[우리 학과 취업스쿨 Day05]

소프트웨어 디자인 패턴

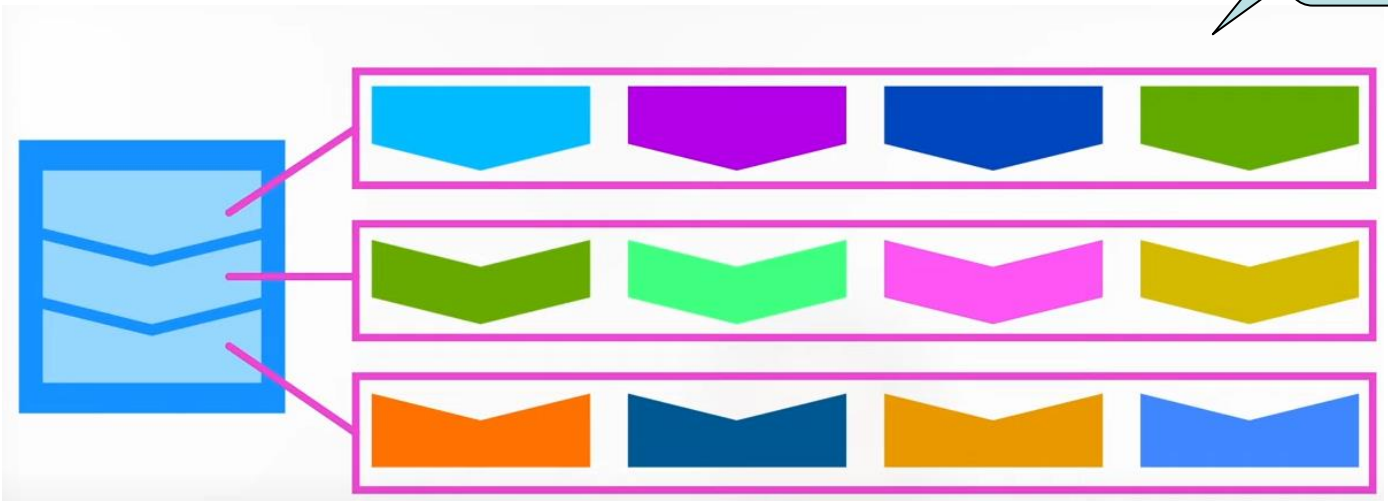
템플릿 메소드 패턴(Template Method Pattern)

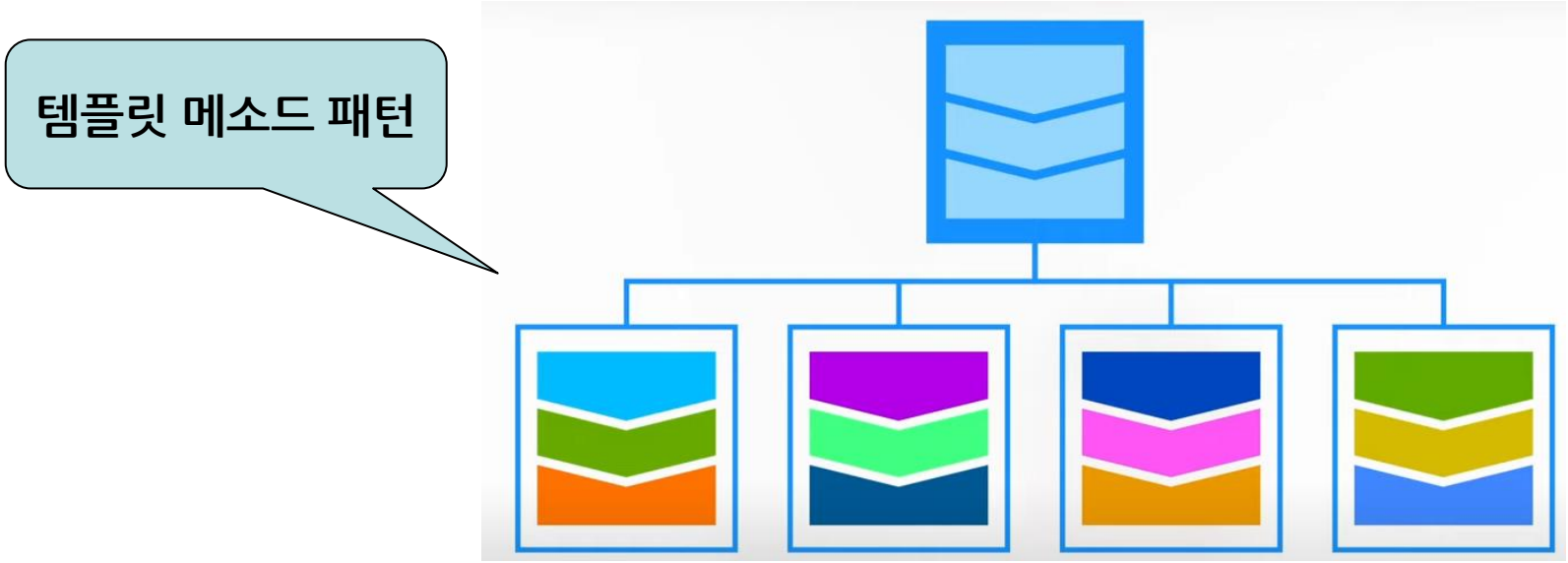
[어떤 같은 형식을 지닌 특정 작업들의 세부 방식을 다양화 하고자 할 때 사용하는 패턴]

OCP(Open-Closed Principle)

- OCP는 가장 중요한 디자인 원칙 중 하나임
  - 클래스는 확장에 대해서는 열려 있어야 하지만 코드 변경에 대해서는 닫혀 있어야 함
  - 기존 코드는 건드리지 않은 채로 확장을 통해서 새로운 행동을 간단히 추가하도록 하는 것

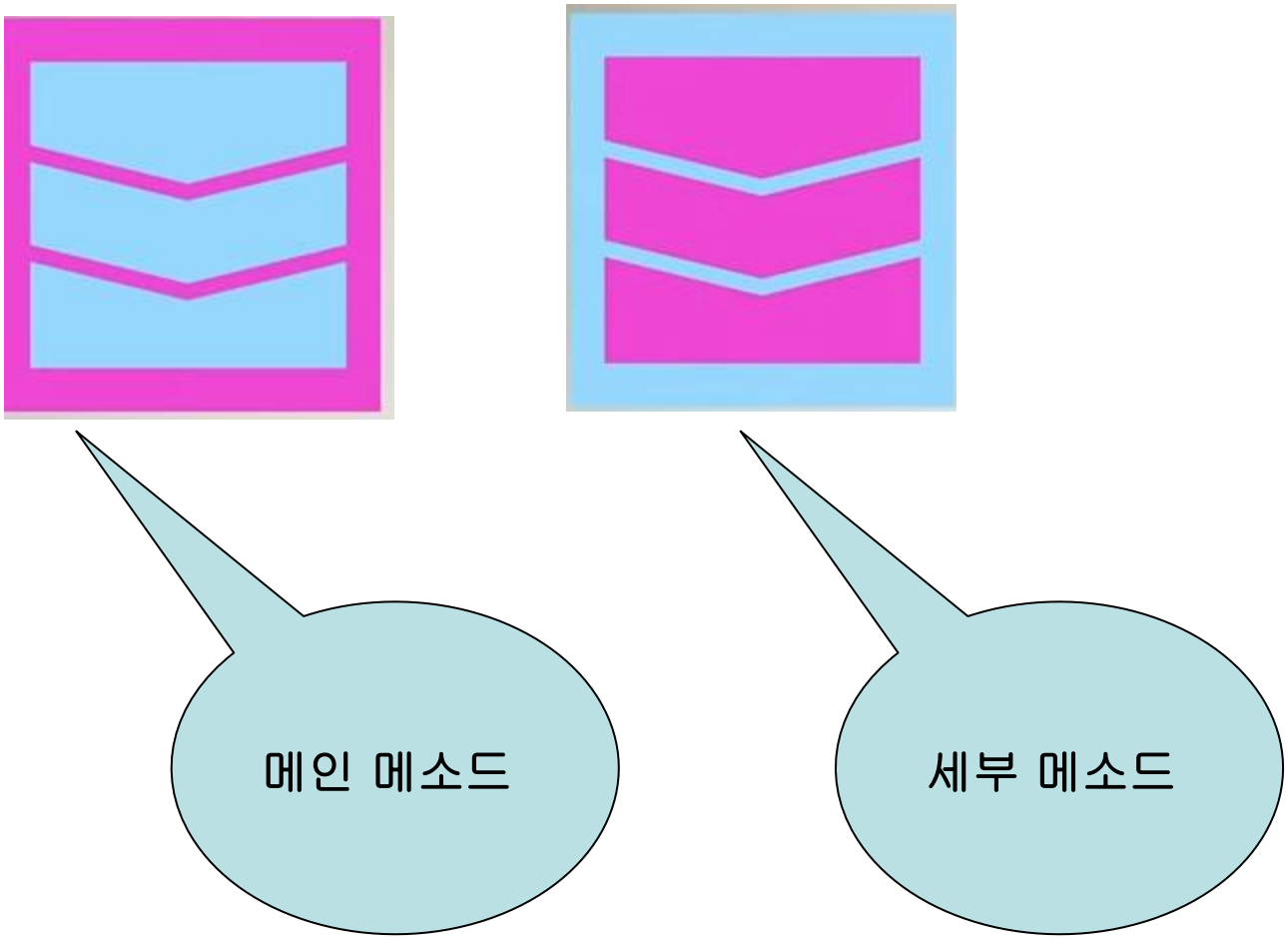
템플릿 메소드 패턴



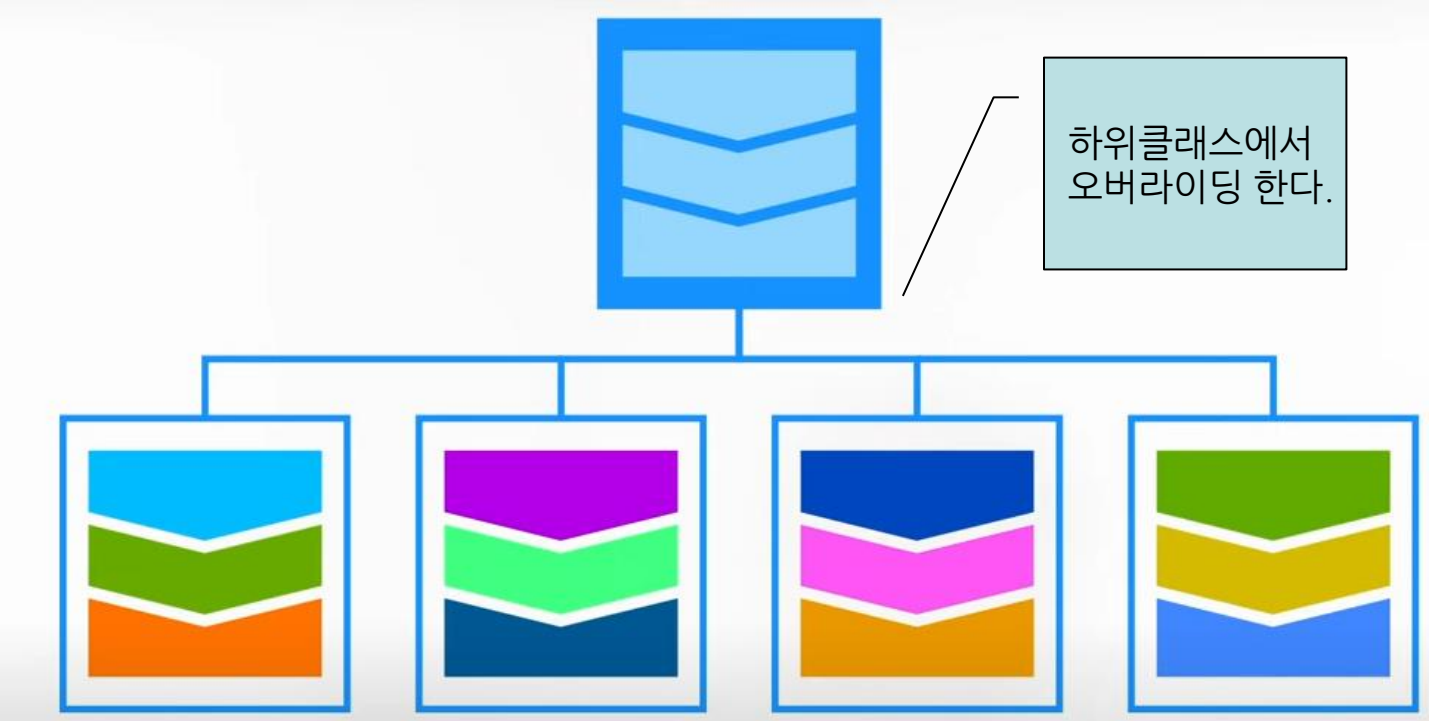


템플릿 메소드 패턴

- 템플릿 메소드에서의 상속은 일정 형식이 있다.



템플릿 메소드 패턴



예제1: 위치 정보 제공 기능 앱 개발

```
3  public abstract class MapView {
4
5      protected abstract void connectMapServer();
6
7      protected abstract void showMapOnScreen();
8
9      protected abstract void moveToCurrentLocation();
10
11     public void initMap() {
12         connectMapServer();
13         showMapOnScreen();
14         moveToCurrentLocation();
15     }
16 }
```

예제1: 위치 정보 제공 기능 앱

```java
package templatemethod;

public class NaverMapView extends MapView {

    @Override
    protected void connectMapServer() {
        System.out.println("네이버 지도 서버에 연결");
    };

    @Override
    protected void showMapOnScreen() {
        System.out.println("네이버 지도를 보여줌");
    };

    @Override
    protected void moveToCurrentLocation() {
        System.out.println("네이버 지도에서 현 좌표로 이동");
    };
}
```

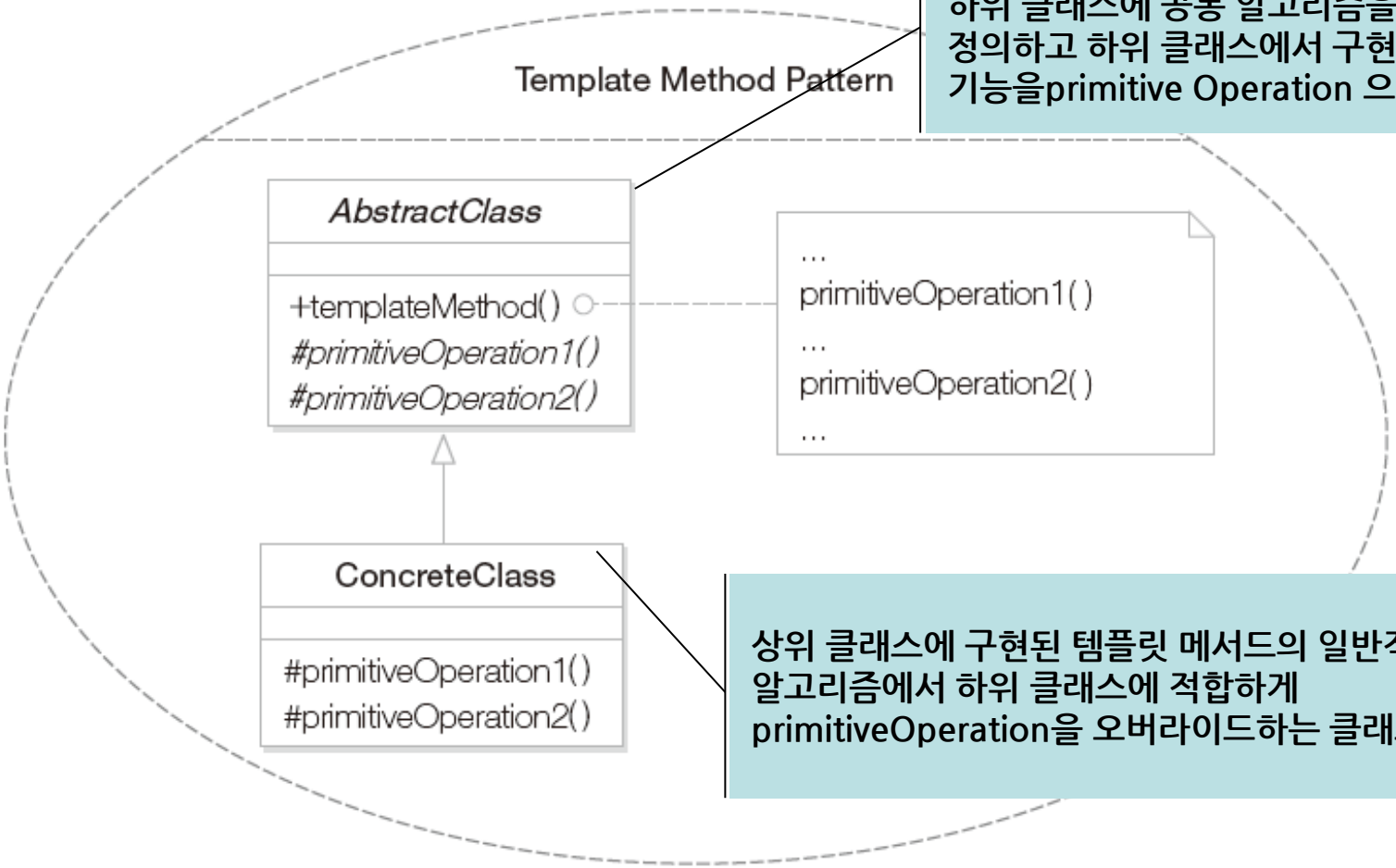
```java
package templatemethod;

public class KakaoMapView extends MapView {

    @Override
    protected void connectMapServer() {
        System.out.println("카카오 지도 서버에 연결");
    };

    @Override
    protected void showMapOnScreen() {
        System.out.println("카카오 지도를 보여줌");
    };

    @Override
    protected void moveToCurrentLocation() {
        System.out.println("카카오 지도에서 현 좌표로 이동");
    };
}
```

예제1: 위치 정보 제공 기능 앱

```
1 package templatemethod;
2
3 public class TemplateExample {
4     public static void main(String[] args) {
5         new NaverMapView().initMap();
6         new KakaoMapView().initMap();
7     }
8 }
```

템플릿 메서드 패턴

그림 11-7 템플릿 메서드 패턴의 컬레보레이션

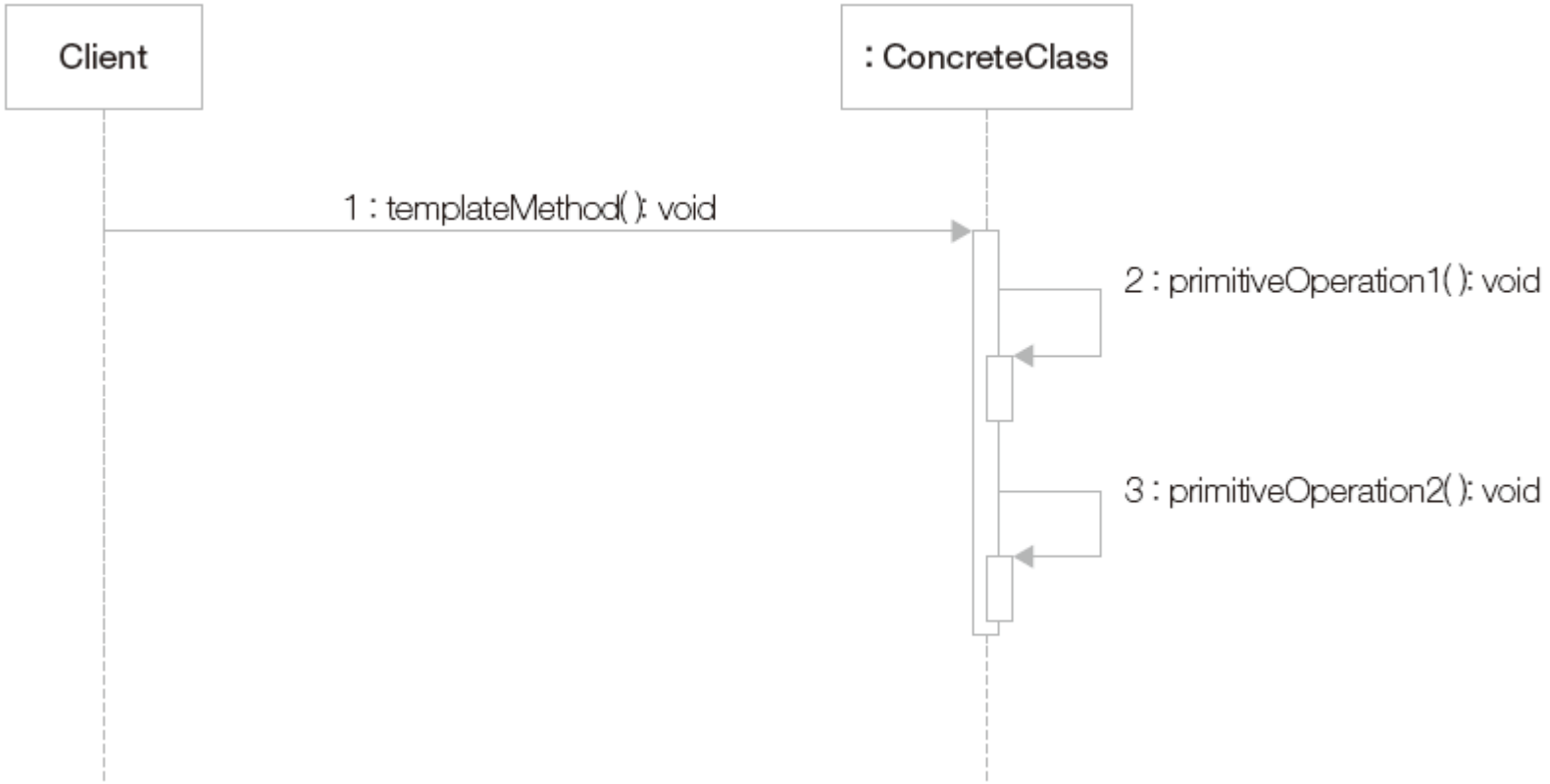


템플릿 메서드를 정의하는 클래스
하위 클래스에 공통 알고리즘을 정의하고 하위 클래스에서 구현될 기능을primitive Operation 으로 정의

상위 클래스에 구현된 템플릿 메서드의 일반적인 알고리즘에서 하위 클래스에 적합하게 primitiveOperation을 오버라이드하는 클래스

템플릿 메서드 패턴

그림 11-8 템플릿 메서드 패턴의 순차 다이어그램



예제2: 게임 프로그램 개발

- 공격하는 괴물들을 피해 보물을 찾으러 가는 탐험가 게임 프로그램 개발
  - 사람: 0부터 100까지의 에너지를 갖음
  - 괴물: 지나가는 탐험가를 공격하여 성공하면 사람의 에너지는 줄어들게 됨
    - 단 사람이 갖고 있는 에너지의 수준에 따라 괴물들이 공격할 수 없는 때도 있음
  - 괴물의 종류에 따라 공격 후 사람의 에너지가 줄어드는 정도가 다름. 만약 공격받은 사람의 에너지가 0이하가 되면 그 사람은 죽게 됨

예제2: 게임 프로그램 개발

❖ 괴물 공격 정보

괴물의 종류	공격가능 기준 (attackable)	공격 시 사람에게 주는 충격 (injury)
용	사람의 에너지<100	에너지 30 감소
드라큘라	사람의 에너지<80	에너지 10 감소

❖ 고려 사항

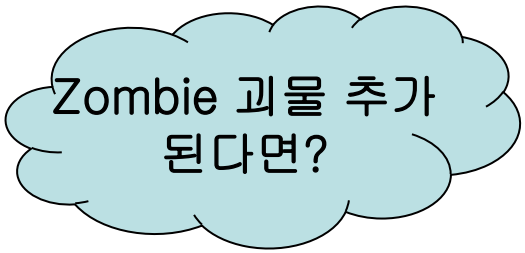
- 새로운 괴물 추가할 수 있음(Zombie)
- 괴물의 공격 가능 기준과 한번 공격으로 사람에게 줄 수 있는 충격이 변경될 수 있음
- 새로운 괴물과 효과음이 추가될 가능성이 있음

예제2: 게임 프로그램 개발(패턴 미적용)

```java
class Monster {
    private int type; // 0: Dragon, 1: Dracula

    public Monster(int type) {
        this.type = type;
    }

    public void attack(Person p) {
        if (type == 0) { // 만약 Dragon이면,
            if (p.getEnergy() < 100) // 100보다 작으면 공격
                p.getInjured(30);// 사람에 주는 충격(에너지 30감소)
        } else if (type == 1) { // 만약 Dracula면,
            if (p.getEnergy() < 80) // 80보다 작으면 공격
                p.getInjured(10);// 사람에 주는 충격(에너지 10감소)
        }
    }
}
```

```java
class Person {
    private int energy;

    public Person() {
        this(100);
    }

    public Person(int energy) {
        this.energy = energy;
    }

    public void getInjured(int injuryLevel) {
        energy -= injuryLevel;
    }

    public boolean isAlive() {
        return (energy > 0); // 에너지가 0보다 크면 살아 있는 것
    }

    public int getEnergy() {
        return energy;
    }
}
```

```java
package templatemethod;

public class TemplateMethodDemoGame {
    public static void main(String[] args) {
        Person hong = new Person(80);

        Monster dragon = new Monster(0);
        Monster dracula = new Monster(1);

        dragon.attack(hong);
        dracula.attack(hong);

        if (hong.isAlive())
            System.out.println("사람의 현재 에너지 : " + hong.getEnergy());
        else
            System.out.println("사람은 죽었다.");
    }
}
```



예제2: 게임 프로그램 개발(패턴 미적용)

```java
class Monster {
    private int type; // 0: Dragon, 1: Dracula, 2: Zombie

    public Monster(int type) {
        this.type = type;
    }

    public void attack(Person p) {
        if (type == 0) { // 만약 Dragon이면,
            if ( p.getEnergy() < 100 ) // 100보다 작으면 공격
                p.getInjured(30);
        } else if ( type == 1) { // 만약 Dracula면,
            if ( p.getEnergy() < 80 ) // 80보다 작으면 공격
                p.getInjured(10);
        } else if ( type == 2) { // 만약 Zombie면,
            if ( p.getEnergy() < 60 )
                p.getInjured(5);
        }
    }
}
```

```java
class Person {
    private int energy;

    public Person() {
        this(100);
    }

    public Person(int energy) {
        this.energy = energy;
    }

    public void getInjured(int injuryLevel) {
        energy -= injuryLevel;
    }

    public boolean isAlive() {
        return (energy > 0); // 에너지가 0보다 크면 살아 있는 것
    }

    public int getEnergy() {
        return energy;
    }
}
```

```java
package templatemethod;

public class TemplateMethodDemoGame {
    public static void main(String[] args) {
        Person hong = new Person(80);

        Monster dragon = new Monster(0);
        Monster dracula = new Monster(1);

        dragon.attack(hong);
        dracula.attack(hong);

        if (hong.isAlive())
            System.out.println("사람의 현재 에너지 : " + hong.getEnergy());
        else
            System.out.println("사람은 죽었다.");
    }
}
```

Zombie 괴물 추가 된다면?

문제 해결(?) 방법

- Monster 클래스를 추상클래스로 정의
  - attack() 메소드를 추상 메소드로 정의
- Dragon, Dracula 클래스에서 **추상 메소드 오버라이딩 하기**

```
class Dragon extends Monster {
    public void attack(Person p) {
      if (p.getEnergy() < 100) // 100보다 작으면 공격
            p.getInjured(30);// 사람에 주는 충격(에너지 30감소)
    }
}
```

```
class Dracula extends Monster {
    public void attack(Person p) {
      if (p.getEnergy() < 100) // 100보다 작으면 공격
            p.getInjured(30);// 사람에 주는 충격(에너지 30감소)
    }
}
```

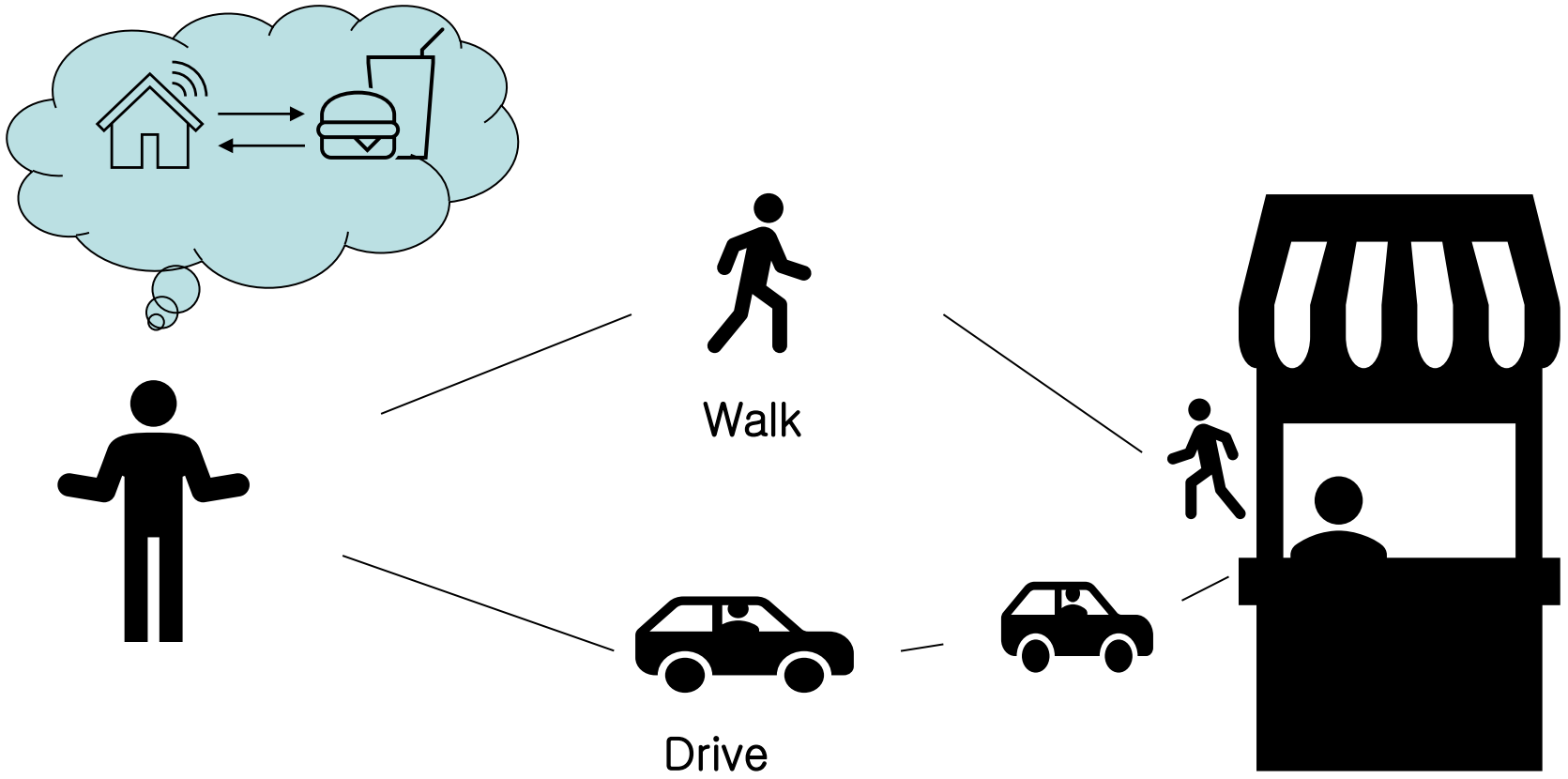
문제 해결 방법

❖ Monster 클래스

- attack() 메소드에 공통된 틀 정의

```
abstract class Monster {
    public void attack(Person p) {
      if (isAttackable(p))
           doInjuryTo(p);
    }
}
```

템플릿 메소드 패턴



템플릿 메소드 패턴을 이용한 프로그램

```
abstract class Monster {

    public void attack(Person p) {
        if (isAttackable(p))
            doInjuryTo(p);
    }

    abstract protected boolean isAttackable(Person p);

    abstract protected void doInjuryTo(Person p);

}
```

```
class Dragon extends Monster {
    protected boolean isAttackable(Person p) {
        boolean attackable = false;

        if (p.getEnergy() < 100) // 100보다 작으면 공격
            attackable = true;

        return attackable;
    }

    protected void doInjuryTo(Person p) {
        p.getInjured(30);
    }
}

class Dracula extends Monster {
    protected boolean isAttackable(Person p) {
        boolean attackable = false;

        if (p.getEnergy() < 80) // 80 보다 작으면 공격
            attackable = true;

        return attackable;
    }

    protected void doInjuryTo(Person p) {
        p.getInjured(10);
    }
}
```

```
    public static void main(String[] args) {
        Person hong = new Person(80);

        Monster dragon = new Dragon();
        Monster dracula = new Dracula();

        dragon.attack(hong);
        dracula.attack(hong);

        if (hong.isAlive())
            System.out.println("사람의 현재 에너지 : " + hong.getEnergy());
        else
            System.out.println("사람은 죽었다.");
    }
}
```

만들어 보자

- 귀신(Ghost)라는 괴물을 추가해 보자. 공격 가능 기준은 사람의 에너지가 40이상, 한 번 공격으로 사람에게 주는 충격은 30이다.

다른 디자인패턴과의 관계

- Strategy 패턴
  - 유사한 알고리즘을 한번에 교체할 수 있게 한 것이고
- Template Method 패턴
  - 알고리즘의 뼈대를 유지한 채 알고리즘의 각 단계의 행위를 하위 클래스에서 변경한다.