5. 스트래티지 패턴

전략

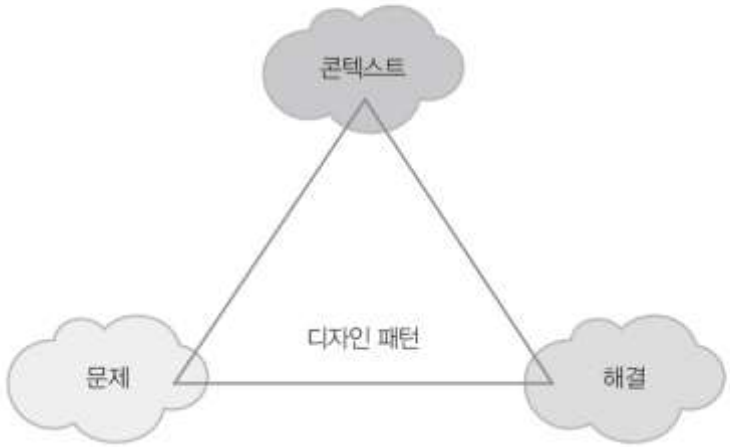
학습목표

학습목표

- 알고리즘 변화를 캡슐화로 처리하는 방법 이해하기
- 스트래티지 패턴을 통한 알고리즘의 변화를 처리하는 방법 이해하기
- 사례 연구를 통한 스트래티지 패턴의 핵심 특징 이해하기

디자인 패턴 구조

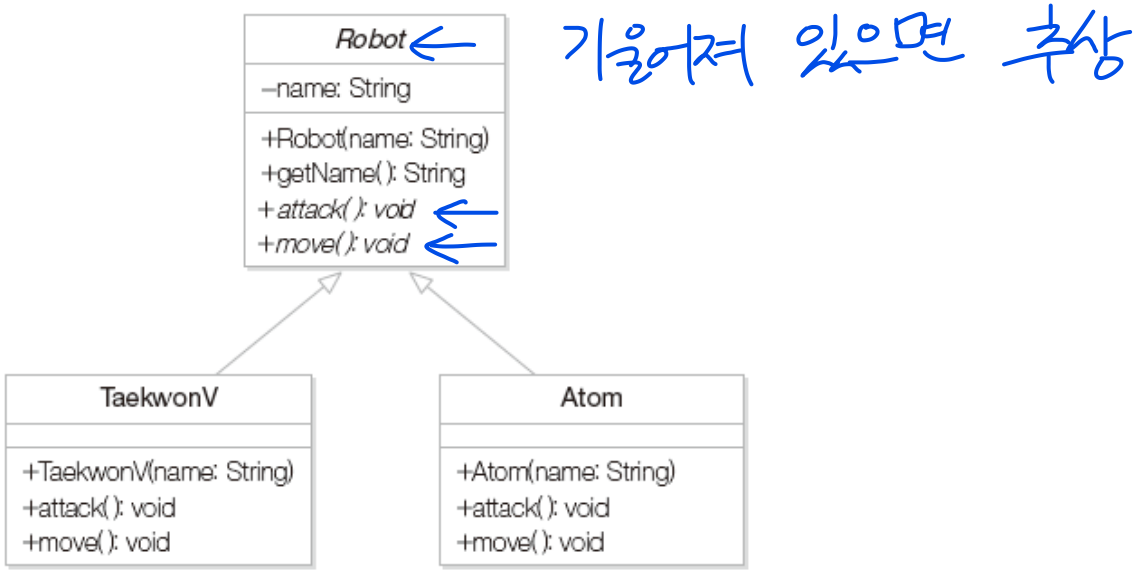
그림 4-3 디자인 패턴의 구성 요소



- 콘텍스트: 문제가 발생하는 여러 상황을 기술한다. 즉, 패턴이 적용될 수 있는 상황을 나타낸다. 경우에 따라서는 패턴이 유용하지 못하는 상황을 나타내기도 한다.
- 문제: 패턴이 적용되어 해결될 필요가 있는 여러 디자인 이슈들을 기술한다. 이때 여러 제약 사항과 영향력도 문제 해결을 위해 고려해야 한다
- 해결: 문제를 해결하도록 설계를 구성하는 요소들과 그 요소들 사이의 관계, 책임, 협력 관계를 기술한다. 해결은 반드시 구체적인 구현 방법이나 언어에 의존적이지 않으며 다양한 상황에 적용할 수 있는 일종의 템플릿이다.

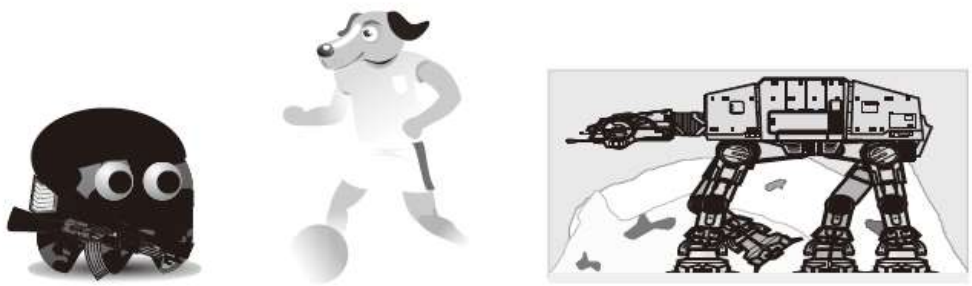
5.1 로봇만들기

그림 5-2 로봇 설계



기울어져 있으면 추상

그림 5-1 각양각색의 로봇



5.2 문제점

- 기존 로봇의 공격 또는 이동 방법을 수정하려면 어떤 변경 작업을 해야 하는가? 예를 들어 아톰이 날 수는 없고 오직 걷게만 만들고 싶다면? 또는 태권V를 날게 하려면?

5.2 문제점

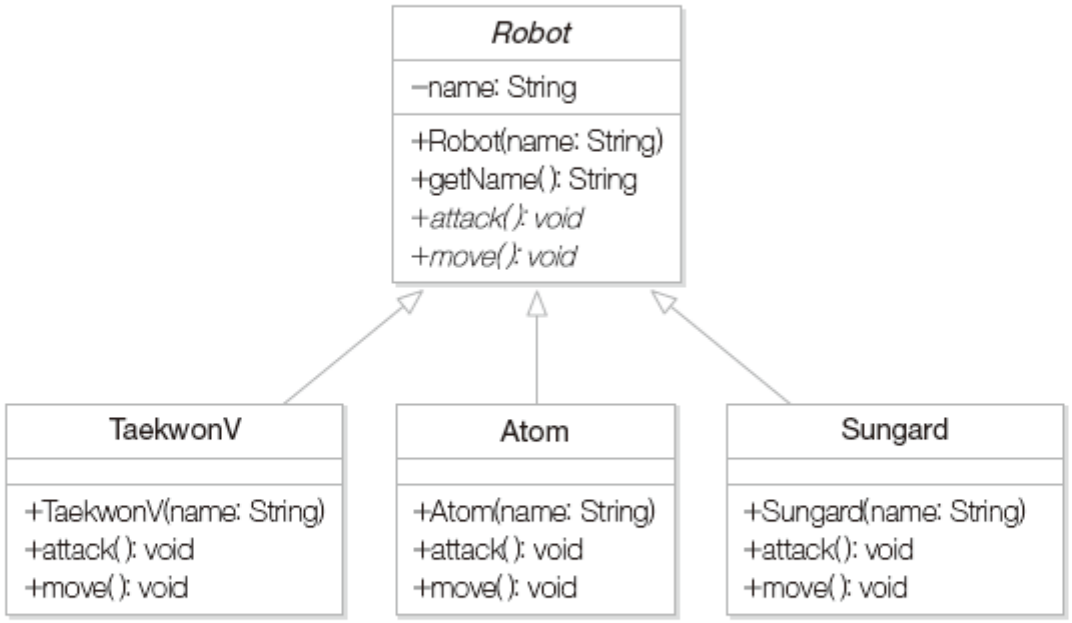
- 아톰이 걷게만 하려면 move() 메소드를 수정해야 함
  - 태권브이와 아톰의 move() 코드가 동일해짐 -> 중복 발생

5.2 문제점

- 새로운 로봇을 만들어 기존의 공격 또는 이동 방법을 추가하거나 수정하려면? 예를 들어 새로운 로봇으로 지구의 용사 선가드(Sungard 클래스)를 만들어 태권V의 미사일 공격 기능을 추가하려면?

새로운 로봇의 추가

그림 5-3 새로운 로봇의 추가



Keypoint_ 현재 시스템의 캡슐화 단위에 따라 새로운 로봇을 추가하기는 매우 쉽다.

새로운 로봇의 추가

- 태권브이와 선가드의 attack 메소드 중복되어 사용되는 문제

5.3 해결책

Keypoint_ 무엇이 변화되었는지를 찾은 후에 이를 클래스로 캡슐화한다.

***이동 방식과 공격 방식이 변화되었음**

-> 새로운 방식의 이동 및 공격이 계속 추가될 수 있으므로 기존 로봇이나 새 로봇이 별다른 코드 변경 없이 기능을 제공받거나 기존 공격이나 이동 방식을 다른 공격이나 이동 방식으로 쉽게 변경할 수 있어야 함

그림 5-4 공격과 이동 전략 인터페이스

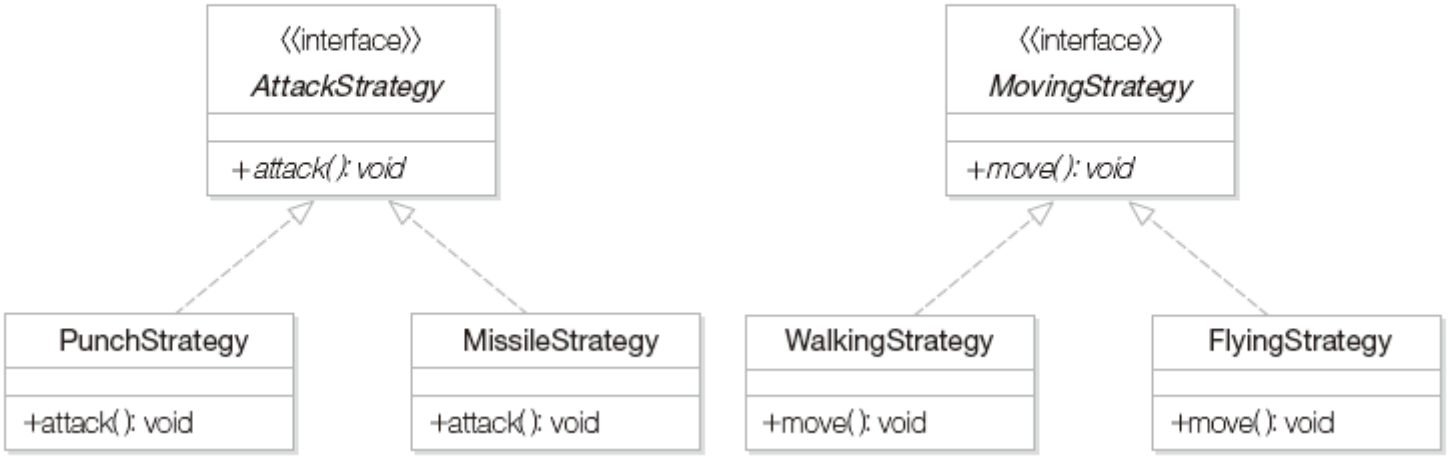
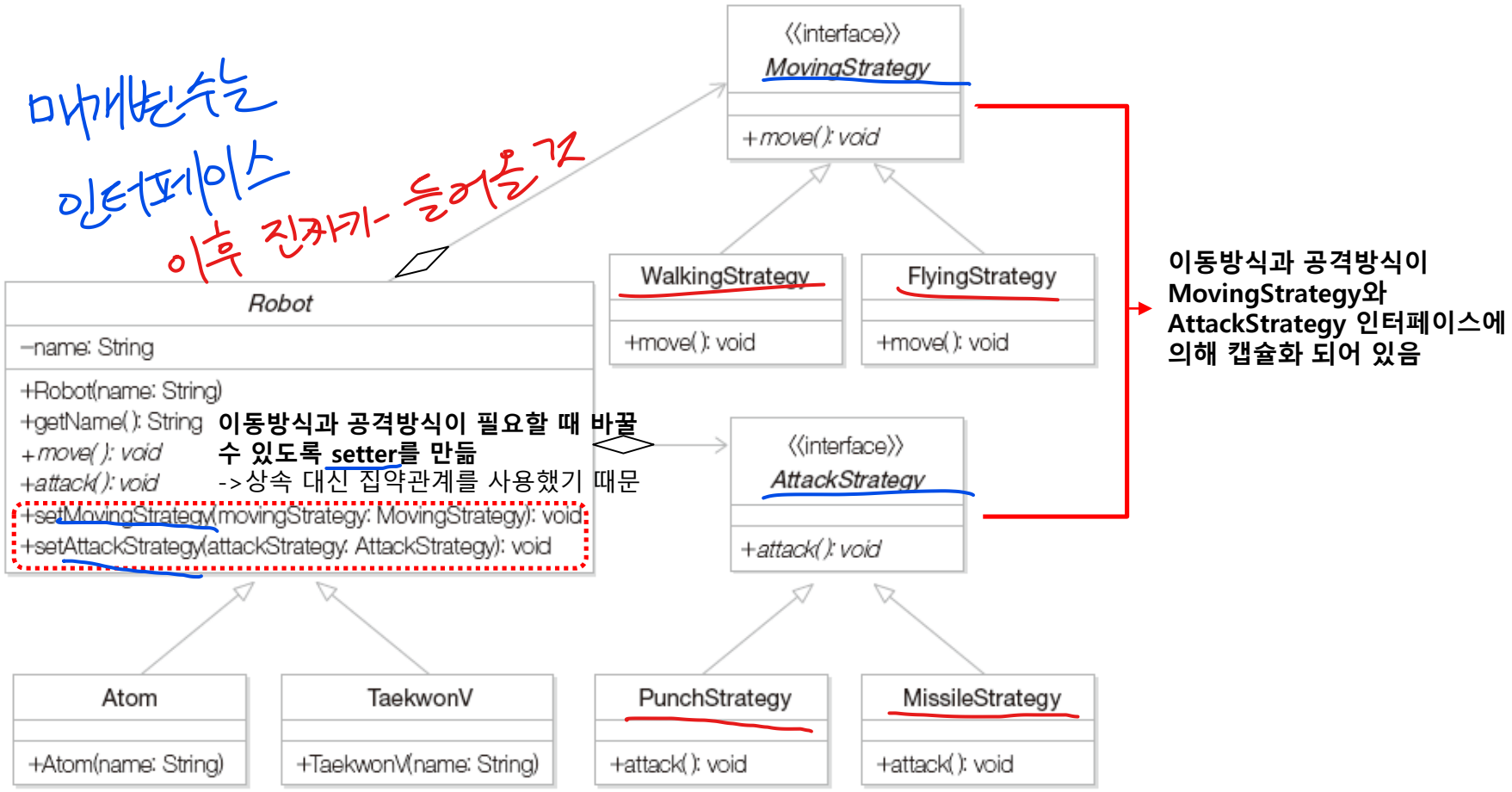


그림 5-5 개선된 인터페이스



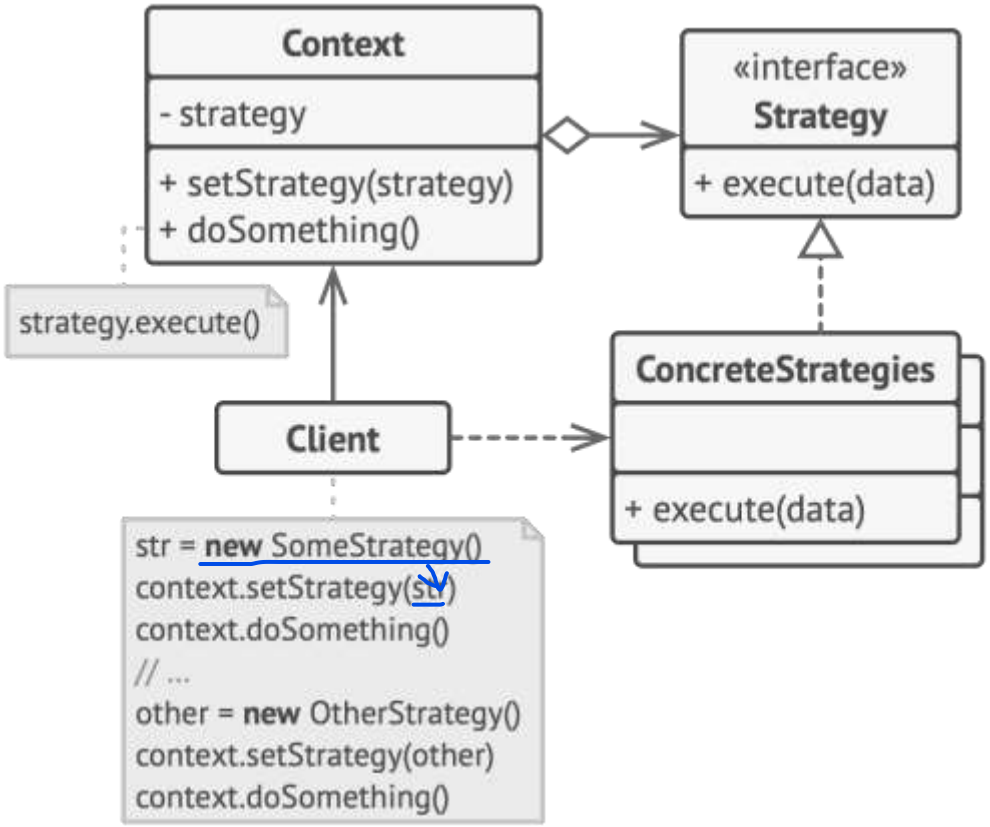
5.4 스트래티지 패턴

- 스트래티지 패턴Strategy Pattern은 전략을 쉽게 바꿀 수 있도록 해주는 디자인 패턴이다.
  - 여기에서 전략이란 어떤 목적을 달성하기 위해 일을 수행하는 방식, 비즈니스 규칙, 문제를 해결하는 알고리즘 등으로 이해할 수 있다.
- 프로그램에서 전략을 실행할 때는 쉽게 전략을 바꿔야 할 필요가 있는 경우가 많이 발생한다.
- 특히 게임 프로그래밍에서 게임 캐릭터가 자신이 처한 상황에 따라 공격이나 행동하는 방식을 바꾸고 싶을 때 스트래티지 패턴은 매우 유용하다.

5.4 스트래티지 패턴

- 전략 패턴(Strategy Pattern)은 알고리즘군을 정의하고 캡슐화해서 각각의 알고리즘군을 수정해서 쓸 수 있게 해줍니다. 이는 클라이언트로부터 알고리즘을 분리해서 독립적으로 변경할 수 있습니다. - 에릭 프리먼 외. (2020). 헤드 퍼스트 디자인 패턴: 서환수 옮김. 한빛미디어
- 전략 패턴에서 콘텍스트는 사용할 전략을 직접 선택하지 않는 대신, 콘텍스트의 클라이언트가 콘텍스트에 사용할 전략을 전달 – 최범균. (2013). 개발자가 반드시 정복해야 할 객체 지향과 디자인 패턴. 인투북스
  - DI(의존 주입)를 이용해서 콘텍스트에 전략을 전달
  - 전략이 어떤 메소드를 제공할지에 대한 여부는 콘텍스트가 전략을 어떻게 사용하냐에 따라 달라짐
- IF-ELSE IF-ELSE/SWITCH와 같은 구조를 사용해야 하는 경우에 전략 패턴이 유용하게 활용될 수 있음
- 전략 패턴은 책에 따라 정책 패턴(Policy Pattern)이라고도 불림

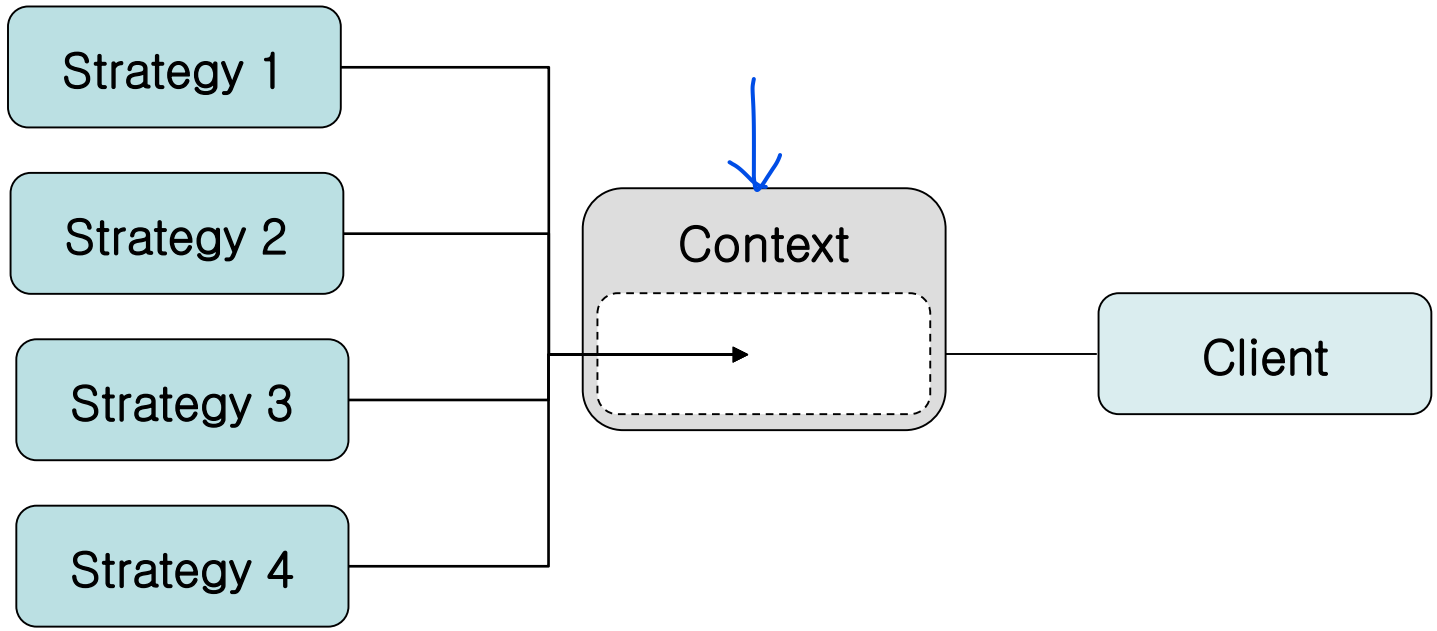
5.4 스트래티지 패턴



출처 : flavono123, 디자인 패턴 – 전략, 방문자, 템플릿 메서드,
https://flavono123.oopy.io/posts/design-patterns-strategy-visitor-template-method

5.4 스트래티지 패턴

- 특정 작업을 하는 전략들을 여러 개를 두고 필요할 때마다 갈아끼우는 패턴



출처 : 얄팍한 코딩사전, 전략(Strategy) 패턴, https://www.youtube.com/watch?v=xlaAiHrZN3U

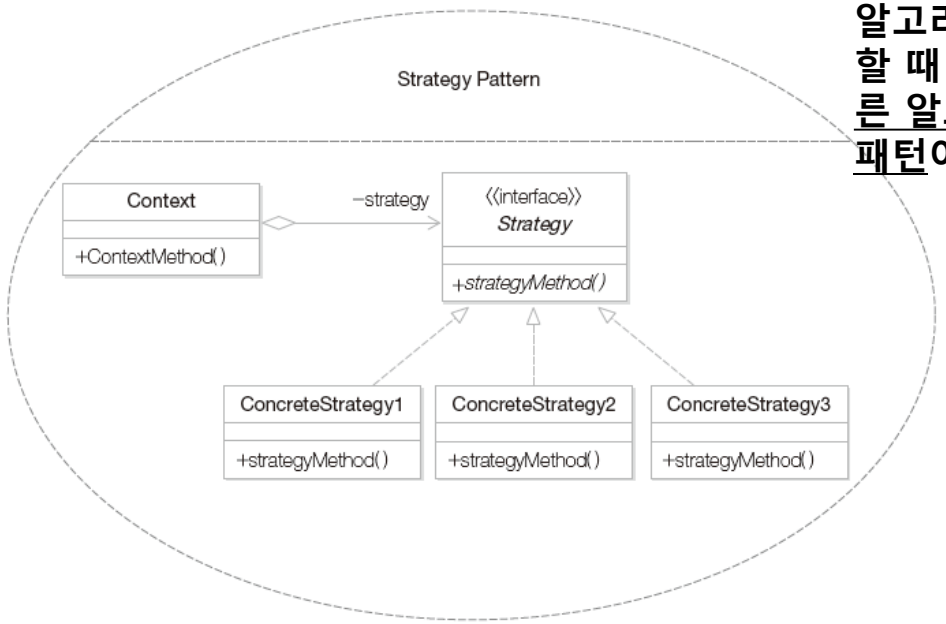
정답은 인터페이스!

- 인터페이스를 활용하면 구조를 유지하면서 호환성 있는 하위 코드의 설계가 가능
- 프로그램 동작 중에 동적으로 알고리즘을 교체하기(갈아끼우기) 위해서는 별도의 setter 메소드 구현이 필요

이호진. (2020). 쉽게 배워 바로 써먹는 디자인 패턴. 한빛미디어

정답은 인터페이스!

그림 5-6 스트래티지 패턴 컬레보레이션



스트래티지 패턴은 같은 문제를 해결하는 여러 알고리즘이 클래스별로 캡슐화되어 있고 필요할 때 교체가능토록 함으로써 동일한 문제를 다른 알고리즘으로 해결할 수 있게 하는 디자인 패턴이다.

- **Strategy**: 인터페이스나 추상 클래스로 외부에서 동일한 방식으로 알고리즘을 호출하는 방법을 명시한다.
- **ConcreteStrategy1, ConcreteStrategy2, ConcreteStrategy3**: 스트래티지 패턴에서 명시한 알고리즘을 실제로 구현한 클래스다.
- **Context**: 스트래티지 패턴을 이용하는 역할을 수행한다. 필요에 따라 동적으로 구체적인 전략을 바꿀 수 있도록 setter 메서드를 제공한다.

그림 5-7 스트래티지 패턴의 순차 다이어그램

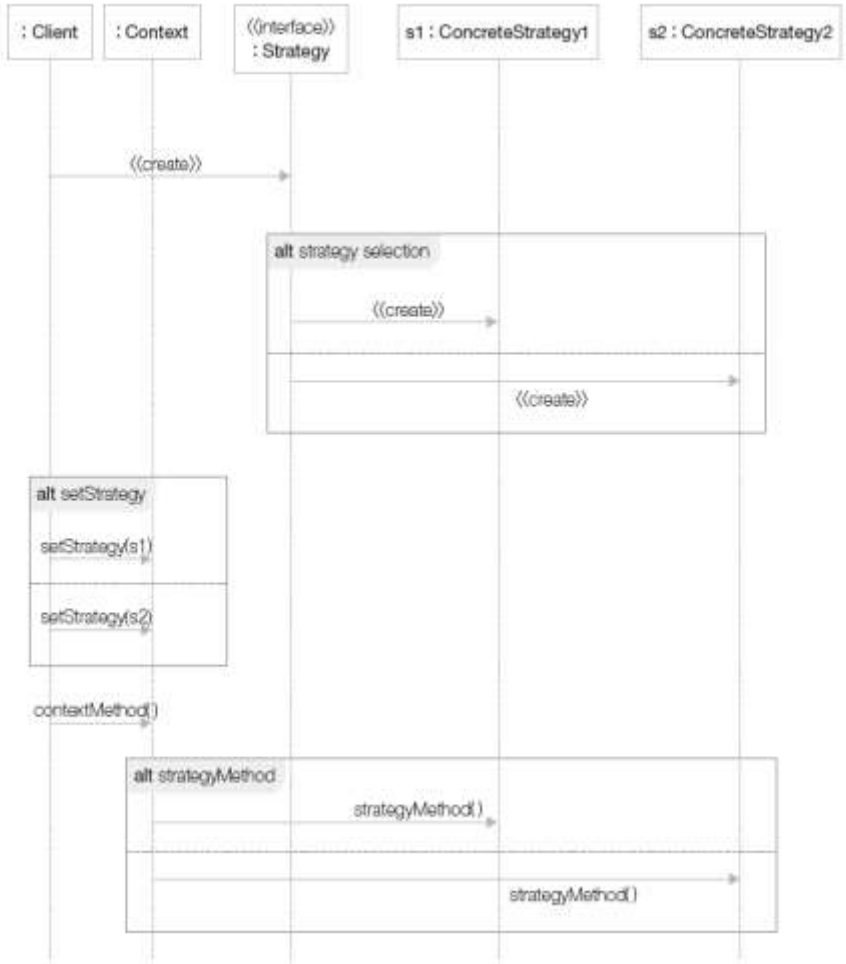
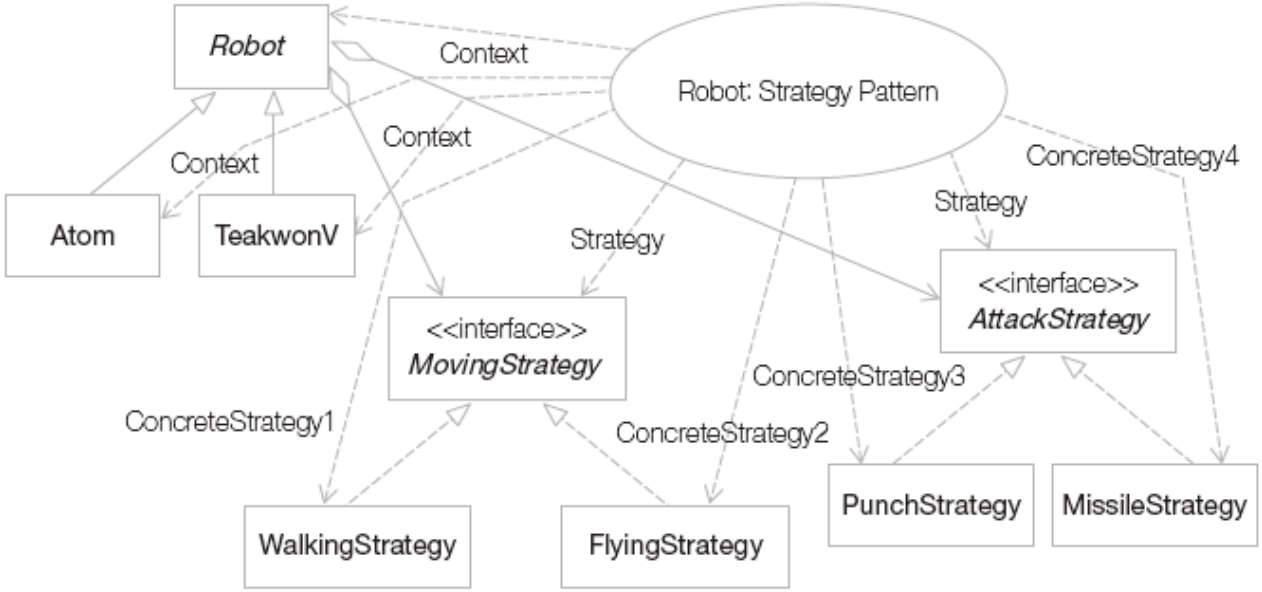


그림 5-8 스트래티지 패턴을 로봇 예제에 적용한 경우



- Robot, Atom, TeakwonV는 Context 역할을 한다.
- MovingStrategy와 AttackStrategy는 각각 Strategy 역할을 한다.
- WalkingStrategy, FlyingStrategy, PunchStrategy, MissileStrategy 클래스는 ConcreteStrategy 역할을 한다.

결제 방법 구현

- 개똥이는 구멍가게를 열었습니다. 아직은 규모가 너무 작아 현금으로만 물건값을 받고 있어요.
- 그런데 현금을 가져오지 않거나 계좌이체를 꺼리는 손님들이 많이 발생하기 시작했습니다. 카드 결제가 되지 않는다고 하니 물건을 사지 않고 돌아가는 사람이 너무 많아졌습니다. 손해가 크다고 판단하고 이제는 카드 결제 기능도 추가하려고 합니다.

결제 방법 구현

- 여기서 현금결제, 카드결제의 공통점은?
  - 결제

- 그렇다면 '결제'라고 하는 부분을 인터페이스로 만들고, 결제와 관련된 부분을 실제로 구현을 하면 어떨까?

결제 방법 구현

- 결제 인터페이스

```
interface PaymentStrategy {  no usages
    void pay(int amount);  no usages
}
```

결제 방법 구현

- 현금 결제를 구현하려면?
  - 결제 인터페이스를 implements로 받아 구현
  - 인터페이스의 pay()도 같이 구현하게 됨

```
public class CashPayment implements PaymentStrategy{  no usages
    @Override  no usages
    public void pay(int amount) {
        System.out.println("현금으로 "+amount+"원 결제합니다.");
    }
}
```

결제 방법 구현

❖ 그렇다면 카드 결제를 구현하려면?

```
public class CreditCardPayment implements PaymentStrategy{  no usages
    private String name;  1 usage
    private String creditCardNumber;  1 usage

    public CreditCardPayment(String name, String creditCardNumber){  no usages
        this.name=name;
        this.creditCardNumber=creditCardNumber;
    }
    @Override  no usages
    public void pay(int amount) {
        System.out.println("신용카드로 "+amount+"원 결제합니다.");
    }
}
```

❖ 만약 여기서 네이버페이로 결제하는 기능을 추가한다면?

결제 방법 구현

❖ 해당 코드의 콘텍스트(Context)는 개똥이네 가게

```
public class Store {  no usages
    private PaymentStrategy paymentStrategy;  2 usages

    public void setPaymentStrategy(PaymentStrategy paymentStrategy){  no usages
        this.paymentStrategy=paymentStrategy;
    }

    public void counter(int amount){  no usages
        paymentStrategy.pay(amount);
    }
}
```

결제 방법 구현

❖ Main 구현

```
public class Main {
    public static void main(String[] args) {
        Store store = new Store();
        store.setPaymentStrategy(new CashPayment());
        store.counter( amount: 50000);

        store.setPaymentStrategy(new CreditCardPayment( name: "홍길동",  creditCardNumber: "1234-5678-1234-7890"));
        store.counter( amount: 200000);
    }
}
```

사용 예시

- 모바일 접속 시 LTE를 사용할지 Wifi를 사용할지 전략 패턴으로 구현
- 게임기의 게임팩 슈퍼 마리오에서 젤다의 전설로 갈아 끼우기

정리

- 전략 패턴을 이용하면 원하는 알고리즘을 선택적으로 교환 가능
- 상속 보다는 위임 처리를 통해 구성하는 형태를 선호하는 패턴
- 알고리즘 객체를 교환하여 사용한다는 측면에서 전략 패턴은 유용한 패턴
- 행동 변경 시 조건문을 사용하지 않고도 원하는 행동으로 교체 가능
- 하지만 알고리즘 객체가 교체된다는 점에서 실행 시 많은 객체를 갖는다는 단점도 있음

이호진. (2020). 쉽게 배워 바로 써먹는 디자인 패턴. 한빛미디어