Command Pattern

514770-1
Fall 2026
10/27/2026
Kyoung Shin Park
Computer Engineering
Dankook University

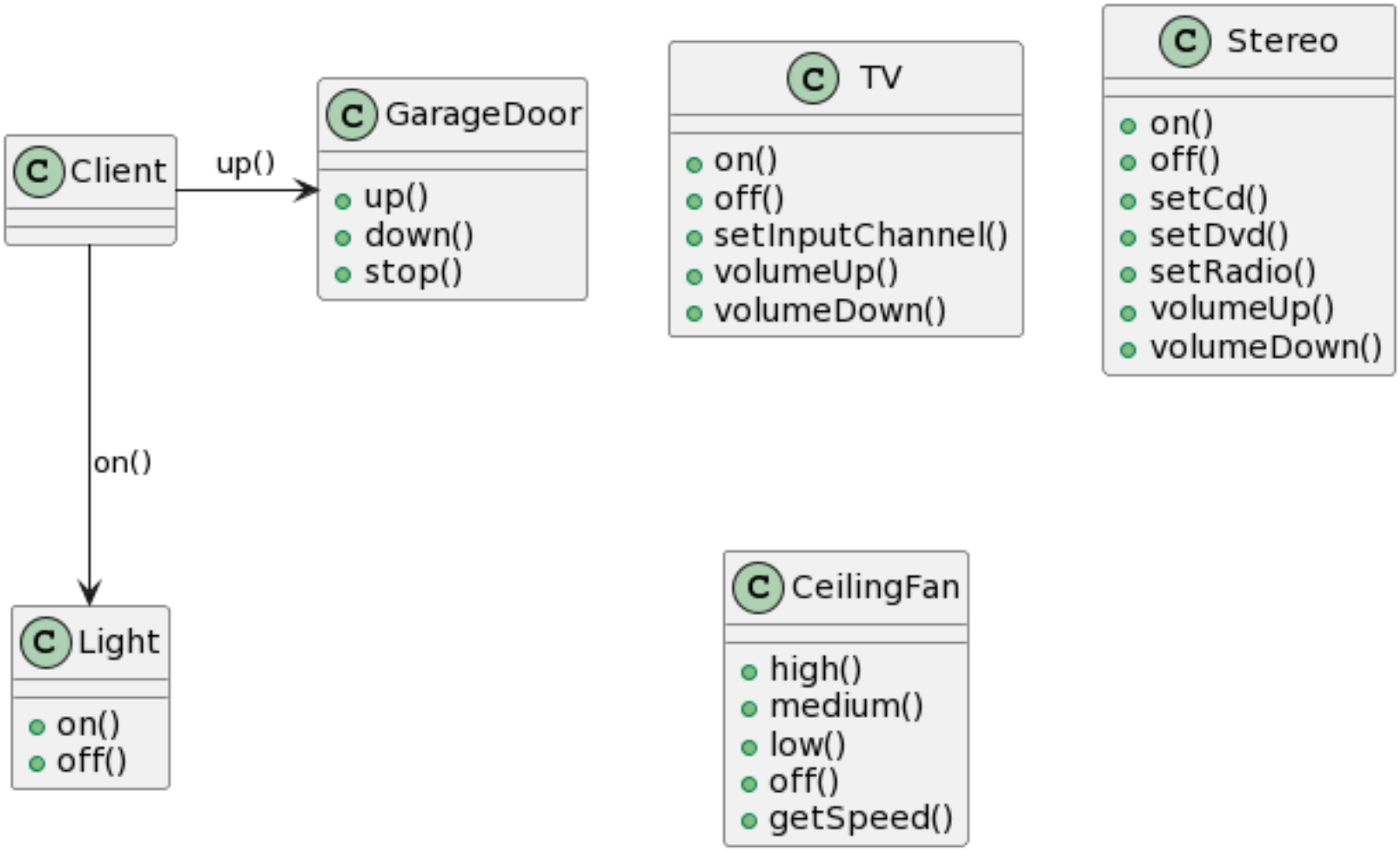
Command Pattern

- "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations."
- Promote "invocation of a method on an object" to full object status
- Also known as "an object-oriented callback"
- Command pattern is useful for "undo" operations.
- All implementations of java.lang.Runnable interface and All implementations of javax.swing.Action interface are good examples of how the command pattern is implemented.

Design Problem

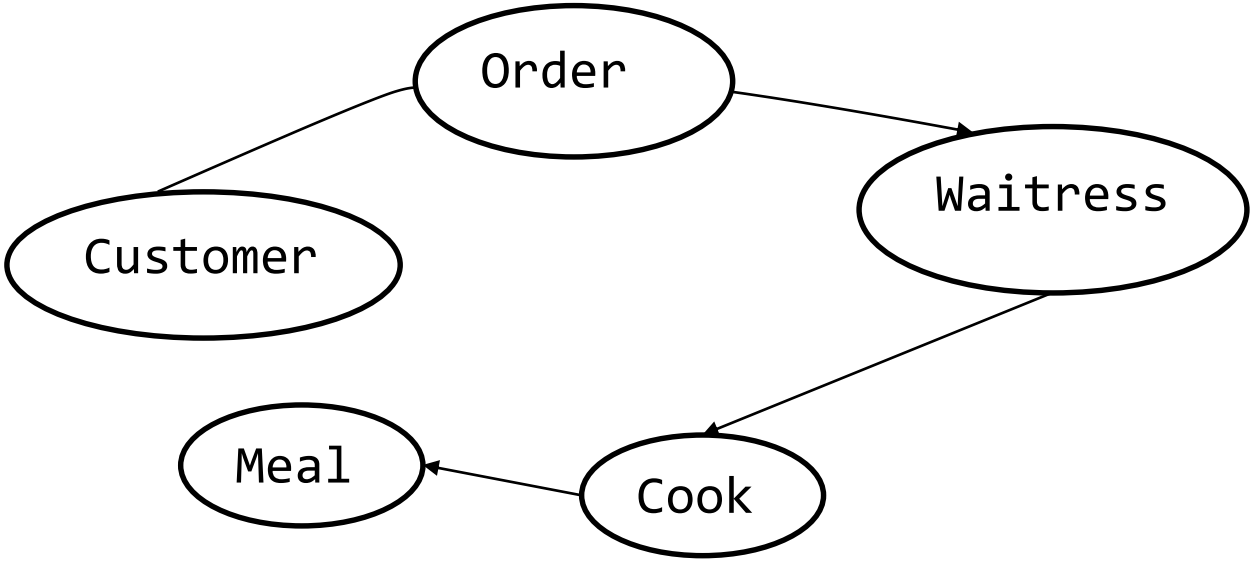
- Suppose you are building a home automation system.
  - There is a programmable remote controller which can be used to turn on and off various items in your home like light, stereo, AC etc.
  - Items have different APIs
    - Garage door up()
    - Light on()
    - TV pressOn()
    - …

Design Problem



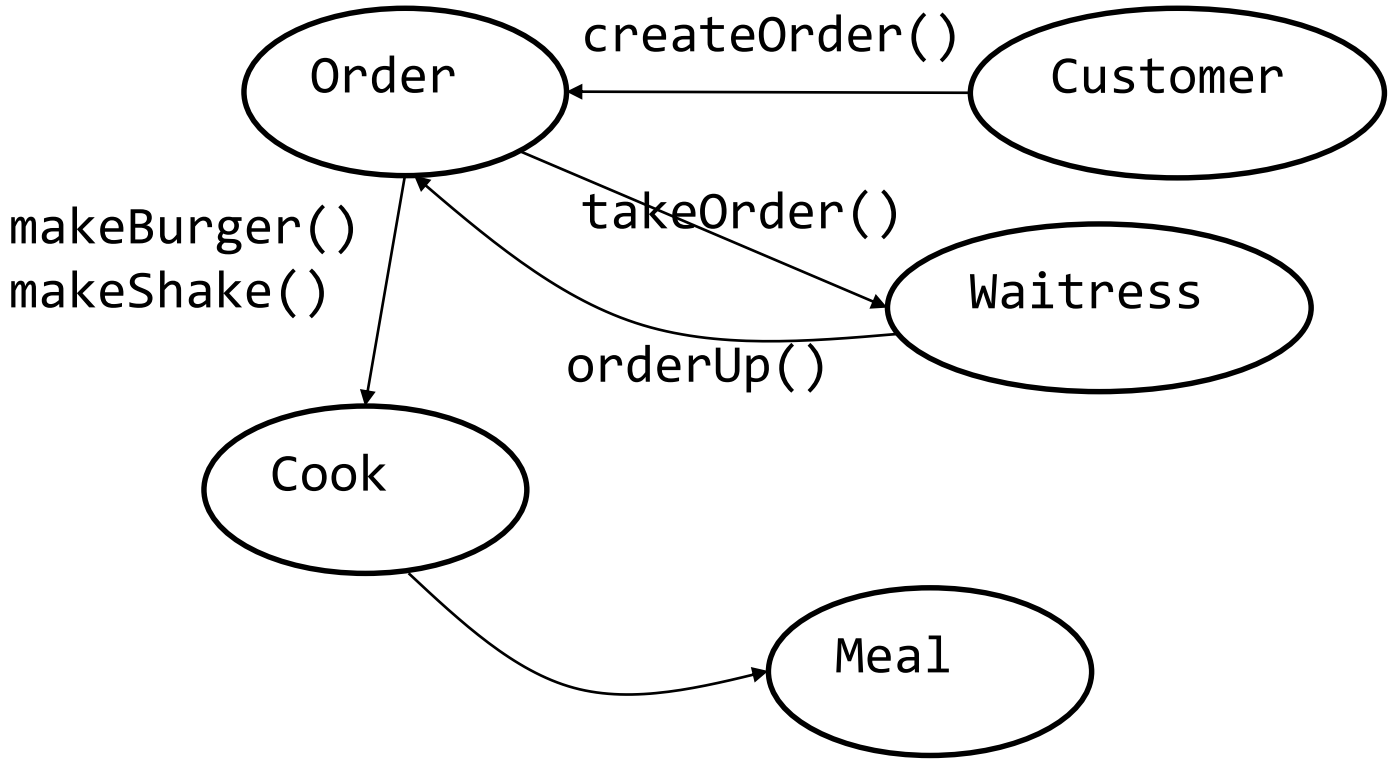
Objectville Diner (HFDP Ch. 6)

- How the Diner operates:
  - The **customer** give the **waitress** your **order**.
  - The **waitress** takes the **order**, place it on the order counter and **says "Order up!"**
  - The **cook** prepares your meal from the **order**.

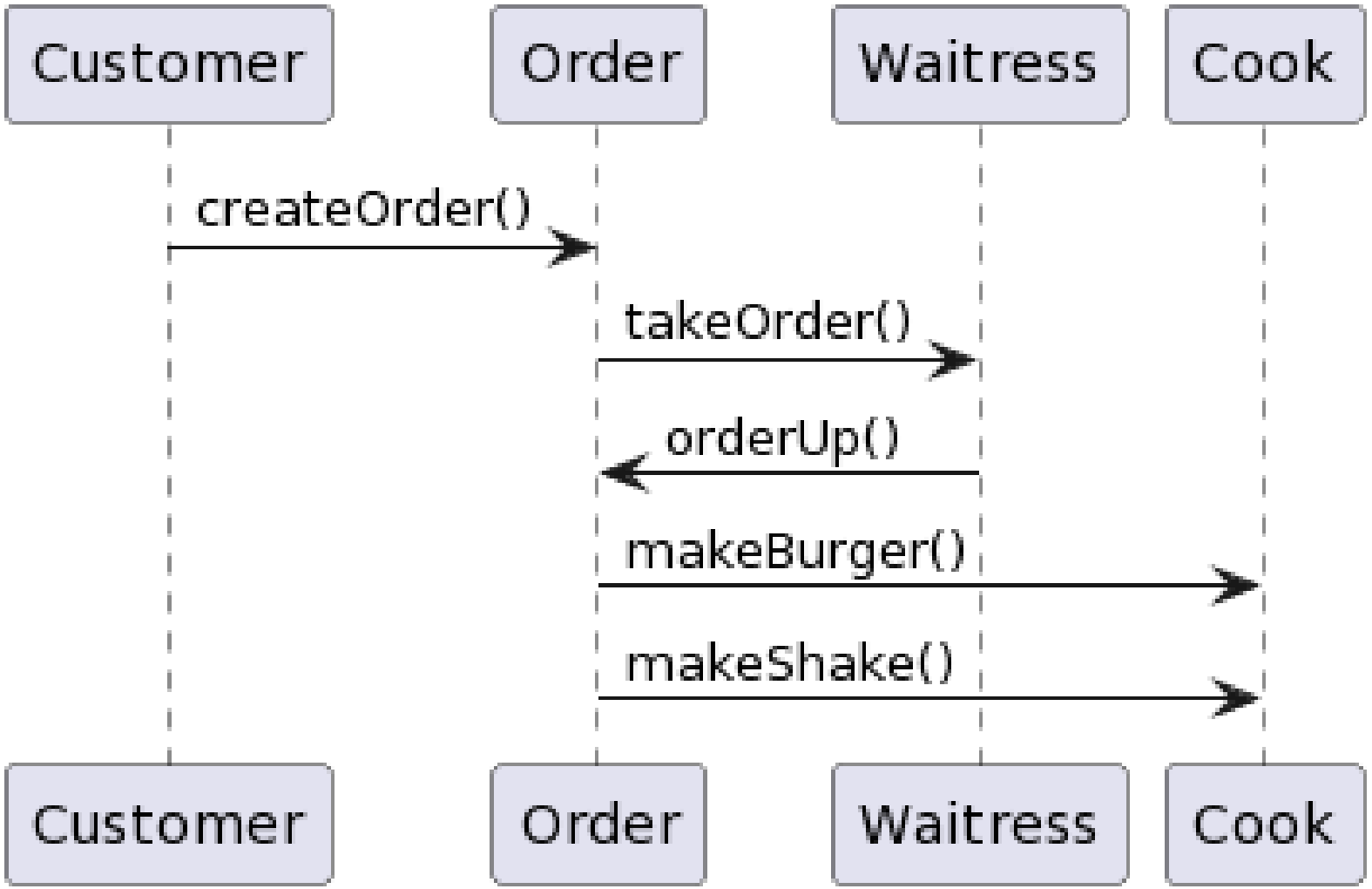


Objectville Diner (HFDP Ch. 6)

- Let's think about the object and method calls.



Objectville Diner (HFDP Ch. 6)



Objectville Diner (HFDP Ch. 6)

- Order
  - Order is an object that acts as a request to prepare a meal.
  - It can be passed around from Waitress to the order counter or to the next Waitress.
  - It has an interface that consists of only one method, **orderUp()**.
    - orderUp() encapsulates the actions needed to prepare the meal.
  - It also has a reference to the object that needs to prepare it (in this case, the Cook).
- Waitress
  - Waitress's job is to take the Order from the Customer, then invoke the orderUp() method to have the meal prepared.
  - Waitress really isn't worried about what's on the Order or who is going to prepare it.
  - Waitress's takeOrder() method gets parameterized with different order from different customers.

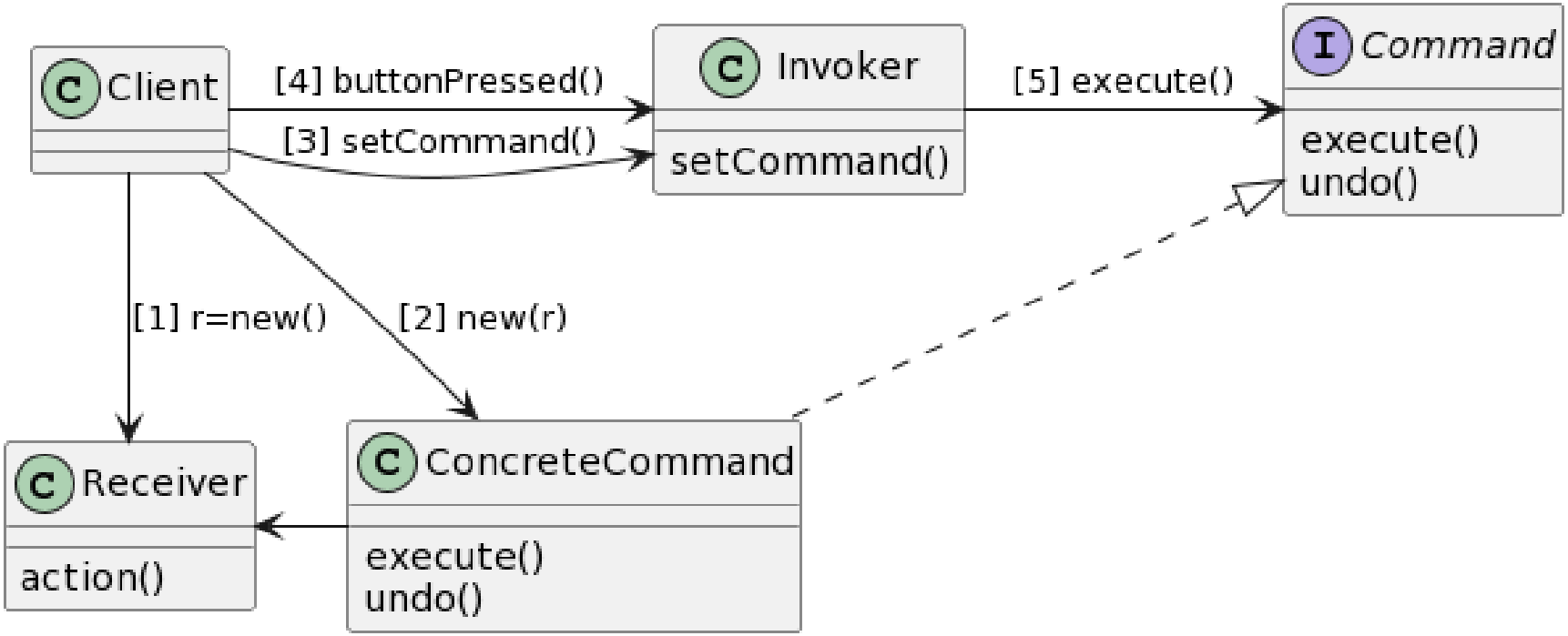
Objectville Diner (HFDP Ch. 6)

- Cook
  - The Cook is the object that really knows how to prepare the meal.
  - Notice the Waitress and the Cook are totally decoupled; the Waitress has the Order that encapsulate the details of the meal; she just calls a method on each order to get it prepared; the Cook gets his instructions from the Order; he never needs to directly communicate with the Waitress.
  - The Waitress has invoked the orderUp(); The Cook takes over and implements all the methods that are needed to create meals.
- In our remote control API, we need to **separate the code** that gets invoked when we press a button from the objects of the vendor-specific classes that carry out those requests.

Command Pattern

	Description
Pattern	Command
Problem	The object APIs do not follow a regular pattern (e.g. camera start/stop recording, light on/off, speaker volume up/down).
Solution	Separate execution and request. The command object contains all the details needed to execute the request.
Result	It creates more (small) classes, but removes and hides the complexity of using objects (the method name becomes the same).

Command Pattern



Define Command Pattern

- Command
  - Defines an interface for executing an operation or set of operations.
- ConcreteCommand
  - Implements the Command interface to perform the operations. Typically acts as an intermediary to a Receiver object.
  - Command knows Receiver, and calls Receiver method
  - Command contains the values of parameters used in Receiver method.
- Receiver
  - Perform the command operations
  - Example: Light on/off, GarageDoor open/close

Define Command Pattern

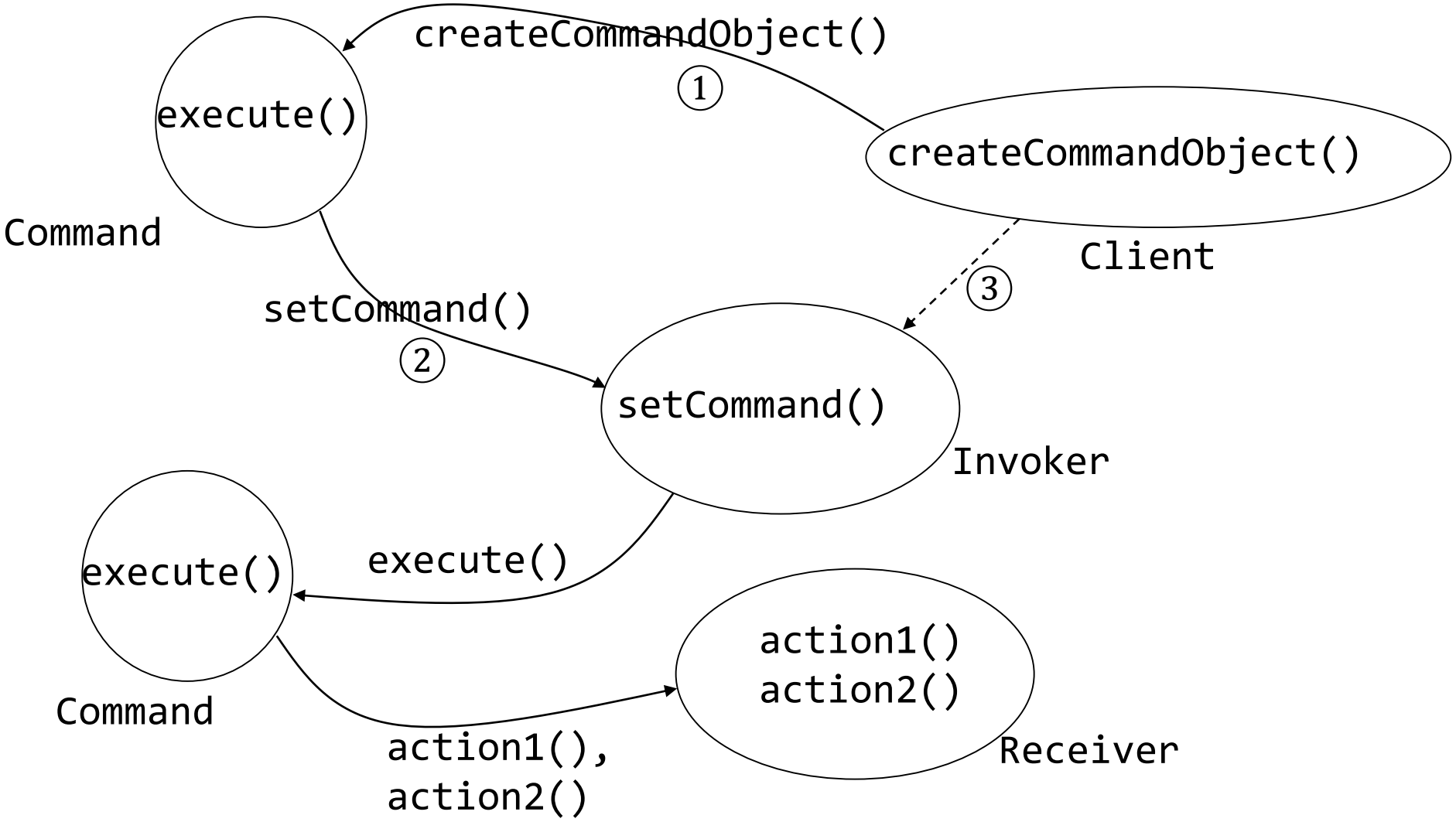
- Invoker
  - Invoker receives a request and bind the Command interface to execute the request.
  - Invoker knows only the Command interface. It doesn't know how the command actually works.
  - Example: RemoteControl
- Client
  - Client decides what to request and pass the request command to the Invoker.
  - Example: main() method

Design

- Decoupling

Object	Description	Diner	Remote Control
Client	Create the command object and set its receiver	Client	Recognize the function of the remote control button and press the button
Command	Define a binding between an action and a receiver	Order	Connect the actual object (TV, light, etc) to the button
Invoker	Connect the Command interface to take order and execute	Waitress	Press the remote control button to execute the function
Receiver	Perform an action	Cook	Actual objects such as TV, light, etc

Objectville Dinner and Command Pattern



Define Command Object

- Implementing the Command interface
  - All command objects implement the same interface, which consists of one method.
    - In the Dinner, we called this method orderUp().
    - Typically, we just use the method **execute()**.

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

Define Command Object

- Implementing a Command to turn a light on
  - The Light class has two methods, on() and off().

```
public class LightOnCommand implements Command {
    Light light; // specific light that is going
to be the Receiver of the request

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

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

Use Command Object

- Let's say we've got a remote control with only one button and corresponding slot to hold a device to control.

```
public class SimpleRemoteControl {
    Command slot;

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

RemoteControlTest

- SimpleRemoteControlTest

```
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();
}
```

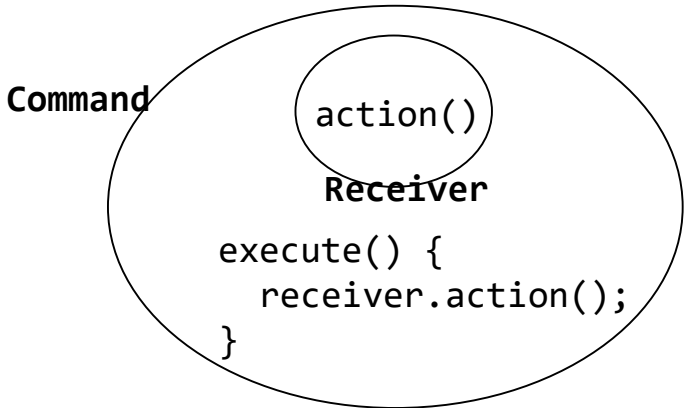
RemoteControlTest

- If you want to add GarageDoor

```
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();
}
```

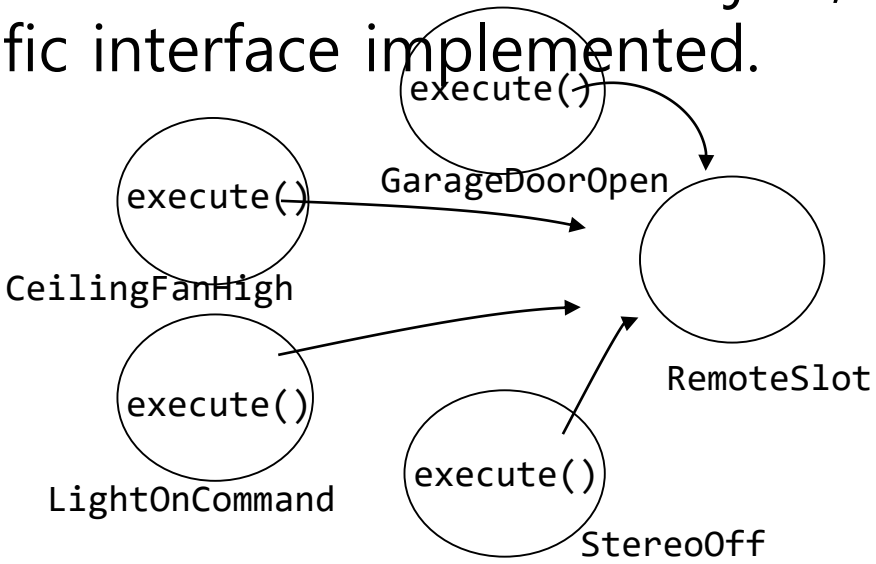
Command Pattern Defined

- Command object
  - A command object encapsulates a request by binding together a set of actions on a specific receiver.
  - To achieve this, it packages the actions and the receiver up into an object that exposes just one method, execute().
  - When called, execute() causes the actions to be invoked on the receiver.
  - From the outside, no other objects really know what actions get performed on what receiver; they just know that if they cal the execute() method, their request will be served.



Command Pattern Defined

- Parameterizing an object with a command
  - In the Diner, the Waitress was **parameterized with multiple orders** throughout the day.
  - In the simple remote control, we first loaded the button slot with a "light on" command, and then later replaced it with a "garage door open" command.
- Invoker(Waitress or remote control) doesn't need to know what actually happens in the command object, as long as it has a specific interface implemented.



Command Pattern Defined

```
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;
  }
```

Command Pattern Defined

```
  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 + "] " +
onCommands[i].getClass().getName() + "    " +
offCommands[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);
  }
}
```

RemoteControlTest

```
public class RemoteLoader {
  public static void main(String[] args) {
    RemoteControl remoteControl = new RemoteControl();
    Light livingRoomLight = new Light("Living Room");
    Light kitchenLight = 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);
```

RemoteControlTest

```
    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() {}
}
```

Adding Undo

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

public class LightOnCommand implements Command {
    Light light; // light is Receiver

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

Adding Undo

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

Command Pattern Define

```
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() {
    // rest of code…
  }
}
```

```
public class RemoteLoader {
  public static void main(String[] args) {
    RemoteControlWithUndo remoteControl = new
RemoteControlWithUndo();
    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();
  }
}
```

Adding Undo for Ceiling Fan Command

```
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; // set speed to HIGH
  }
  public void medium() { speed = MEDIUM; }
  public void low() { speed = LOW; }
  public void off() { speed = OFF; }
  public int getSpeed() { return speed; }
}
```

Adding Undo for Ceiling Fan Command

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

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

Adding Undo for Ceiling Fan Command

```
  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();
    }
  }
}
```

RemoteControlWithUndoTest

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

RemoteControlWithUndoTest

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

MacroCommand

- Add the macro command that darkens the light by pressing a button, turns on audio and TV, changes to DVD mode, and even fills the bathtub with water.

```
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();
    }
  }
}
```

MacroCommand

```
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);
```

MacroCommand

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

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