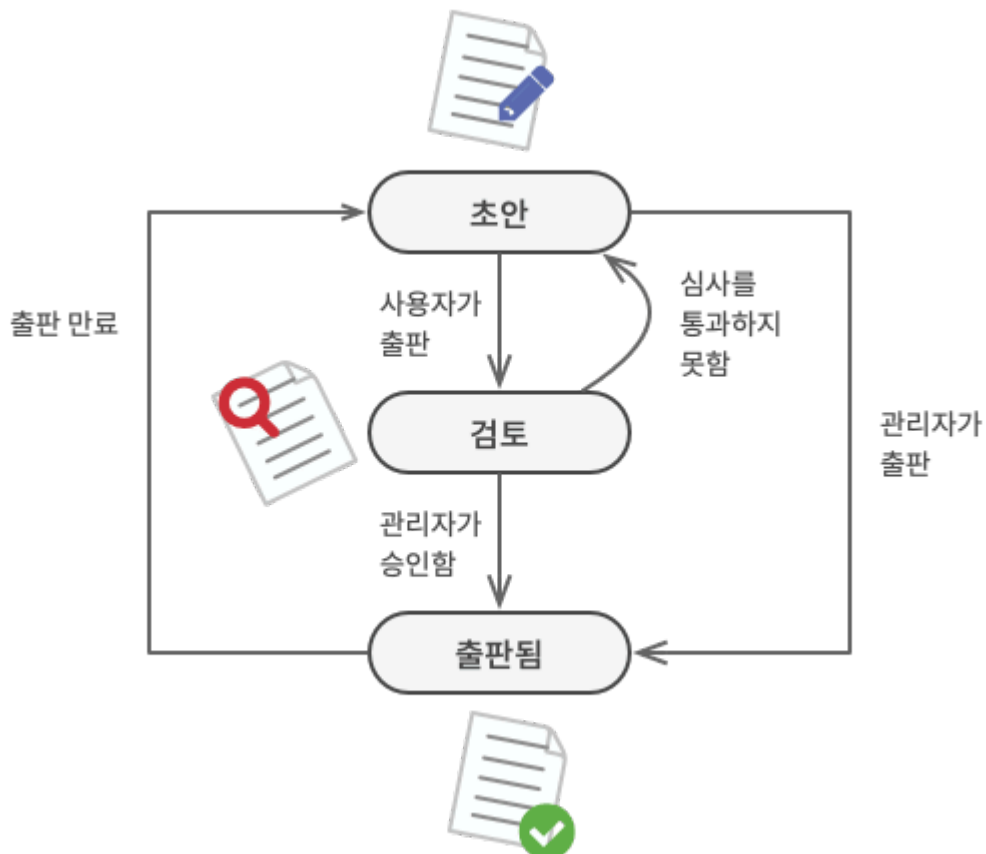
문제 제기



유한 상태 머신

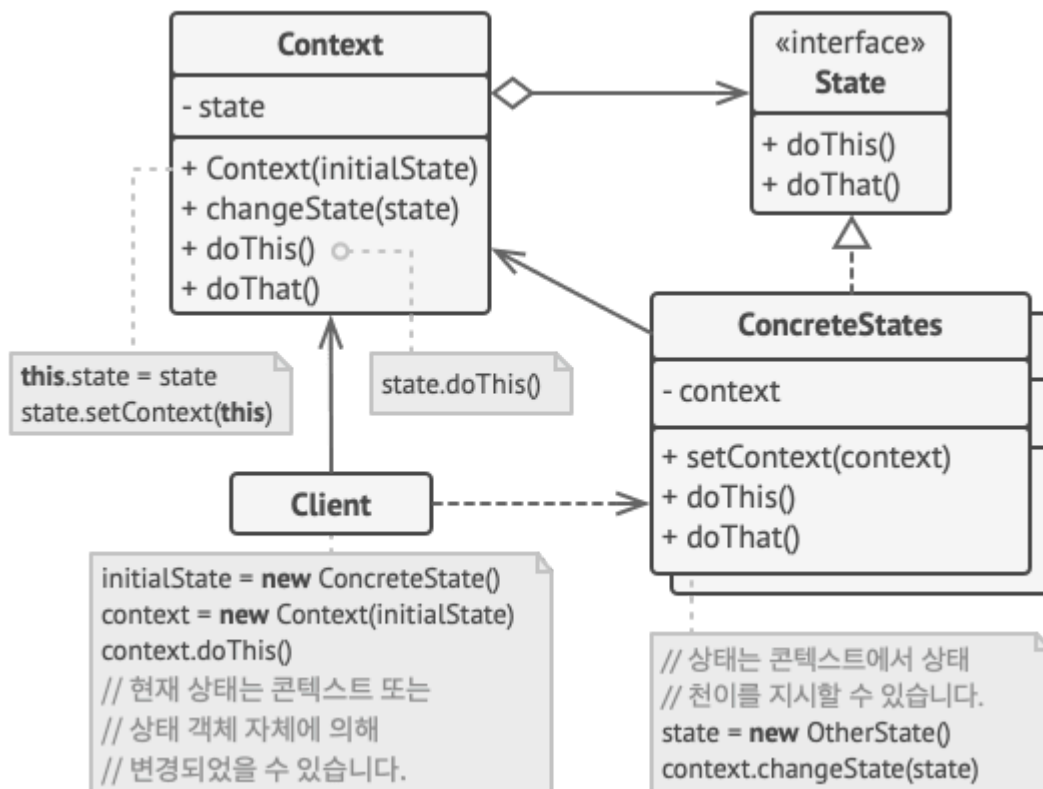
“문이 ‘열림’, ‘닫힘’, ‘잠금’ 상태일 때, 각 상태에 따라 버튼의 동작이 달라진다. if문으로 처리하려니 코드가 지저분하다.”

예시



- **실생활:** 문 상태 – 열기, 닫기, 잠금 상태별 동작 다름
- **프로그래밍:** TCP 연결 상태, 게임 캐릭터 상태 (공격/방어 등)

구현 코드



```

interface State {
    void handle();
}

class OpenState implements State {
    public void handle() {
        System.out.println("문이 열려 있습니다.");
    }
}

class ClosedState implements State {
    public void handle() {
        System.out.println("문이 닫혀 있습니다.");
    }
}

class Door {
    private State state;
}
    
```

```

    public void setState(State state) {
        this.state = state;
    }

    public void pressButton() {
        state.handle();
    }
}

// 사용
public class Main {
    public static void main(String[] args) {
        Door door = new Door();

        door.setState(new OpenState());
        door.pressButton();

        door.setState(new ClosedState());
        door.pressButton();
    }
}

```

장점

- 조건문 제거 → 코드 깔끔
- 상태별 로직을 **독립적으로 관리**
- 상태 전이가 명확하게 표현됨

단점

- 상태 클래스 수 증가
- 단순 상태에는 과할 수 있음

2. 퀴즈

? Q1. 객체의 상태 변화 시, 의존 객체들에 자동으로 알림을 주는 패턴은?

▼ 정답

옵저버(Observer) 패턴

? Q2. 객체의 내부 상태에 따라 행동이 달라지는 패턴은?

▼ 정답

상태(State) 패턴

? Q3. 알고리즘을 런타임에 바꿀 수 있도록 하는 행위 패턴은?

▼ 정답

전략(Strategy) 패턴