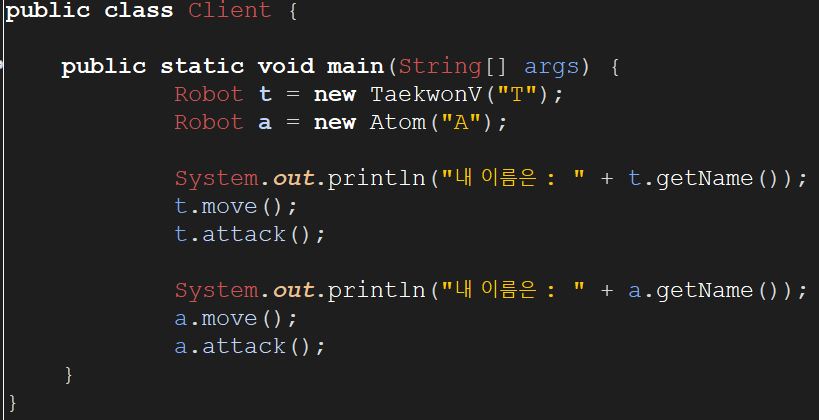
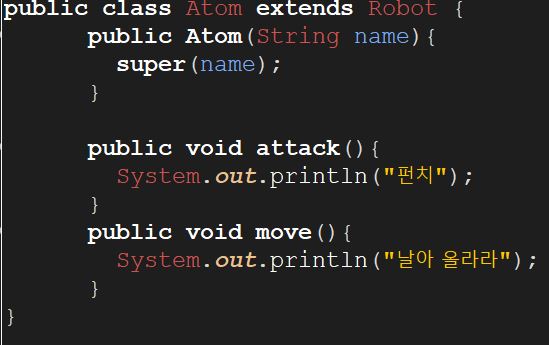
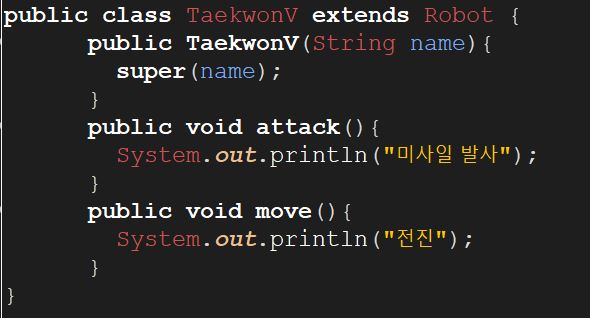
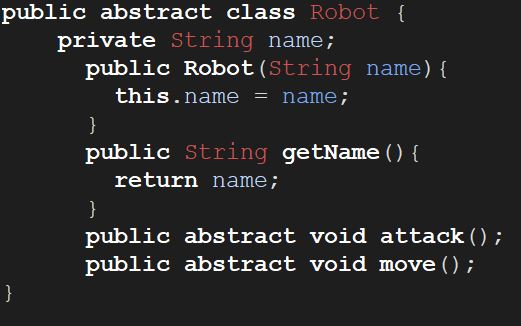
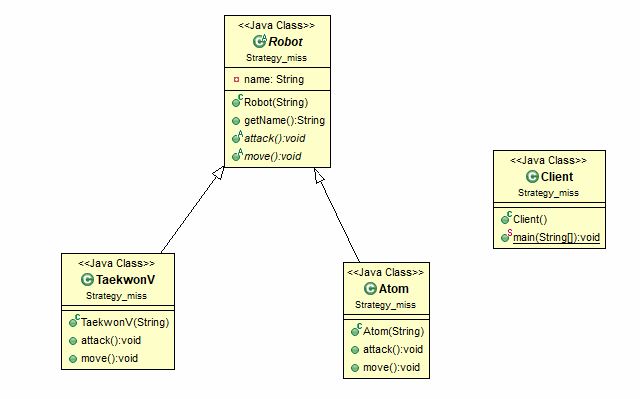
스트래티지 패턴은 구현한 클래스의 알고리즘이 변경될 경우, 모든 클래스를 바꿔야하는 어려움을 해결하기 위한 패턴이다. 즉, 클래스의 행동을 메서드(추상화 – 인터페이스)로 만들어 변화에 유연한 캡슐화 처리를 해주기 위한 패턴이다.



위의 코드는 Robot 추상 클래스를 상속받은 TaekwonV와 Atom 클래스를 나타낸다.

클래스 다이어 그램으로 나타내면 아래의 그림과 같이 그릴수 있다.



위의 클래스 다이어그램 형식으로 코드를 짜면 상위 클래스의 캡슐화로 인해 새로운 객체 로봇을 추가하기가 매우 쉬워진다. 하지만 매우 큰 결함도 함께 가지게 되는데 만약 TaekownV 또는 Atom 객체에 fly() 기능을 추가한다거나, 똑 같은 기능이 있는 새로운 Robot를 상속받는 객체가 나타난다면 어떻게 될까? 이러한 경우에서는 기존의 코드를 수정해주거나 코드의 중복이 반드시 생긴다. 만약에 미사일 코드의 내용이 바뀐다면 일일이 찾아가서 다 수정을 해줘야 하는 등 많은 번거로운 일이 발생하게 된다. 그래서 위의 코드는 캡슐화가 매우 잘 되어있지만 객체 안의 내용이 변경될 가능성이 매우 높기 때문에 치명적인 오류가 발생 하는 것 이다.

그래서 기능들도 클래스로 따로 빼주어 상황에 따라 쉽게 바뀌도록 해 주는 것이 스트레이지 패턴이다.

public interface AttackStrategy {

public void attack();

}

public interface MovingStrategy {

public void move();

}

public class FlyingStrategy implements MovingStrategy {

@Override

public void move() {

System.out.println("날아올라라");

}

}

public class WalkingStrategy implements MovingStrategy {

@Override

public void move() {

System.out.println("걷는다.");

}

}

public class PunchStrategy implements AttackStrategy {

@Override

public void attack() {

System.out.println("앞찌르기!");

}

public class MissileStrategy implements AttackStrategy {

@Override

public void attack() {

System.out.println("미사일 발사");

}

}

super(name);

}

}

public class Robot {

private String name;

private AttackStrategy attackStrategy;

private MovingStrategy movingStrategy;

public Robot(String name){

this.name = name;

}

public String getName(){

return name;

}

public void move(){

movingStrategy.move();

}

public void attack(){

attackStrategy.attack();

}

public void setMovingStrategy(MovingStrategy movingStrategy){

this.movingStrategy = movingStrategy;

}

public void setAttackStrategy(AttackStrategy attackStrategy){

this.attackStrategy = attackStrategy;

}

}

public class TaekwonV extends Robot {

public TaekwonV(String name){

super(name);

}

}

public class Atom extends Robot {

public Atom(String name){

super(name);

}

}

public class Client {

public static void main(String[] args) {

Robot t = new TaekwonV("태꿘브이");

Robot a = new Atom("아톰");

t.setMovingStrategy(new WalkingStrategy());

t.setAttackStrategy(new MissileStrategy());

a.setMovingStrategy(new FlyingStrategy());

a.setAttackStrategy(new PunchStrategy());

t.move();

t.attack();

a.move();

}

}

위의 코드처럼 스트레이지 패턴을 적용시키면 새로운 행동이 생겨도 기존의 코드에는 아무런 영향을 주지 않는다.(AttackStrategy를 상속받은 클래스만 추가하고 Main에서 사용해 주기만 하면된다.)

또한, 여러 클래스가 생성되어 적용된 상태에서 Missile 객체 내용만 바꿔줘도 해당 클래스들은 문제가 없다.(MissileStrategy만 바꿔주면 끝!)

