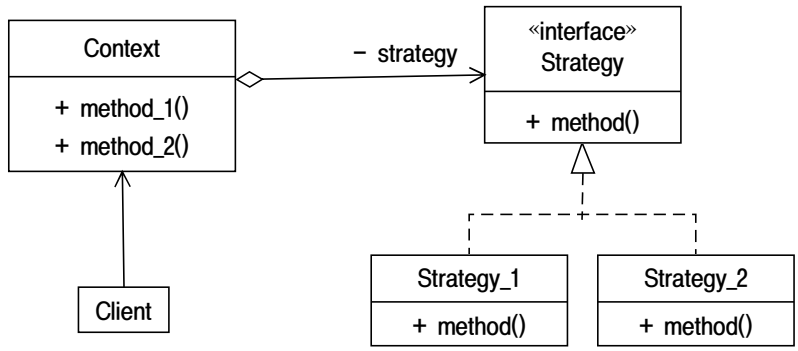
21. 디자인 패턴(design pattern)

디자인 패턴의 하나인 **전략 패턴**(strategy pattern)에 대해서 살펴본다.

- 전략 패턴(strategy pattern) : **알고리즘 교체**에 유용(알고리즘 변형이 필요한 경우에 유용)
- 전략 패턴은 알고리즘을 객체화하여 같은 문제에 **다양한 알고리즘**을 적용할 수 있다.
- 전략 패턴은 알고리즘을 사용하는 클라이언트와 독립적으로 알고리즘을 변경할 수 있다.
- 전략 패턴은 참조하는 객체가 자주 변경되는 경우에 적용할 수 있다.
- 전략 패턴은 **위임(delegation)**을 통해서 어떤 행동을 사용할지 것인지 결정한다.
- 전략 패턴은 인터페이스를 상속받은 클래스에서 알고리즘을 구현한다.

〈전략 패턴〉

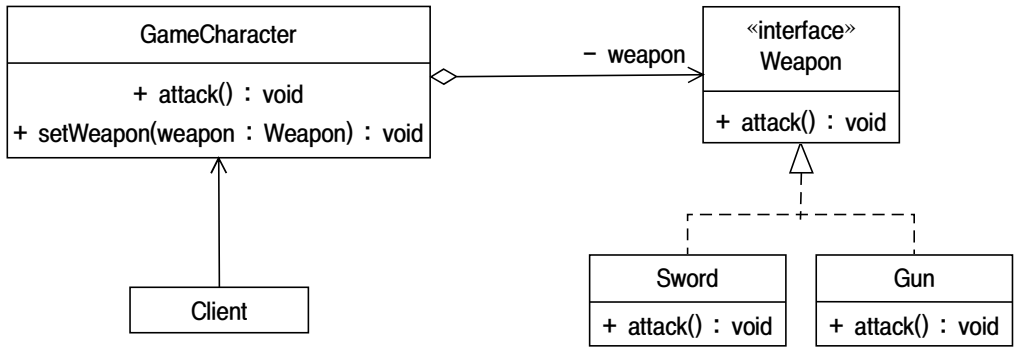


- 클래스 **Conext**는 객체를 생성하여 인터페이스 **Strategy**를 직접 참조한다.
- 여러 클래스들(Starategy_1, Starategy_2)이 인터페이스 **Strategy**를 구현하고 있다.
- 클래스 **Conext**가 인터페이스 Strategy를 참조하므로 다양한 전략(strategy)들을 가지는 여러 클래스가 있어도 코드 변경이 용이하다.

◈ **의존성 역전의 원칙(Dependency Inversion Principle, DIP)**

- 프로그래머는 **추상화**에 의존해야지, **구체화**에 의존하면 안 된다.
- 객체 참조는 부모클래스의 인터페이스에 의존해야 한다.
- 전략 패턴은 개방-폐쇄 원칙과 의존성 역전 원칙을 준수한다.
- 의존성 역전 법칙을 준수하면 객체의 결합도를 낮추고 유지보수가 편하다.

〈전략 패턴〉



↓
↓ 프로그램
↓

〈전략 패턴 프로그램〉

```
interface Weapon        //인터페이스는 선언과 구현을 분리하고, 기능을 사용할 수 있는 통로
{
  public void attack();  //메서드 선언
}
class Sword implements Weapon{ public void attack(){ System.out.println("검 공격"); } }
class Gun implements Weapon{ public void attack(){ System.out.println("총 공격"); } }
class GameCharacter{                          //각종 무기 사용 기능을 위임하는 클래스
  private Weapon weapon;                      //인터페이스 Weapon과 연관 설정
  public void setWeapon(Weapon weapon){ this.weapon=weapon; } //무기설정(의존성 주입)
  public void attack(){
    if(weapon==null) System.out.println("맨손으로 공격");
    else weapon.attack();                     //무기 사용 기능을 다른 클래스로 위임
  }
}
public class Client{
  public static void main(String args[]){
    GameCharacter gamecharacter = new GameCharacter();
    gamecharacter.attack();                   //맨손으로 공격(현재 무기가 없음)
    gamecharacter.setWeapon(new Sword());     //무기를 검으로 설정
    gamecharacter.attack();                   //검을 이용한 공격
    gamecharacter.setWeapon(new Gun());       //무기를 총으로 설정
    gamecharacter.attack();                   //총을 이용한 공격
  }
} //실행 결과
   맨손으로 공격
   검 공격
   총 공격
```