

# Command

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 17

# Acknowledgement & Materials Copyright

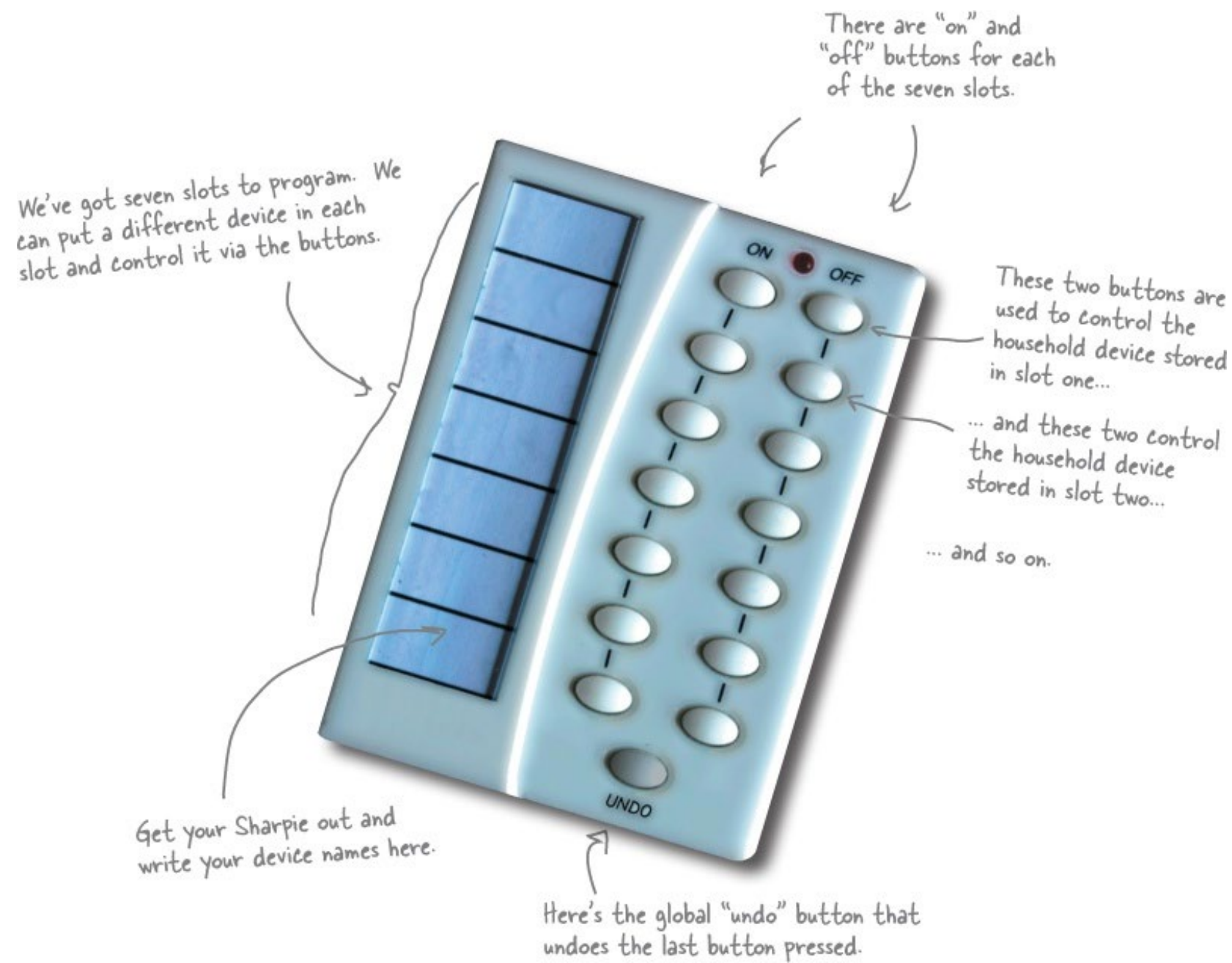
- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

# Command Pattern

- Today's discussion on Command is largely from Chapter 6 of the Head First Design Patterns book
  - Read through it when you can for more details...

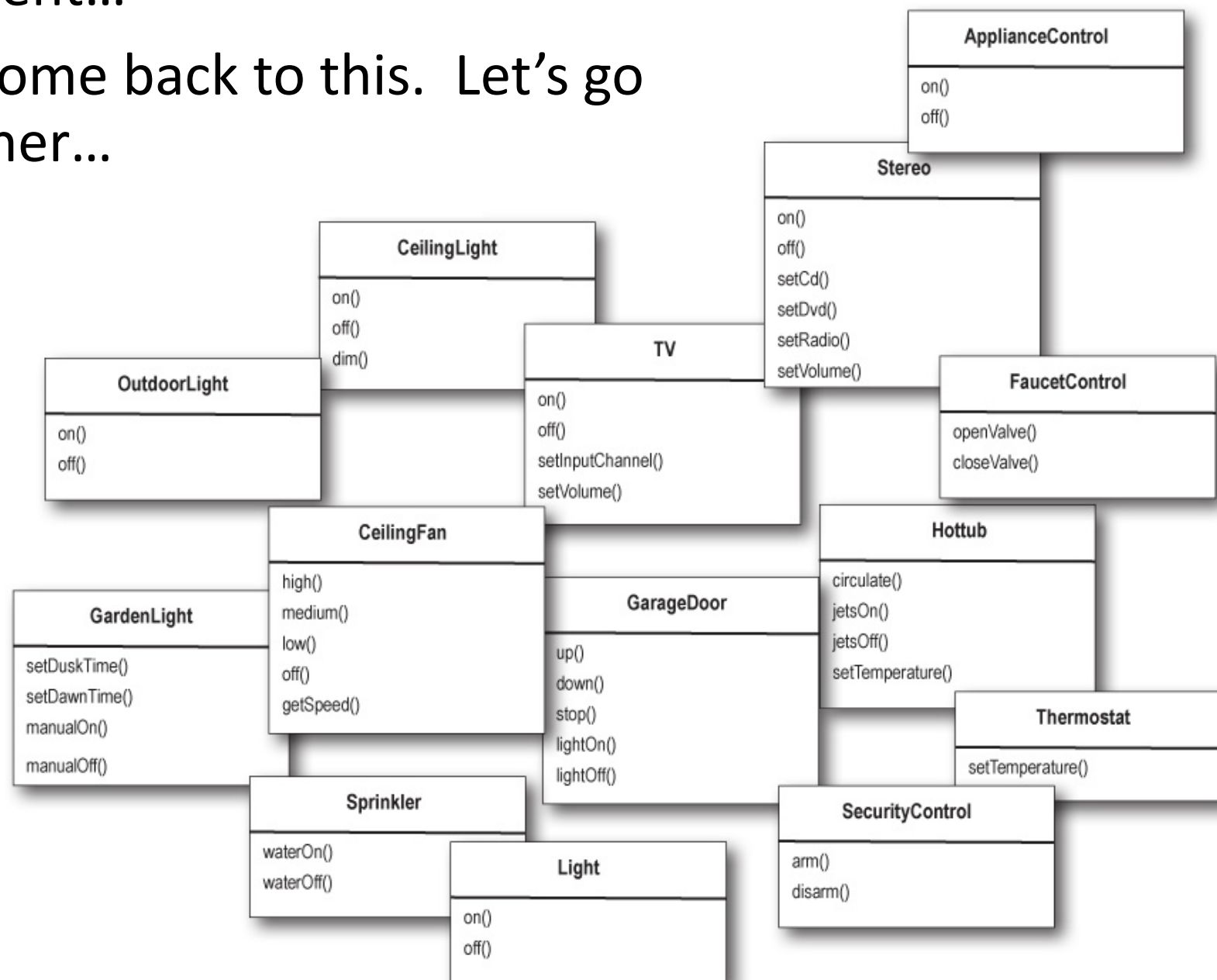
# The Problem – A Remote Control

- The book presents a remote control with 7 programmable slots, each with an on and off button, plus a global undo button



# The Other Problem – Many Different Devices

- The book presents a set of classes for the different commands each device controlled by the remote can respond to, and they're not consistent...
- We'll come back to this. Let's go to a diner...



# The Diner

- Customer -> Order -> Waitress -> Order Slip
- Waitress -> Order Slip -> Order Counter
- Cook -> gets Order Slip -> makes Order



# The Command Pattern is Here!

- Customer -> Order -> Waitress -> Order Slip
- Waitress -> Order Slip -> Order Counter
- Cook -> gets Order Slip -> makes Order
  
- Customer is a **Client**, needs this action, an Order, to be executed
- Order Slip – Encapsulates a Request – a **Request** object
  - It has one method – OrderUp() – containing the actions needed to prepare the Order
  - In fact, the Waitress does not need to know what's in the Order or who prepares the Order, they just have to deliver, or invoke, the request
- The Waitress is the **Invoker**
  - The Cook doesn't really care who asked for the Order, they just need to see an Order has arrived and act on it per the Order Slip
- The Cook is the **Receiver** – they do the action outlined in the request

# Command Pattern

- Start at Client
- **Client** Object – CreateCommandObject() - a Request
  - Command\_Object knows
    - Who is the target Receiver
    - What actions do I need the Receiver to execute? (Receiver methods)
- **Command\_Object** – defines Execute()
  - Command.Execute() will be called to invoke specified Receiver actions
- Client Object - Invoker.SetCommand(Command\_Object)
  - Client tells Invoker I have a Command\_Object for you
- **Invoker** – Command\_Object.Execute()
  - The Invoker calls Command\_Object.Execute()...
  - What the timing is of executing that command may vary
- **Receiver** – executes the Receiver actions in the Command\_Object when the Invoker says to



# Implementing a Command Interface and a Command

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

```
public class LightOnCommand implements Command {  
    Light light; //reference to the command Receiver  
  
    // constructor – sets the specific light to command  
    public LightOnCommand(Light light) {  
        this.light = light;  
    }  
  
    // override for execute with specifically what action the light needs to do  
    public void execute() {  
        light.on();  
    }  
}
```

# Using the Command Object

- If we had a remote control with one button, we could have it hold a command and control a device. The remote is the Invoker:

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

# Testing a Command

```
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();  
    }  
}
```

This is our Client in Command Pattern—speak.

The remote is our Invoker; it will be passed a command object that can be used to make requests.

Now we create a Light object. This will be the Receiver of the request.

Here, create a command and pass the Receiver to it.

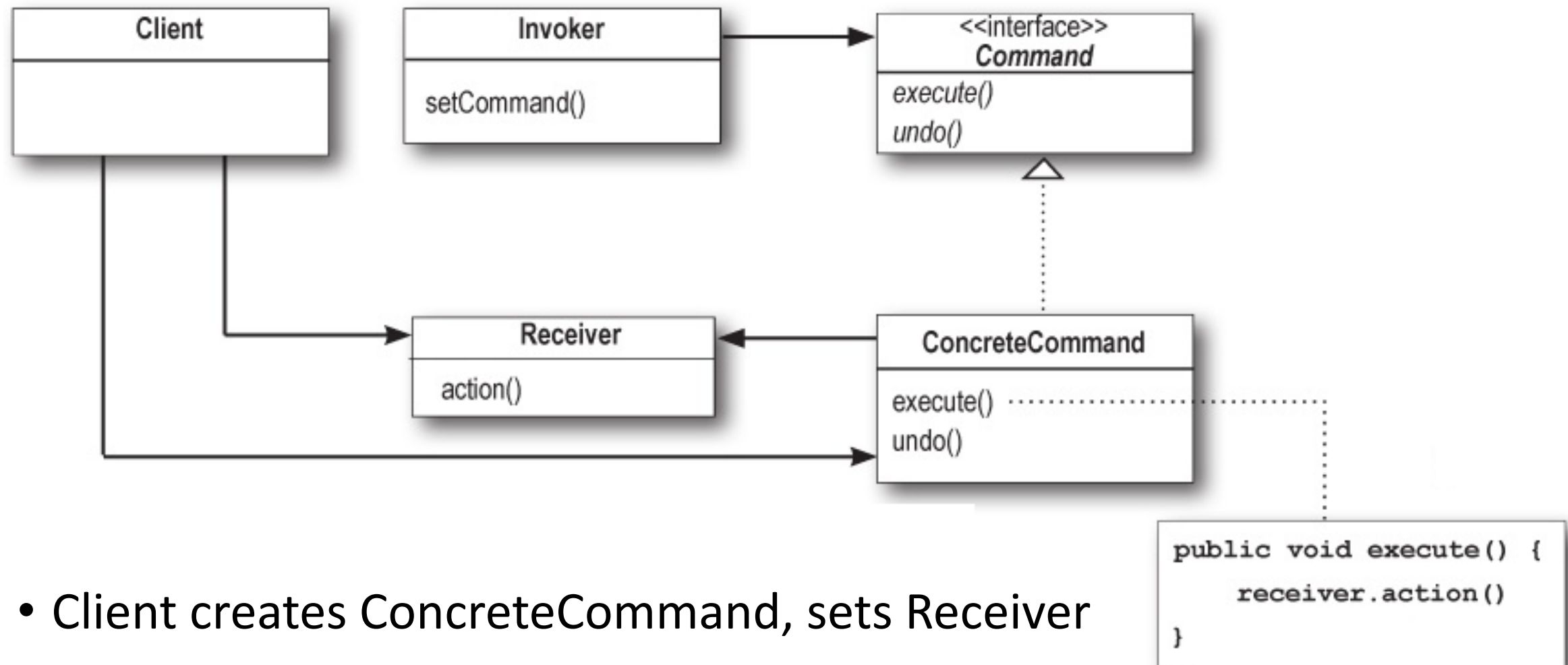
Here, pass the command to the Invoker.

And then we simulate the button being pressed.

Here's the output of running this test code.

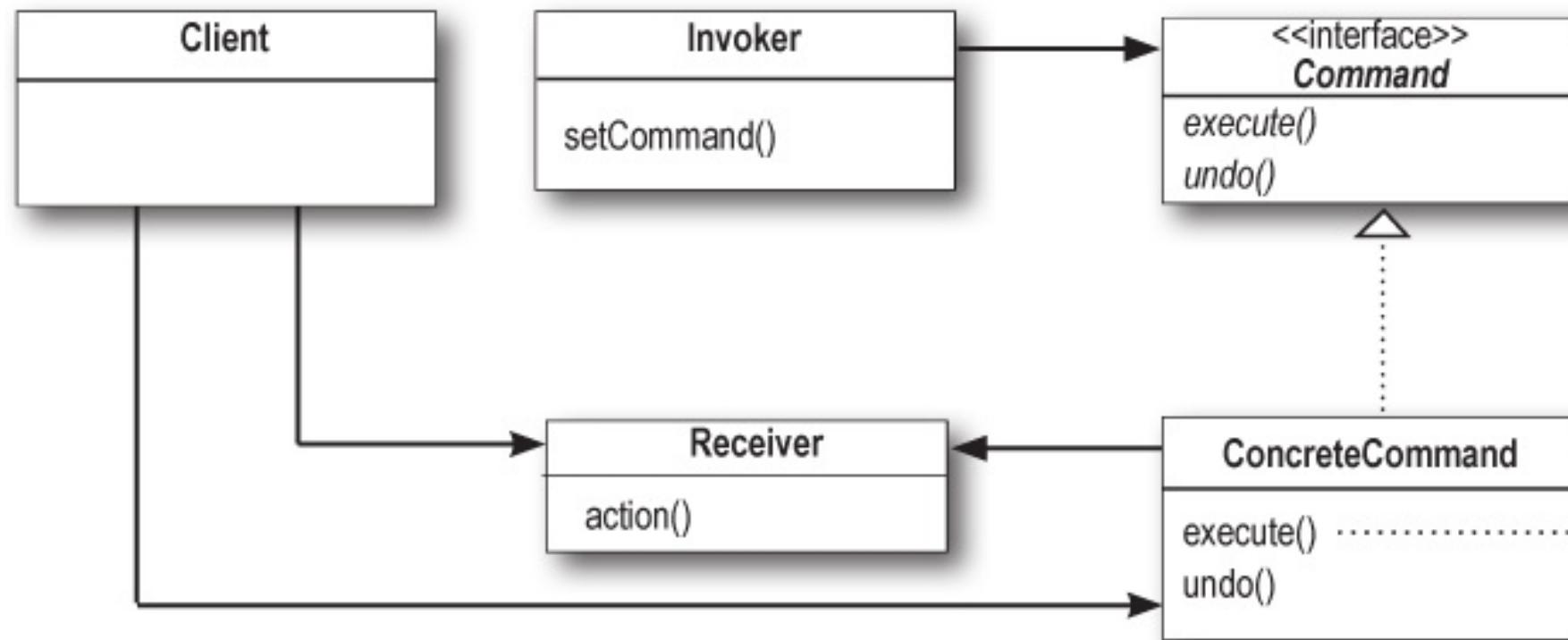
```
File Edit Window Help DinerFoodYum  
%java RemoteControlTest  
  
Light is On  
  
%
```

# UML for Command Pattern



- Client creates ConcreteCommand, sets Receiver
- Invoker holds command, calls execute()
- Command interface specifies common command methods
- ConcreteCommand is the binding between Receivers and actions
- Receiver provide actions to execute (any class can be a Receiver)

# UML for Command Pattern



Note here that the command is delegating the `action()` to the receiver set by the client. If the action was simple, the command could do it directly (eliminates receiver)...

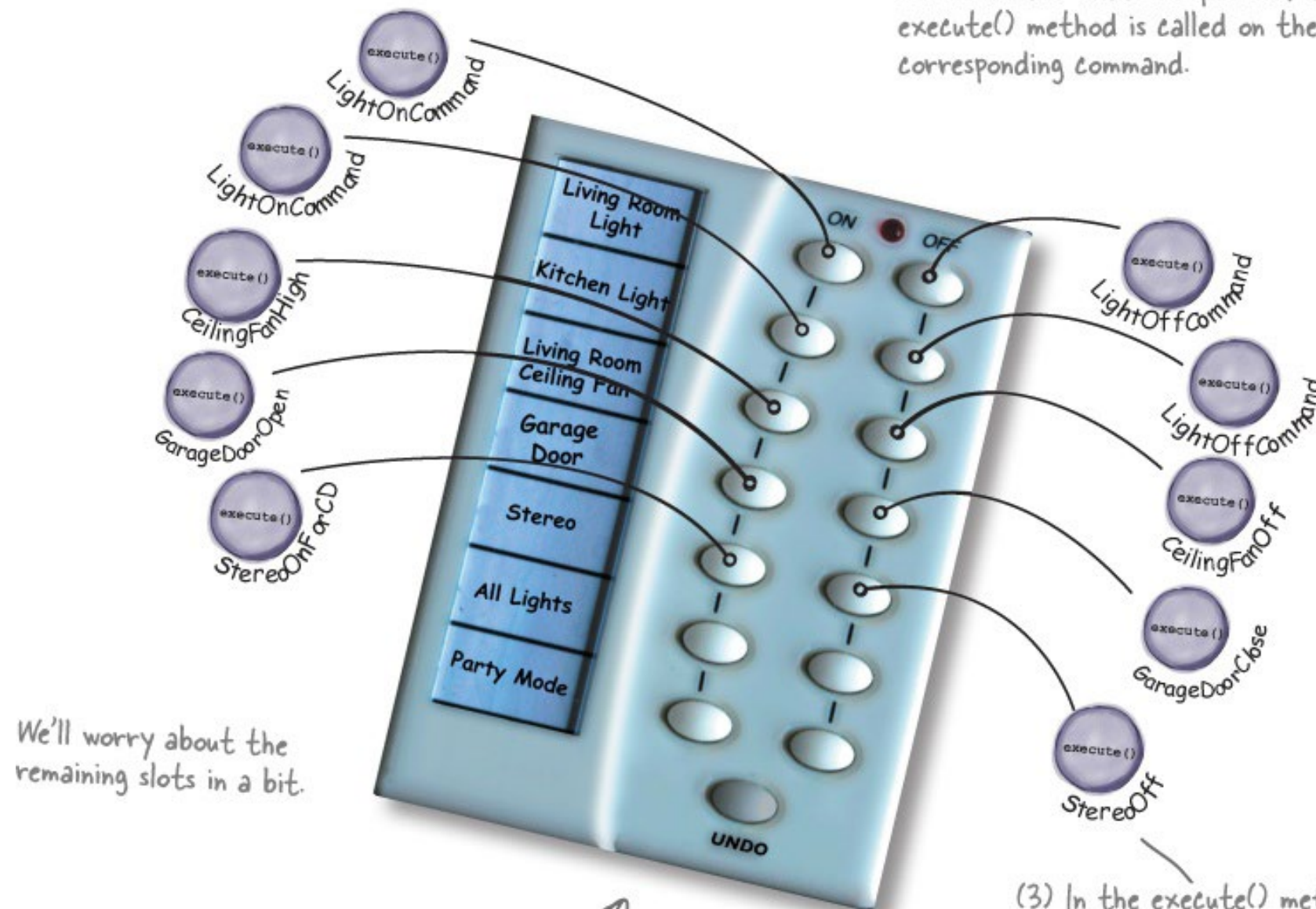
```
public void execute() {
    receiver.action()
}
```

- Client creates **ConcreteCommand**, sets **Receiver**
- **Invoker** holds command, calls `execute()`
- **Command** interface specifies common command methods
- **ConcreteCommand** is the binding between **Receivers** and actions
- **Receiver** provide actions to execute (any class can be a **Receiver**)

# That Remote Control...

(1) Each slot gets a command.

(2) When the button is pressed, the `execute()` method is called on the corresponding command.



We'll worry about the remaining slots in a bit.

(3) In the `execute()` method actions are invoked on the receiver.



You can look in the book at the Invoker code for the remote. It's big, but simple.

The Invoker

# More Complicated Commands

- Most of the devices execute in their implemented Command class just call on or off – but they could do more – consider the Stereo

Stereo
on() off() setCd() setDvd() setRadio() setVolume()

```
public class StereoOnWithCDCommand implements Command {  
    Stereo stereo;
```

```
    public StereoOnWithCDCommand(Stereo stereo) {  
        this.stereo = stereo;  
    }
```

```
    public void execute() {  
        stereo.on();  
        stereo.setCD();  
        stereo.setVolume(11);  
    }
```

```
}
```

Just like the LightOnCommand, we get passed the instance of the stereo we are going to be controlling and we store it in a local instance variable.

To carry out this request, we need to call three methods on the stereo: first, turn it on, then set it to play the CD, and finally set the volume to 11. Why 11? Well, it's better than 10, right?

# What about the remote buttons without commands?

- Create a command that does nothing:

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


- Sneakily, this is actually another pattern...
- This is a **Null Object Pattern**
- Null objects are used when you don't have anything to return, but you don't want the client to have to handle null cases



# What about the undo? Undo is a method in the command interface...

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

*execute() turns the light on, so undo() simply turns the light back off.*



- The opposite case for LightOff and undo is probably easy to see?

# Macro Commands

- Once we have a set of commands, it's easy to build combinations of them

```
Light light = new Light("Living Room");  
TV tv = new TV("Living Room");  
Stereo stereo = new Stereo("Living Room");  
Hottub hottub = new Hottub();
```

Create all the devices: a light,  
tv, stereo, and hot tub.

```
LightOnCommand lightOn = new LightOnCommand(light);  
StereoOnCommand stereoOn = new StereoOnCommand(stereo);  
TVOnCommand tvOn = new TVOnCommand(tv);  
HottubOnCommand hottubOn = new HottubOnCommand(hottub);
```

Now create all the On  
commands to control them.

```
Command[] partyOn = { lightOn, stereoOn, tvOn, hottubOn};  
Command[] partyOff = { lightOff, stereoOff, tvOff, hottubOff};
```

Create an array for  
On and an array for  
Off commands...

```
MacroCommand partyOnMacro = new MacroCommand(partyOn);  
MacroCommand partyOffMacro = new MacroCommand(partyOff);
```

...and create two  
corresponding macros  
to hold them.

# That's a LOT of little Command Classes!

Light
on() off()

- Maybe we could use **lambda expressions**?
- `Light livingRoomLight = new Light("Living Room");`
- Now, when I set the command in the Invoker, instead of passing commands, I could pass lambdas.

`remoteControl.setCommand(0, () -> { livingRoomLight.on(); }, () -> { livingRoomLight.off(); } );`

Here are the two lambda expressions.

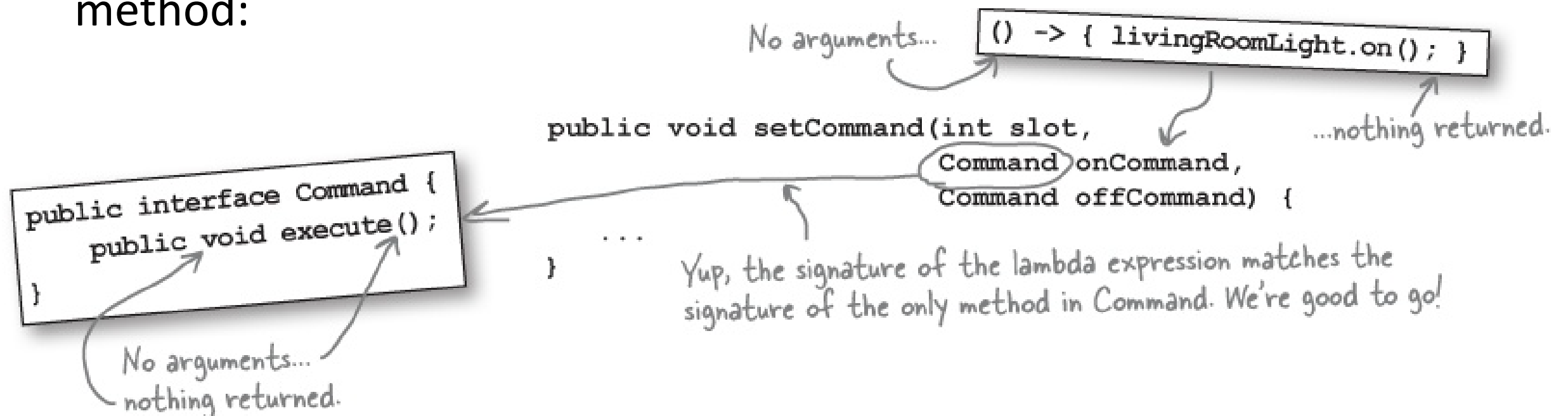
The lambdas get passed as commands to `setCommand`.

```
public void setCommand(int slot, Command onCommand, Command offCommand) {  
    onCommands[slot] = onCommand;  
    offCommands[slot] = offCommand;  
}
```

- How could this work if the system is looking for an execute method to call for normal Command objects?

# Lambda Magic

- The Lambda Expressions can stand in for a Command object if that Command interface has **one** method: execute() ...AND...
- The Lambda Expression must have the same signature as that one method:




- The compiler will look to see that Command has one method with a matching signature, and will use the lambda instead

# Java Method References


- If the lambda you're passing in has just one method, you can use a method reference to replace a single-method lambda expression
- Looks like this:

```
remoteControl.setCommand(0, livingRoomLight::on, livingRoomLight::off);
```

This is a reference to the `on()` method  
of the `livingRoomLight` object.



This is a reference to the `off()`  
method of the `livingRoomLight` object.



- Maybe  
    `livingRoomLight::on`
- is a little cleaner than  
    `() -> { livingRoomLight.on(); }`
- but otherwise, not much different...

# Multiple actions in a Lambda

- If the lambda signature matches the signature of the one method we're calling (in this case, `execute()`), we can bundle up multiple actions...

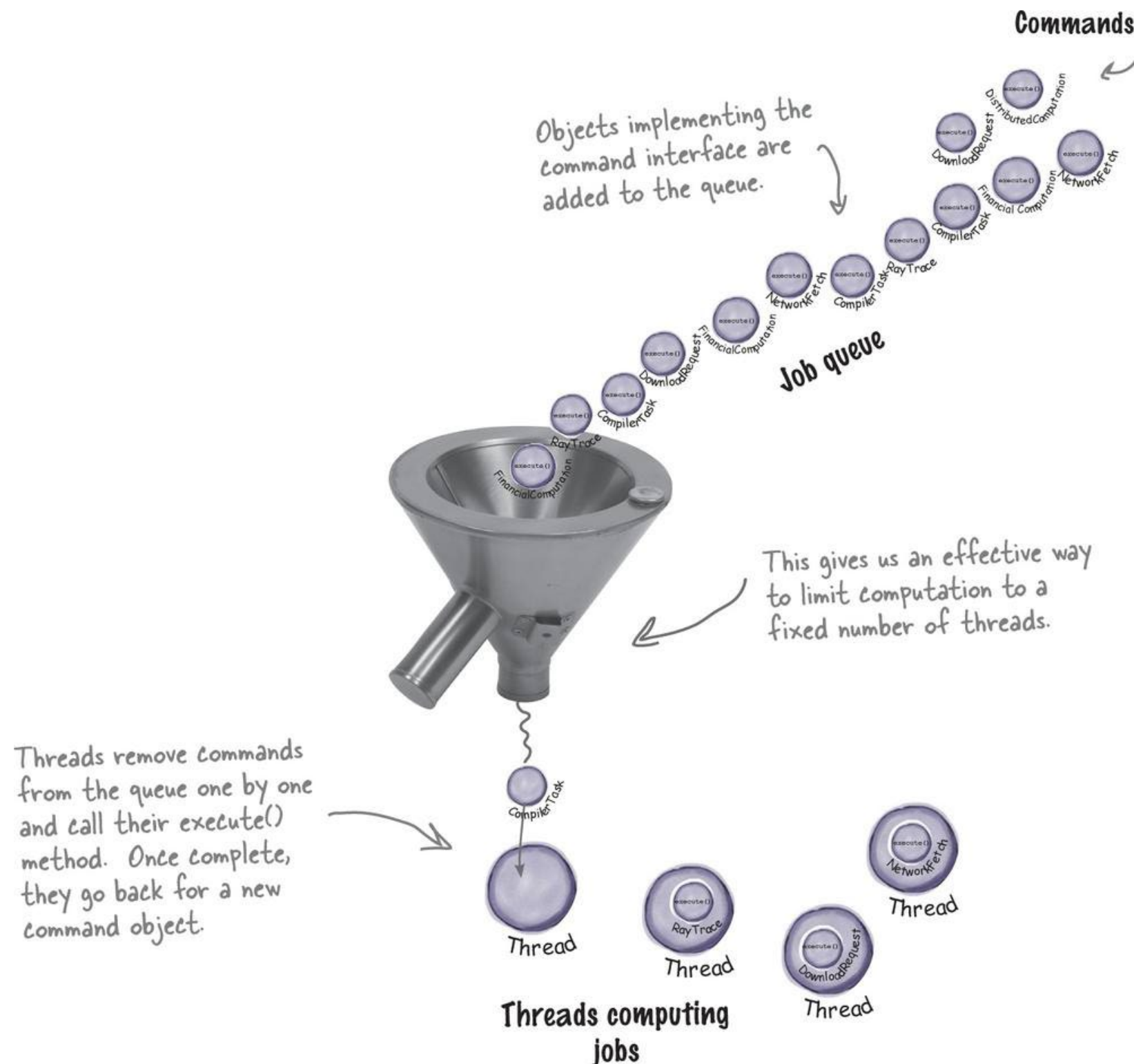
```
Command stereoOnWithCD = () -> {  
    stereo.on(); stereo.setCD(); stereo.setVolume(11);  
};  
remoteControl.setCommand(3, stereoOnWithCD, stereo::off);
```

← This lambda expression does three things (just like the `stereoOnWithCDCommand`'s `execute()` method did).

← We can pass the lambda expression using its name.

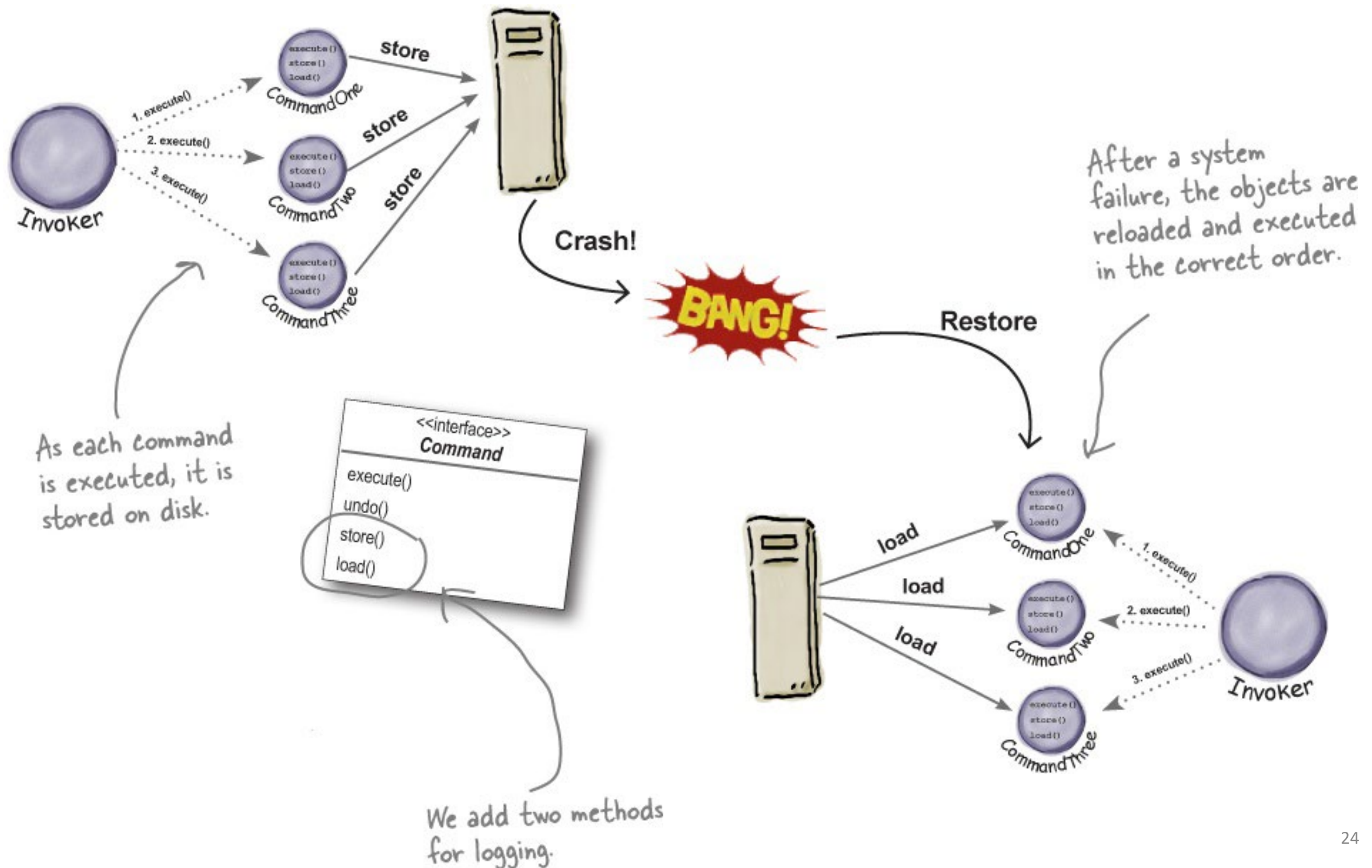
- Using lambdas can really drive down the number of classes in an implementation. For the remote control example, it goes from 22 classes down to 9.
- Note that lambdas can have 0 to n parameters and return values (see the Java docs)
- Also note that this lambda implementation only works if there is a single method in the Command interface, like `execute()`. If you add `undo()`, you'll have to consider other implementation...

# Other uses for Command: Queuing Requests



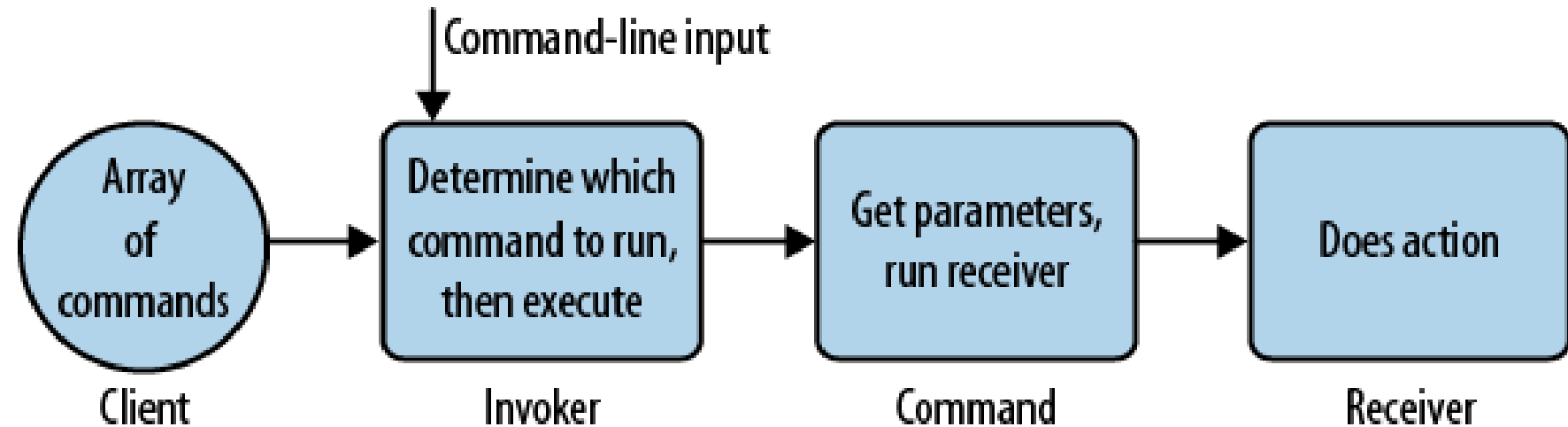


# Other uses for Command: Logging Requests

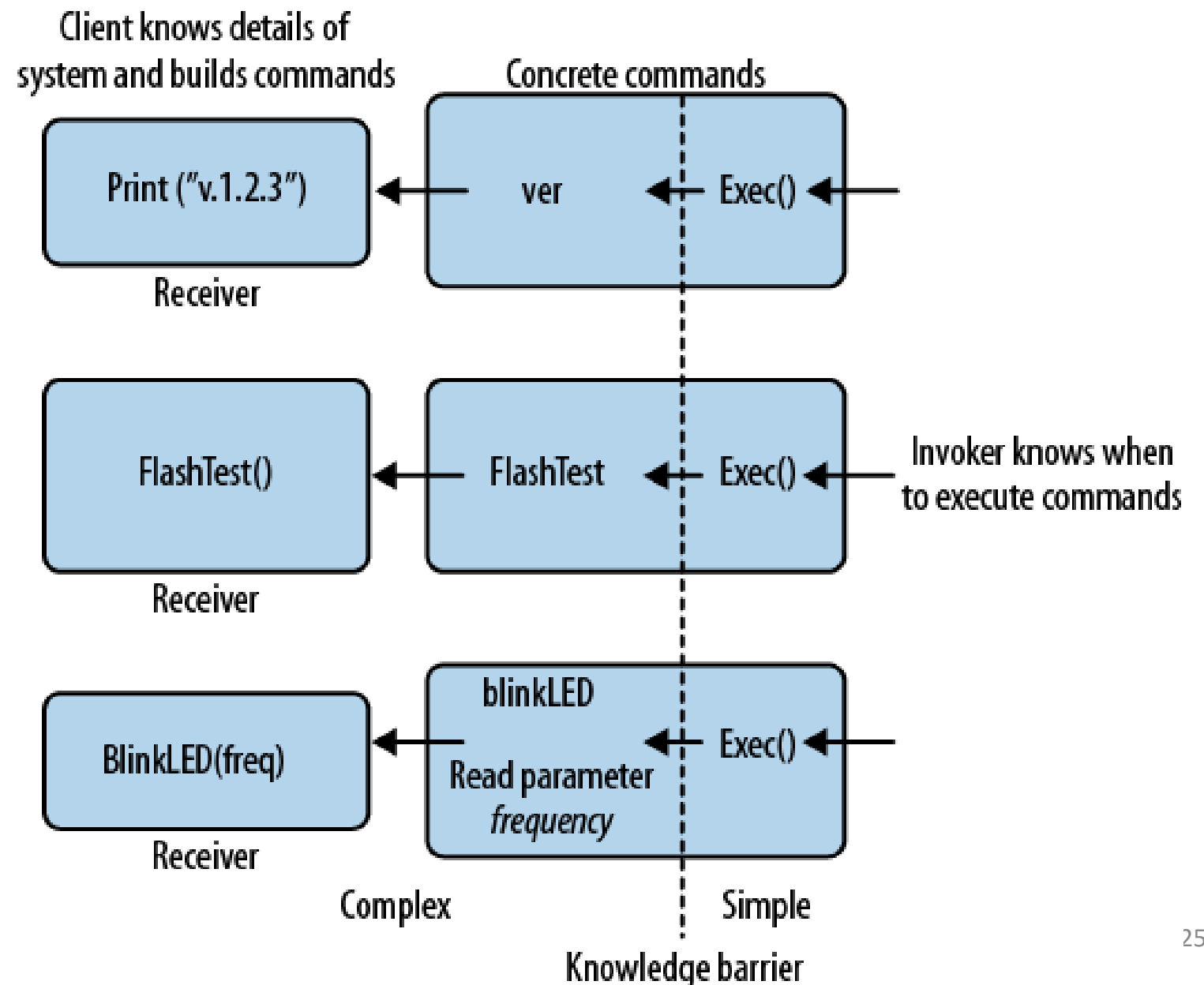




# One more...



- The Command Pattern popped up in my Firmware class...
  - From Making Embedded Systems, Elecia White, 2011, O'Reilly
- The pattern was used to create a framework of commands to test new boards, and to be able to easily add more tests
- Commands are C structures with a function pointer for executing an operation



# Python Command Pattern Implementation

```
import abc
```

```
class Command(metaclass=abc.ABCMeta):
```

```
    # The command interface that declares a method (execute) for a particular action.
```

```
    @abc.abstractmethod
```

```
    def execute(self):
```

```
        pass
```

```
class Sandwich:
```

```
    # Receiver with method for action
```

```
    def make_sandwich(self):
```

```
        print("A sandwich is being made")
```

```
class SandwichCommand(Command):
```

```
    #A concrete / specific Command class, implementing execute()
```

```
    def __init__(self, sandwich: Sandwich):
```

```
        self._sandwich = sandwich
```

```
    def execute(self):
```

```
        self._sandwich.make_sandwich()
```

<https://medium.com/@rrfd/strategy-and-command-design-patterns-wizards-and-sandwiches-applications-in-python-d1ee1c86e00f>

# Python Command Pattern Implementation

```
class MealInvoker:
```

```
    # Has a reference to the Command, and can execute the method
```

```
    def __init__(self, command: Command):
```

```
        self._command = command
```

```
        self._command_list = [] # type: List[Command]
```

```
    def set_command(self, command: Command):
```

```
        self.command = command
```

```
    def get_command(self):
```

```
        print(self.command.__class__.__name__)
```

```
    def add_command_to_list(self, command: Command):
```

```
        self._command_list.append(command)
```

```
    def execute_commands(self):
```

```
        # Execute all the saved commands, then empty the list.
```

```
        for cmd in self._command_list:
```

```
            cmd.execute()
```

```
        self._command_list.clear()
```

```
    def invoke(self):
```

```
        self._command.execute()
```

```
    # Command pattern in action
```

```
    sandwich = Sandwich() # receiver
```

```
    command_sandwich = SandwichCommand(sandwich) # concrete command
```

```
    meal_invoker = MealInvoker(command_sandwich) # invoker
```

```
    meal_invoker.invoke() # Starting the method calls
```

```
    meal_invoker.add_command_to_list(command_sandwich)
```

```
    meal_invoker.execute_commands()
```

```
>> A sandwich is being made
```

```
>> A sandwich is being made
```

<https://medium.com/@rrfd/strategy-and-command-design-patterns-wizards-and-sandwiches-applications-in-python-d1ee1c86e00f>

# Command Summary

- The Command Pattern decouples an object making a Request from the one that knows how to perform it
- A Command object is at the center of this decoupling and encapsulates a Receiver with an action (or set of actions)
- An Invoker makes a request of a Command object by calling its execute() method, which invokes those actions on the receiver
- Invokers can be parameterized with Commands, even dynamically at runtime
- Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called
- Meta or Macro Commands are a simple extension of Command that allow multiple commands to be invoked at once (by, for instance, creating an array of commands)
- In practice, it is not uncommon for “smart” Command objects to implement the request themselves rather than delegating to a receiver
- Commands may also be used to implement logging and transactional systems
- And don’t forget the secret Null Object Pattern we discovered

# Next Steps

- Wednesday AM first thing – An in class live bonus exercise on UML of patterns...
  - Review your patterns
  - Bring some paper/pencil
- If you're not on a team for projects, you should be! Use Piazza!
- Assignments
  - New participation topic up now on Piazza (your best work) – keep up with your responses
  - Project 3 part 2 on Wed 2/23, we'll also review Project 4 (the last music store code)
    - One part of Project 4 will be your proposal for your semester project topic (Project 5/6/7)
  - Quiz 5 is up now, due Thur 2/24
    - Quiz 6 goes up Sat 2/26 – Thur 3/3
    - Then Midterm exam on Canvas Sat 3/5 – Thur 3/10
  - The Graduate Research Draft Presentation is due Fri 3/11
  - Readings from the Head First Design Patterns textbook
    - Lecture 14 on Factory – Chapter 4
    - Lecture 16 on Singleton – Chapter 5
    - Lecture 17 on Command – Chapter 6
- Coming up
  - Command, Façade/Adapter, Expanding Horizons, Template...
- Please come find us for any help you need or questions you have!