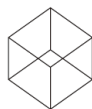
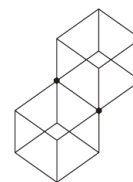


8. 커맨드 패턴



JAVA 개체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



학습목표

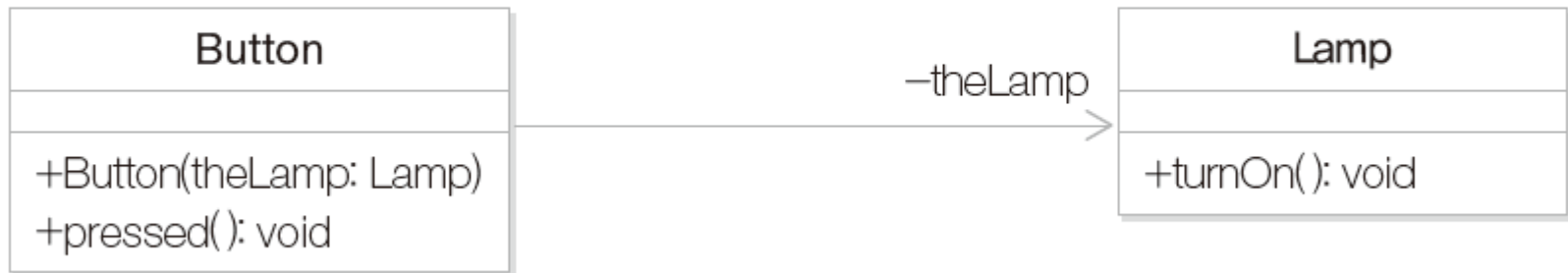
학습목표

- 기능을 캡슐화로 처리하는 방법 이해하기
- 커맨드 패턴을 통한 기능의 캡슐화 방법 이해하기
- 사례 연구를 통한 커맨드 패턴의 핵심 특징 이해하기

8.1 만능 버튼 만들기

- ❖ 만능 버튼: 눌리면 특정 기능을 수행
- ❖ 예) 램프를 켜는 버튼
 - Button 클래스: 버튼이 눌렸음을 인식하는 클래스
 - Lamp 클래스: 불을 켜는 기능을 제공

그림 8-1 램프 켜는 버튼을 설계한 클래스 다이어그램



만능 버튼 소스 코드

코드 8-1

```
public class Lamp {  
    public void turnOn() {  
        System.out.println("Lamp On");  
    }  
}
```

```
public class Button {  
    private Lamp theLamp ;  
    public Button(Lamp theLamp) {  
        this.theLamp = theLamp ;  
    }  
    public void pressed() {  
        theLamp.turnOn() ;  
    }  
}
```

```
public class Client {  
    public static void main(String[] args) {  
        Lamp lamp = new Lamp() ;  
        Button lampButton = new Button(lamp) ;  
        lampButton.pressed() ;  
    }  
}
```

8.2 문제점

- ❖ 버튼이 눌렸을 때 램프를 켜는 대신에 다른 기능을 수행하기 위해서는 어떤 변경 작업을 해야 되는가? 예를 들어 버튼이 눌리면 알람을 시작시키려면?
- ❖ 뿐만 아니라 버튼이 눌렸을 때 수행되는 기능을 프로그램이 동작할 때 결정하기 위해서는? 예를 들어 버튼이 처음 눌렸을 때는 램프를 켜고, 두 번째 눌렸을 때는 알람을 동작시키려면?

8.2.1 버튼을 눌렀을 때 다른 기능을 실행하는 경우

❖ 버튼을 눌렀을 때 알람을 시작하게 하려면

코드 8-2

```
public class Alarm {  
    public void start() {  
        System.out.println("Alarming...");  
    }  
}
```

```
public class Button {  
    private Alarm theAlarm ;  
    public Button(Alarm theAlarm) {  
        this.theAlarm = theAlarm ;  
    }  
    public void pressed() {  
        theAlarm.start() ;  
    }  
}
```

```
public class Client {  
    public static void main(String[] args) {  
        Alarm alarm = new Alarm() ;  
        Button alarmButton = new Button(alarm) ;  
        alarmButton.pressed() ;  
    }  
}
```

- Button 클래스의 pressed 메서드 수정이 필요함
- 기능 변경을 위해서 기존 소스 코드를 수정하므로 OCP를 위반하는 것임

8.2.2 버튼을 누르는 동작에 따라서 다른 기능을 실행하는 경우

❖ 처음 눌렀을 때는 램프를 켜고 두번째 눌렀을 때는 알람을 동작하게

Listing 8-3

```
public class Lamp {
    public void turnOn() { System.out.println("Lamp On") ; }
}

public class Alarm {
    public void start() { System.out.println("Alarming...") ; }
}

enum Mode { LAMP, ALARM} ;
public class Button {
    private Lamp theLamp ;
    private Alarm theAlarm ;
    private Mode theMode ;
    public Button(Lamp theLamp, Alarm theAlarm) {
        this.theLamp = theLamp ;
        this.theAlarm = theAlarm ;
    }
    public void setMode(Mode mode) { this.theMode = mode ; }
    public void pressed() {
        switch ( theMode ) {
            case LAMP: theLamp.turnOn() ; break ;
            case ALARM: theAlarm.start() ; break ;
        }
    }
}
```

Mode에 따라서 램프와 알람을 동작시킴

8.2.2 버튼을 누르는 동작에 따라서 다른 기능을 실행하는 경우

- ❖ 처음 눌렀을 때는 램프를 켜고 두번째 눌렀을 때는 알람을 동작하게

Listing 8-3

```
public class Client {  
    public static void main(String[] args) {  
        Lamp lamp = new Lamp();  
        Alarm alarm = new Alarm();  
        Button button = new Button(lamp, alarm);  
  
        button.setMode(Mode.LAMP);  
        button.pressed();  
  
        button.setMode(Mode.ALARM);  
        button.pressed();  
    }  
}
```

Mode를 설정함으로써 버튼의 동작을 변경시킴

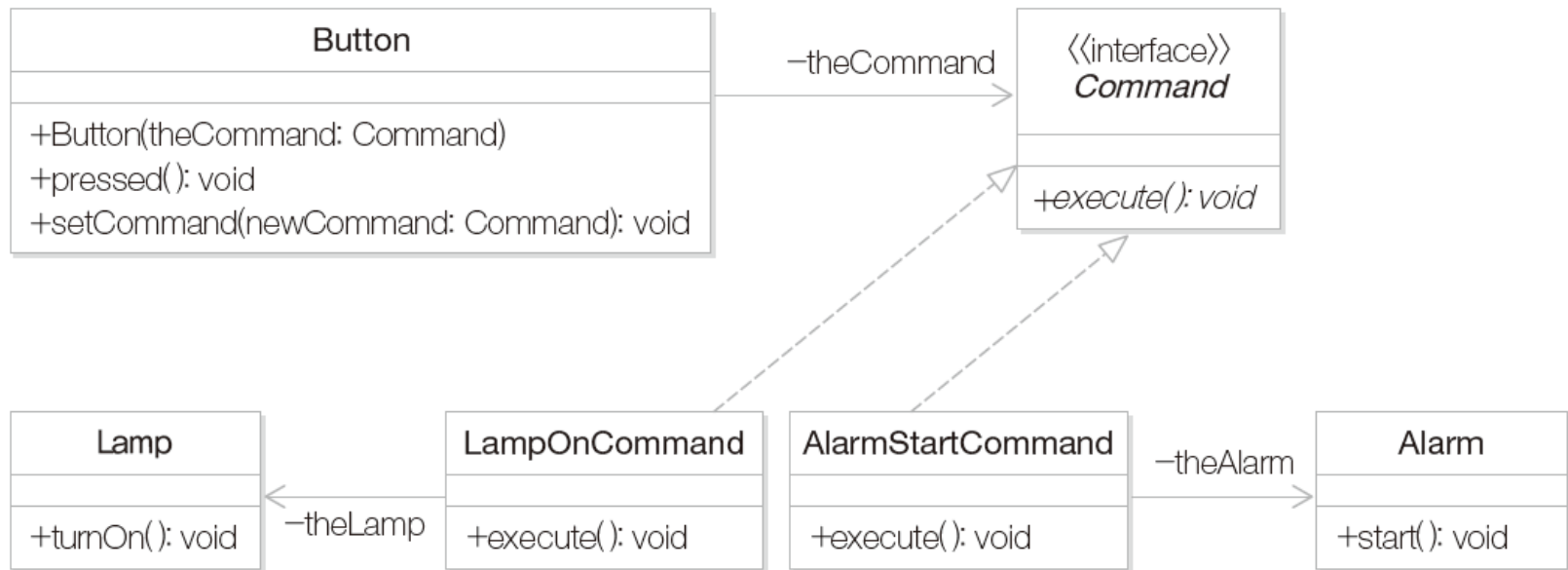
- ❖ 문제점: 기능의 변경 또는 새로운 기능의 추가 때마다 Button 클래스를 수정해야 함 → OCP를 위반함

8.3. 해결책

❖ 버튼이 눌렸을 때 수행될 기능을 캡슐화

- 버튼은 수행될 기능을 캡슐화된 객체로서 전달 받음
- 버튼이 눌리면 전달 받은 객체를 호출함으로써 구체적인 기능을 수행

그림 8-2 개선된 Button 클래스의 다이어그램



8.3. 해결책: 소스 코드

Listing 8-4

```
public interface Command {  
    abstract public void execute();  
}
```

```
public class Lamp {  
    public void turnOn() { System.out.println("Lamp On"); }  
}
```

```
public class LampOnCommand implements Command { // 램프를 켜는 기능의 캡슐화  
    private Lamp theLamp;  
    public LampOnCommand(Lamp theLamp) {  
        this.theLamp = theLamp ;  
    }  
    public void execute() { theLamp.turnOn(); }  
}
```

```
public class Alarm {  
    public void start() { System.out.println("Alarming..."); }  
}
```

```
public class AlarmOnCommand implements Command { // 알람을 울리는 기능의 캡슐화  
    private Alarm theAlarm ;  
    public AlarmOnCommand(Alarm theAlarm) {  
        this.theAlarm = theAlarm ;  
    }  
    public void execute() { theAlarm.start(); }  
}
```

8.3. 해결책: 소스 코드

Listing 8-4

```
public class Button {  
    private Command theCommand ;  
  
    public Button(Command theCommand) {  
        setCommand(theCommand) ;  
    }  
    public void setCommand(Command newCommand) {  
        this.theCommand = newCommand ;  
    }  
    // 버튼이 눌리면 주어진 Command의 execute 메서드를 호출함  
    public void pressed() {  
        theCommand.execute() ;  
    }  
}
```

8.3. 해결책: 소스 코드

Listing 8-4

```
public class Client {  
    public static void main(String[] args) {  
        Lamp lamp = new Lamp();  
        Command lampOnCommand = new LampOnCommand(lamp);  
  
        Button button1 = new Button(lampOnCommand); // 램프를 켜는 기능을 설정함  
        button1.pressed();  
  
        Alarm alarm = new Alarm();  
        Command alarmOnCommand = new AlarmOnCommand(alarm); // 알람을 울리는 기능을 설정함  
        Button button2 = new Button(alarmOnCommand);  
        button2.pressed();  
  
        button2.setCommand(lampOnCommand); // 알람을 울리는 기능을 설정함  
        button2.pressed();  
    }  
}
```

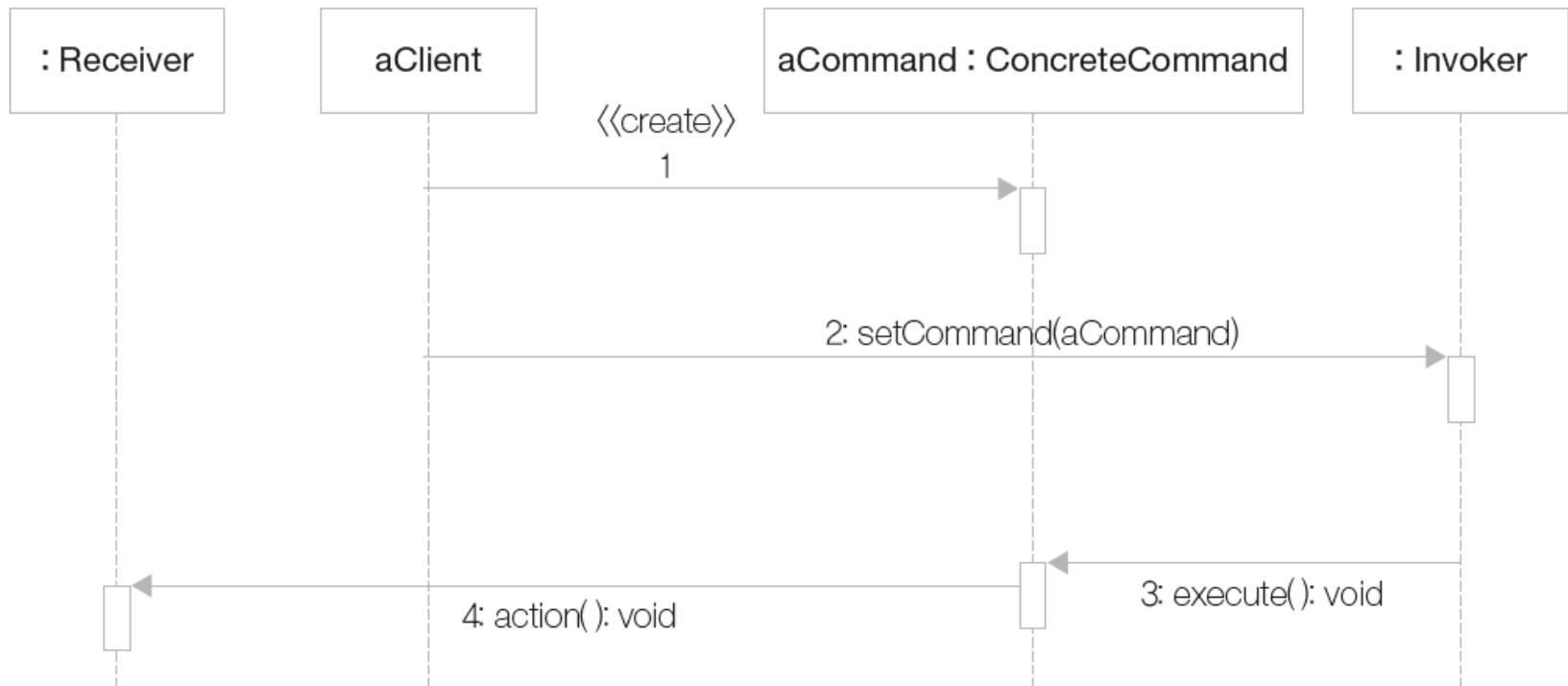
8.4 커맨드 패턴

- ❖ 이벤트가 발생했을 때 실행될 기능이 다양하면서 변경이 필요한 경우 이벤트를 발생시키는 클래스의 변경없이 재사용하고자 할 때

커맨드 패턴은 실행될 기능을 캡슐화함으로써 기능의 실행을 요구하는 호출자 클래스(Invoker)와 실제 기능을 실행하는 수신자 클래스(Receiver) 사이의 의존성을 제거한다. 따라서 실행될 기능의 변경에도 호출자 클래스를 수정없이 그대로 사용할 수 있도록 해준다.

8.4 커맨드 패턴

그림 8-5 커맨드 패턴의 순차 다이어그램



커맨드 패턴의 적용

그림 8-6 커맨드 패턴을 만능 버튼 예제에 적용한 경우

