CS 351
Design of Large Programs
Command Pattern

Brooke Chenoweth

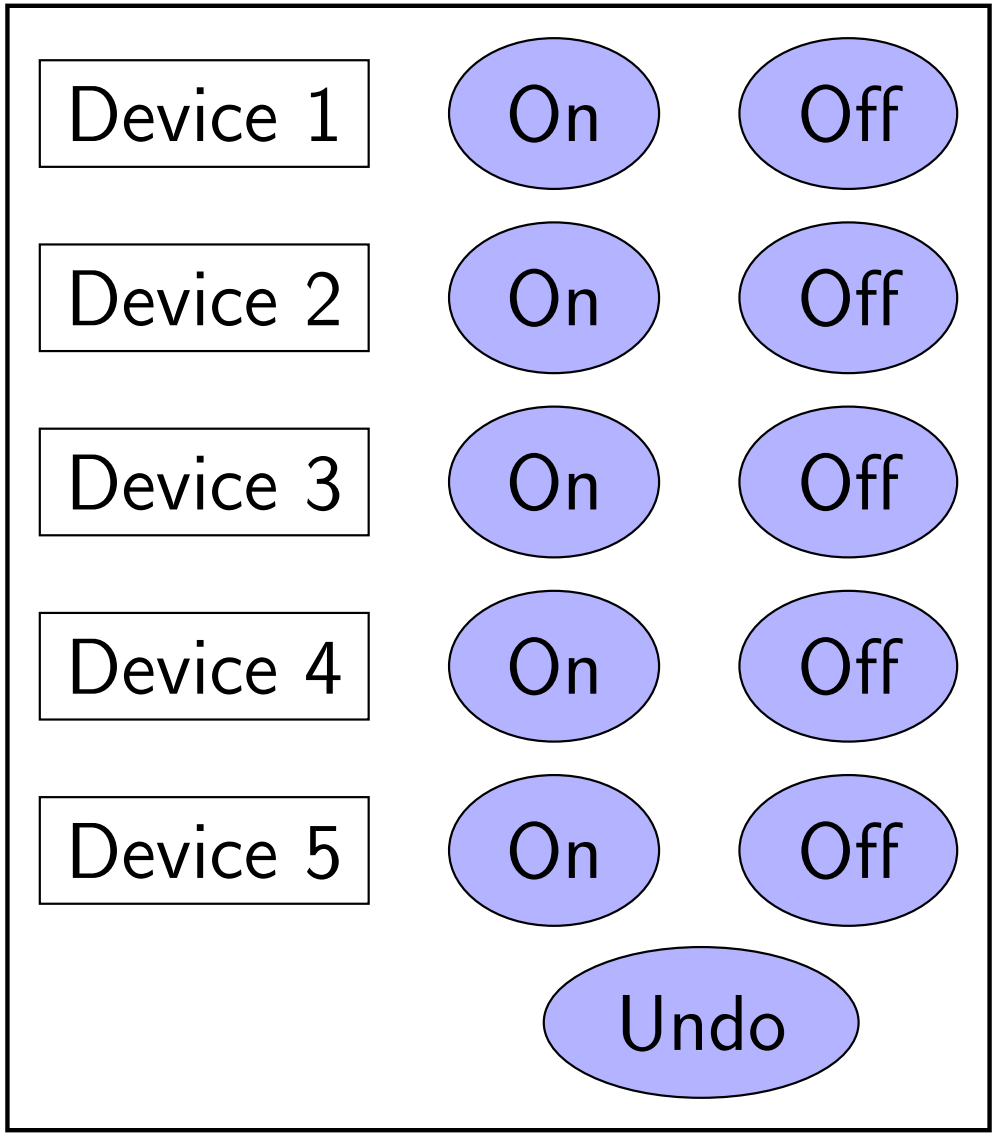
University of New Mexico

Spring 2024

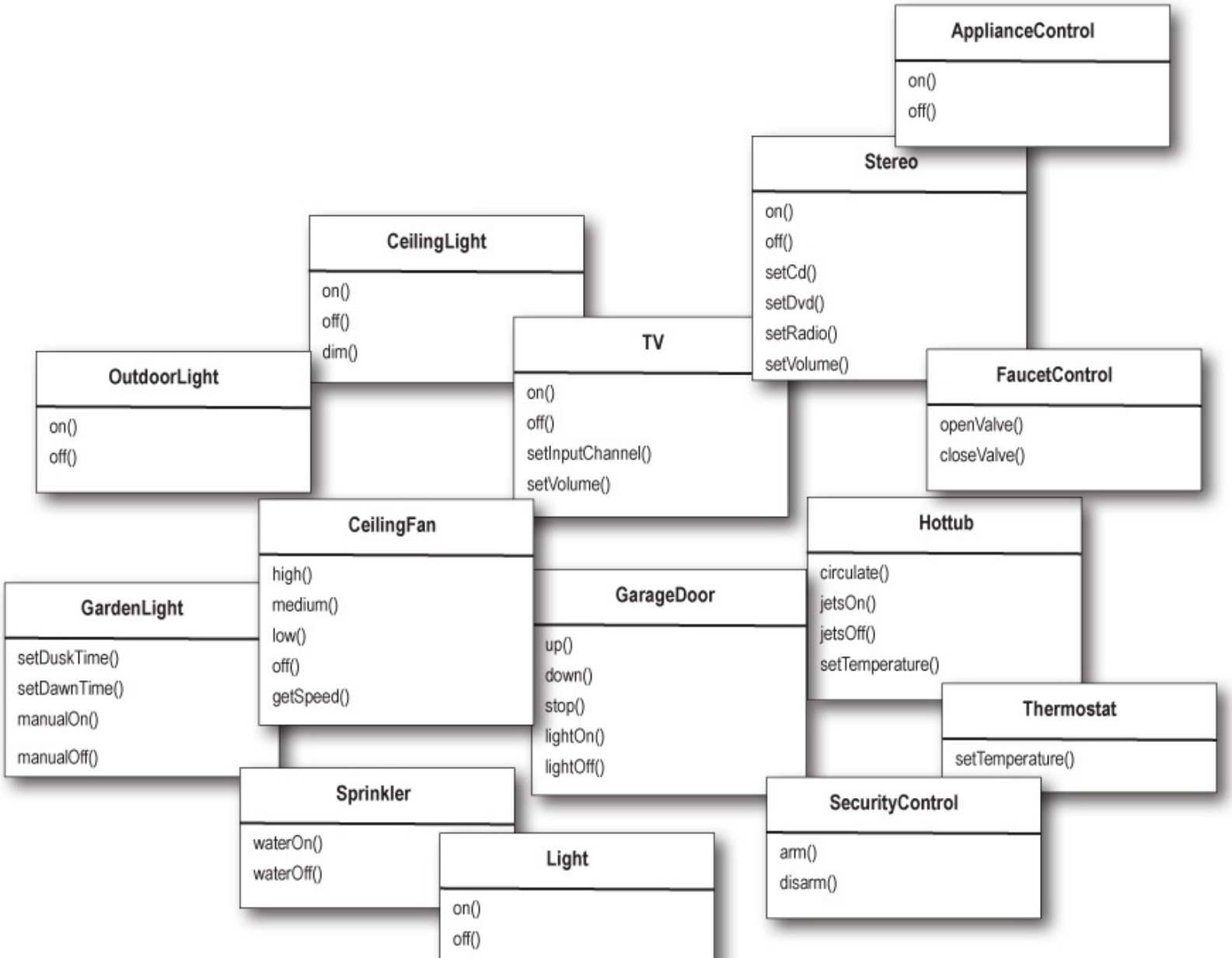
The Mission: A Remote Control

- Remote has multiple programmable slots. We can associate each with a different device.
- Each slot has on and and off buttons.
- Remote has a global undo button.
- Vendors have provided classes to control home automation devices.
- Can we make an API for programming the remote?

The Remote Control



The Vendor Classes



Towards a Design

- The remote is simple, but the devices are not!
- So many different method names!
- We need some *information hiding* and *separation of concerns*
- The remote shouldn't have a bunch of switch statements that select between devices...
- We really need to *decouple* the requester of the action from the object that performs the action

Command Objects (in context)

We introduce *command objects* into the design

- A command object encapsulates a request to do something (e.g., turn on a light) on a specific object (e.g., the living room lamp)
- We can then just store a command object for each button such that when the button is pressed, the command is invoked
- The button doesn't have to know anything about the command

An Analogy: Ordering in a Diner

1. The customer gives the server their order.
2. The server takes the order, places it on the counter, and says "Order up!"
3. The short-order cook prepares the meal from the order.

An Analogy: Ordering in a Diner

1. The customer gives the server their order.
    - An order slip encapsulates a request to prepare a meal. Its method `orderUp()` encapsulates the actions needed to prepare the meal; it also carries its own reference to the appropriate Cook
2. The server takes the order, places it on the counter, and says "Order up!"
3. The short-order cook prepares the meal from the order.

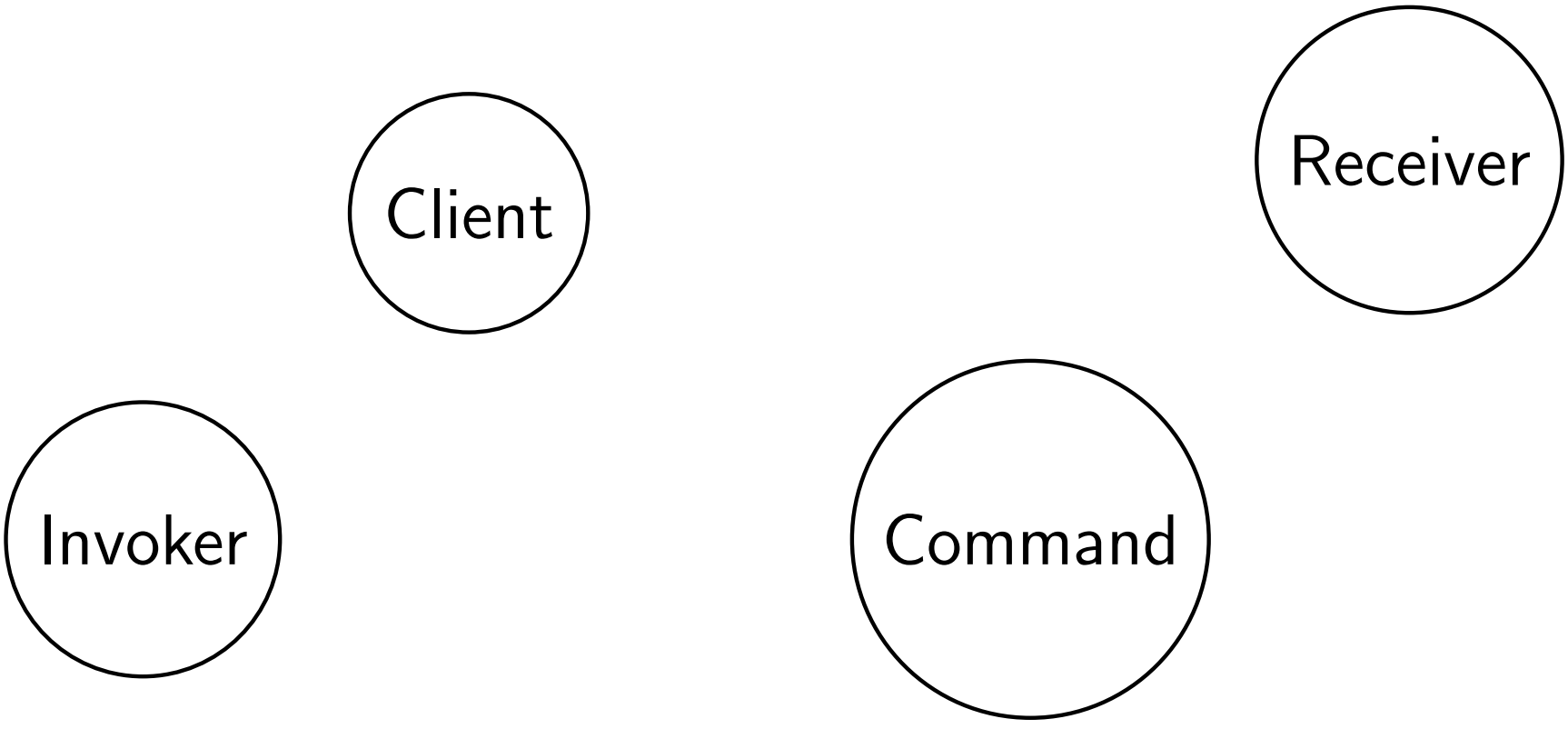
An Analogy: Ordering in a Diner

1. The customer gives the server their order.
   - An order slip encapsulates a request to prepare a meal. Its method `orderUp()` encapsulates the actions needed to prepare the meal; it also carries its own reference to the appropriate Cook
2. The server takes the order, places it on the counter, and says "Order up!"
   - The server just creates order slips and invokes the `orderUp()` method.
3. The short-order cook prepares the meal from the order.

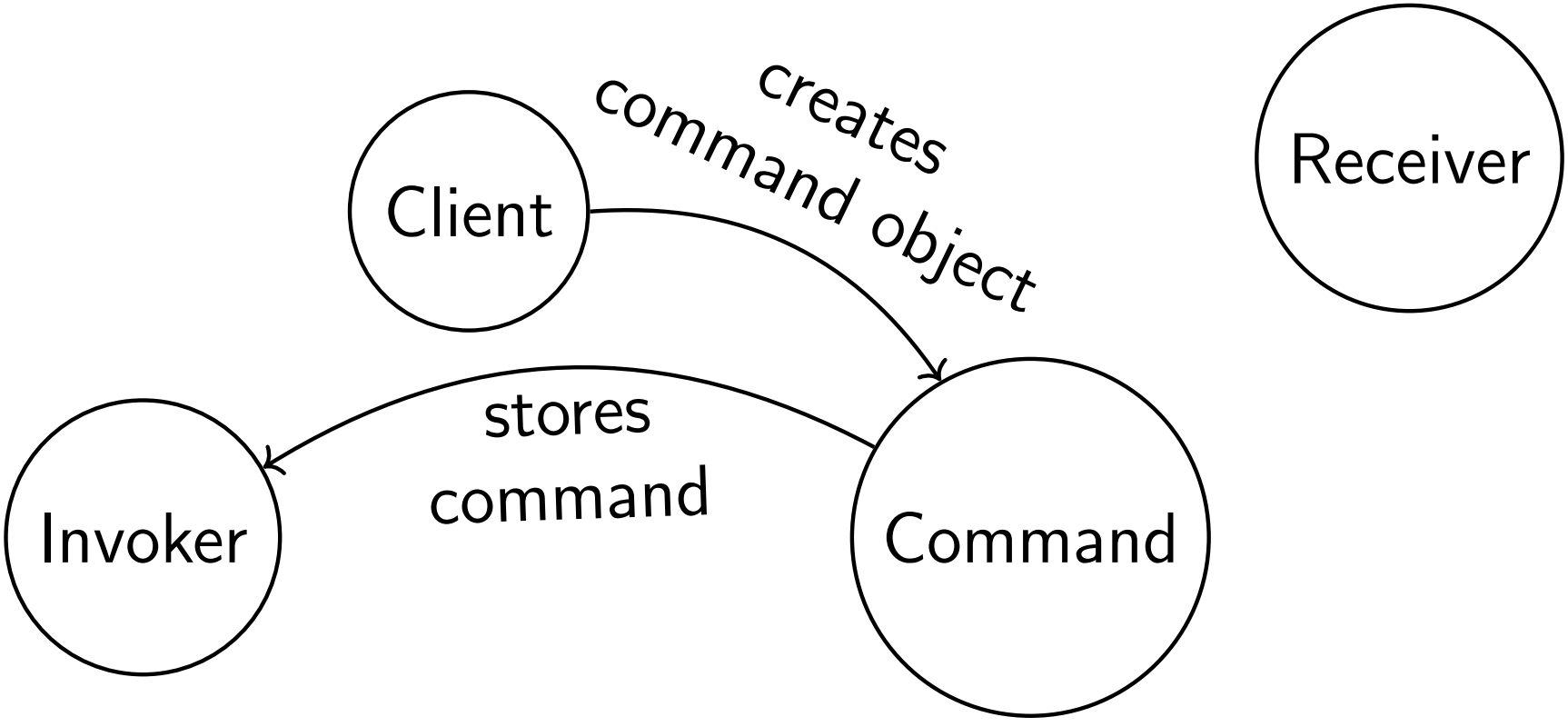
An Analogy: Ordering in a Diner

1. The customer gives the server their order.
   - An order slip encapsulates a request to prepare a meal. Its method `orderUp()` encapsulates the actions needed to prepare the meal; it also carries its own reference to the appropriate Cook
2. The server takes the order, places it on the counter, and says "Order up!"
   - The server just creates order slips and invokes the `orderUp()` method.
3. The short-order cook prepares the meal from the order.
   - The Cook knows how to prepare the meals; but is completely decoupled from the server (they need never directly communicate)

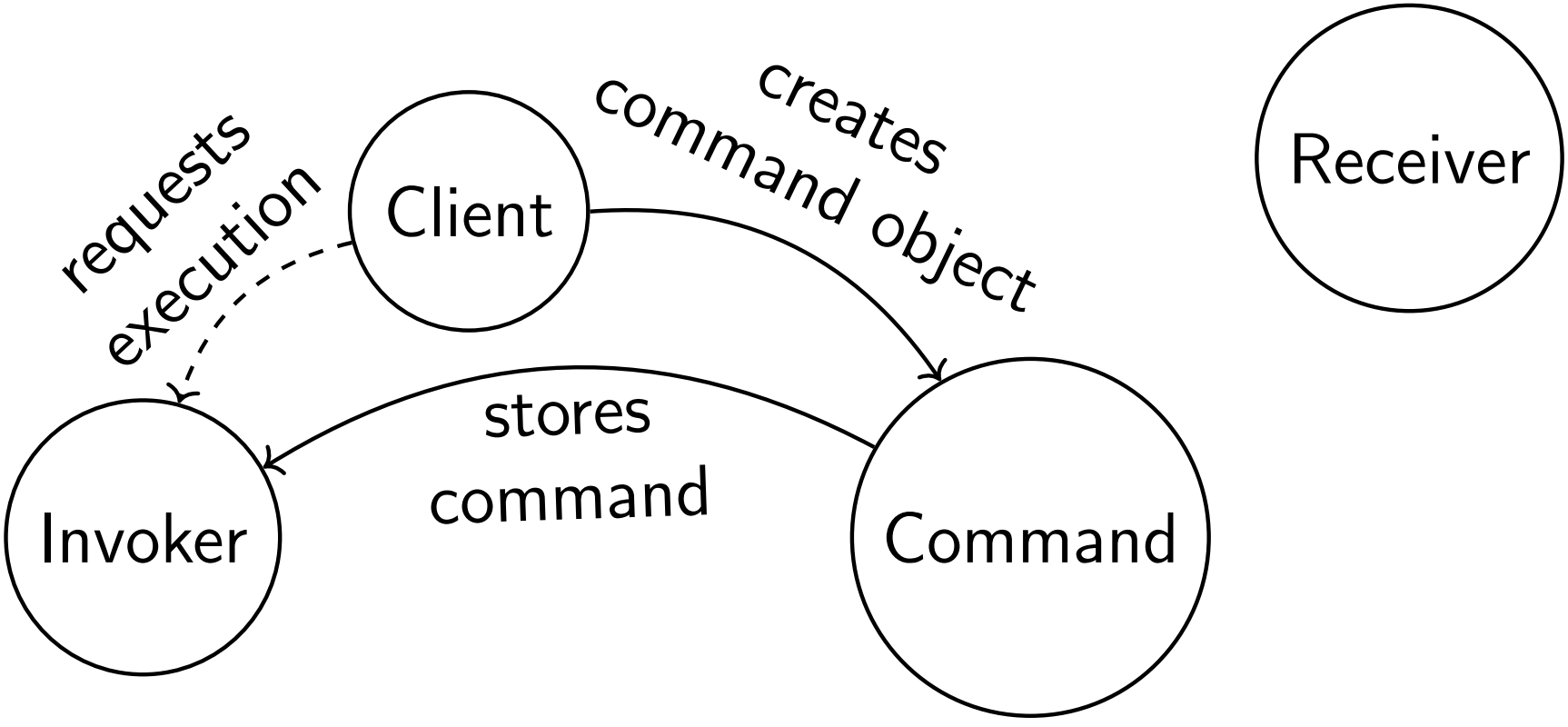
The Command Pattern in General



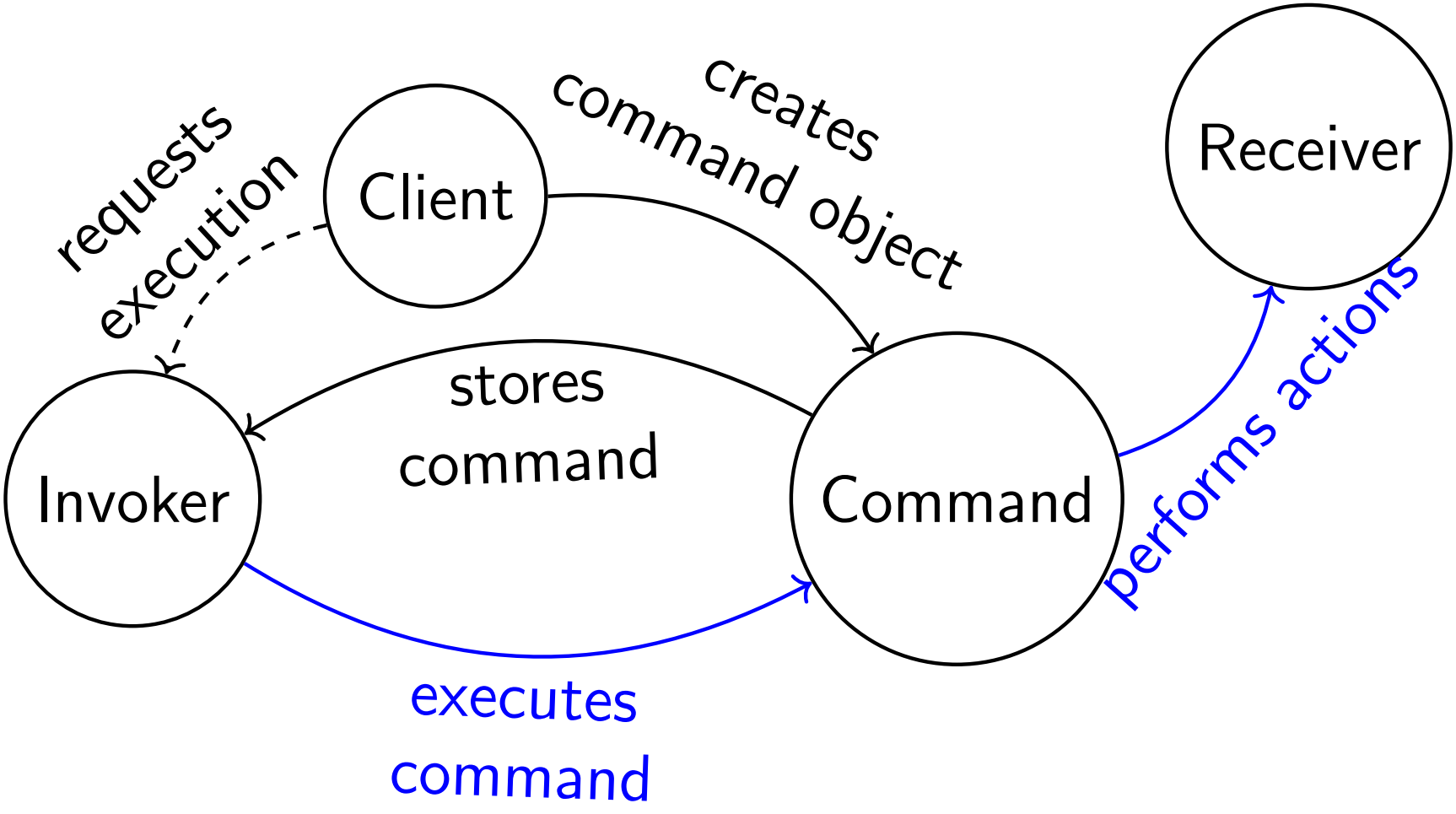
The Command Pattern in General



The Command Pattern in General



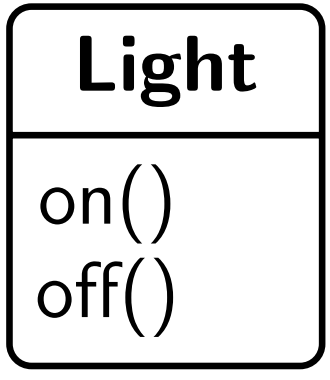
The Command Pattern in General



A First Command Object

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

```
public class LightOnCommand implements Command {
  private Light light;

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

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



Using the Command Object

```
public class SimpleRemoteControl {
  private Command slot;

  public SimpleRemoteControl () { }

  public void setCommand(Command command) {
    slot = command;
  }

  public void buttonWasPressed() {
    slot.execute();
  }
}
```

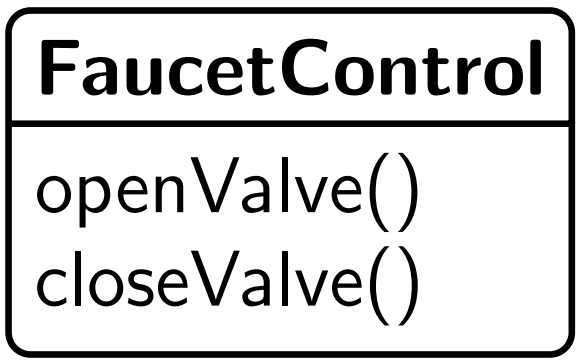
A Simple Test of the Remote

```
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();
  }
}
```

Can you do it?

- Implement the `FaucetOffCommand` class
- Here's the new test code:

```
public class RemoteControlTest {
  public static void main(String[] args) {
    SimpleRemoteControl remote =
            new SimpleRemoteControl();
    Faucet faucet = new FaucetControl();
    FaucetOffCommand faucetOff =
            new FaucetOffCommand(faucet);
    remote.setCommand(faucetOff);
    remote.buttonWasPressed();
  }
}
```



Solution

```
public class FaucetOffCommand implements Command {
  private FaucetControl faucet;

  public FaucetOffCommand(Faucet faucet) {
    this.faucet = faucet;
  }

  public void execute () {
    faucet.closeValve();
  }
}
```

Solution

```
public class FaucetOffCommand implements Command {
  private FaucetControl faucet;

  public FaucetOffCommand(Faucet faucet) {
    this.faucet = faucet;
  }

  public void execute () {
    faucet.closeValve();
  }
}
```

For a very simple command object, you might choose to use an anonymous inner class (perhaps using lambda syntax!) instead of a named implementation.

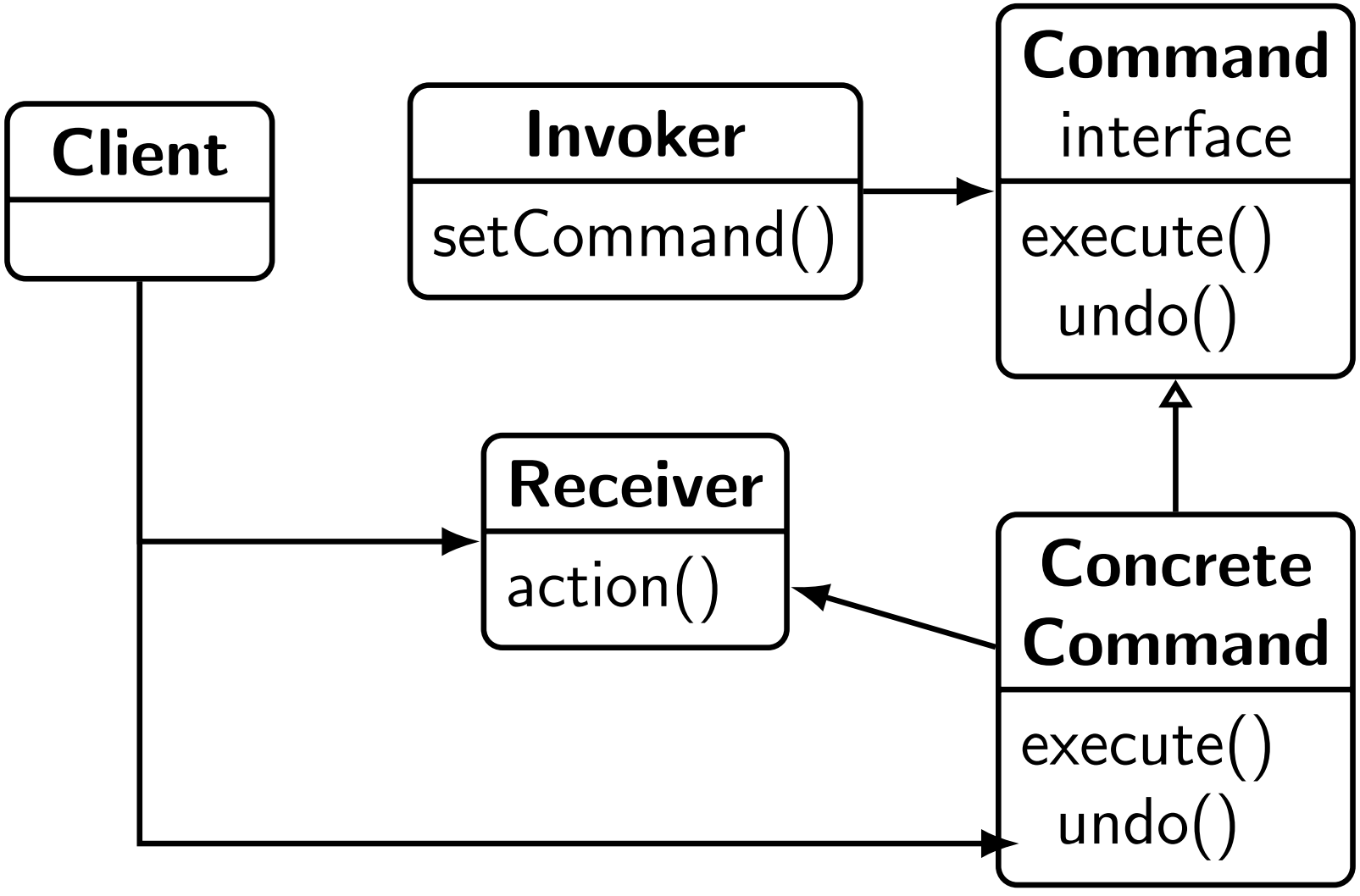
The Command Pattern

The *Command Pattern* encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

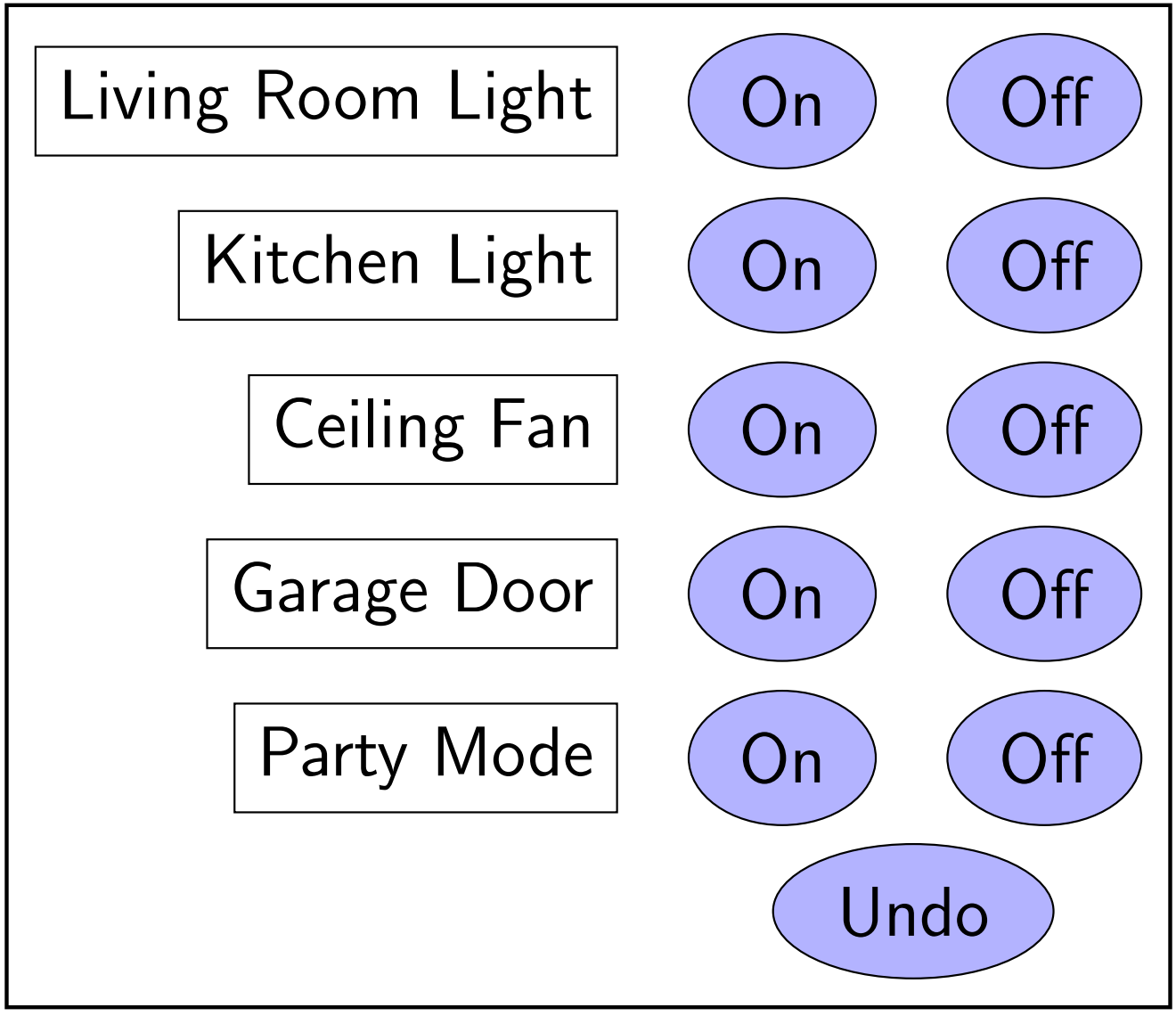
The Command Pattern

- The command object *encapulates a request* by binding together a set of actions on a specific receiver.
- Command object only exposes the execute method.
- When execute is called, causes actions to be invoked on receiver.
- Other objects don't know what actions get performed on what receiver; they just know that their request will be serviced if they call execute.

The Command Pattern Class Diagram



Back to our Remote...



Question

How does the remote know the difference between the kitchen light and the living room light?

Question

How does the remote know the difference between the kitchen light and the living room light?

- It doesn't have to! The Receiver is encapsulated in the command that is inserted in the slot.

Let's Add Support for the Undo Button

- First, let's expand the Command interface:

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

- Now, every Command should be undoable

Let's Add Support for the Undo Button

- So we add an implementation for `undo()` for every command that we implement
  - E.g., for `LightOnCommand`, `undo()` simply calls `light.off()`
  - Some `undo()` method implementations are more complicated; e.g., undoing a change in speed of a ceiling fan
- All that's left is to add support to the remote control to handle tracking which `undo()` method to call
  - Store the last command executed; if the undo button is pressed, we can just invoke the `undo()` method on that command

Light On with Undo

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

Remote with Undo

```
public class RemoteControl {
  private Command[] slots = new Command[10];
  private Command recent;

  public void setCommand(int index, Command command) {
    slot[index] = command;
  }

  public void buttonWasPressed(int index) {
    slot[index].execute();
    recent = slot[index];
  }

  public void undoWasPressed() {
    if(recent != null) {
      recent.undo();
    }
  }
}
```

More Questions

Do I *have* to have a Receiver?

More Questions

Do I *have* to have a Receiver?

- A Command object could just implement the execute() functionality itself; but having the Command object just pass the invocation from the Invoker to the Receiver gets the highest decoupling

More Questions

Do I *have* to have a Receiver?

- A Command object could just implement the execute() functionality itself; but having the Command object just pass the invocation from the Invoker to the Receiver gets the highest decoupling

How would you implement a history of undo operations?

Remote with Undo History

```
public class RemoteControl {
  private Command[] slots = new Command[10];
  private Deque<Command> stack = new LinkedList<>;

  public void setCommand(int index, Command command) {
    slot[index] = command;
  }

  public void buttonWasPressed(int index) {
    slot[index].execute();
    stack.push(slot[index]);
  }

  public void undoWasPressed() {
    if(!stack.isEmpty()) {
      Command recent = stack.pop();
      recent.undo();
    }
  }
}
```