

전략 패턴 (Strategy Pattern)

Wikipedia의 Strategy pattern 문서를 한글로 번역한 뒤 스터디 발표를 위해 정리한 문서입니다.

👉 https://en.wikipedia.org/wiki/Strategy_pattern

개요

전략 패턴(Strategy Pattern)은 실행 중에 알고리즘을 선택할 수 있게 하는 행위 소프트웨어 디자인 패턴의 종류 중 하나입니다.

전략 패턴은 알고리즘군을 정의하고, 각각을 캡슐화하여 상호 교환 가능하게 만듭니다. 전략 패턴을 사용하면 알고리즘을 사용하는 클라이언트가 알고리즘을 독립적으로 변경할 수 있습니다.

이 패턴은 Gang of Four의 "Design Patterns" 책에서 소개되었으며, 소프트웨어 엔지니어링에서 널리 사용되는 디자인 패턴 중 하나입니다.

주요 특징

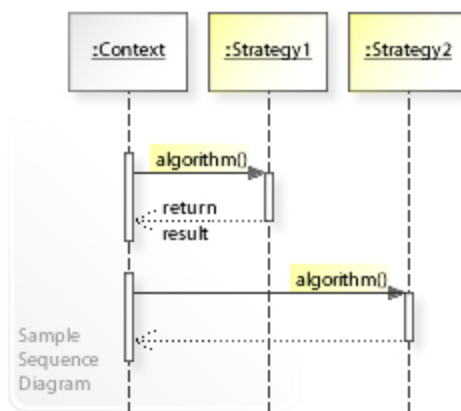
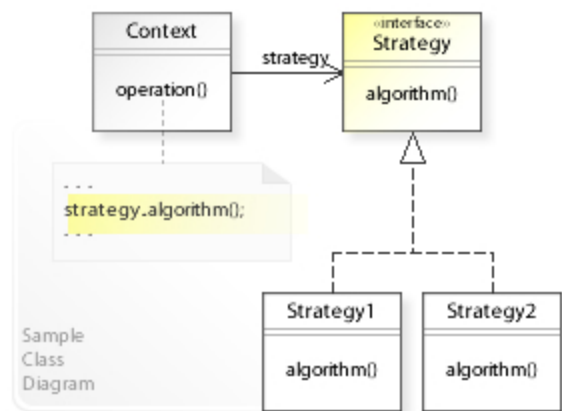
- 알고리즘 캡슐화 🖱️ 각 알고리즘을 별도의 클래스로 캡슐화
- 런타임 선택 🖱️ 실행 시점에 알고리즘 구현을 선택 가능
- 합성 우선 🖱️ 상속보다 합성(composition) 사용
- 개방-폐쇄 원칙 🖱️ 기존 코드 수정 없이 새로운 전략 추가 가능
- 클라이언트로부터 독립 🖱️ 알고리즘 변경이 클라이언트 코드에 영향을 주지 않음

핵심 구조

전략 패턴은 다음과 같은 주요 구성 요소로 이루어져 있습니다:

1. **Context** 클래스: Strategy 인터페이스에 대한 참조를 유지하고 사용
2. **Strategy** 인터페이스: 모든 전략이 구현해야 하는 공통 메서드의 정의
3. **Concrete Strategy** 클래스들: Strategy 인터페이스를 구현하여 특정 알고리즘을 제공
4. **Client**: Context에 적절한 Strategy를 주입

전략 패턴의 UML 구조



Java 예제: 자동차 브레이크 시스템

Strategy 인터페이스

```
// 브레이크 동작을 정의하는 전략 인터페이스  
public interface IBrakeStrategy {  
    void brake();  
}
```

Java 예제: 구체적인 전략 구현

```
// 일반 브레이크 전략
public class Brake implements IBrakeStrategy {
    @Override
    public void brake() {
        System.out.println("Simple Brake applied");
    }
}

// ABS 브레이크 전략
public class BrakeWithABS implements IBrakeStrategy {
    @Override
    public void brake() {
        System.out.println("Brake with ABS applied");
        System.out.println("- Anti-lock system activated");
        System.out.println("- Brake pressure modulated");
    }
}
```

Java 예제: Context 클래스

```
/**
 * Context 클래스 - 자동차
 * 브레이크 전략을 사용하고 런타임에 변경 가능
 */
public abstract class CarContext {
    private IBrakeStrategy brakeStrategy;

    public Car(IBrakeStrategy brakeStrategy) {
        this.brakeStrategy = brakeStrategy;
    }

    public void applyBrake() {
        brakeStrategy.brake();
    }

    // 런타임에 전략 변경 가능
    public void setBrakeStrategy(IBrakeStrategy brakeType) {
        this.brakeStrategy = brakeType;
    }
}
```


Java 예제: 구체적인 Context 클래스

```
// 세단 - 기본 브레이크 사용
public class SedanContext extends CarContext {
    public SedanContext() {
        super(new Brake());
    }
}

// SUV - ABS 브레이크 사용
public class SUVContext extends CarContext {
    public SUVContext() {
        super(new BrakeWithABS());
    }
}
```

Java 예제: 클라이언트 코드

```
public class Main {  
    public static void main(String[] args) {  
        CarContext sedan = new SedanContext(); // 세단은 일반 브레이크로 시작  
        sedan.applyBrake();  
        // Simple Brake applied  
  
        CarContext suv = new SUVContext(); // SUV는 ABS 브레이크로 시작  
        suv.applyBrake();  
        // Brake with ABS applied  
        // - Anti-lock system activated  
        // - Brake pressure modulated  
  
        sedan.setBrakeBehavior(new BrakeWithABS()); // 런타임에 세단의 브레이크를 ABS로 변경  
        sedan.applyBrake();  
        // Brake with ABS applied  
    }  
}
```

장점

- 유연성 향상: 런타임에 알고리즘을 동적으로 변경할 수 있습니다
- 코드 재사용: 알고리즘을 독립적인 클래스로 분리하여 재사용 가능합니다
- 테스트 용이: 각 전략을 독립적으로 테스트할 수 있습니다
- 조건문 제거: if-else나 switch 문을 제거하고 다형성 활용
- 개방-폐쇄 원칙: 기존 코드 수정 없이 새로운 전략 추가 가능
- 단일 책임 원칙: 각 전략이 하나의 알고리즘에만 집중합니다

단점

- 클래스 수 증가: 전략마다 새로운 클래스가 필요합니다
- 클라이언트 인지 필요: 클라이언트가 전략들의 차이를 알아야 합니다
- 컨텍스트와 전략 간 데이터 공유: 때로는 불필요한 매개변수가 전달될 수 있습니다
- 단순한 경우 과도할 수 있음: 알고리즘이 거의 변하지 않는 경우 오버엔지니어링일 수 있습니다

사용 사례

전략 패턴은 다음과 같은 상황에서 유용하게 사용됩니다:

1. 정렬 알고리즘: 퀵소트, 병합정렬, 버블정렬 등 데이터 특성에 따라 선택
2. 압축 알고리즘: ZIP, RAR, TAR 등 압축 방식 선택
3. 결제 처리: 신용카드, PayPal, 암호화폐 등 결제 수단 선택
4. 검증 전략: 이메일, 전화번호, 주민등록번호 등 다양한 검증 로직
5. 렌더링 전략: HTML, PDF, XML 등 출력 형식 선택
6. 경로 탐색: A*, 다익스트라, BFS 등 상황에 맞는 알고리즘 선택

실제 적용 예시

1. 정렬 전략

```
interface SortStrategy {  
    void sort(int[] array);  
}  
  
class QuickSort implements SortStrategy {  
    public void sort(int[] array) {  
        // 퀵 정렬 구현  
        System.out.println("Quick Sort applied");  
    }  
}  
  
class MergeSort implements SortStrategy {  
    public void sort(int[] array) {  
        // 병합 정렬 구현  
        System.out.println("Merge Sort applied");  
    }  
}  
  
class Sorter {  
    private SortStrategy strategy;  
  
    public void setStrategy(SortStrategy strategy) {  
        this.strategy = strategy;  
    }  
  
    public void sort(int[] array) {  
        strategy.sort(array);  
    }  
}
```

2. 결제 처리

```
interface PaymentStrategy {
    void pay(int amount);
}

class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;

    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card");
    }
}

class PayPalPayment implements PaymentStrategy {
    private String email;

    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal");
    }
}

class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void checkout(int amount) {
        paymentStrategy.pay(amount);
    }

    public void setPaymentMethod(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }
}
```


3. 압축 전략

```
interface CompressionStrategy {
    void compress(String file);
}

class ZipCompression implements CompressionStrategy {
    public void compress(String file) {
        System.out.println("Compressing " + file + " using ZIP");
    }
}

class RarCompression implements CompressionStrategy {
    public void compress(String file) {
        System.out.println("Compressing " + file + " using RAR");
    }
}

class FileCompressor {
    private CompressionStrategy strategy;

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

    public void compressFile(String file) {
        strategy.compress(file);
    }
}
```

Strategy vs Template Method

측면	Strategy Pattern	Template Method Pattern
구조	합성(Composition) 사용	상속(Inheritance) 사용
알고리즘 변경	런타임에 변경 가능	컴파일 타임에 고정
유연성	높음 (객체 교체)	낮음 (클래스 확장)
알고리즘 전체	전체를 교체	일부 단계만 재정의
복잡도	더 많은 클래스 필요	상속 계층 구조
사용 시기	알고리즘 자체가 다를 때	알고리즘 구조는 같고 단계만 다를 때

관련 디자인 원칙

합성 우선 원칙 (Composition over Inheritance)

전략 패턴은 상속 대신 합성을 사용하여 더 유연한 설계를 제공합니다:

- ❌ 상속: 컴파일 타임에 고정, 단일 부모 클래스만 가능
- ✅ 합성: 런타임에 변경 가능, 다양한 전략 조합 가능

개방-폐쇄 원칙 (Open-Closed Principle)

- 개방: 새로운 전략 클래스 추가에 열려 있음
- 폐쇄: 기존 Context 코드 수정에는 닫혀 있음

의존성 주입과의 관계

전략 패턴은 의존성 주입(Dependency Injection)과 함께 사용되는 경우가 많습니다:

```
// 생성자 주입
public class Car {
    private final IBrakeBehavior brakeBehavior;
    public Car(IBrakeBehavior brakeBehavior) {
        this.brakeBehavior = brakeBehavior;
    }
}

// 세터 주입
public class Car {
    private IBrakeBehavior brakeBehavior;
    public void setBrakeBehavior(IBrakeBehavior brakeBehavior) {
        this.brakeBehavior = brakeBehavior;
    }
}
```

함수형 프로그래밍에서의 전략 패턴

현대 언어에서는 고차 함수를 사용하여 더 간단하게 구현 가능:

```
public class Car {  
    private Consumer<Void> brakeStrategy;  
    public void setBrakeStrategy(Consumer<Void> strategy) {  
        this.brakeStrategy = strategy;  
    }  
    public void applyBrake() {  
        brakeStrategy.accept(null);  
    }  
}  
  
Car car = new Car();  
car.setBrakeStrategy(v -> System.out.println("ABS brake"));  
car.applyBrake();
```





실무에서의 전략 패턴

전략 패턴은 많은 프레임워크와 라이브러리에서 사용됩니다:




1. **Java Collections:** `Comparator` 인터페이스
2. **Spring Framework:** 다양한 `Strategy` 인터페이스들
3. **Android:** `OnClickListener`, `TextWatcher` 등
4. **JavaScript:** 콜백 함수, `Promise handlers`
5. **Flutter:** `Builder` 패턴과 함께 사용

요약

전략 패턴은 다음과 같은 경우에 유용함:

-  런타임에 알고리즘을 선택해야 할 때
-  같은 목적의 다양한 알고리즘이 존재할 때
-  조건문을 제거하고 싶을 때
-  알고리즘을 독립적으로 테스트하고 싶을 때

전략 패턴은 다음과 같은 경우에 피해함:

-  전략이 거의 변하지 않을 때
-  알고리즘이 매우 단순할 때
-  전략 간 공유 데이터가 많을 때