고급객체지향 프로그래밍
강의노트 #08

Command Pattern

조용주
ycho@smu.ac.kr

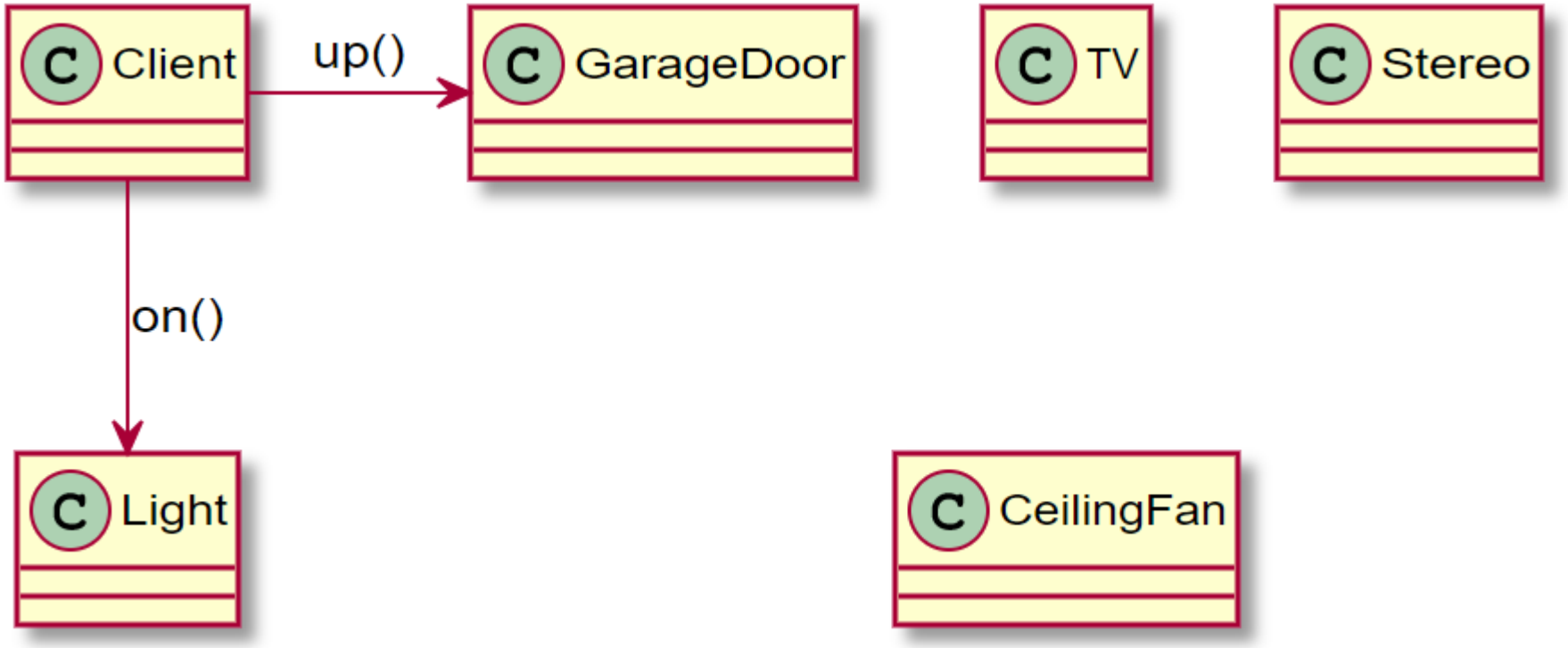
코맨드 패턴 (Command Pattern)

- 목적
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
  - 요구사항 (요청 , 명령) 을 객체로 캡슐화시킴 . 이를 이용해서 다른 요구사항을 지닌 클라이언트를 매개변수화 시킬 수 있고 , 요구사항을 큐에 넣거나 로그로 남길 수 있으며 작업 취소 (undo) 기능을 지원할 수도 있음

문제

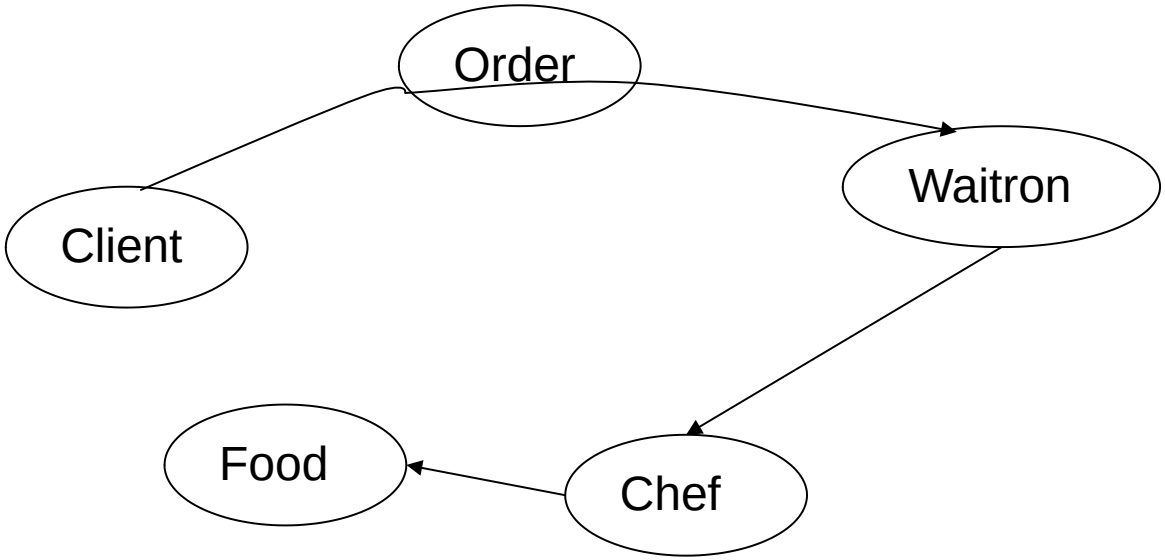
- 홈오토메이션용 리모컨
  - 사용하려는 객체가 많고 , API 가 서로 다른 경우
    - 차고문 up()
    - 전등 on()
    - TV pressOn()
    - …
  - 예 : 홈오토메이션 (Home Automation) 용 리모컨을 개발하는데 , 차고문 , 전등 , TV, Stereo, 에어컨 등 사용해야 하는 객체가 너무 많고 , 서로 다른 명령들로 구성되어 있음

문제



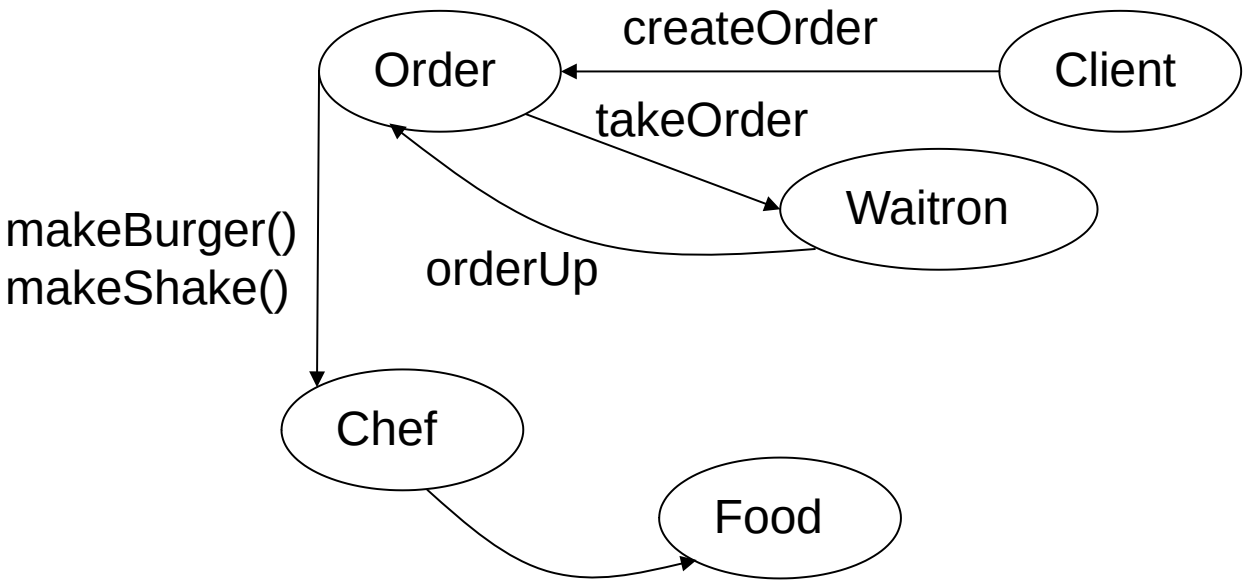
문제 – 객체마을 식당

- 식당에서 주문
  - 고객이 종업원에게 주문함
  - 종업원이 주문을 받아 카운터에 갖다 주고 " 주문 받아요 !" 라고 주방장에게 얘기함
  - 주방장이 주문대로 음식을 준비함

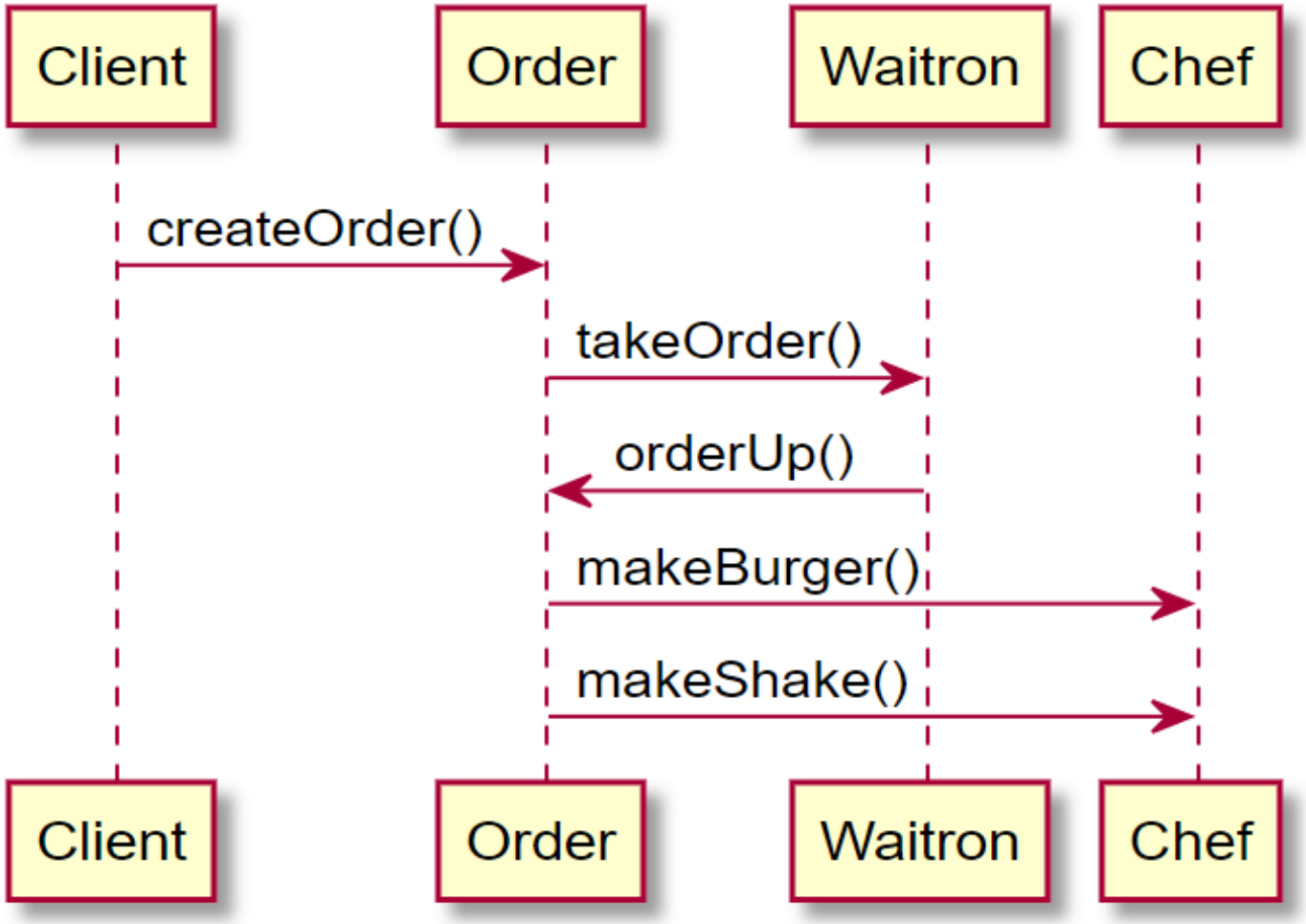


문제 – 객체마을 식당

- 객체와 메소드로 표현



문제 – 객체마을 식당


Client
Order
Waitron
Chef
createOrder()
takeOrder()
orderUp()
makeBurger()
makeShake()
Client
Order
Waitron
Chef

문제 – 객체마을 식당

- Order
  - 주문한 메뉴를 요구하는 역할
  - 웨이트런 (Waitron) 이 계산대 또는 다른 웨이트런에게 전달 가능
  - orderUp() 이라는 인터페이스가 포함되어 있음
    - 식사를 준비하기 위한 행동을 캡슐화한 메소드
  - 주방장 (chef) 에 대한 레퍼런스가 포함될 수 있음
- Waitron
  - 손님에게 주문 받고 orderUp() 메소드를 호출하여 식사 준비를 시키는 역할
  - Order 에 무슨 내용이 있는지 누가 식사를 준비하는지 알 필요 없음
  - takeOrder() 메소드 포함 (여러 Client 이 여러 Order 를 전달)

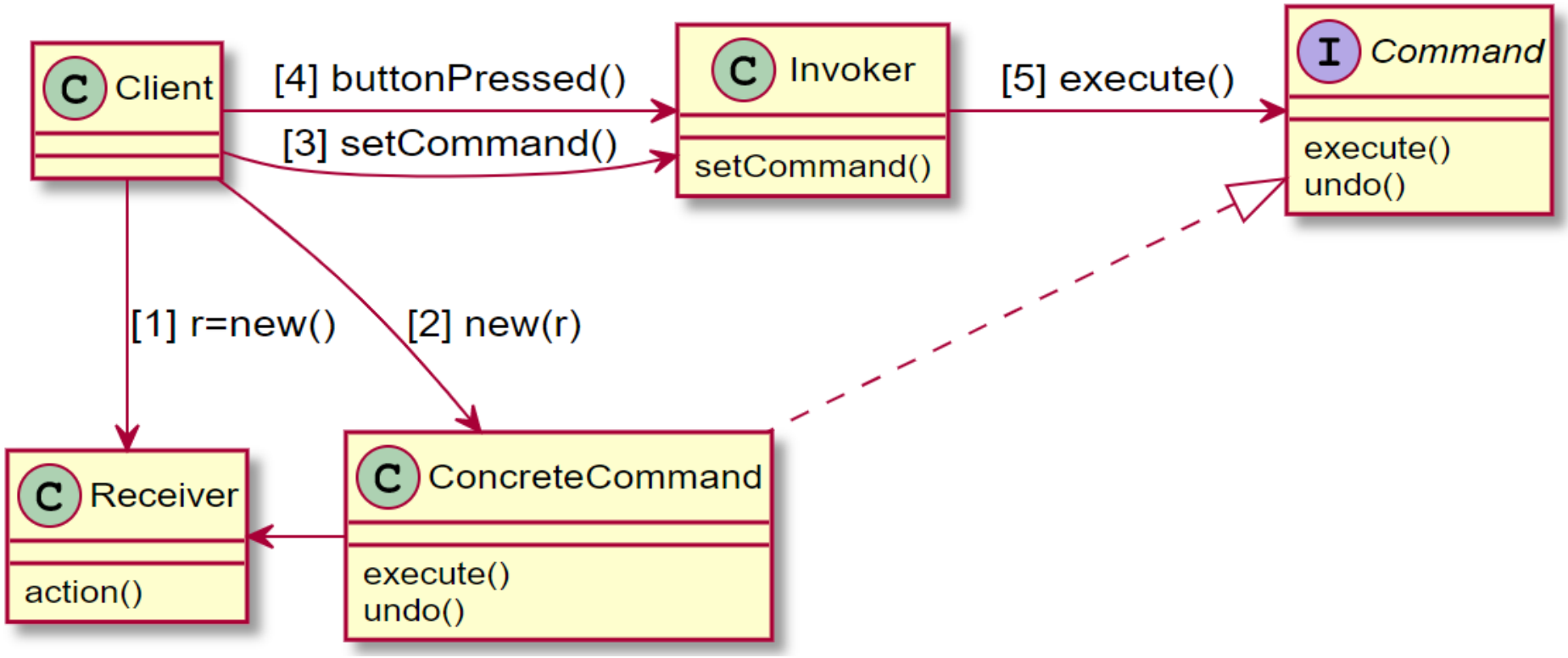
문제 – 객체마을 식당

- Chef
  - 식사를 준비하는 방법을 알고 있음
  - 웨이트런과 분리되어 있음
  - 웨이트런은 각 주문서에 있는 orderUp() 을 호출하고 Chef 는 Order 로부터 할 일을 전달 받음
- 여기서는 어떤 것을 요구하는 객체와 그 요구를 받아들이고 처리하는 객체를 분리시킴
- 리모컨 API 에서 리모컨 버튼이 눌렸을 때 호출되는 코드와 특정업체에서 제공한, 실제로 일을 처리하는 코드를 분리시키는 것이 필요함

디자인 패턴 요소

요소	설명
이름	커맨드 (Command)
문제	사용 객체의 API 가 서로 다름
해결방안	실행과 요청을 분리
결과	(작은) 클래스가 많아지지만 , 객체 사용에 필요한 복잡성을 제거하고 감춤 (함수 이름이 동일해짐)

커맨드 패턴 클래스 다이어그램



설계

- Decoupling
  - 요청과 실행을 분리

Object	설명	레스토랑	리모컨
Client	커맨드 객체 생성	Client	리모컨 버튼의 기능을 인지하고 버튼 누름
Command (커맨드)	어떤 Receiver 를 실행할 지 연결	Order	버튼에 실제 사용 객체를 연결해놓음
Invoker	주문을 받아서 , 실행하기 위해 Command 인터페이스 연결	Waitron	리모컨 버튼을 누르면 기능을 실행함
Receiver	실제 명령을	Chef	TV 전등 같은 실제

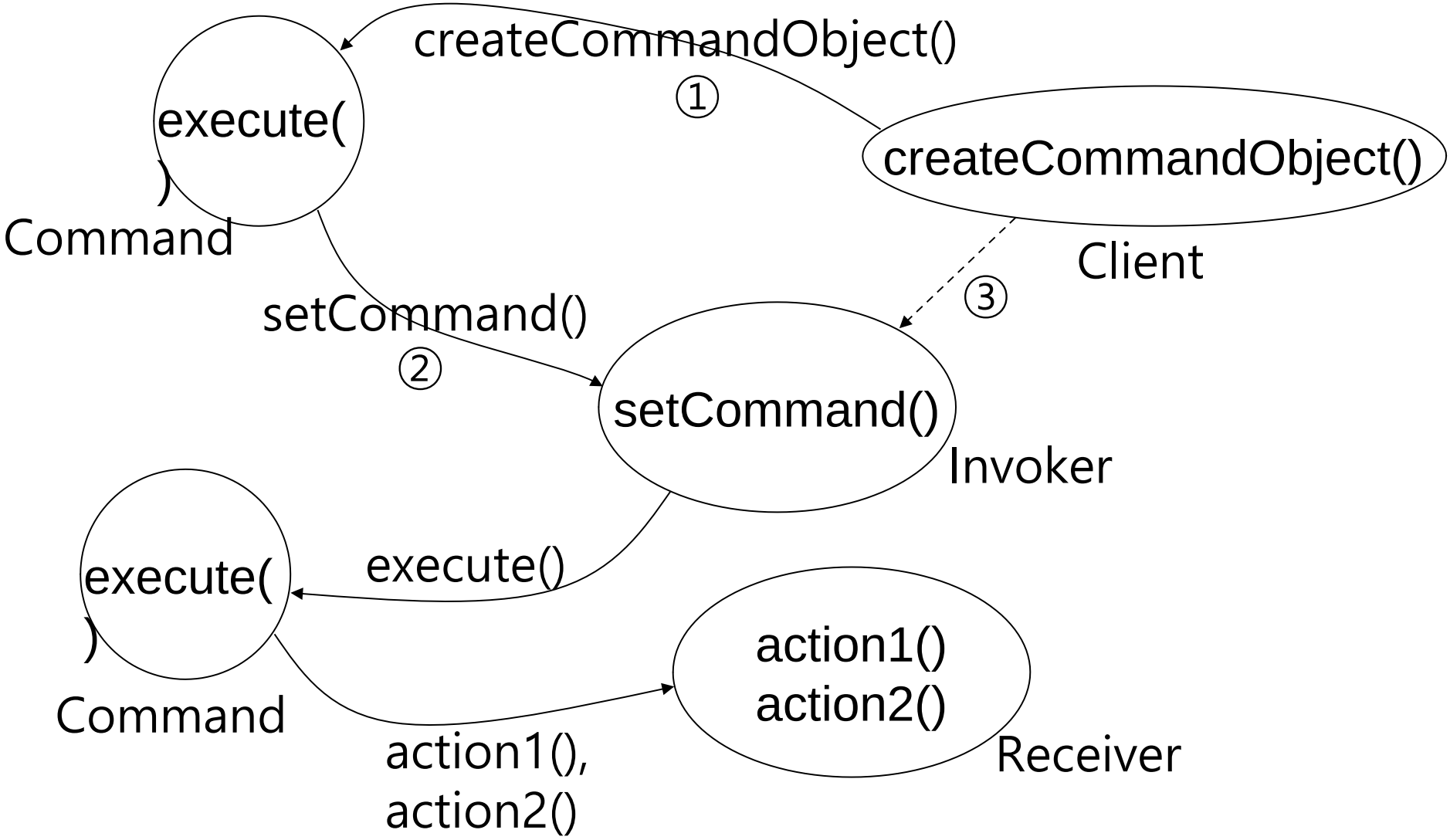
설계 (역할)

- Command
  - Receiver 를 알고 있고 , Receiver 의 메소드를 호출
  - Receiver 의 메소드에서 사용되는 매개변수 (parameters) 의 값들은 Command 에 저장됨
  - 예 : Command, ConcreteCommand
- Receiver
  - 실제 명령 (command) 수행
  - 예 : Light, GarageDoor

설계 (역할)

- Invoker
  - 요청을 받아서, 요청을 실행하기 위해 Command 인터페이스 연결
  - Command 인터페이스만 알고 있음. Command 가 실제 어떻게 실행되는지 모름
  - 예: 리모컨 (RemoteControl)
- Client
  - 무엇을 요청할지 결정하고, 요청 Command 를 Invoker 에 넘김
  - 예: main() 함수

객체마을 식당과 커맨드 패턴



첫 번째 커맨드 객체

- Command 인터페이스
  - 커맨드 객체는 모두 같은 인터페이스를 구현해야 함
    - 객체마을 식당에서는 orderUp()
    - 일반적으로는 execute()

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

첫 번째 커맨드 객체

- 전등을 켜기 위한 커맨드 클래스 구현
  - 전자제품 공급 업체에서 제공한 클래스인 Light 에는 on() 과 off() 라는 두 개의 메소드가 있음

```
public class LightOnCommand implements Command {
    Light light; // light 객체가 Receiver 가 됨

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```

첫 번째 커맨드 객체

- 전등을 켜기 위한 커맨드 클래스 구현
  - 전자제품 공급 업체에서 제공한 클래스인 Light 에는 on() 과 off() 라는 두 개의 메소드가 있음

```
public class GarageDoorOpenCommand implements
Command {
    GarageDoor door;

    public GarageDoorOpenCommand(GarageDoor d) {
        door = d;
    }

    public void execute() {
        door.open();
    }
}
```

커맨드 객체 사용하기

- 버튼이 하나 밖에 없는 리모컨이 있다고 가정

```
public class SimpleRemoteControl {
    Command slot;

    public SimpleRemoteControl() {}
    public void setCommand(Command command) {
        slot = command;
    }
    public void buttonWasPressed() {
        slot.execute();
    }
}
```

리모컨 사용을 위한 간단한 테스트 클래스

- SimpleRemoteControl 사용 코드

```
public class RemoteControlTest {
  public static void main(String[] args) {
    SimpleRemoteControl remote
                    = new SimpleRemoteControl();
    Light light = new Light();
    LightOnCommand lightOn
                    = new LightOnCommand(light);
    remote.setCommand(lightOn);
    remote.buttonWasPressed();
}
```

리모컨 사용을 위한 간단한 테스트 클래스

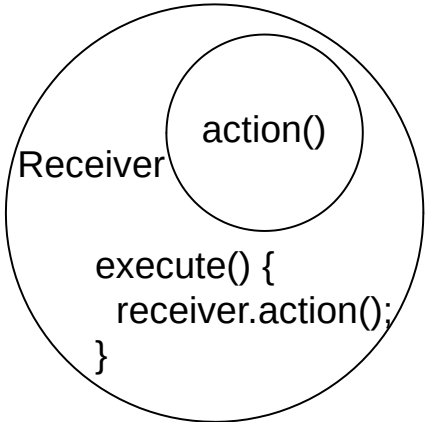
- GarageDoor 가 추가된다면

```
public class RemoteControlTest {
  public static void main(String[] args) {
    SimpleRemoteControl remote
                  = new SimpleRemoteControl();
    Light light = new Light();
    LightOnCommand lightOn
                  = new LightOnCommand(light);
    GarageDoor garageDoor = new GarageDoor();
    GarageDoorOpenCommand garageOpen =
      new GarageDoorOpenCommand(garageDoor);

    remote.setCommand(lightOn);
    remote.buttonWasPressed();
    remote.setCommand(garageOpen);
    remote.buttonWasPressed();
}
```

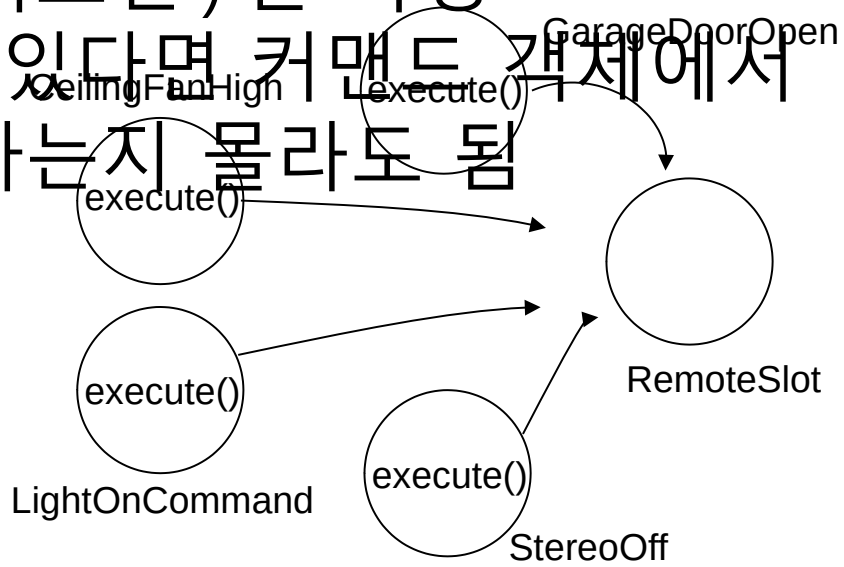
커맨드 패턴의 정의

- 커맨드 객체
  - 일련의 행동을 특정 리시버하고 연결시킴으로써 요구 사항을 캡슐화 한 것
  - 행동과 리시버를 한 객체에 집어넣고 , execute() 라는 메소드 하나만 외부에 공개함
  - 메소드 호출을 통해 리시버에서 일련의 작업들이 처리됨
  - 외부에서 볼 때에는 어떤 객체가 리시버 역할을 하는지 , 그 리시버에서 실제로 어떤 일을 하는지 알 수 없음 . 그냥 execute() 메소드를 호출하면 요구 사항이 처리된다는 것만 알게 됨



커맨드 패턴의 정의

- 명령을 통해서 객체를 매개변수화하는 예
  - 객체마을 식당에서 Waitron 에게 여러 개의 Order 를 전달
  - 리모컨 예제에서 버튼 슬롯에 "전등 켜기" 명령을 로딩했다가 나중에 "차고문 열기" 명령을 로딩하기도 함
- Invoker(Waitron 또는 리모컨)은 특정 인터페이스만 구현되어 있다면 커맨드 객체에서 실제로 어떤 일이 일어나는지 몰라도 됨



커맨드 패턴의 정의

```
public class RemoteControl {
  Command[] onCommands;
  Command[] offCommands;

  public RemoteControl() {
    onCommands = new Command[7];
    offCommands = new Command[7];
    Command noCommand = new NoCommand();
    for (int i = 0; i < 7; i++) {
      onCommands[i] = noCommand;
      offCommands[i] = noCommand;
    }
  }
  public void setCommand(int slot,
        Command onCommand, Command offCommand) {
    onCOmmands[slot] = onCommand;
    offCommands[slot] = offCommand;
  }
```

커맨드 패턴의 정의

```
  public void onButtonWasPushed(int slot) {
    onCommands[slot].execute();
  }
  public void offButtonWasPushed(int slot) {
    offCommands[slot].execute();
  }
  public String toString() {
    StringBuffer stringBuff = new StringBuffer();
    stringBuff.append("\n------ Remote Control
------\n");
    for (int i = 0; i < onCommands.length; i++) {
      stringBuff.append("[slot " + i + "] " + on-
Commands[i].getClass().getName() + "    " + off-
Commands[i].getClass().getName() + "\n");
    }
    return stringBuff.toString();
  }
}
```

```
public class LightOffCommand implements Command {
  Light light;
  public LightOffCommand(Light light) {
    this.light = light;
  }
  public void execute() {
    light.off();
  }
}

public class StereoOnWithCDCommand
                          implements Command {
  Stereo stereo;
  public StereoOnWithCDCommand(Stereo stereo) {
    this.stereo = stereo;
  }
  public void execute() {
    stereo.on();
    stereo.setCD();
    stereo.setVolume(11);
  }
}
```

리모컨 테스트

```
public class RemoteLoader {
  public static void main(String[] args) {
    RemoteControl remoteControl = new RemoteControl();
    Light livingRoomLight = new Light("Living Room");
    Light livingRoomLight = new Light("Kitchen");
    Stereo stereo = new Stereo("Living Room");
    LightOnCommand livingRoomLightOn =
                  new LightOnCommand(livingRoomLight);
    LightOffCommand livingRoomLightOff =
                  new LightOffCommand(livingRoomLight);
    LightOnCommand kitchenLightOn =
                  new LightOnCommand(kitchenLight);
    LightOffCommand kitchenLightOff =
                  new LightOffCommand(kitchenLight);
    StereoOnWithCDCommand stereoOnWithCD =
                  new StereoOnWithCDCommand(stereo);
    StereoOffWithCDCommand stereoOff =
                  new StereoOffCommand(stereo);
```

리모컨 테스트

```
    remoteControl.setCommand(0, livingRoomLightOn,
livingRoomLightOff);
    remoteControl.setCommand(1, kitchenLightOn,
kitchenLightOff);
    remoteControl.setCommand(3, stereoOnWithCD,
stereoOff);
    System.out.println(remoteControl);
    remoteControl.onButtonWasPushed(0);
    remoteControl.offButtonWasPushed(0);
    remoteControl.onButtonWasPushed(1);
    remoteControl.offButtonWasPushed(1);
    remoteControl.onButtonWasPushed(3);
    remoteControl.offButtonWasPushed(3);
  }
}

public class NoCommand implements Command {
  public void execute() {}
}
```

Undo 기능 추가

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

public class LightOnCommand implements Command {
    Light light; // light 객체가 Receiver 가 됨

    public LightOnCommand(Light light) {
        this.light = light;
    }
    public void execute() {
        light.on();
    }
    public void undo() {
      light.off();
    }
}
```

Undo 기능 추가

```
public class LightOffCommand implements Command {
  Light light;
  public LightOffCommand(Light light) {
    this.light = light;
  }
  public void execute() {
    light.off();
  }
  public void undo() {
    light.on();
  }
}
```

커맨드 패턴의 정의

```
public class RemoteControlWithUndo {
  Command[] onCommands;
  Command[] offCommands;
  Command undoCommand;

  public RemoteControlWithUndo() {
    onCommands = new Command[7];
    offCommands = new Command[7];
    Command noCommand = new NoCommand();
    for (int i = 0; i < 7; i++) {
      onCommands[i] = noCommand;
      offCommands[i] = noCommand;
    }
    undoCommand = noCommand;
  }
```

```
  public void setCommand(int slot,
        Command onCommand, Command offCommand) {
    onCOmmands[slot] = onCommand;
    offCommands[slot] = offCommand;
  }
  public void onButtonWasPushed(int slot) {
    onCommands[slot].execute();
    undoCommand = onCommands[slot];
  }
  public void offButtonWasPushed(int slot) {
    offCommands[slot].execute();
    undoCommand = offCommands[slot];
  }
  public void undoButtonWasPushed() {
    undoCommand.undo();
  }
  public String toString() {
    // 기존 코드…
  }
}
```

```
public class RemoteLoader {
  public static void main(String[] args) {
    RemoteControlWithUndo remoteControl = new Remote-
ControlWithUndo();
    Light livingRoomLight = new Light("Living Room");
    LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
    LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);
    remoteControl.setCommand(0, livingRoomLightOn,
livingRoomLightOff);

    remoteControl.onButtonWasPushed(0);
    remoteControl.offButtonWasPushed(0);
    System.out.println(remoteControl);
    remoteControl.undoButtonWasPushed();
    remoteControl.offButtonWasPushed(0);
    remoteControl.onButtonWasPushed(0);
    System.out.println(remoteControl);
    remoteControl.undoButtonWasPushed();
  }
}
```

Undo 기능을 구현할 때 상태를 사용하는 방법

```
public class CeilingFan {
  public static final int HIGH = 3;
  public static final int MEDIUM = 2;
  public static final int LOW = 1;
  public static final int OFF = 0;
  String location;
  int speed;
  public CeilingFan(String location) {
    this.location = location;
    speed = OFF;
  }
  public void high() {
    speed = HIGH; //  속도를 HIGH 로 맞추기
  }
  public void medium() { speed = MEDIUM; }
  public void low() { speed = LOW; }
  public void off() { speed = OFF; }
  public int getSpeed() { return speed; }
}
```

선풍기 명령어에 작업취소 기능 추가하기

```
public class CeilingFanHighCommand implements Command {
  CeilingFan ceilingFan;
  int prevSpeed;

  public CeilingFanHighCommand(CeilingFan ceilingFan) {
    this.ceilingFan = ceilingFan;
  }
  public void execute() {
    prevSpeed = ceilingFan.getSpeed();
    ceilingFan.high();
  }
```

선풍기 명령어에 작업취소 기능 추가하기

```
  public void undo() {
    if (prevSpeed == CeilingFan.HIGH) {
      ceilingFan.high();
    } else if (prevSpeed == CeilingFan.MEDIUM) {
      ceilingFan.medium();
    } else if (prevSpeed == CeilingFan.LOW) {
      ceilingFan.low();
    } else if (prevSpeed == CeilingFan.OFF) {
      ceilingFan.off();
    }
  }
}
```

선풍기 테스트 준비

```
public class RemoteLoader {
  public static void main(String[] args) {
    RemoteControlWithUndo remoteControl
                        = new RemoteControlWithUndo();
    CeilingFan ceilingFan
                       = new CeilingFan("Living Room");
    CeilingFanMediumCommand ceilingFanMedium =
       new CeilingFanMediumCommand(ceilingFan);
    CeilingFanHighCommand ceilingFanMedium =
       new CeilingFanHighCommand(ceilingFan);
    CeilingFanOffCommand ceilingFanMedium =
       new CeilingFanOffCommand(ceilingFan);
    remoteControl.setCommand(0, ceilingFanMedium,
                                ceilingFanOff);
    remoteControl.setCommand(1, ceilingFanHigh,
                                ceilingFanOff);
```

선풍기 테스트 준비

```
        remoteControl.onButtonWasPushed(0); // medium
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();// medium again
        remoteControl.onButtonWasPushed(1);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();// medium again
    }
}
```

리모컨에 매크로 버튼 추가

- 버튼 한 개를 누르면 전등이 어두워지면서, 오디오, TV 가 켜지고, DVD 모드로 변경되고, 욕조에 물이 채워지는 것 까지 한꺼번에 처리하는 기능

```
public class MacroCommand implements Command {
  Command[] commands;
  public MacroCommand(Command[] commands) {
    this.commands = commands;
  }
  public void execute() {
    for (int i = 0; i < commands.length; i++) {
      commands[i].execute();
    }
  }
}
```

매크로 커맨드 사용 방법

```
Light light = new Light("Living Room");
TV tv = new TV("Living Room");
Stereo stereo = new Stereo("Living Room");
Hottub hottub = new Hottub();
LightOnCommand lightOn = new LightOnCommand(light);
StereoOnCommand stereoOn = new StereoOnCommand(stereo);
TVOnCommand tvOn = new TVOnCommand(tv);
HottubOnCommand hottubOn = new HottubOnCommand(hottubOn);

Command[] partyOn = {lightOn, stereoOn, tvOn, hottubOn};
Command[] partyOff = {lightOff, stereoOff, tvOff, hut-
tubOff};
MacroCommand partyOnMacro = new MacroCommand(partyOn);
MacroCommand partyOffMacro = new MacroCommand(partyOff);
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);

remoteControl.onButtonWasPushed(0);
remoteControl.offButtonWasPushed(0);
```

디자인 패턴 요소

요소	설명
이름	커맨드 (Command)
문제	사용 객체의 API 가 서로 다름
해결방안	실행과 요청을 분리
결과	(작은) 클래스가 많아지지만 , 객체 사용에 필요한 복잡성을 제거하고 감춤 (함수 이름이 동일해짐)

커맨드 패턴 클래스 다이어그램