04/09 스터디 노트

날짜	@2025년 4월 9일
태그	디자인패턴

1. 행위 패턴

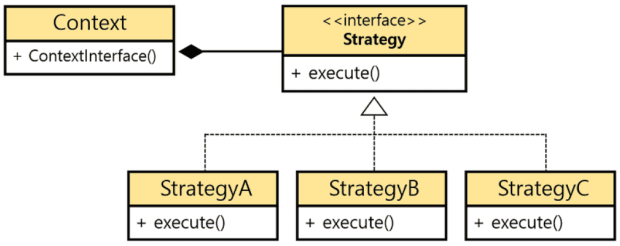
생성(Creational) 패턴	구조(Structural) 패턴	행위(Behavioral) 패턴
• 추상 팩토리(Abstract Factory) • 빌더(Builder) • 팩토리 메서드(Factory Methos) • 프로토타입(Prototype) • 싱글턴(Singleton)	• 어댑터(Adapter) • 브리지(Bridge) • 컴퍼지트(Composite) • 데커레이터(Decorator) • 퍼사드(Facade) • 플라이웨이트(Flyweight) • 프록시(Proxy)	• 책임 연쇄(Chain of Responsibility) • 커맨드(Command) • 인터프리터(Interpreter) • 이터레이터(Iterator) • 미디에이터(Mediator) • 메멘토(Memento) • 옵서버(Observer) • 테이트(State) • 스트래티지(Strategy) • 템플릿 메서드(Template Method) • 비지터(Visitor)

1-1. 개념

- **행위(Behavioral) 패턴**은 객체나 클래스 간의 **책임 분산**과 **커뮤니케이션 방법**에 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 (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?)

"어플리케이션에서 할인 정책이 자주 바뀐다. 기존에는 if-else로 할인 방식을 분기했지만, 조건문이 많아지고 변경이 잦다 보니 코드가 **복잡**하고 **확장**도 어렵다."

---

3. 예시

- **실생활**: 네비게이션 앱에서 사용자 설정에 따라 '가장 빠른 길', '최소 요금', '최소 거리' 등의 경로 탐색 전략을 선택할 수 있음.
- **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

- 전략 클래스가 **많아질 수 있음**
- 클라이언트가 전략을 **직접 선택**해야 할 수도 있음

옵저버 패턴 (Observer Pattern)

개념

- 옵저버 패턴은 **객체의 상태 변화**를 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

"게시판에 새 글이 등록되면 구독자들에게 알림을 보내고 싶다. 하지만 구독자 목록은 자주 변경되고, 특정 사용자에게만 보내고 싶을 수도 있다."

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 결합**으로 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

커맨드 패턴 (Command Pattern)

개념

- 커맨드 패턴은 요청을 **객체 형태로 캡슐화**하여 요청의 실행, 취소, 저장 등을 유연하게 처리하는 패턴이다.
- 명령의 발신자(Invoker)와 수행자(Receiver)를 분리시켜서 **재사용성과 유연성**을 높인다.
- **실행 취소(Undo), redo, 매크로 기능** 등에 적합하다.

문제 제기

“리모컨 버튼을 누르면 각기 다른 가전제품이 작동해야 하고, 최근 실행한 명령을 다시 취소하거나 기록하고 싶다.”

예시

- **실생활**: 스마트 리모컨 – TV 켜기, 에어컨 끄기, 명령 기록
- **프로그래밍**: GUI 버튼 동작 처리, 명령 로그 기록, undo/redo 기능

구현 코드

```
// Command 인터페이스
interface Command {
    void execute();
}

// Receiver
class Light {
    void on() {
        System.out.println("불 켜짐");
    }
}

// ConcreteCommand
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}

// Invoker
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
```

```
    }
}

// 사용
public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOn = new LightOnCommand(light);

        RemoteControl remote = new RemoteControl();
        remote.setCommand(lightOn);
        remote.pressButton();
    }
}
```

장점

- 요청을 **저장**, **취소**, **재실행** 가능
- 클라이언트와 Receiver 간 **결합도 낮음**
- 명령을 큐나 로그로 관리 가능

단점

- 클래스 수 증가
- 단순한 요청엔 과한 구조일 수 있음

템플릿 메서드 패턴 (Template Method Pattern)

개념

- 템플릿 메서드 패턴은 알고리즘의 **공통적인 뼈대(Template)** 를 상위 클래스에 정의하고,

  구체적인 세부 동작은 하위 클래스에서 구현하도록 하는 패턴이다.
- 알고리즘 흐름은 고정하되, **일부 단계만 변경 가능**하게 한다.

문제 제기

“데이터를 파일로 읽는 작업은 ‘파일 열기 → 읽기 → 닫기’ 흐름은 같지만, 파일 형식에 따라 읽는 방식만 다르다.”

예시

- **실생활**: 커피/차 만들기 – 물 끓이고 컵에 붓는 건 같고, 우유/레몬 첨가만 다름
- **프로그래밍**: 게임 루프, 웹 프레임워크의 추상 컨트롤러

구현 코드

```
abstract class DataProcessor {
    public final void process() {
        openFile();
        readData();
        closeFile();
    }

    protected void openFile() {
        System.out.println("파일 열기");
    }

    protected abstract void readData();

    protected void closeFile() {
        System.out.println("파일 닫기");
    }
}

class CSVProcessor extends DataProcessor {
    protected void readData() {
        System.out.println("CSV 데이터 읽기");
    }
}
```

장점

- **중복 코드 제거**, 재사용성 증가
- 알고리즘의 구조 보호 (수정 방지)
- **후크 메서드 제공 가능**

단점

- **상속에 의존**하므로 유연성 떨어질 수 있음
- 하위 클래스가 많아지면 **복잡성 증가**

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

"문이 '열림', '닫힘', '잠김' 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. 요청을 캡슐화하여 저장하거나 실행 취소가 가능하게 하는 패턴은?

▼ 정답

커맨드(Command) 패턴

❓ Q2. 알고리즘을 런타임에 바꿀 수 있도록 하는 행위 패턴은?

▼ 정답

전략(Strategy) 패턴

❓ Q3. 객체의 상태 변화 시, 의존 객체들에 자동으로 알림을 주는 패턴은?

▼ 정답

옵저버(Observer) 패턴

❓ Q4. 객체의 내부 상태에 따라 행동이 달라지는 패턴은?

▼ 정답

상태(State) 패턴

❓ Q5. 알고리즘의 골격은 상위 클래스에 두고, 일부 단계를 하위 클래스에서 구현하는 패턴은?

▼ 정답

템플릿 메서드(Template Method) 패턴