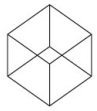
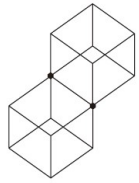
7. 스테이트패턴



JAVA 객체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



학습목표

학습목표

- UML 상태 머신 이해하기
- 상태를 캡슐화로 처리하는 방법 이해하기
- 스테이트 패턴을 통한 상태 변화의 처리 방법 이해하기
- 새로운 상태를 추가할 수 있는 처리 방법 이해하기

7.1 상태 머신 다이어그램

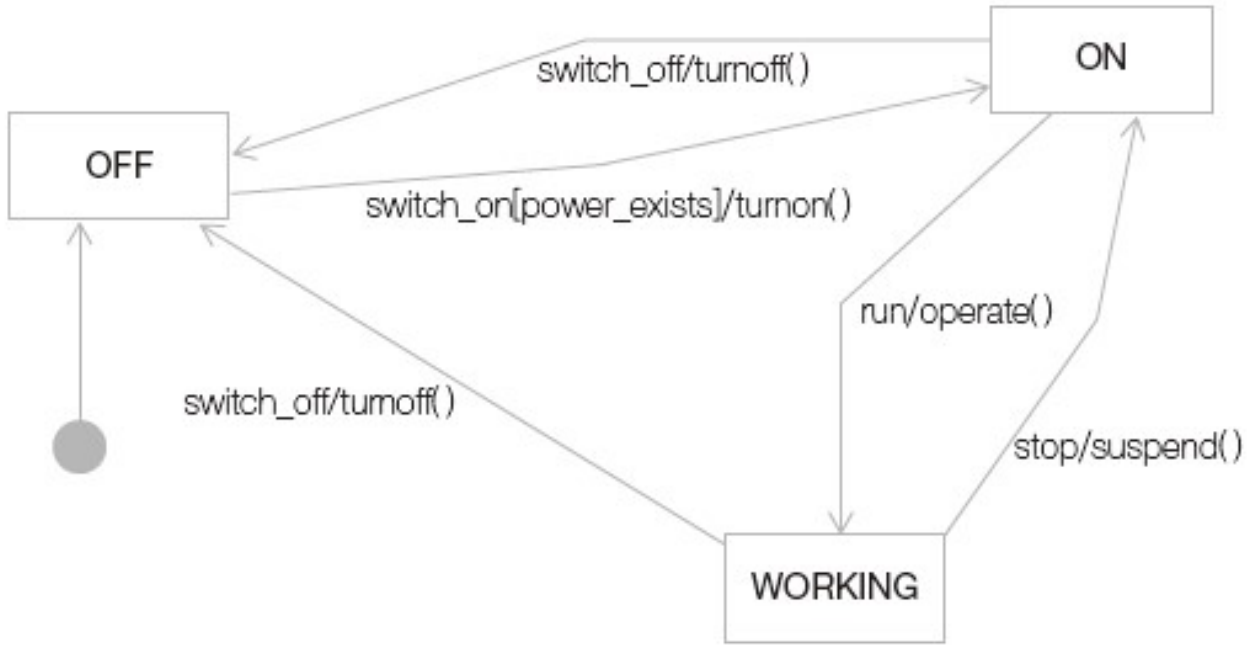
- 상태(State)
  - 실세계의 많은 시스템은 다양한 상태가 있고, 상태에 따라 다른 행위를 함
  - 예) 행복한 상태 → 돈을 빌리기쉬움
  - 예) 우울한 상태 → 돈을 빌리기 어려움

- UML에서의 상태변화를 모델링하는 도구
  - 상태머신 다이어그램
  - State machine diagram

7.1 상태 머신 다이어그램

❖ 그림 7-1. 선풍기 상태 머신 다이어그램

그림 7-1 선풍기 상태 머신 다이어그램



❖ 그림 7-1

- **단순한 기능만 실행하는 선풍기**
  - 바람세기를 선택하는 기능 없음
  - 회전할수 없음
  - 정해진 바람 세기로만 동작하는 선풍기를 의미

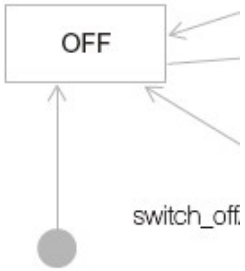
- **상태(state) : 모서리가 둥근 사각형**
- **상태전이(state transition) : 화살표**

❖ 상태란?

- **객체가 시스템에 존재하는 동안, 즉 객체의 라이프 타임 동안 객체가 가질수 있는 어떤 조건이나 상황**
- **객체는 어떤 상태에 있는 동안 어떤 액티버티를 수행하거나, 특정 이벤트가 발생하기를 기다림**

❖ 의사 상태 (pseudo state)

- 예) 시작 : initial , 종료: final , 히스토리:history, 선택: choice, 교차 : junction , 포크 : fork, 조인: join, 진입점: entry point, 진출점 : exit point
- 시작 : initial
  - 객체가 시작하는 처음 상태



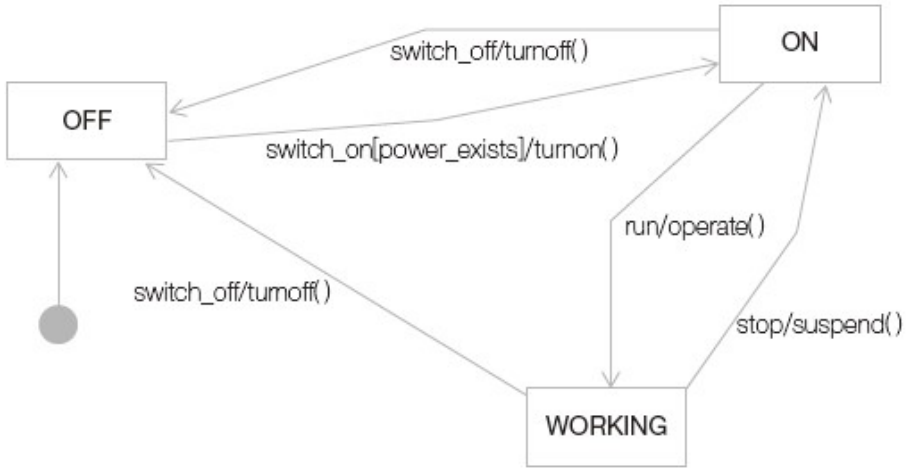
- 종료 : final
- 상태진입 : 객체의 한 상태에서 다른 상태로 이동함
  - 특정 이벤트가 발생한 후 명세된 조건을 만족한 경우에 이뤄짐
  - 이벤트(인자리스트) [조건] / 액션

그림

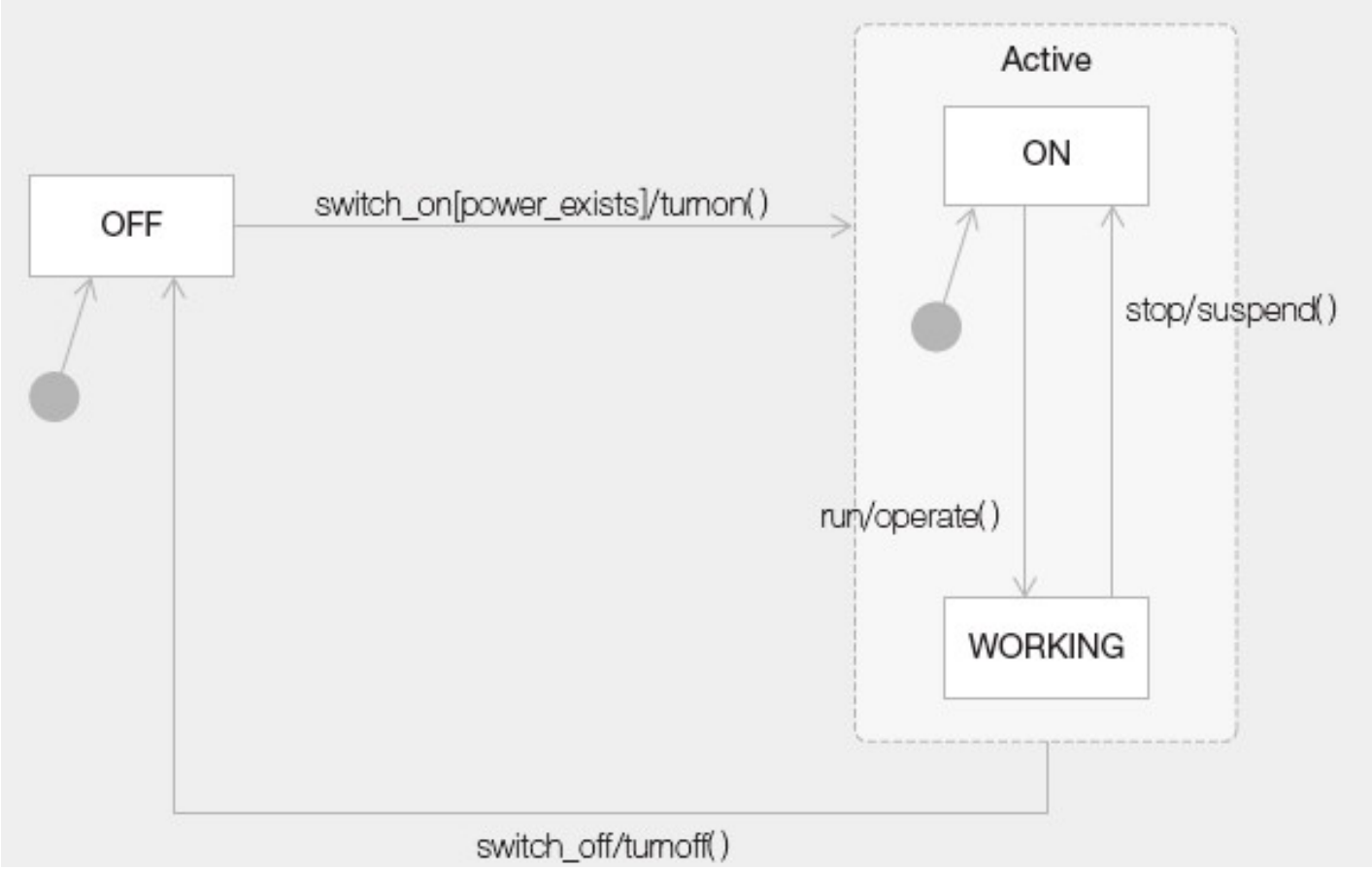
❖ 그림 7-1 해석

- 선풍기는 기본적으로 OFF 상태에서 시작한다.
- OFF 상태에서 사용자가 선풍기 스위치를 켜면 switch_on 이벤트를 발생시킨다. 이때 전원이 들어온 상태라면 (power_exists 조건) ON 상태로 진입한다. 이때 turnon 액션을 실행하게 된다.
- OFF 상태에서 사용자가 선풍기 스위치를 켜면 switch_on 이벤트를 발생시킨다. 이때 전원이 들어오지 않은 상태라면 (power_exists 조건) OFF 상태에 머무른다.
- 사용자가 ON 상태에서 동작 버튼을 누르면 run 이벤트를 발생시키고 WORKING 상태로 진입한다. 이때 operate 액션을 실행하게 된다.
- 선풍기가 ON 상태나 WORKING 상태에 머무를 때 사용자가 스위치를 끄면 switch_off 이벤트가 발생하고 이 이벤트로 인해 OFF 상태로 진입한다.

그림 7-1 선풍기 상태 머신 다이어그램



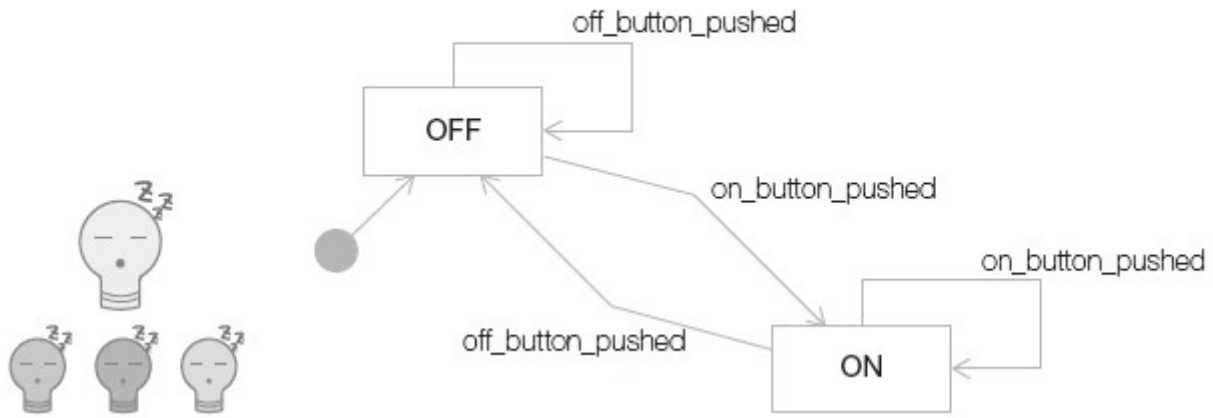
복합상태(Composite state)



- 서브 상태(Substate) : ON 상태나 WORKING
- Active 상태 : 복합상태 (composite state)
- Active에서는 ON 상태나 WORKING 중 어떤 상태에 있든 switch_off 이벤트가 발생하면 OFF 상태로 진입
  - 복합 상태는 동일한 진입으로 인한 상태 머신의 복잡성을 줄일 수 있음
- 복합 상태 안에서도 시작 상태가 존재
  - OFF 상태에서 switch_on 이벤트가 발생했을 때는 Active 복합 상태로 진입하는데, 이때 묵시적으로 ON 상태로 진입이 일어남

7.2 형광등 만들기

그림 7-2 형광등의 상태 머신 다이어그램



❖ **형광등의 각 상태를 표현하는 상수 정의**

```
private static int ON = 0;
private static int OFF = 1;
```

❖ **형광등의 상태를 저장하는 변수**

```
private int state;
```

코드 7-1

```
public class Light {
    private static int ON = 0;  // 형광등이 켜진 상태
    private static int OFF = 1; // 형광등이 꺼진 상태
    private int state; // 형광등의 현재 상태

    public Light() {
        state = OFF; // 형광등 초기상태는 꺼져 있는 상태임
    }

    public void on_button_pushed() {
        if (state == ON) {
            System.out.println("반응 없음");
        }
        else {  // 형광등이 꺼져 있을 때 On 버튼을 누르면 켜진 상태로 전환됨
            System.out.println("Light On!");
            state = ON;
        }
    }
…
```

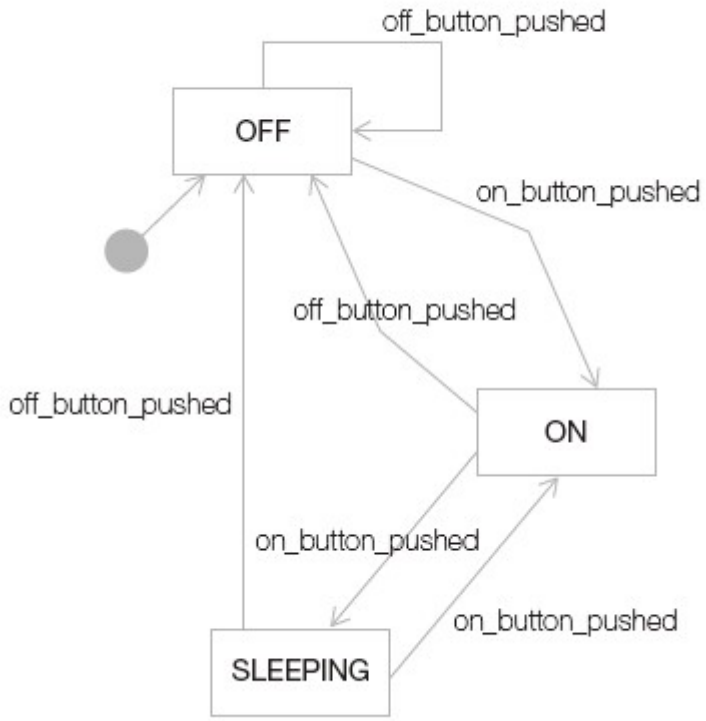
코드 7-1

```
public void off_button_pushed() {
        if (state == OFF) {
            System.out.println("반응 없음");
        }
        else {  // 형광등이 꺼져 있을 때 Off 버튼을 누르면 켜진 상태로 전환됨
            System.out.println("Light Off!");
            state = OFF;
        }
    }
}

public class Client {
    public static void main (String[] args) {
        Light light = new Light();
        light.off_button_pushed();   // 반응 없음
        light.on_button_pushed();
        light.off_button_pushed();
    }
}
```


7.3 문제점

❖ 형광등에 새로운 상태를 추가할 때, 가령 형광등에 '취침등' 상태를 추가하려면?

그림 7-3 '취침등' 상태를 추가한 상태 머신 다이어그램



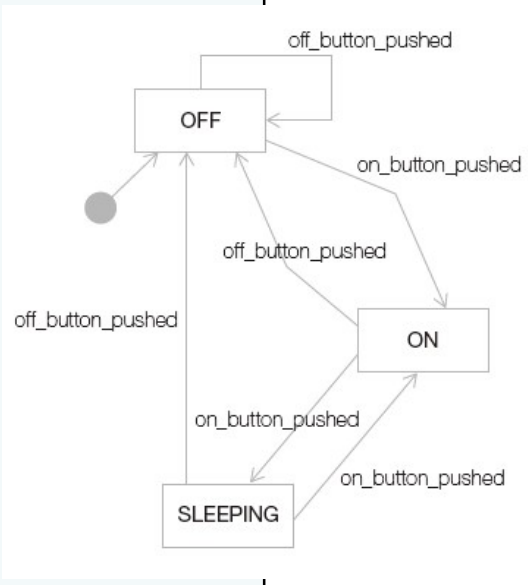
❖ 1) 취침등 상태를 나타내는 상수 추가

```
private static int SLEEPING = 2;
```


- On_button_pushed, off_button_pushed 에서 SLEEPING 변수값과 같은지 비교하고, 이에 대한 행위를 하도록 수정
- 예) On_button_pushed()

코드 7-1

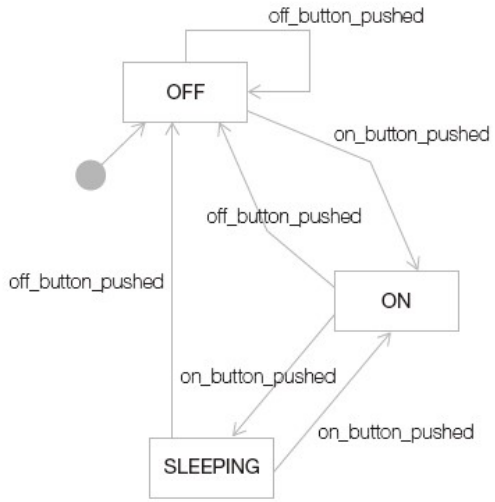
```
public void on_button_pushed() {
    if (state == ON) {
        System.out.println("취침등 상태");
        state = SLEEPING;
    }
    else if (state == SLEEPING) {  // 형광등이 취침등 상태로 있는 경우
        System.out.println("Light On!");  // On 버튼을 누르면 켜진 상태로 전환됨
        state = ON;
    }
    else { // 상태가 꺼져 있는 경우에 On 버튼을 누르면 켜진 상태로 전환됨
        System.out.println("Light On!");
        state = ON;
    }
}
```



```
public class Light {
private static int ON = 0;
private static int OFF = 1;
private static int SLEEPING = 2;
private int state;
public Light() {
state = OFF; // 초기 상태는 형광등이 꺼져 있는 상태
}
public void off_button_pushed() {
    if (state = = OFF)
        System.out.println("반응 없음");
    else if (state = = SLEEPING) {
        System.out.println("Light OFF!");
        state = OFF;
    }
   else {
        System.out.println("Light Off!");
       state = OFF;
   }
 }
}
```

```
public void on_button_pushed() {
   if (state = = ON) {
      System.out.println("취침등 상태");
      state = SLEEPING;
   }
else if (state = = SLEEPING) {
     System.out.println("Light On!"); // On 버튼을 누르면 켜진 상태로 전환됨
     state = ON;
}
else {
      System.out.println("Light On!");
      state = ON;
  }
}
```

그림 7-3 '취침등' 상태를 추가한 상태 머신 다이어그램



❖ 수정에 대한 평가

- If 문 안에 반복적이고 다양한 내용으로 구성
- 시스템의 상태변화를 파악하기에 용이하지 않음
- 새로운 상태가 추가되는 경우, 상태 변화를 초래하는 모든 메소드에 이를 반영하기 위한 코드를 수정해야 함

7.4 해결책

❖ 상태를 캡슐화

- **무엇이 변하는지 찾아야 함**
- **변하는 부분을 찾아서 캡슐화하는 것이 중요**
- **목표**
  - 현재 시스템이 어떤 상태에 있는지와 상관없이 구성하고, 상태변화에도 독립적이도록 코드를 수정하는 것
- **상태를 클래스로 분리해 캡슐화함**
- **상태에 의존적인 행위들도 상태 클래스에 같이 두어, 특정 상태에 대한 행위를 구현하도록 바꿈**
- **➔ 상태에 따른 행위가 각 클래스에 국지화되어 이해하고 수정하기 쉬움.**

❖ 그림 7-4

- 5장의 스트래티지 패턴과 구조가 비슷(똑같음)
- Light Class는 추상화된 State Interface만 참조
- 현재 어떤 상태에 있는지 무관하게 코드 작성 가능

그림 7-4 스테이트 패턴으로 구현한 형광등 상태 머신 다이어그램

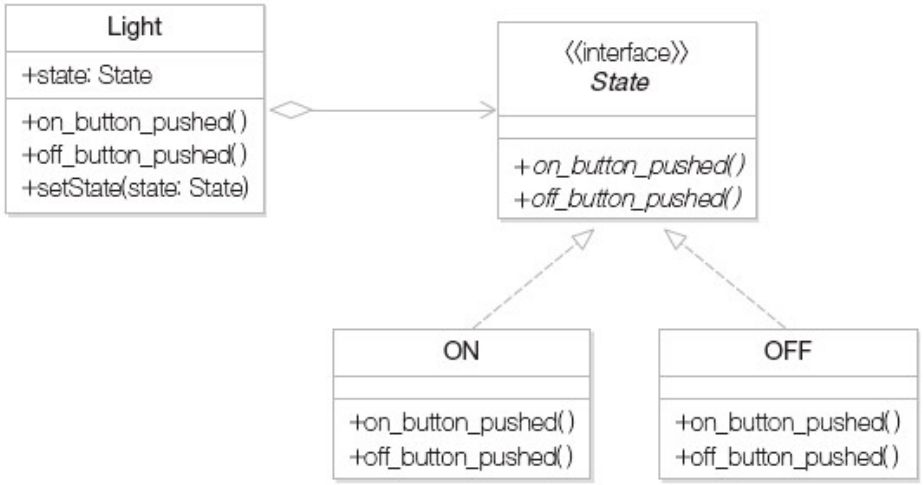
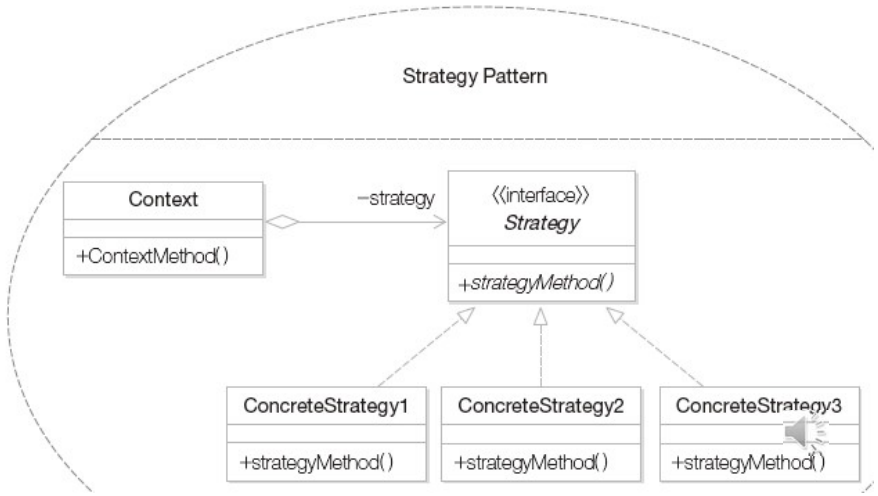


그림 5-6 스트래티지 패턴 컬레보레이션



❖ 코드 7-2
- Interface State

❖ 코드 7-3
- Class ON
- Class OFF

❖ 코드 7-4
- If 문이나 switch 문을 사용해 상태 변화를 나타낼 필요가 없음

코드 7-2

```
interface State {
    public void on_button_pushed(Light light);
    public void off_button_pushed(Light light);
}
```

코드 7-3

```
public class ON implements State {
    public void on_button_pushed(Light light) {
        System.out.println("반응 없음");
    }

    public void off_button_pushed(Light light) {
        System.out.println("Light Off!");
        light.setState(new OFF(light));
    }
}
```

코드 7-4

```
Public class OFF implements State {
    public void on_button_pushed(Light light) {
        System.out.println("Light On!");
        light.setState(new Oon(light));
    }
    public void off_button_pushed(Light light) {
        System.out.println("반응 없음");
    }
}
```

❖ 코드 7-5

- Light Class
- 구체적인 상태를 나타내는 객체를 참조하지 않음
- Light Class는 시스템이 어느 상태에 있는지와 무관
- 상태가 새로운 상태로 교체되더라도 Ligth Class는 전혀 영향을 받지 않음
- 문제점
  - 상태 변화가 생길 때마다 새로운 상태 객체를 생성하므로 메모리 낭비와 성능 저하를 가져올수 있음
  - 상태 객체는 한번만 생성해도 좋음
  - 객체를 하나만 만들수 있는 방법인 싱글톤 패턴 적용 필요

코드 7-5

```
Public class Light {
    private State state;

    public Light() {
        state = new OFF();
    }

    public void setState(State state) {
        this.state = state;
    }

    public void on_button_pushed() {
        state.on_button_pushed(this);
    }

    public void off_button_pushed() {
        state.off_button_pushed(this);
    }
}
```

코드 7-6

```
public class ON implements State {
    private static ON on = new ON(); // ON 클래스의 인스턴스로 초기화됨
    private ON() {  }

    public static ON getInstance() { // 초기화된 ON 클래스의 인스턴스를 반환함
        return on;
    }

    public void on_button_pushed(Light light) {   // ON 상태일때 On 버튼을 눌러도
변화 없음
        System.out.println("반응 없음");
    }

   public void off_button_pushed(Light light) {
        light.setState(OFF.getInstance());
        System.out.println("Light off!");
    }
}
```

❖ 코드 7-6, 7-7

- **싱글턴 패턴으로 ON과 OFF 클래스를 변경함**

코드 7-7

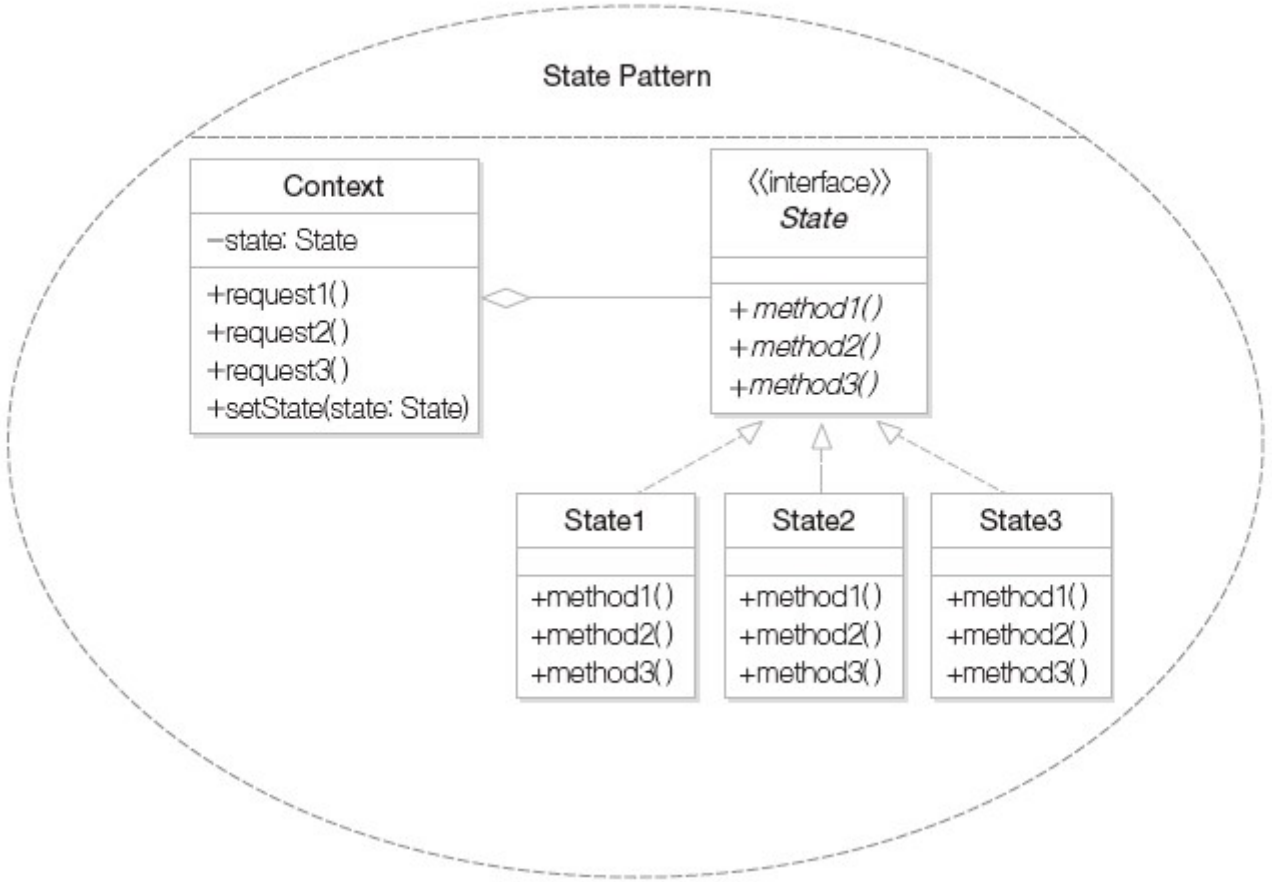
```
public class OFF  implements State {
    private static OFF off = new OFF(); // OFF 클래스의 인스턴스로 초기화됨
    private OFF() {  }

    public static OFF getInstance() { // 초기화된 OFF 클래스의 인스턴스를 반환함
        return off;
    }

    public void on_button_pushed(Light light) {   // Off 상태일때 On 버튼을 누르면 On
상태임
        light.setState(On.getInstance());
        System.out.println("Light On!");
    }

   public void off_button_pushed(Light light) { // Off 상태일때 Off 버튼을 눌러도 변화
없음
        System.out.println("반응 없음");
    }
}
```


7.5 스테이트 패턴

❖ 스테이트 패턴의 적용상황

- **일을 수행할 때의 상태에 따라 상태 하나하나가 어떤 상태인지 검사해, 일을 다르게 수행할 필요가 있음**
- **조건식이 필요할수 있는 문제 ……**
- **➔ 스테이트 패턴**
  - 어떤 행위를 수행할 때, 상태에 행위를 수행하도록 위임함
  - 시스템의 상태를 클래스로 분리해 표현
  - 각 클래스에서 수행하는 행위를 구현함
  - 이러한 상태들을 외부로부터 캡슐화하기 위해 인터페이스를 만들어, 시스템의 각 상태를 나타내는 클래스로 하여금 실체화하게 함

그림 7-5 스테이트 패턴의 컬레보레이션



❖ 그림 7-6. 스테이트 패턴을 형광등 예제에 적용한 예

그림 7-6 스테이트 패턴을 형광등 예제에 적용한 경우