

7. 스테이트패턴



JAVA 개체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



학습목표

학습목표

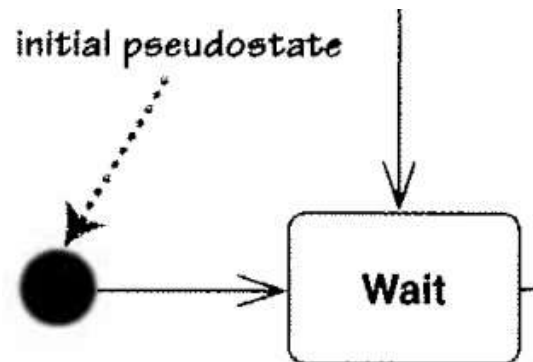
- UML 상태 머신 이해하기
- 상태를 캡슐화로 처리하는 방법 이해하기
- 스테이트 패턴을 통한 상태 변화의 처리 방법 이해하기
- 새로운 상태를 추가할 수 있는 처리 방법 이해하기

UML state diagrams

- ❖ 상태 다이어그램: 객체의 생애 동안 데이터와 행동을 나타냄
 - 상태 집합 (초기 시작 상태 포함)
 - 상태 간 전이(transition)
 - 전체 다이어그램은 해당 객체의 관점에서 그려짐
- ❖ 유한 상태 기계(DFA, NFA, PDA 등)와 유사
- ❖ 상태 다이어그램에 가장 적합한 객체는 무엇인가?
 - 크고 복잡한 객체로 긴 생애를 가진 것
 - 도메인("모델") 객체
 - 시스템의 모든 클래스에 대해 상태 다이어그램을 만드는 것은 유용하지 않음

States

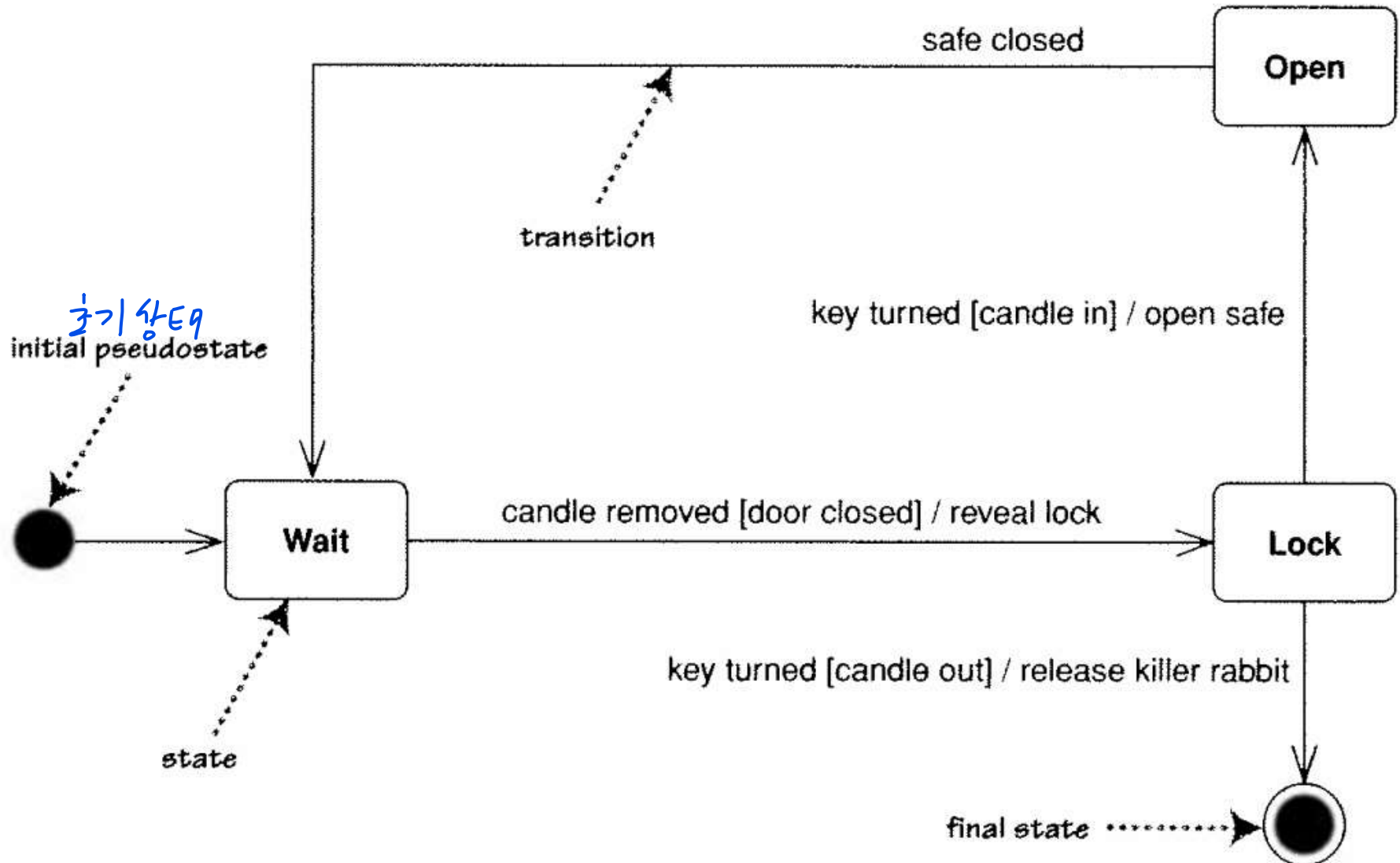
- ❖ 상태: 객체의 데이터에 대한 개념적 설명
 - 객체의 필드 값으로 표현
- ❖ 전체 다이어그램은 중심 객체의 관점에서 그려지며,
 - 이 객체가 볼 수 있고 영향을 미칠 수 있는 상태/개념만 포함
 - 필드의 모든 가능한 값을 포함하지 않고, 개념적으로 다른 값만 포함



Killer rabbit의 탈옥을 막기



State diagram example



Killer rabbit 탈옥 막기 state diagram

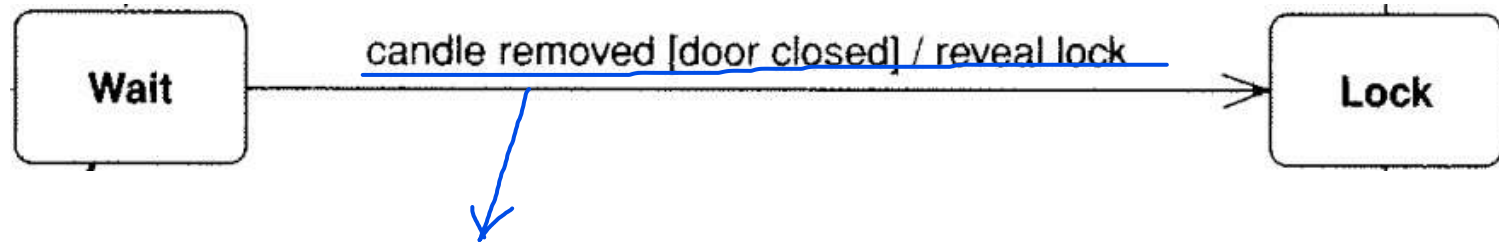
- ❖ 초기 상태 (Initial Pseudostate): 다이어그램의 가장 왼쪽에 있는 검은 원은 다이어그램의 시작 상태를 나타냄. 여기서 시스템은 " 대기 (Wait) " 상태로 전환
- ❖ 대기 상태 (Wait): 시스템은 대기 상태에 있으며, 여기서 특정 조건이 충족될 때 다른 상태로 전환됨
 - "촛불이 제거되고 문이 닫힌 경우" -> "잠금(Lock)" 상태로 전환."

Killer rabbit 탈옥 막기 state diagram

- ❖ 잠금 상태 (Lock): 금고가 잠금 상태에 있으며, 특정 조건이 충족되면 상태가 변함
 - "열쇠가 돌려지고 촛불이 안에 있는 경우" -> "금고를 열고(Open)" 상태로 전환.
 - "열쇠가 돌려지고 촛불이 밖에 있을 때" -> "킬러 토끼를 방출"이라는 트리거와 함께 시스템은 최종 상태로 감
- ❖ 열림 상태 (Open): 금고가 열려 있는 상태로, 여기서 금고는 다시 닫힐 수 있으며, 이 경우 "대기(Wait)" 상태로 되돌아감
- ❖ 최종 상태 (Final State): 킬러 토끼가 방출된 후 시스템이 이 최종 상태로 전환됨. 이 상태는 종료를 의미하며, 더 이상 상태 전환이 없음

Transitions

- ❖ 전이(transition): 한 상태에서 다른 상태로의 이동

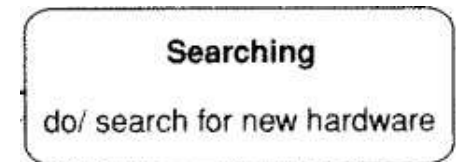


- 시그니처 [가드] / 활동

- 시그니처(signature) : 상태 변화를 유발하는 이벤트
- 가드(guard): 참이어야 하는 boolean 조건
- 활동(activity): 전이 중에 실행되는 행동 (선택 사항)

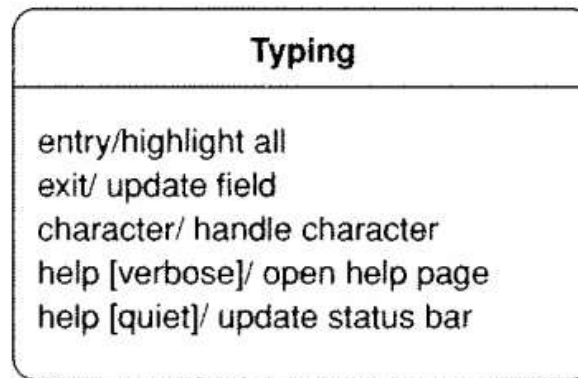
- ❖ 전이는 상호 배타적(mutually exclusive)이어야 하며(결정론적),

- 이벤트에 대해 어떤 전이를 선택해야 하는지가 명확해야 함
- 대부분의 전이는 즉각적이며, "수행(do)" 활동을 제외함



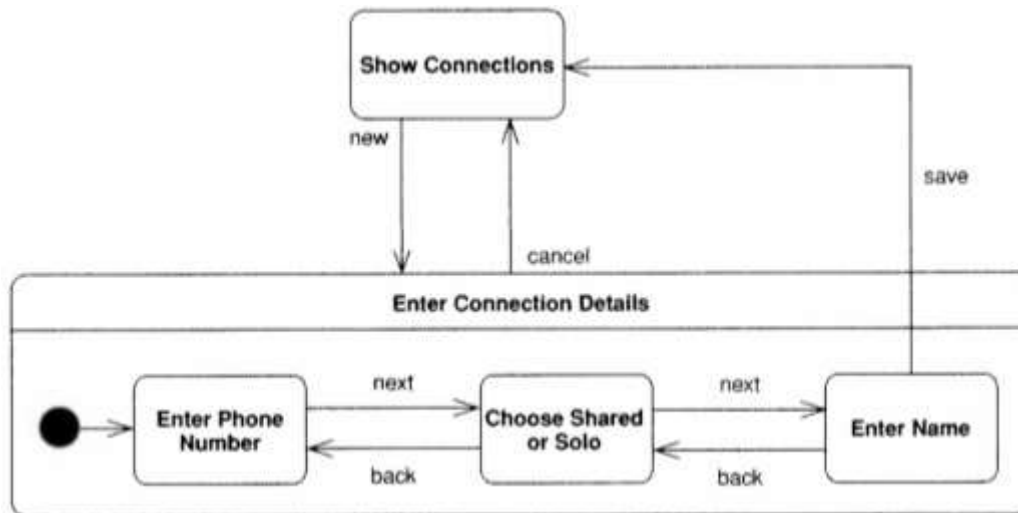
Internal activities

- ❖ 내부 활동(internal activity): 중심 객체(central object)가 스스로 수행하는 행동
 - 때때로 자기 전이(self-transitions)로 그려지며, (같은 상태에 머무는 이벤트)
- ❖ entry/exit 활동
 - 해당 상태에 들어가거나 나오는 이유



Super/substates

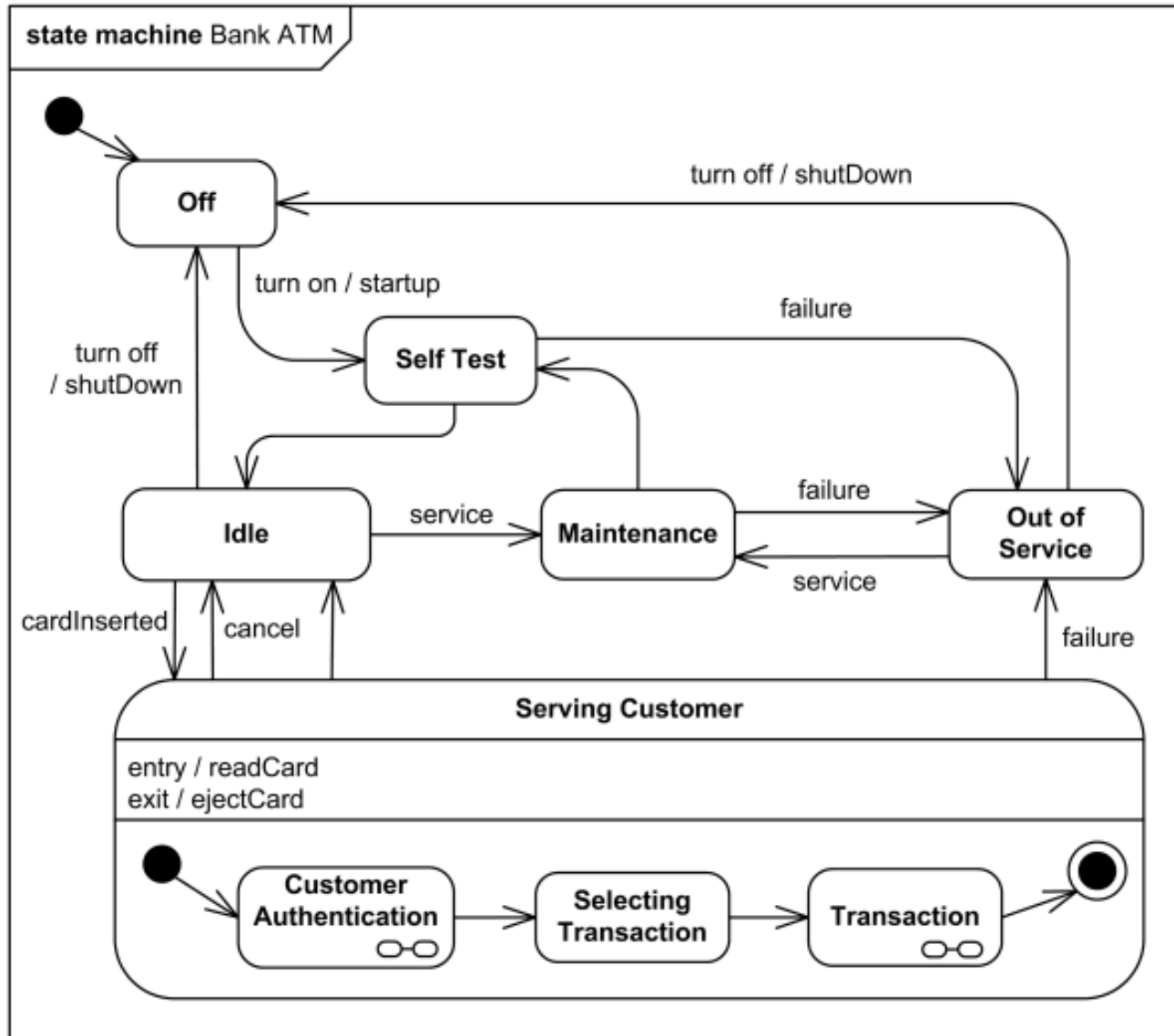
- ❖ 복잡한 상태의 경우, 그 안에 하위 상태(substate)를 포함할 수 있음
 - 더 큰 상태 안에 중첩된 둥근 사각형으로 그립니다.
- ❖ 주의: 이 기능을 과도하게 사용하지 않는 것이 좋음
 - 하나의 상태 안에서 하위 상태와 별개의 상태를 혼동하기 쉬운 점을 염두에 두어야 함



*출처 : Martin Fowler. (2003). UML Distilled, Addison-Wesley Professional
Slides created by Marty Stepp, <http://www.cs.washington.edu/403/>

State diagram example

ATM software states at a bank



상태 다이어그램 예제

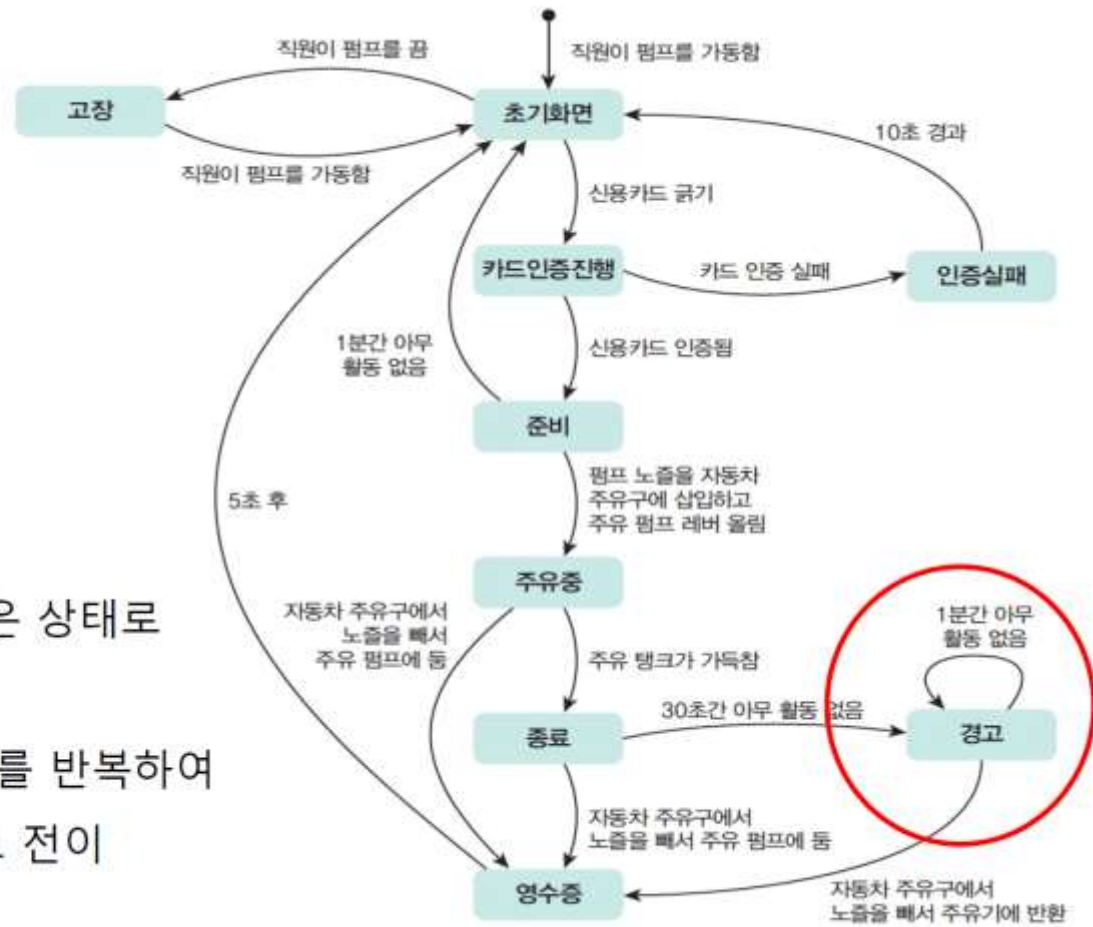


그림 6.12 주유소 펌프 상태 다이어그램

❖ 하나의 상태에서 같은 상태로 전이 가능

- 예) 자신의 상태를 반복하여 루프모양으로 전이

상태 다이어그램 예제(cont')

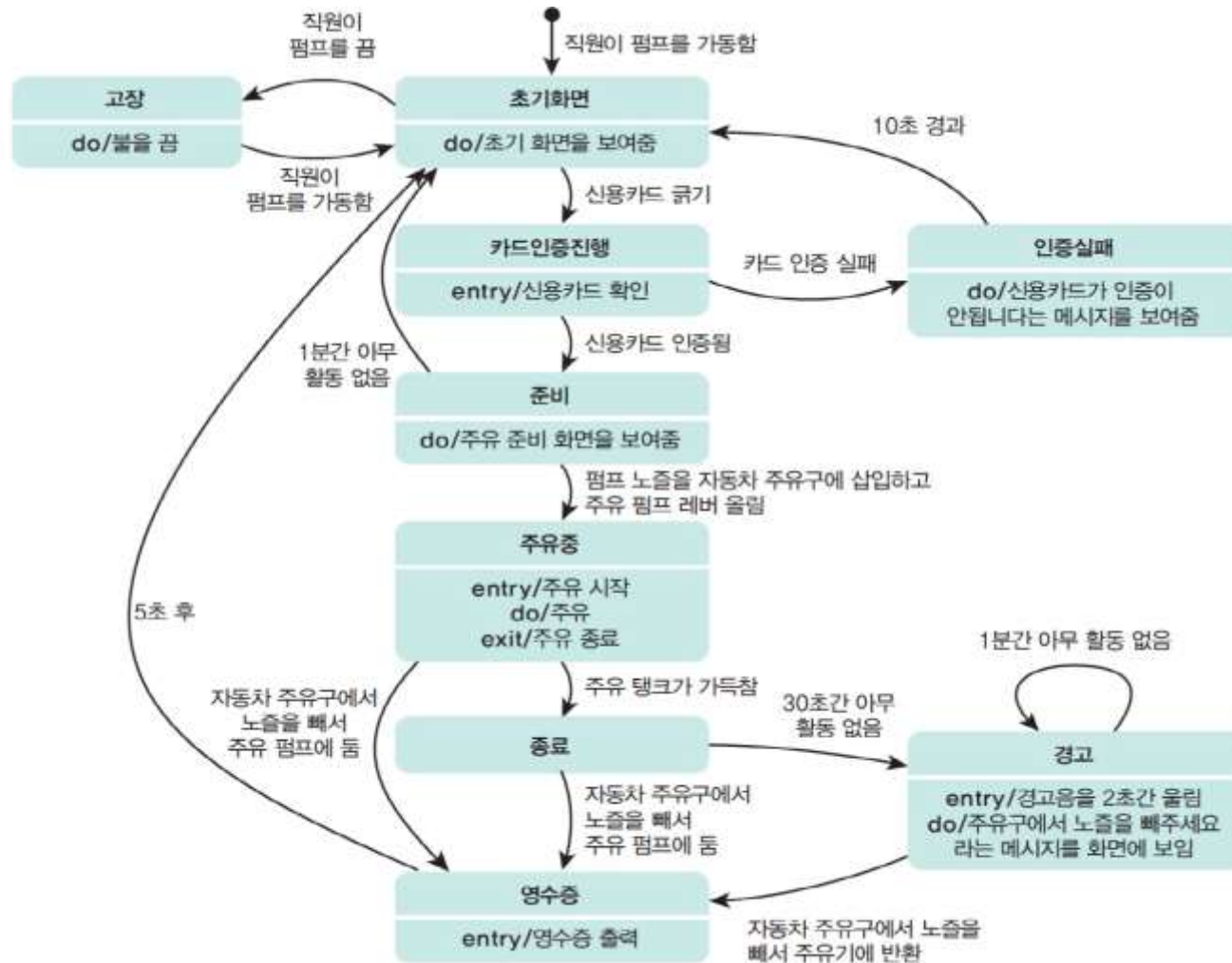
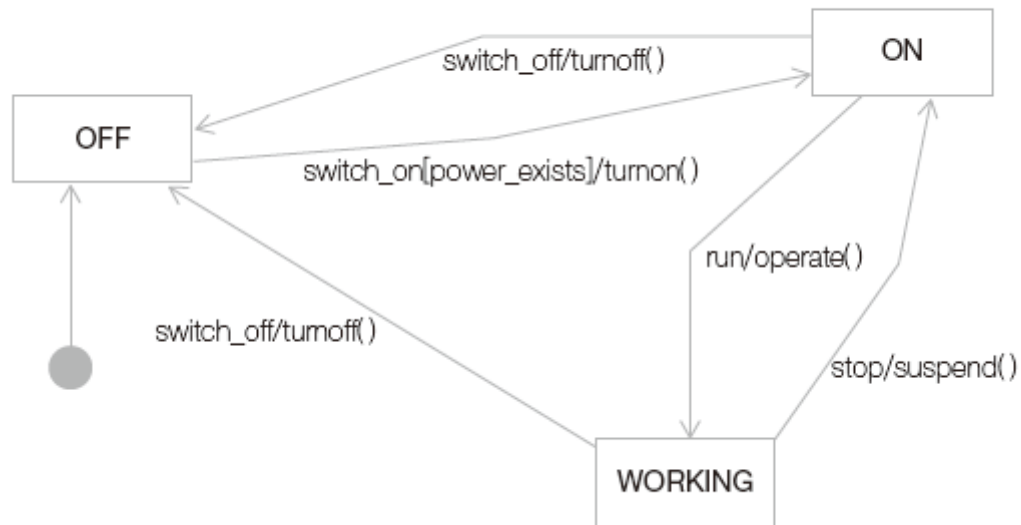


그림 6.13 액션을 추가한 주유소 펌프 상태 다이어그램

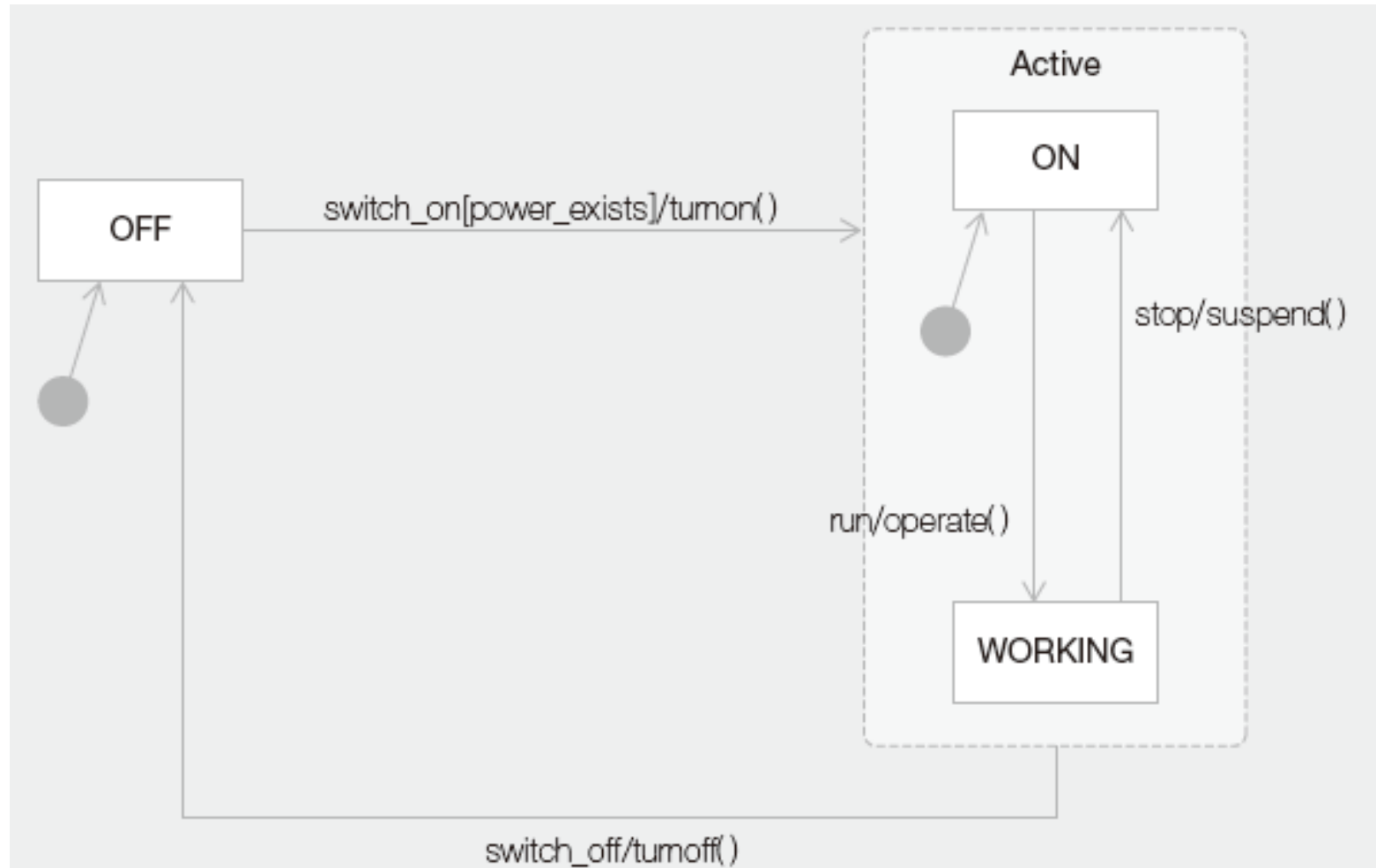
7.1 상태 머신 다이어그램

그림 7-1 선풍기 상태 머신 다이어그램



-
- ❖ 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.

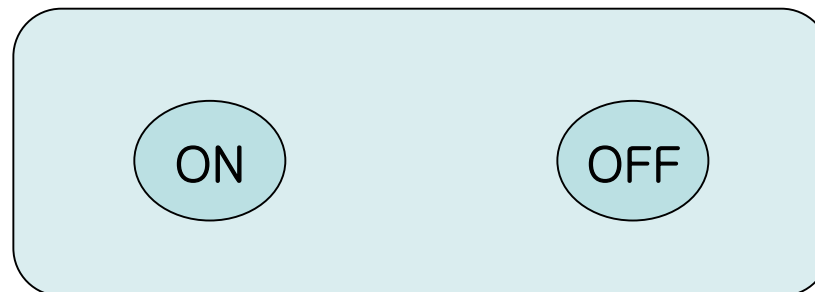
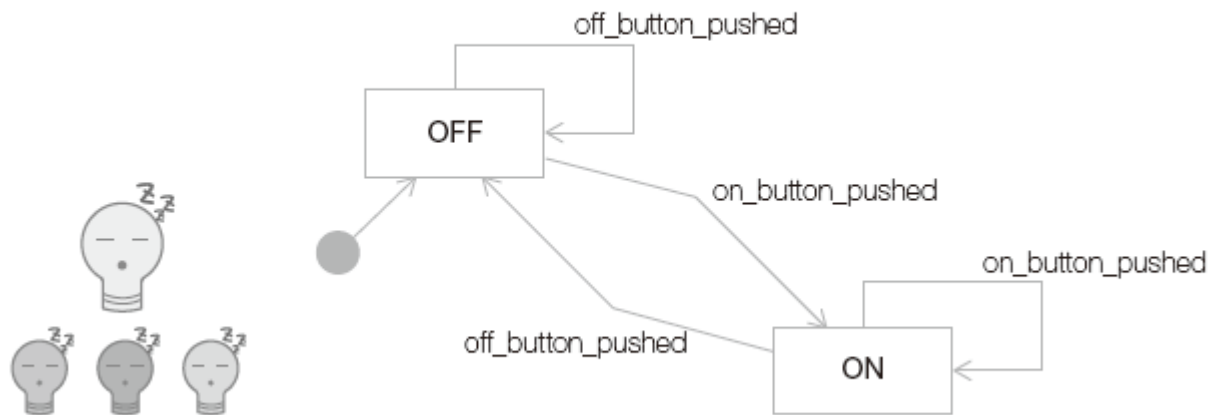
Composite state



-
- ❖ Active에서는 ON 상태나 WORKING 중 어떤 상태에 있든 switch_off 이벤트가 발생하면 OFF 상태로 진입한다. 이와 같이 복합 상태는 동일한 진입으로 인한 상태 머신의 복잡성을 줄일 수 있다 .
 - ❖ 또한 여기에서 눈여겨볼 하나의 사실은 복합 상태 안에서도 시작 상태가 존재한다는 점이다. OFF 상태에서 switch_on 이벤트가 발생했을 때는 Active 복합 상태로 진입하는데, 이때 묵시적으로 ON 상태로 진입이 일어난다.

7.2 형광등 만들기

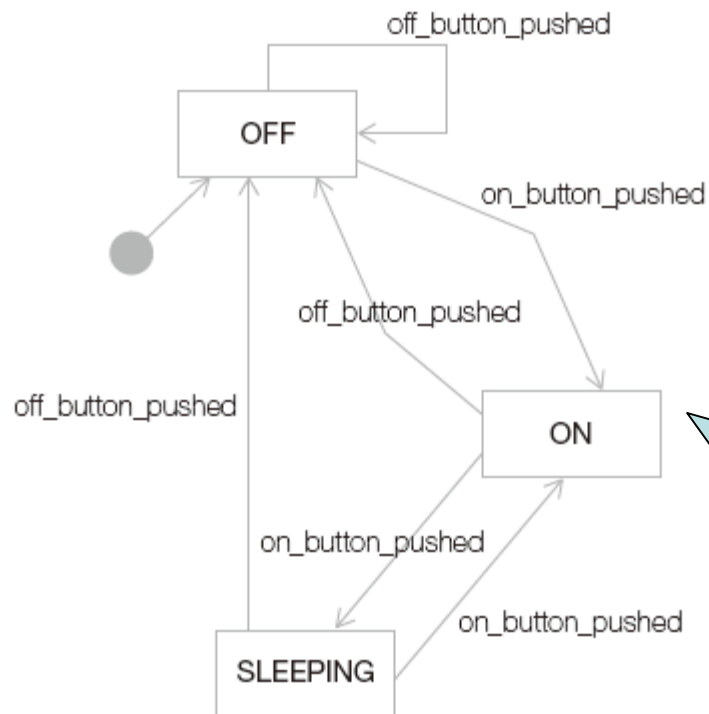
그림 7-2 형광등의 상태 머신 다이어그램



7.3 문제점

- ❖ 형광등에 새로운 상태를 추가할 때, 가령 형광등에 ‘취침등’ 상태를 추가하려면?

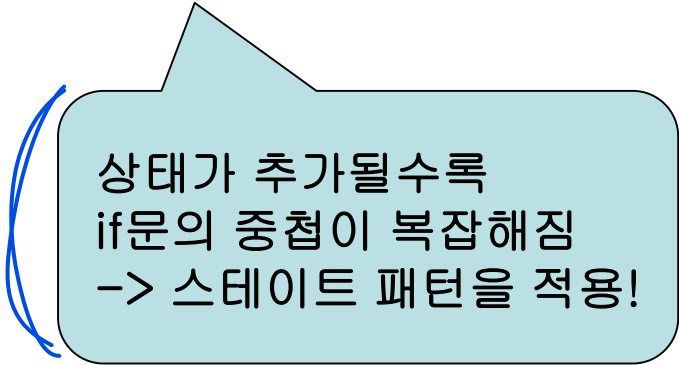
그림 7-3 ‘취침등’ 상태를 추가한 상태 머신 다이어그램



이미 ON인 상태에서
한 번 더 ON을 누르면
SLEEPING으로 전환

```
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("취침 등 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;
    }
}
```

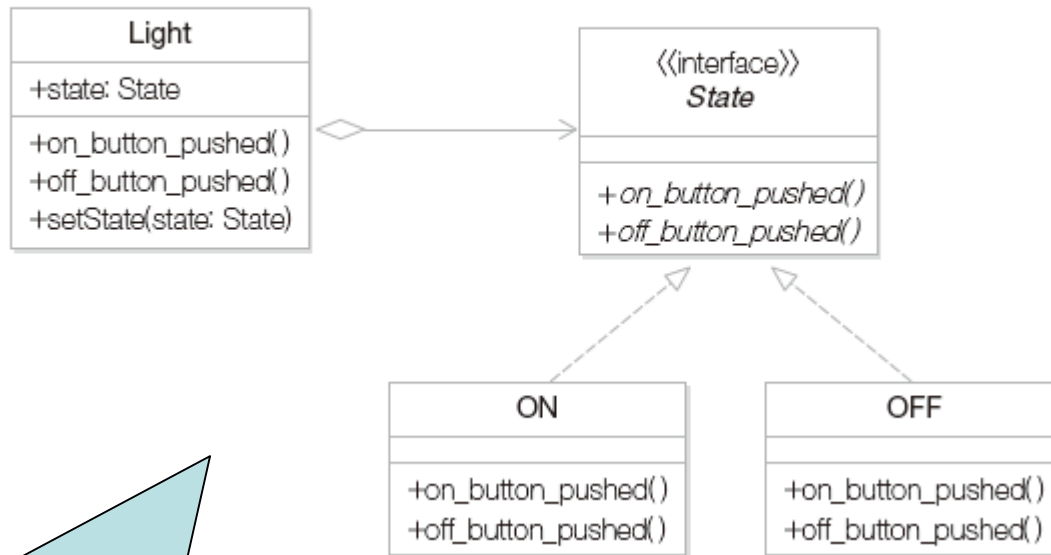


상태가 추가될수록
if문의 중첩이 복잡해짐
→ 스테이트 패턴을 적용!

7.4 해결책

❖ 상태를 캡슐화

그림 7-4 스테이트 패턴으로 구현한 형광등 상태 머신 다이어그램



스트래티지 패턴과 비슷한 형태이나,
스테이트 패턴은 컨텍스트의 '상태'가
내부에 있다는 차이점이 있음

코드

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

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));
    }
}
```

```
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);
    }
}
```

그림 7-5 스테이트 패턴의 컬레보레이션

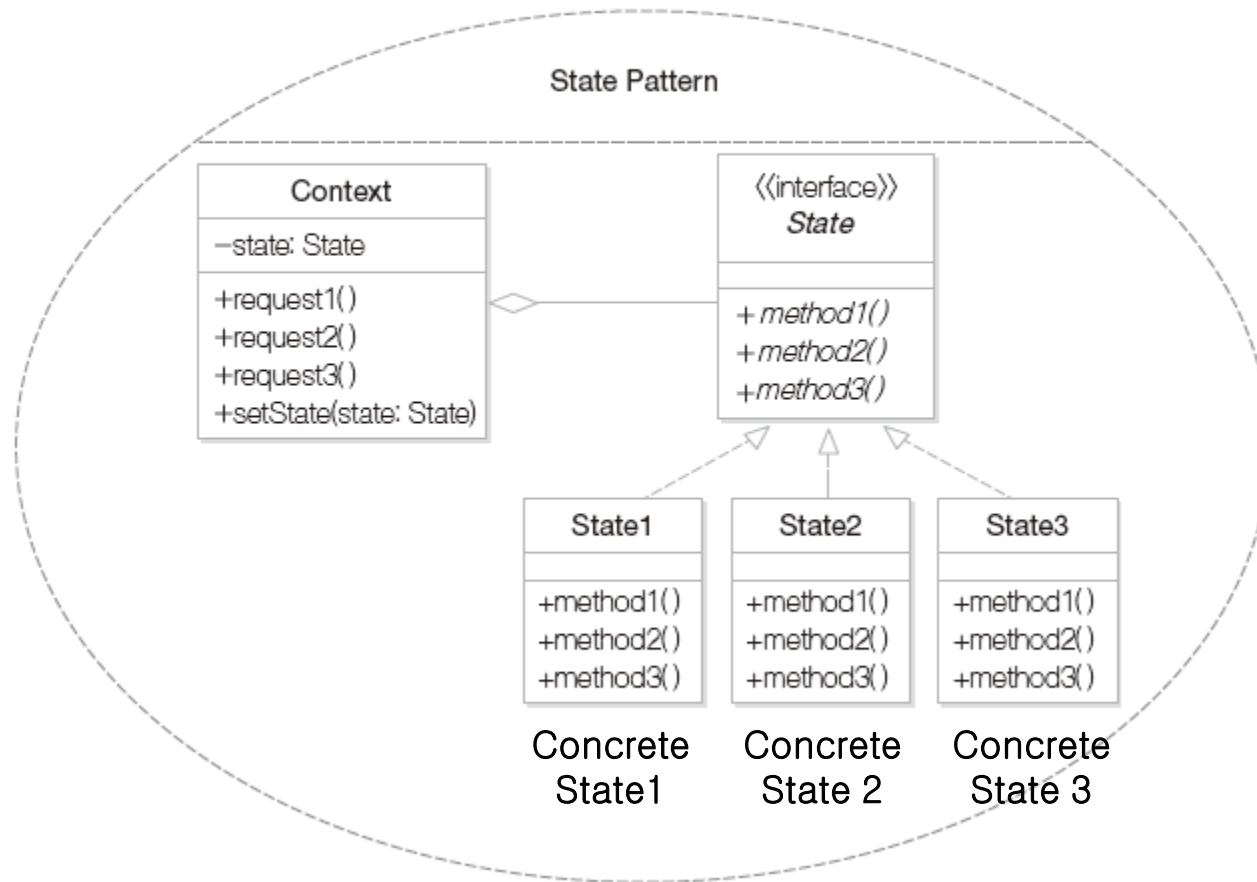
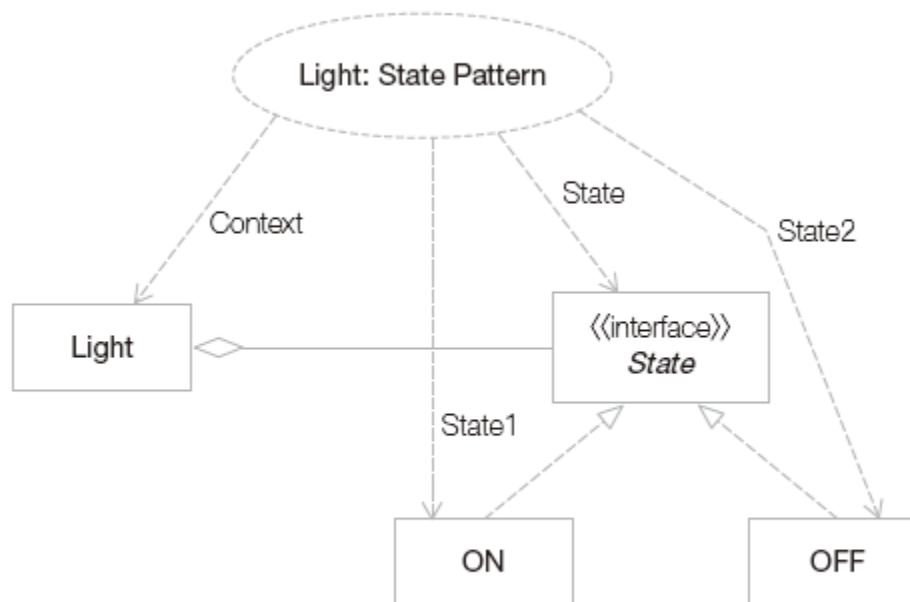


그림 7-6 스테이트 패턴을 형광등 예제에 적용한 경우



호텔 방 예약 시스템

- ❖ 방 예약
- ❖ 체크인
- ❖ 체크아웃
- ❖ 예약 취소

```
public class HotelSystem {
    // attributes
    private HotelState hotelState;

    // usage: new
    public HotelSystem() {
        this.hotelState = HotelState.AVAILABLE; // 예약 가능 상태
    }

    // new
    public void reserveRoom() {
        if (this.hotelState == HotelState.AVAILABLE) {
            System.out.println("예약 가능 상태입니다.");
            this.hotelState = HotelState.RESERVED; // 예약 완료 상태로 변경
        } else if (this.hotelState == HotelState.RESERVED) {
            System.out.println("이미 예약된 객실입니다.");
        } else if (this.hotelState == HotelState.OCCUPIED) {
            System.out.println("이미 체크인 되었습니다.");
        } else {
            System.out.println("잘못된 예약 상태입니다.");
        }
    }
}
```

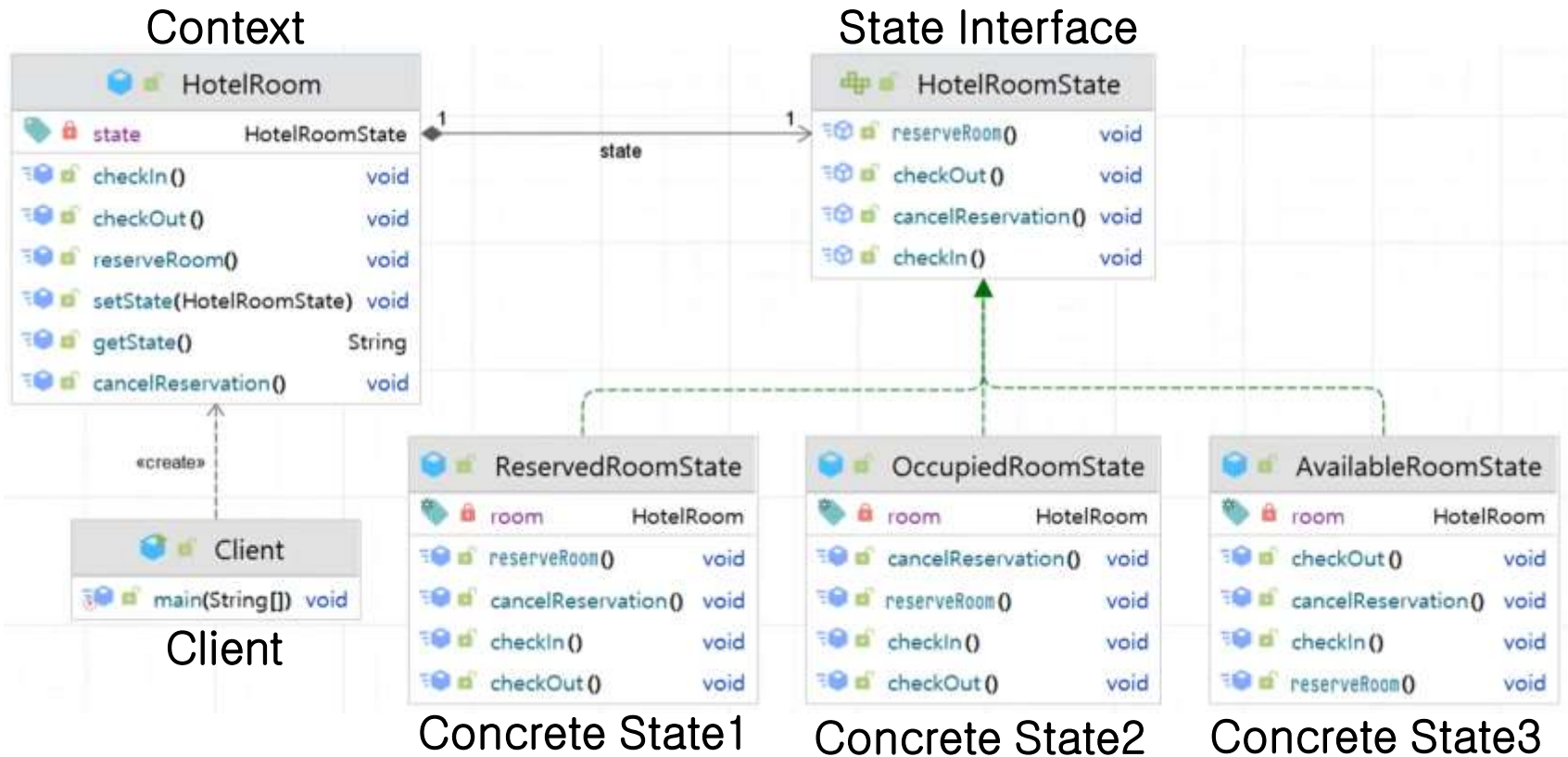
```
public void checkIn() {
    if (this.hotelState == HotelState.RESERVED) {
        System.out.println("체크인 완료");
        this.hotelState = HotelState.OCCUPIED; // 체크인 완료 상태로 변경
    } else if (this.hotelState == HotelState.AVAILABLE) {
        System.out.println("예약이 필요합니다.");
    } else if (this.hotelState == HotelState.OCCUPIED) {
        System.out.println("이미 체크인 되었습니다.");
    } else {
        System.out.println("잘못된 예약 상태입니다.");
    }
}

public void checkOut() {
    if (this.hotelState == HotelState.OCCUPIED) {
        System.out.println("체크아웃 완료");
        this.hotelState = HotelState.AVAILABLE; // 예약 가능 상태(초기 상태)로 변경
    } else if (this.hotelState == HotelState.AVAILABLE) {
        System.out.println("아직 체크아웃이 완료되지 않았습니다.");
    } else if (this.hotelState == HotelState.RESERVED) {
        System.out.println("아직 체크아웃이 완료되지 않았습니다.");
    } else {
        System.out.println("잘못된 예약 상태입니다.");
    }
}
```

복잡한 if문을 없애고
스테이트 패턴을 적용!

*출처: 아벨(2023), 아벨의 상태 패턴, <https://www.youtube.com/watch?v=BeoilfvAmpE>

호텔 방 예약 시스템



*출처: 아벨(2023), 아벨의 상태 패턴, <https://www.youtube.com/watch?v=BeoilfvAmpE>

비디오 플레이어



*출처: 알파한 코딩사전(2023). 상태(State) 패턴. <https://www.youtube.com/watch?v=RfJ7IUcCs04>

비디오 플레이어

```
public class VideoPlayer {
    private State state;

    public VideoPlayer() {
        // Set initial state Stopped
        this.state = new StoppedState();
    }

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

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

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

```
public class Main {
    public static void main(String[] args) {
        VideoPlayer player = new VideoPlayer();

        player.play(); // "Starting the video."
        player.play(); // "Video is already playing."
        player.stop(); // "Pausing the video."
        player.play(); // "Resuming the video."
        player.stop(); // "Pausing the video."
        player.stop(); // "Stopping the video."
        player.stop(); // "Video is already stopped."
    }
}
```

*출처: 알파한 코딩사전(2023). 상태(State) 패턴. <https://www.youtube.com/watch?v=RfJ7IUcCs04>

SNS의 좋아요/싫어요 버튼

- ❖ 스테이트 패턴을 사용해서 SNS의 좋아요/싫어요 버튼을 만들어 봅시다.

카페의 음료 주문/제작 상태

- ❖ 스테이트 패턴을 사용해서 카페에서 주문받은 음료를 “주문완료”, “제작중”, “제작완료”로 상태를 표시할 수 있도록 구현해봅시다.

스태이트 패턴의 장점

- ❖ 코드의 가독성 향상: 상태별로 행동을 분리함으로써 코드가 더 깔끔하고 이해하기 쉬워짐
- ❖ 유연한 상태 전이: 객체의 상태가 변화할 때, 각 상태에 대한 구체적인 클래스를 통해 유연하게 전이할 수 있음
- ❖ 응집도 증가: 상태와 관련된 행동이 동일한 클래스에 모여 있어 응집도가 높아짐
- ❖ 확장 용이: 새로운 상태를 추가하거나 기존 상태를 수정할 때, 전체 시스템에 미치는 영향을 최소화할 수 있음

스테이트 패턴의 단점

- ❖ 클래스 수 증가: 각 상태를 클래스별로 분리해야 하므로 클래스 수가 많아져 복잡성이 증가할 수 있음
- ❖ 상태 전이 관리의 복잡성: 상태 간의 전이를 관리하는 로직이 복잡해질 수 있으며, 실수로 잘못된 전이가 발생할 수 있음
- ❖ 디버깅 어려움: 상태가 많을 경우 디버깅이 복잡해질 수 있으며, 특정 상태에서 발생하는 버그를 추적하기 어려울 수 있음
- ❖ 초기 설계의 부담: 상태와 전이에 대한 명확한 이해가 필요하므로, 초기 설계에 더 많은 노력이 필요

스트래티지 패턴과 스테이트 패턴의 차이

- ❖ 초점: 전략 패턴은 알고리즘을 변경하는 데 중점을 두며, 상태 패턴은 객체의 상태에 따른 행동 변화를 관리함
- ❖ 전이 방식: 전략 패턴은 클라이언트가 명시적으로 알고리즘을 선택하는 반면, 상태 패턴은 객체가 내부적으로 상태를 변경하면서 적절한 행동을 자동으로 선택함
- ❖ 구조적 차이: 전략 패턴은 알고리즘을 독립적으로 분리하지만, 상태 패턴은 상태 간의 전이와 행동을 밀접하게 연결함