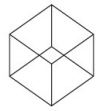
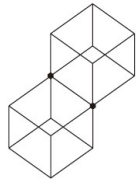
5. 스트래티지 패턴

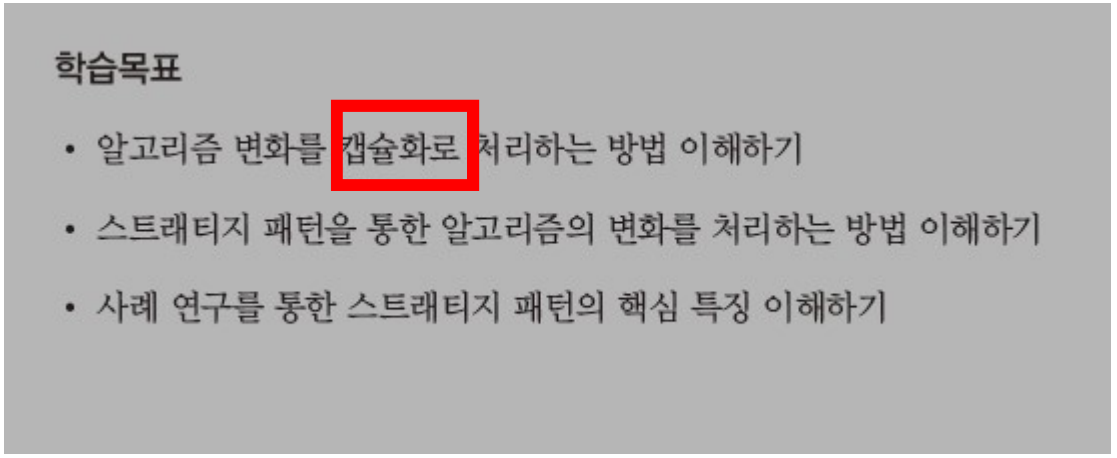


JAVA 객체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



학습목표


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

5.1 로봇만들기

❖ 로봇만들기

- 아톰 , 태권 V
- 공격기능
- 이동기능
- 아톰 : 공격할때 주먹만 사용 , 하늘을 날수 있음
- 태권V : 미사일공격가능, 걷기만 함

그림 5-1 각양각색의 로봇

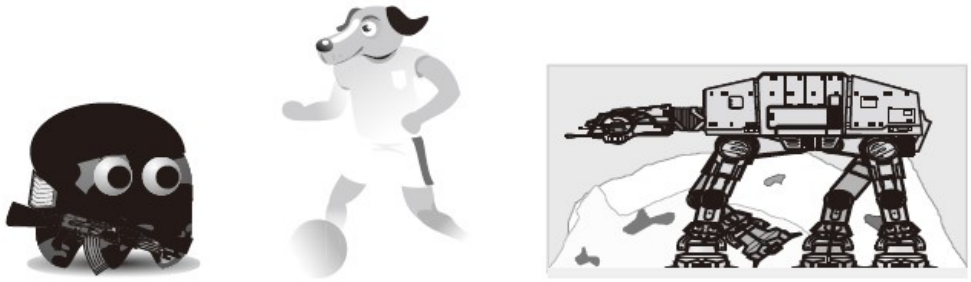
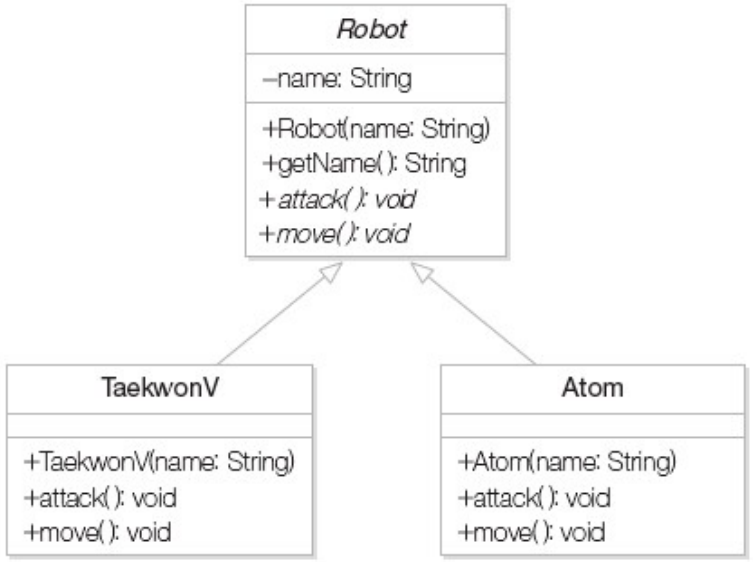


그림 5-2 로봇 설계



코드 5-1

```
public abstract class Robot {
    private String name;
    public Robot(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract void attack();
    public abstract void move();
}

public class TaekwonV extends Robot {
    public TaekwonV(String name) {
        super(name);
    }
    public void attack() {
        System.out.println("I have missile and can attack with it.");
   }
    public void move() {
       System.out.println("I can only walk.");
    }
}
```

코드 5-1

```
public class  Atom extends Robot {
    public Atom(String name) {
        super(name);
    }
    public void attack() {
        System.out.println("I have strong punch and can attack with it.");
    {

    public void move() {
       System.out.println("I can only fly.");
    }
}

public class Client {
    public static void main(String[] args) {
        Robot taekwonV = new Taekwon("TaekwonV");
        Robot atom = new Atom("Atom);

        System.out.println("My name is " + taekwonV.getName());
        taekwonV.move();
        taekwonV.attack();

        System.out.println();

        System.out.println("My name is " + atom.getName());
        atom.move();
        atom.attack();
        }
}
```

5.2 문제점 (확장 요구사항)

- 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?
- 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?

5.2.1 기존 로봇의 공격과 이동 방법을 수정하는 경우

❖ 수정사항

- 아톰이 날수 없고 오직 걷게만 하도록 변경

코드 5-2

```
public abstract class Robot {
    private String name;

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

    public String getName() {
        return name;
    }

    public abstract void attack();
    public abstract void move();
}
```

코드 5-2

```
public class TaekwonV extends Robot {
    public TaekwonV(String name) {
        super(name);
    }

    public void attack() {
        System.out.println("I have Missile and can attck with it.");
    }

    public void move() {
        System.out.println("I can only walk.");
    }
}
```

```
public class Atom extends Robot {
    public Atom(String name) {
        super(name);
   }

    public void attack() {
       System.out.println("I have strong punch and can attack with it.");
    }

    public void move() {
       System.out.println("I can only walk.");
    }
}
```

코드 5-1

```
public class  Atom extends Robot {
    public Atom(String name) {
        super(name);
    }
    public void attack() {
        System.out.println("I have strong punch and can attack with it.");
    {

    public void move() {
       System.out.println("I can only fly.");
    }
}
```

❖ 변경(코드 5-2)의 문제점

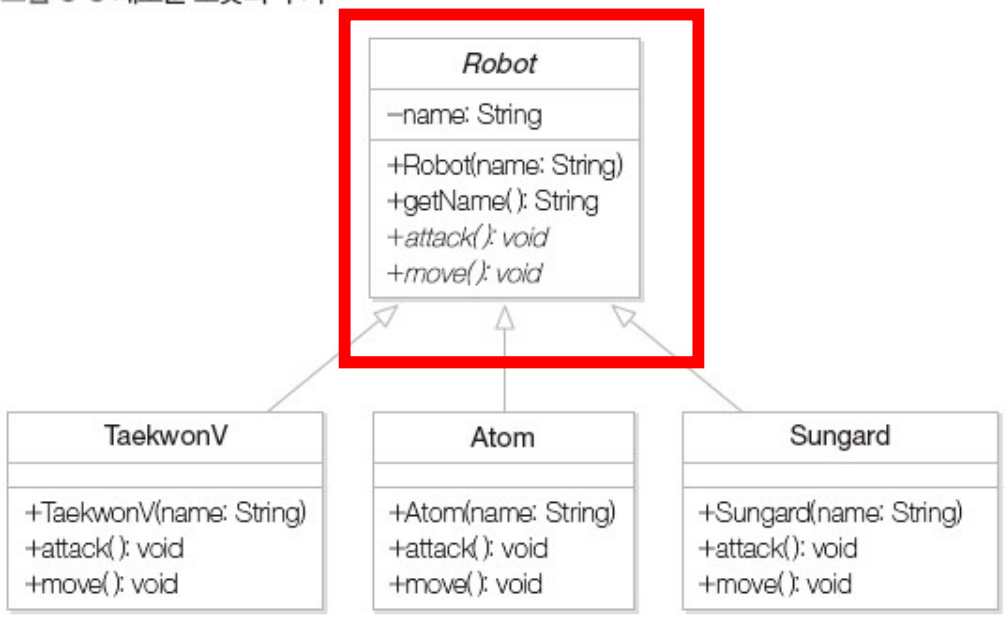
- 1) 새로운 기능을 변경하려고 기존 코드의 내용을 수정하는 것 → OCT에 위배
- 2) Atom 클래스의 move 메소드와 TaekwonV 클래스의 move가 동일한 기능을 실행하므로 기능이 중복됨
  - 로봇의 종류가 많아질수록 중복코드를 일관되게 유지관리하는 것은 힘듬

5.2.2 새로운 로봇에 공격/이동 방법을 추가/수정하는 경우

❖ 새로운 로봇의 추가

- 그림 5-3. Sungard 를 추가

그림 5-3 새로운 로봇의 추가



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

❖ 그림 5-3의 문제점

- 새로운 로봇에 기존의 공격 또는 이동 방법을 추가하거나 변경하려고 할 경우
- 예) Sungard 클래스에 태권 V의 미사일 공격 능력을 사용하려고 할 경우
  - TaekwonV 클래스와 Sungard 클래스의 attack 메소드가 중복해서 사용

- 새로운 방식의 이동기능과 공격기능이 추가될 경우 기존의 모든 코드를 수정해야함

5.3 해결책

❖ 변화내용

- 무엇이 변화되는지를 찾자 → 이동기능과 공격기능
- 이를 캡슐화
- 외부에서 구체적인 이동방식과 공격방식을 담은 구체적인 클래스를 은닉함
- 공격과 이동을 위한 인터페이스를 만듬
- 이들을 실제 실현한 클래스를 만듬

Keypoint_ 무엇이 변화되었는지를 찾은 후에 이를 클래스로 캡슐화한다.

❖ 그림 5-4

- MovingStrategy : 이동기능을 캡슐화
- AttackStrategy : 공격기능을 캡슐화
- (액션)에 해당하는 부분을 Object로 처리

그림 5-4 공격과 이동 전략 인터페이스

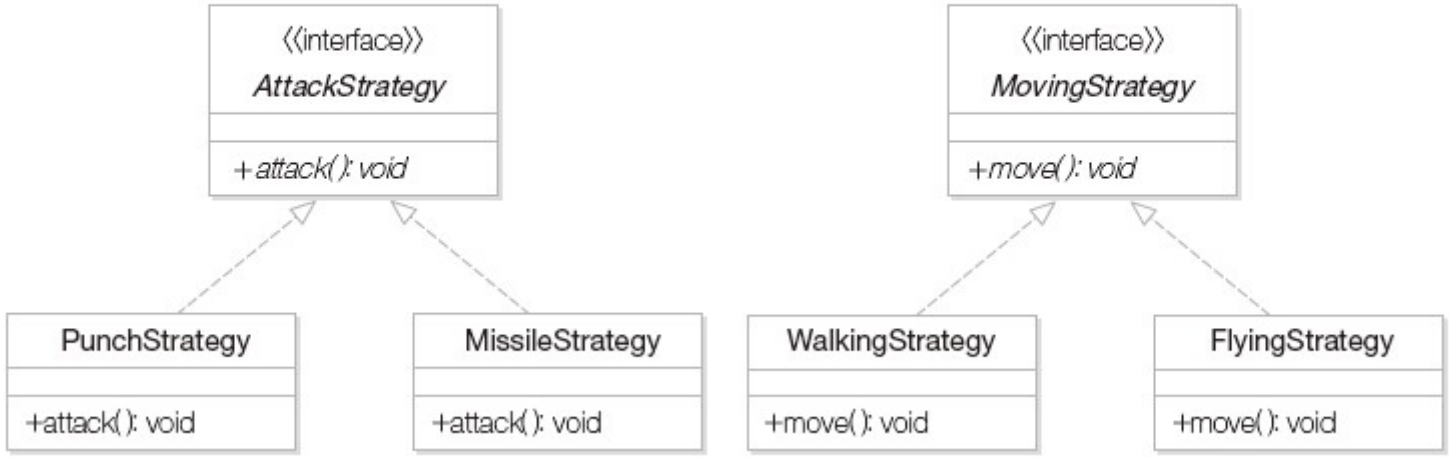
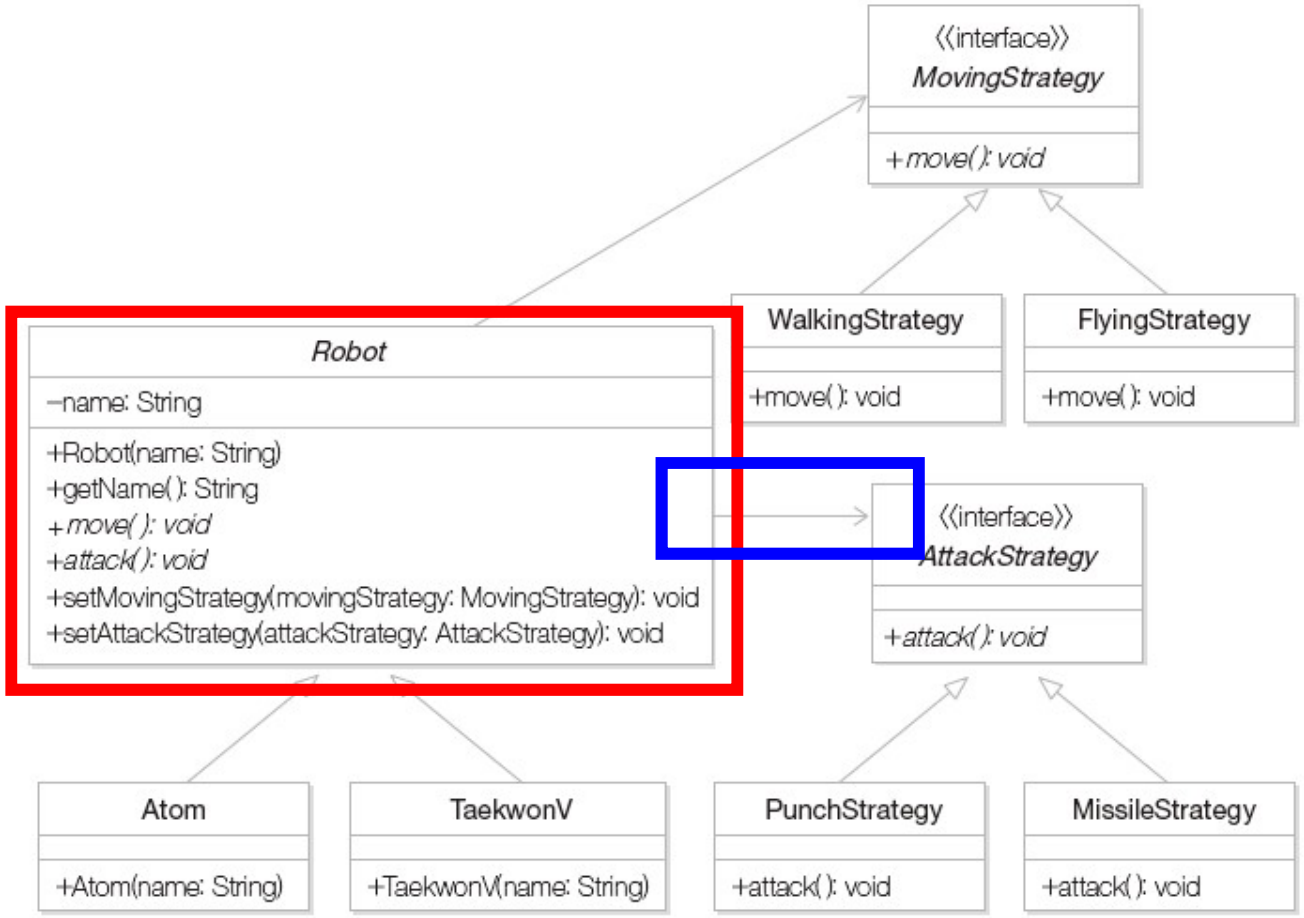


그림 5-5 개선된 인터페이스



❖ 그림 5-5에 대한 평가

– Robot class 입장에서 보기

- 구체적인 이동방식과 공격방식이 두개의 Interface에 의해 캡슐화
  - ➢ 이동방식과 공격방식을 저장하는 attribute 존재
- 향후 등장할 이동 방식과 공격방식의 변화에 무관
- 새로운 공격 방식이 나와도 Robot class를 수정할 필요는 없음

– 외부에서 공격방식과 이동방식을 바꾸는 통로

- setMoveStrategy
- setAttackStrategy

표 5-1 클래스 설명

클래스 이름	클래스 설명
Robot	Robot 클래스. 이동과 공격을 실행하는 메서드가 있고, 이를 상속받아 구체적인 로봇을 만듦
Atom, TaekwonV	Robot 클래스를 상속 받아 실제 로봇을 구현하
<<interface>> AttackStrategy	각 로봇이 취할 수 있는 공격 방법에 대한 인터페이스
PunchStrategy. MissileStrategy	각 공격 방법을 실제로 구현함
<<interface>> MovingStrategy	각 로봇이 취할 수 있는 이동 방법에 대한 인터페이스
WalkingStrategy. FlyingStrategy	각 이동 방법을 실제로 구현함

코드 5-3

```
public abstract class Robot {
    private String name;
    private MovingStrategy movingStrategy;
    private AttackStrategy attackStrategy;

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

    public String getName() {
        return name;
    }

    public void move() {
        movingStrategy move();
    }

    public void attack() {
        attackStrategy.attack();
    }
```

코드 5-3

```
public void setMovingStrategy(MovingStrategy movingStrategy) {
    this.movingStrategy = movingStrategy;
}

public void  setAttackStrategy(AttackStrategy attackStrategy) {
    this.attackStrategy = attackStrategy;
}
```

```
    }
}

public class Atom extends Robot {
    public Atom(String name) {
       super(name);
     }
}

public class TaekwonV extends Robot {
    public TaekwonV(String name) {
        super(name);
    }
}

interface MovingStrategy {
    public void move();
}
```

코드 5-3

```
public class FlyingStrategy implements MovingStrategy {
    public void move() {
        System.out.println("I can fly.");
    }
}

public class WalkingStrategy implements MovingStrategy {
    public void move() {
        System.out.println("I can only walk.");
    }
}

interface AttackStrategy {
    public void attack();
}

public class MissileStrategy implements AttackStrategy {
    public void attack() {
        System.out.println("I have Missile and can attack with it.");
    }
}

public class PunchStrategy implements AttackStrategy {
    public void attack() {
        System.out.println("I have strong punch and can attack with it.");
    }
}
```

코드 5-3

```
public class Client {
    public static void main(String[] args) {
        Robot taekwonV = new TaekwonV("TaekwonV");
        Robot atom = new Atom("Atom");

        taekwonV.setMovingStrategy(new WalkingStrategy());
        taekwonV.setAttackStrategy(new MissileStrategy());

        atom.setMovingStrategy(new FlyingStrategy()); // 이동 전략을 날아간다는 전략으로 설정됨
        atom.setAttackStrategy(new PunchStrategy()); // 공격 전략을 펀치를 구사하는 전략으로 설정됨

        System.out.println("My name is " + taekwonV.getName());
        taekwonV.move();
        taekwonV.attack();

        System.out.println();

        System.out,println("My name is " + atom.getName());
        atom.move();
        atom.attack();
    }
}
```

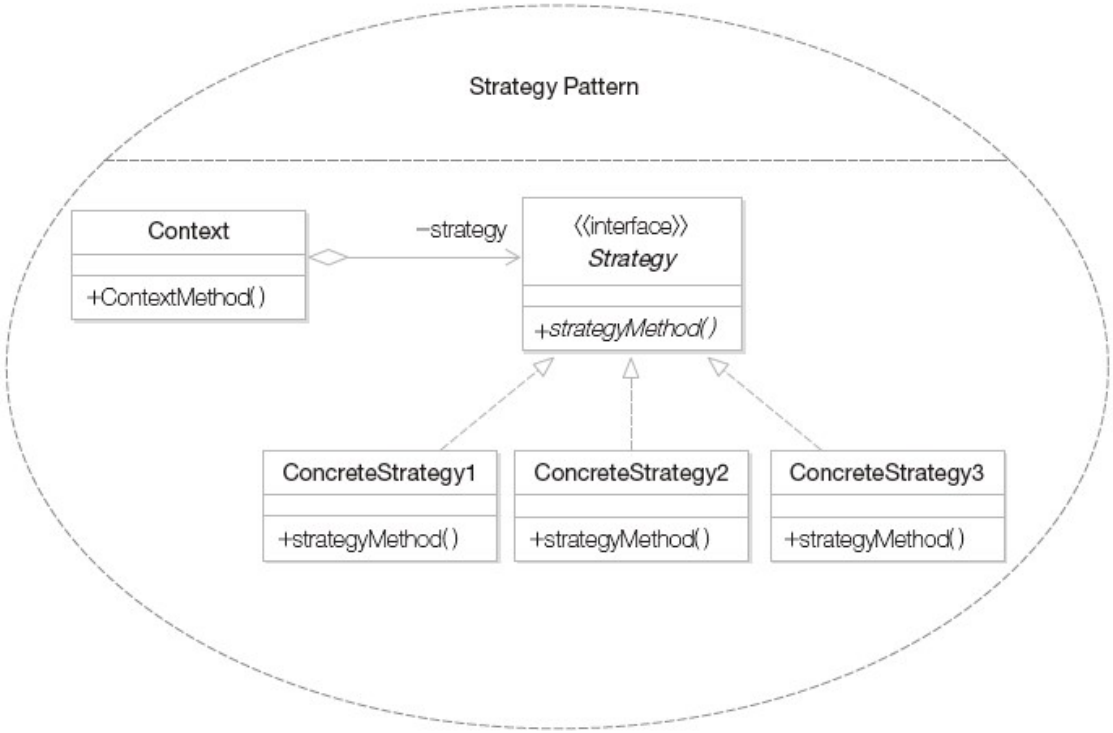
5.4 스트래티지 패턴

❖ 스트래티지 패턴(Strategy Pattern)

- 전략을 쉽게 바꿀 수 있도록 해주는 디자인 패턴
- 전략이란 어떤 목적을 달성하기 위해 일을 수행하는 방식, 비즈니스 규칙, 문제를 해결하는 알고리즘 등
- 쉽게 전략을 바꿔야 할 필요가 있는 경우가 많이 발생
- 특히 게임 프로그래밍에서 게임 캐릭터가 자신이 처한 상황에 따라 공격이나 행동하는 방식을 바꾸고 싶을 때 스트래티지 패턴은 매우 유용

Key point_ 스트래티지 패턴은 같은 문제를 해결하는 여러 알고리즘(방식)이 클래스별 캡슐화되어 있고 이들이 필요할 때 교체할 수 있도록 함으로써 동일한 문제를 다른 알고리즘으로 해결할 수 있게 하는 디자인 패턴이다.

그림 5-6 스트래티지 패턴 컬레보레이션



- **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

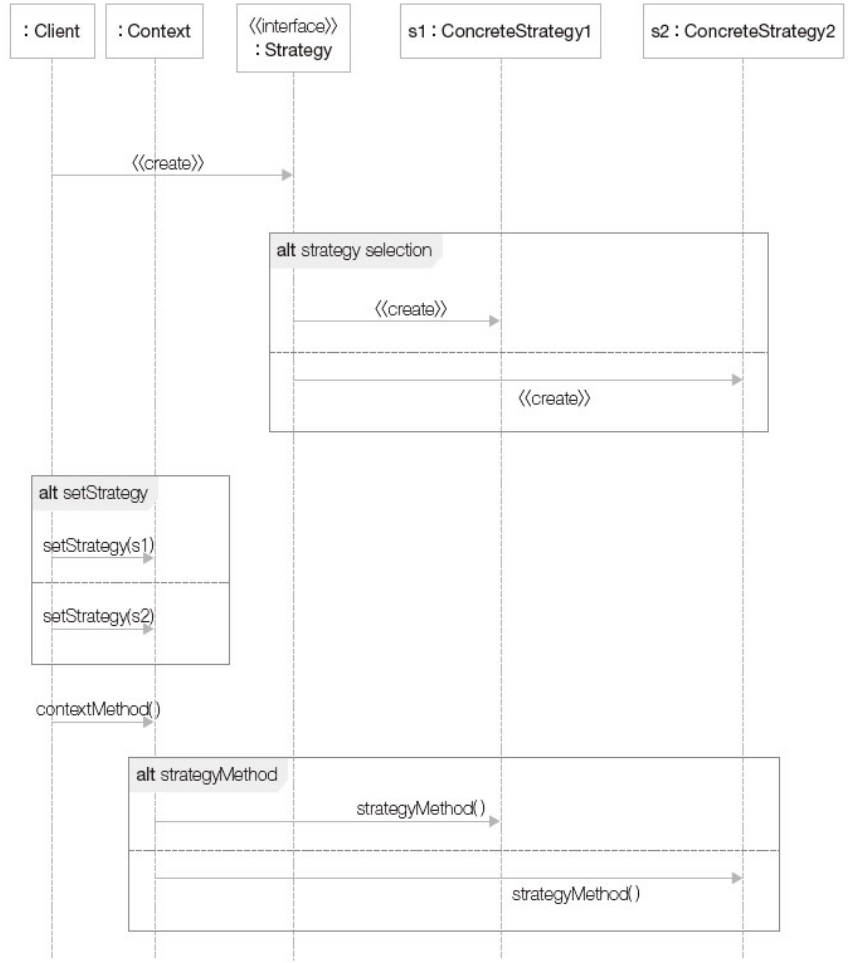
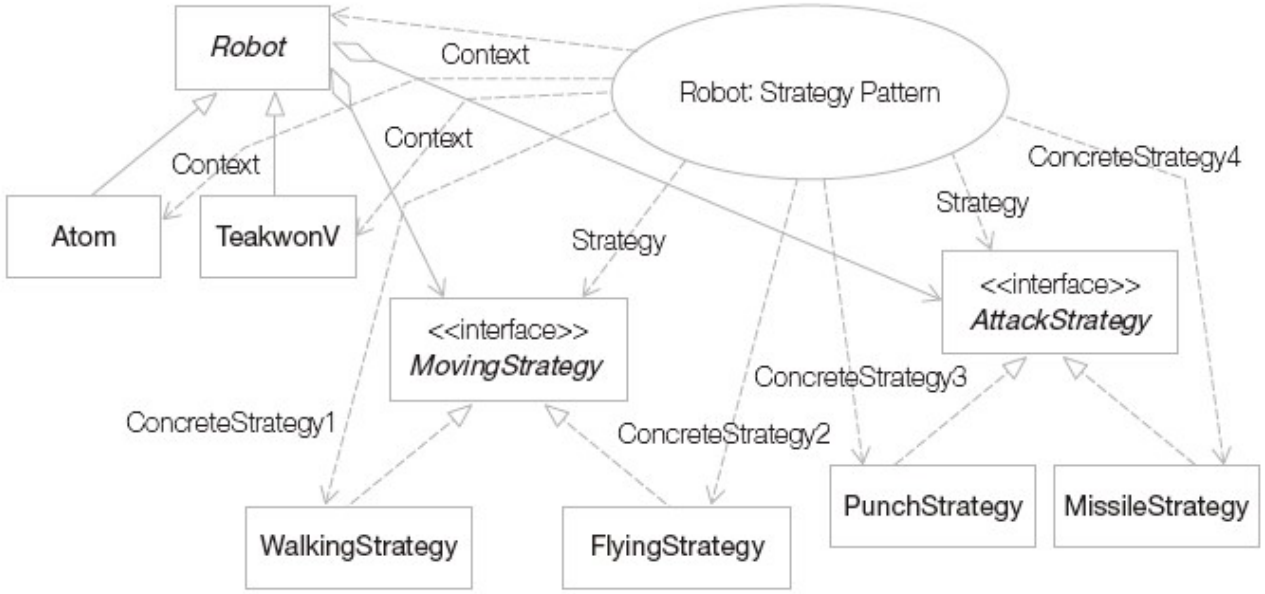


그림 5-8 스트래티지 패턴을 로봇 예제에 적용한 경우



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