8. 커맨드 패턴

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: 누르면 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

- 버튼을 눌렀을 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 ;
      }
   }
}
```

열거형

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() ;
   }
}
```

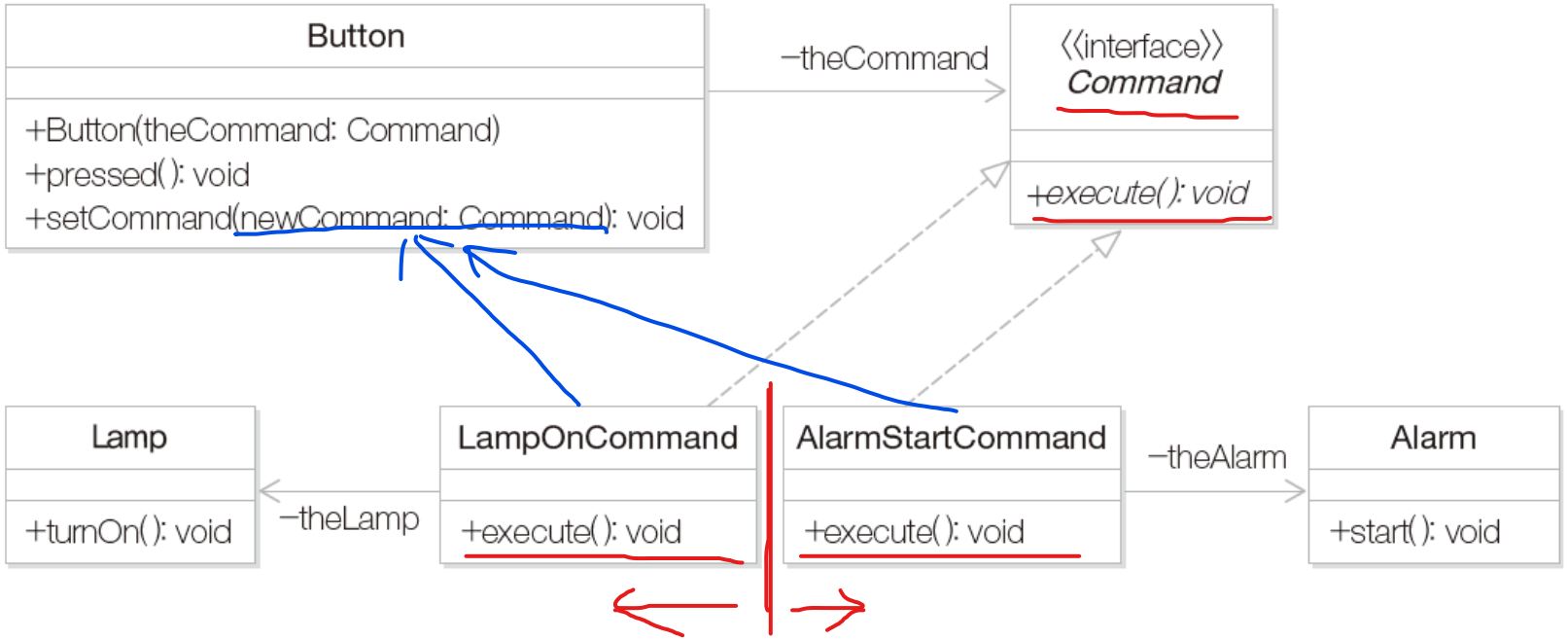
Mode를 설정함으로써 버튼의 동작을 변경시킴

- 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 구체적 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) ; // 램프를 켜는 기능을 설정함
      button2.pressed() ;
   }
}
```

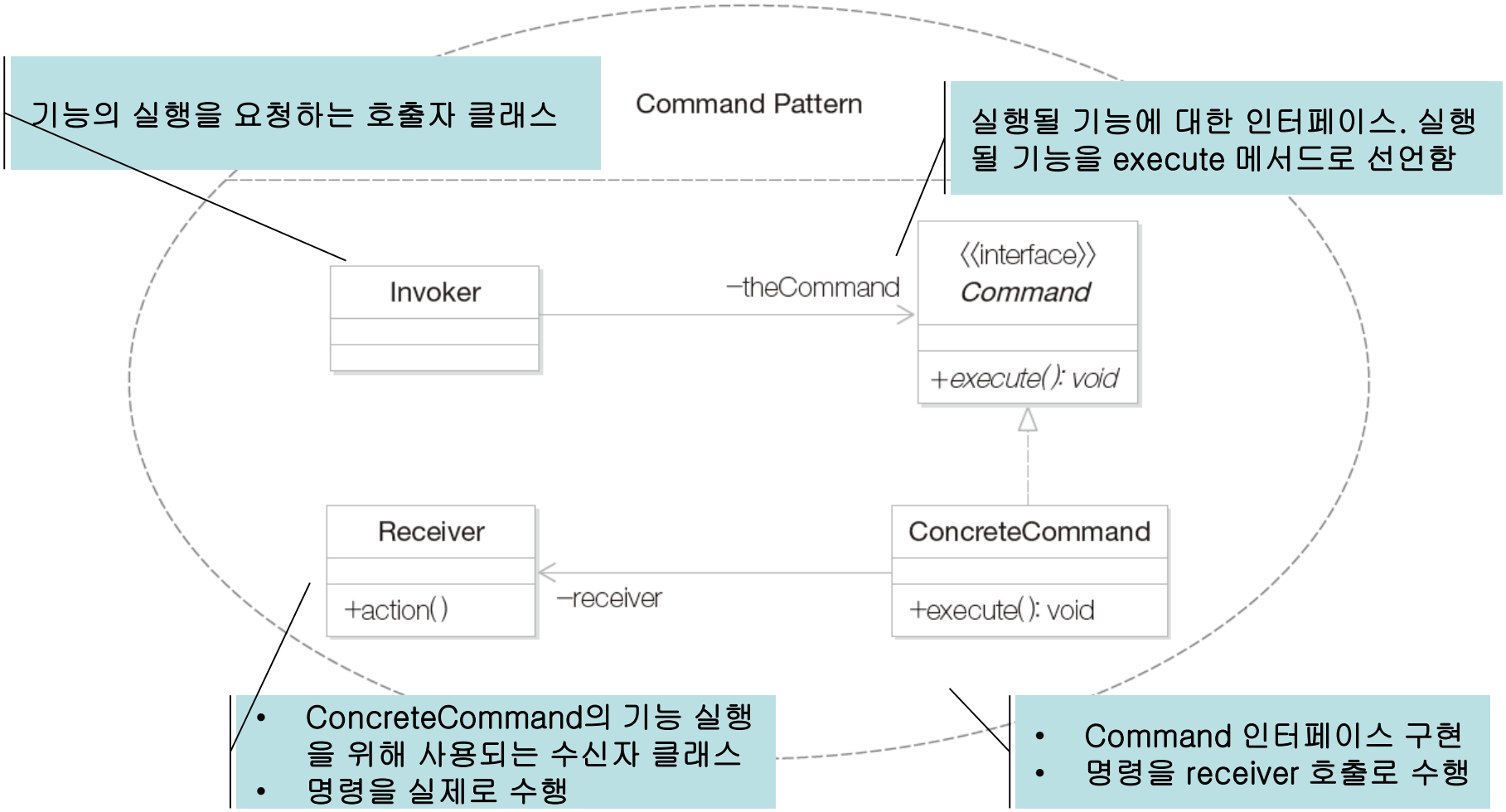
8.4 커맨드 패턴

- 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 8-4 커맨드 패턴의 컬레보레이션

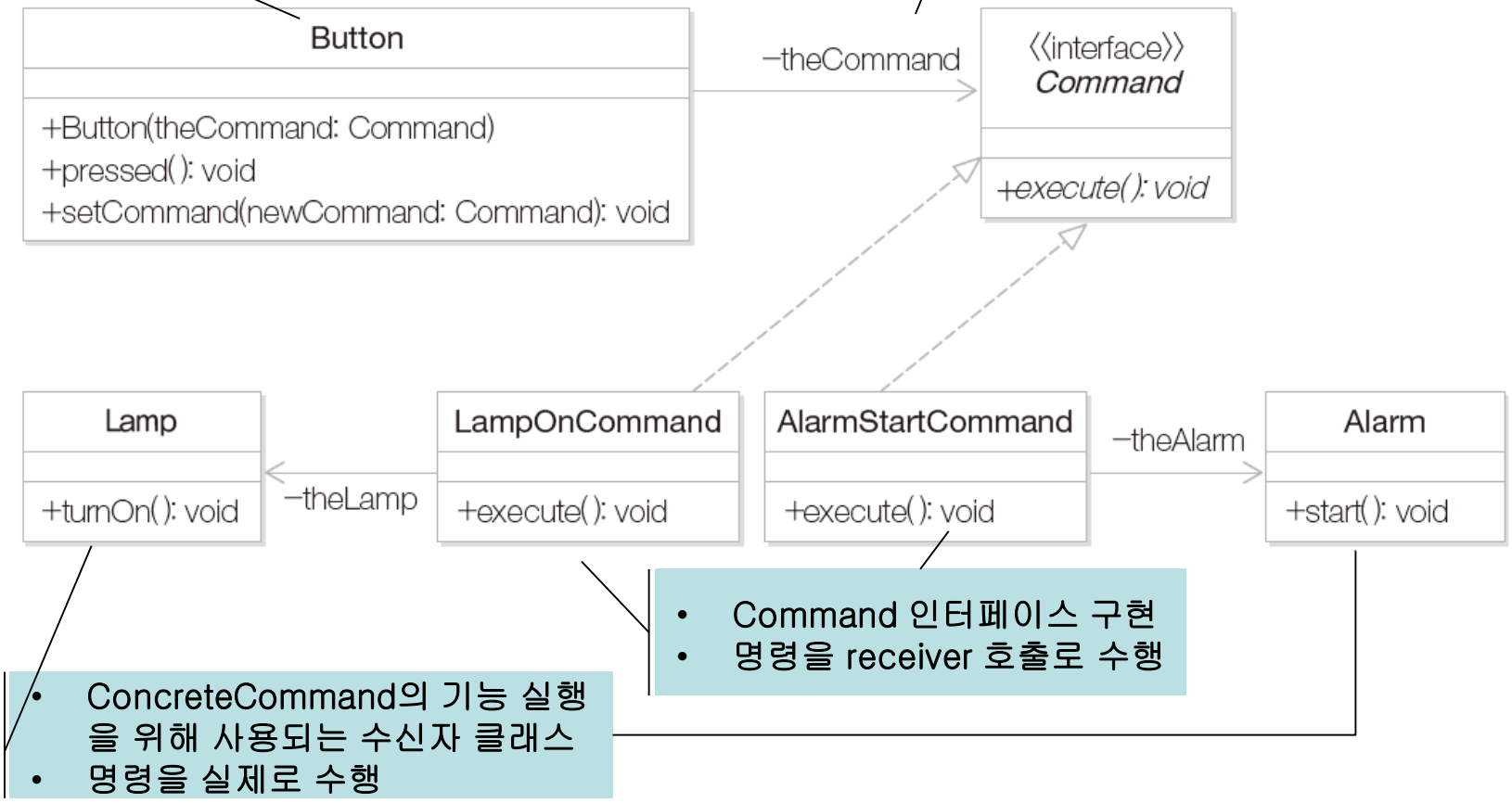


8.4 커맨드 패턴

기능의 실행을 요청하는 호출자 클래스

실행될 기능에 대한 인터페이스. 실행될 기능을 execute 메서드로 선언함

그림 8-2 개선된 Button 클래스의 다이어그램

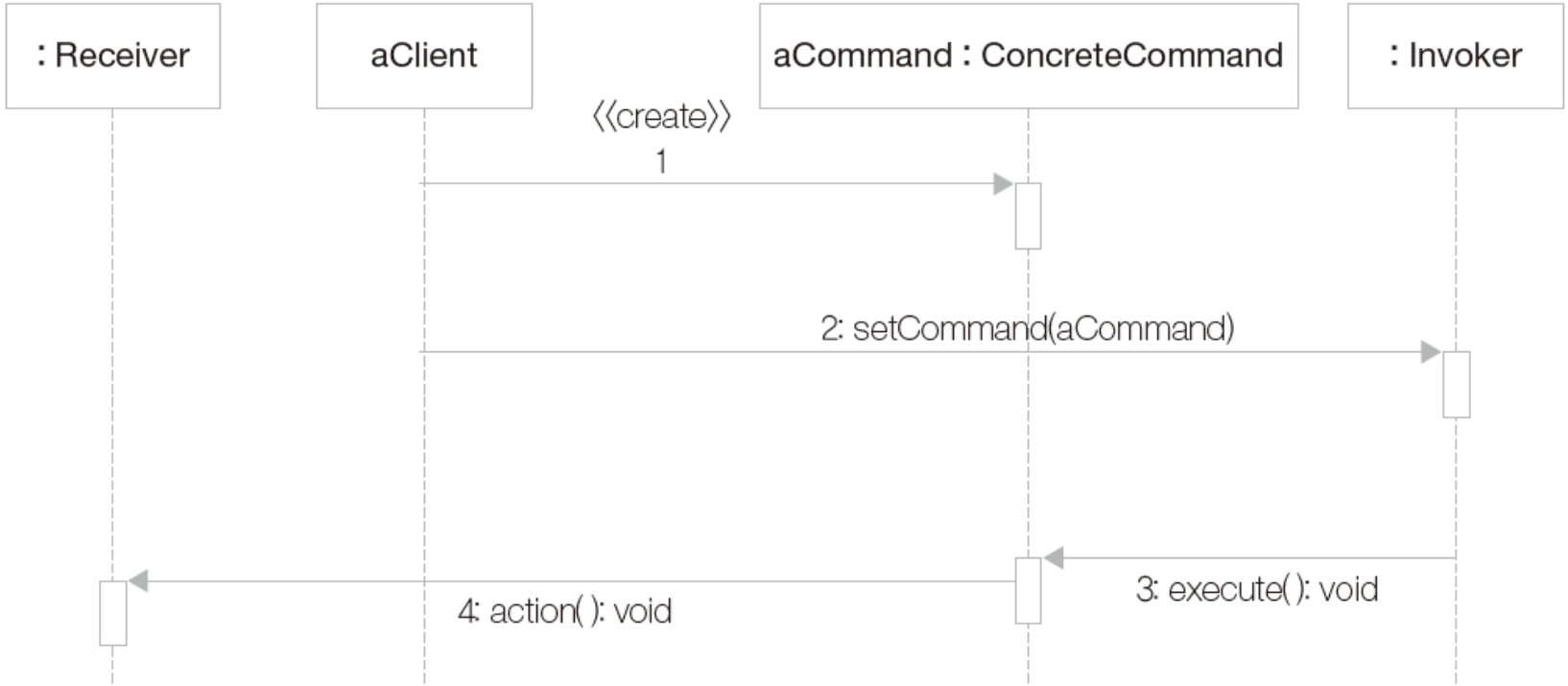


- Command 인터페이스 구현
- 명령을 receiver 호출로 수행

- ConcreteCommand의 기능 실행을 위해 사용되는 수신자 클래스
- 명령을 실제로 수행

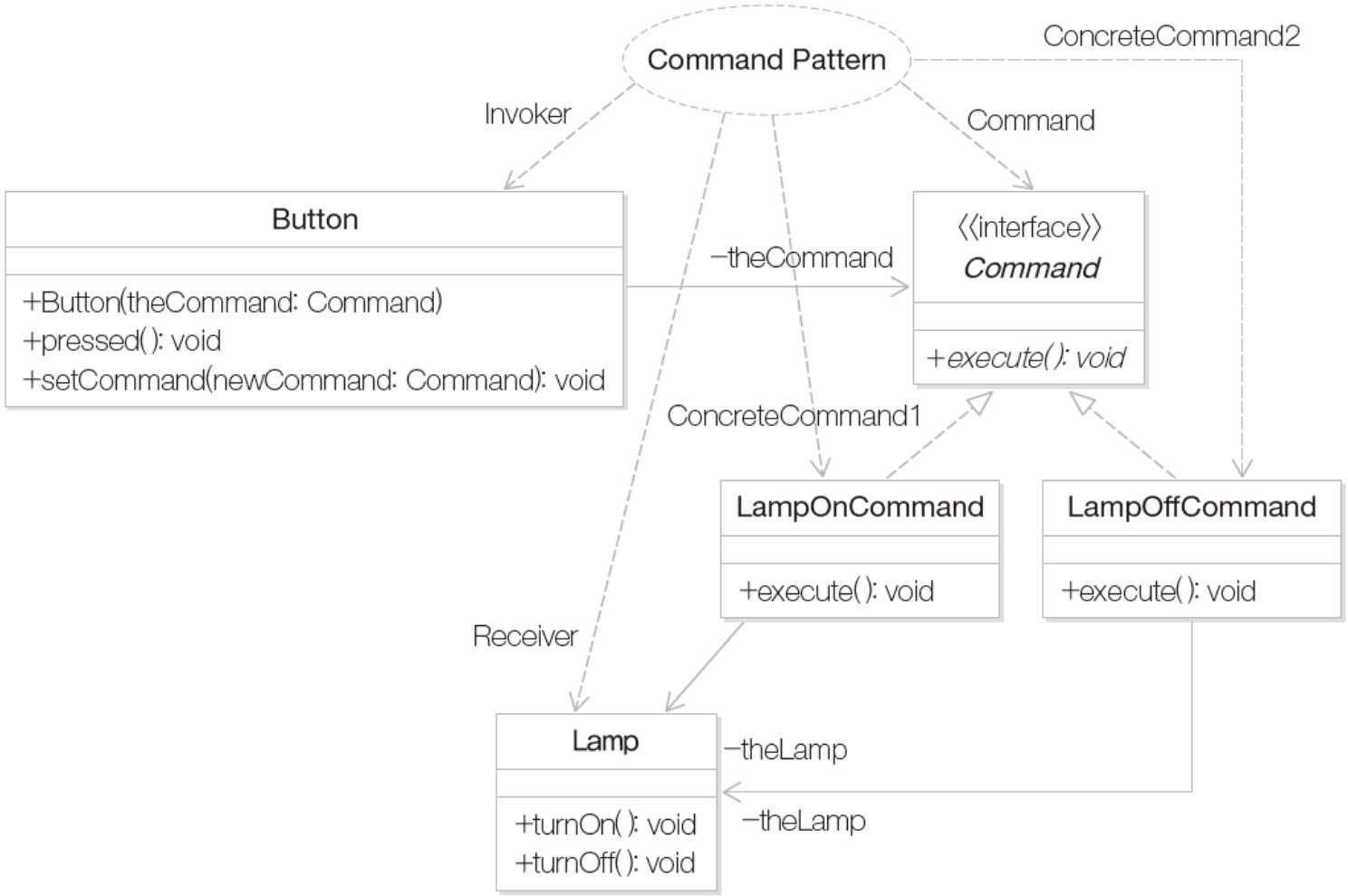
8.4 커맨드 패턴

그림 8-5 커맨드 패턴의 순차 다이어그램



커맨드 패턴의 적용

그림 8-6 커맨드 패턴을 만능 버튼 예제에 적용한 경우



커맨드 인터페이스

- 커맨드 패턴에는 커맨드 인터페이스가 항상 등장
- 명령을 내리고 싶을 때, 다뤄야 할 커맨드를 전부 이 인터페이스를 상세하게 구현하는 방식으로 만들어주면 캡슐화를 통해 같은 방법으로 명령을 실행할 수 있음

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

추상메소드는 execute()

클라이언트

```
public class Client {
    public static void main(String[] args) {
        Lamp lamp = new Lamp() ; //리시버
        Command lampOnCommand = new LampOnCommand(lamp) ; //커맨드 객체

        Button button1 = new Button(lampOnCommand) ; //인보커
        button1.pressed() ;

        Alarm alarm = new Alarm() ; //리시버
        Command alarmOnCommand = new AlarmOnCommand(alarm) ; //커맨드
        Button button2 = new Button(alarmOnCommand) ; //인보커
        button2.pressed() ;

        button2.setCommand(lampOnCommand) ;
        button2.pressed() ;
    }
}
```

인보커(Invoker) 클래스

- 생성자는 어떤 커맨드 객체를 컨트롤 할 것인지 파라미터로 받음
- setCommand는 다른 커맨드 객체로 교체하고자 할 때 사용

```
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() ;
        }
}
```

커맨드 클래스

- 커맨드 클래스이므로 Command 인터페이스를 구현
- 생성자는 이 커맨드 객체(LampOnCommand)로 제어하게 될 특정 조명 정보를 받음
- execute메소드가 호출되면 theLamp 인스턴스가 리시버(receiver)가 됨

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

리시버(receiver) 클래스

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

만능 리모콘을 만들어봅시다.

- 리모콘의 기능은 다음과 같습니다.
  - 기기의 전원을 끄고 켬
  - 라디오의 볼륨을 조절
  - TV의 채널을 변경함

출처 : [디자인 패턴] 커맨드 패턴(Command Pattern), https://jaimemin.tistory.com/2546

리시버 구현

```
public class Device {
    public void turnOn() {
        System.out.println("전원이 켜졌습니다.");
    }

    public void turnOff() {
        System.out.println("전원이 꺼졌습니다.");
    }
}
```

```
public class Stereo {
    public void adjustVolume() {
        System.out.println("볼륨 조절");
    }
}
```

```
public class TV {
    public void changeChannel() {
        System.out.println("TV 채널 변경");
    }
}
```

커맨드 부분

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

```
public class TurnOnCommand implements Command {
    private Device device;
    public TurnOnCommand(Device device) {
        this.device = device;
    }

    @Override
    public void execute() {
        device.turnOn();
    }
}
```

커맨드 부분

```
public class TurnOffCommand implements Command {
    private Device device;
    public TurnOffCommand(Device device) {
        this.device = device;
    }

    @Override
    public void execute() {
        device.turnOff();
    }
}
```

커맨드 부분

```
public class AdjustVolumeCommand implements Command {
    private Stereo stereo;
    public AdjustVolumeCommand(Stereo stereo) {
        this.stereo = stereo;
    }


    @Override
    public void execute() {
        stereo.adjustVolume();
    }
}
```

커맨드 부분

```
public class ChangeChannelCommand implements Command {
    private TV tv;
    public ChangeChannelCommand(TV tv) {
        this.tv = tv;
    }

    @Override
    public void execute() {
        tv.changeChannel();
    }
}
```

인보커(Invoker) 부분

```
public class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

클라이언트(Client) 부분

```
public class Client {
    public static void main(String[] args) {
    // Receiver 객체들 생성
    Device device = new Device();
    Stereo stereo = new Stereo();
    TV tv = new TV();

    // Command 객체들 생성
    Command turnOnDevice = new TurnOnCommand(device);
    Command turnOffDevice = new TurnOffCommand(device);
    Command adjustVolumeStereo = new AdjustVolumeCommand(stereo);
    Command changeChannelTV = new ChangeChannelCommand(tv);

    // Invoker 객체 생성 및 명령 실행
    RemoteControl remote = new RemoteControl();
    remote.setCommand(turnOnDevice);
    remote.pressButton();
    remote.setCommand(adjustVolumeStereo);
    remote.pressButton();
    remote.setCommand(changeChannelTV);
    remote.pressButton();
    remote.setCommand(turnOffDevice);
    remote.pressButton();
    }
}
```

커맨드 패턴의 장점

- 1. 요청의 캡슐화: 커맨드 패턴을 사용하면 요청을 객체로 캡슐화할 수 있어, 코드의 유연성을 높이고, 요청을 다양한 방식으로 처리할 수 있게 함
- 2. 요청의 큐잉 및 로그: 커맨드 객체를 사용하여 요청을 큐에 저장하거나 로그를 남길 수 있음
- 3. 실행 취소 및 재실행: 커맨드 패턴은 요청을 객체로 만들기 때문에, 실행 취소(undo) 및 재실행(redo) 기능을 구현하기가 용이
- 4. 결합도 감소: 클라이언트 코드와 요청을 처리하는 수신자(receiver) 간의 결합도를 낮출 수 있음
- 5. 확장성: 새로운 커맨드를 추가하는 것이 쉬움. 기존 코드를 수정하지 않고도 새로운 기능을 추가할 수 있어, 시스템의 확장성이 높아짐

커맨드 패턴의 단점

- 1. 복잡성 증가: 커맨드 패턴을 구현하기 위해서는 각 요청을 표현하는 커맨드 객체를 만들어야 하므로 코드의 복잡성이 증가할 수 있음
- 2. 메모리 사용량 증가: 각 커맨드 요청이 객체로 생성되기 때문에, 메모리 사용량이 늘어날 수 있음
- 3. 디버깅의 어려움: 커맨드 패턴을 사용하면 요청이 객체로 분리되기 때문에, 디버깅 시 요청의 흐름을 추적하는 것이 어려울 수 있음
- 4. 과도한 추상화: 간단한 요청의 경우, 커맨드 패턴을 적용하면 오히려 과도한 추상화가 되어 코드가 복잡해질 수 있음.
- 5. 상태 관리의 복잡성: 요청의 상태를 관리해야 하는 경우, 커맨드 객체 내에서 상태를 적절히 관리해야 함. 이로 인해 상태 관리가 복잡해질 수 있음

과제1

- 전략 패턴(Strategy Pattern)
- 싱글턴 패턴(Singleton Pattern)
- 상태 패턴(State Pattern)

각각의 패턴을 적용하기 알맞은 사례를 1개씩 생각하고, 그 사례를 실제로 패턴을 이용하여 구현해서 과제로 제출하세요.

*제출물 : 사례에 대한 설명, 만든 소스코드 전체 캡처 제출(실행결과 확인할 수 있게 캡처 요망)

제출 기한 : 11/10(일) 밤 11:59까지 e-class 제출

*앞으로 격주에 한번씩 패턴을 실제로 구현하는 과제가 부여될 예정입니다.