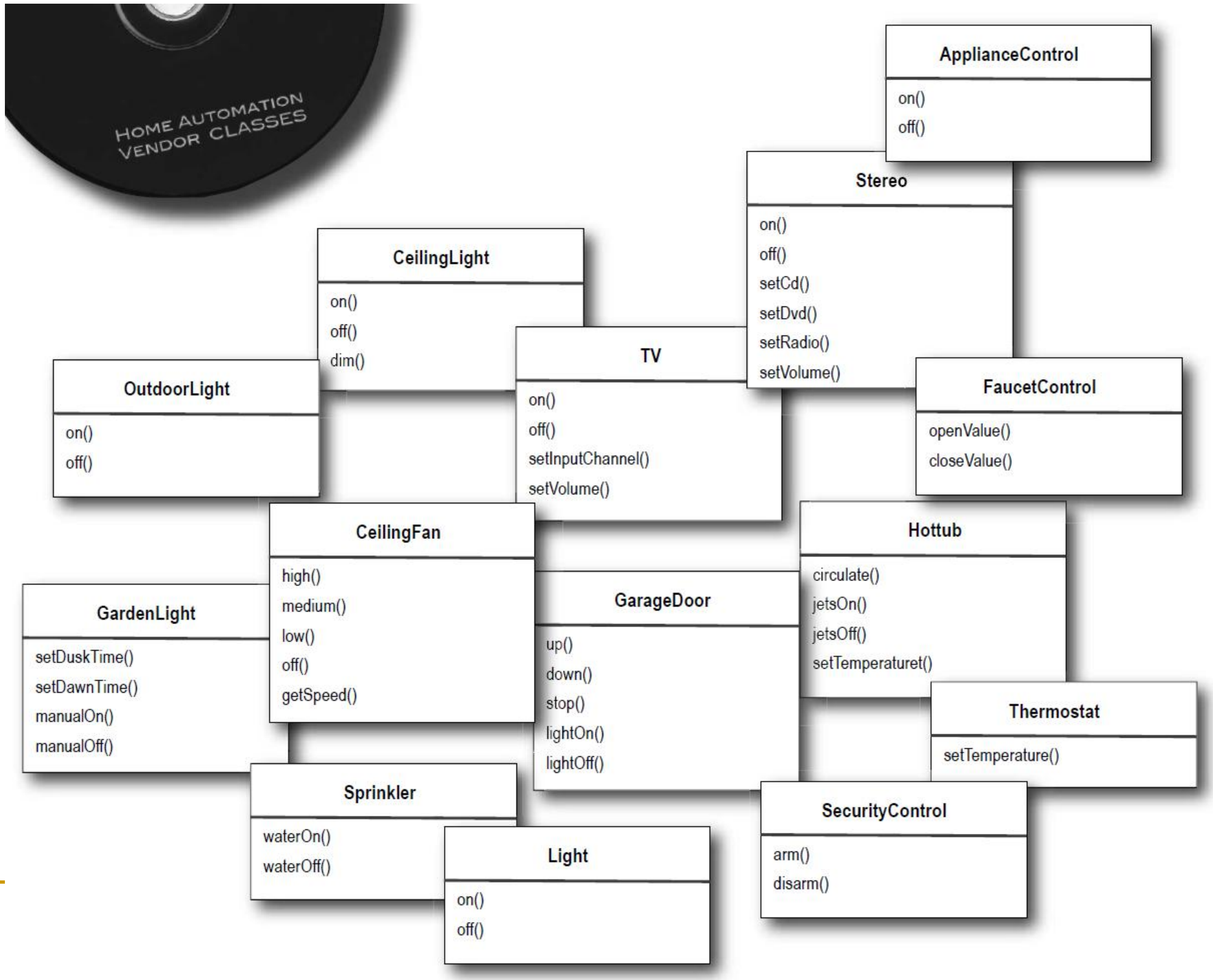
Command (命令模式, Behavioral Pattern)

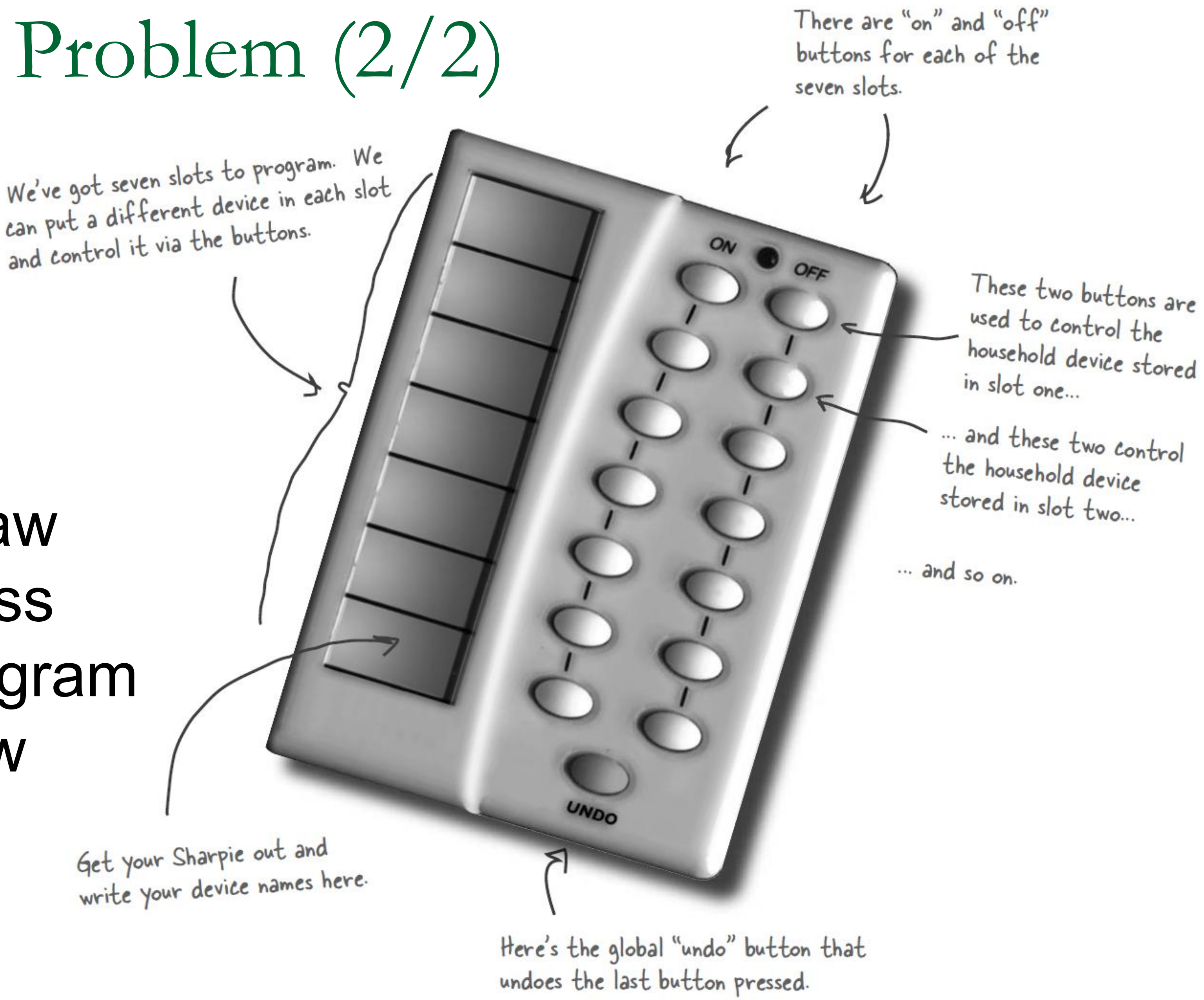
Kai SHI

The Problem (1/2)



The Problem (2/2)

- Draw class diagram now


There are "on" and "off" buttons for each of the seven slots.
We've got seven slots to program. We can put a different device in each slot and control it via the buttons.
ON
OFF
These two buttons are used to control the household device stored in slot one...
... and these two control the household device stored in slot two...
... and so on.
UNDO
Get your Sharpie out and write your device names here.
Here's the global "undo" button that undoes the last button pressed.

Command Pattern

- Intent
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable (redoable) operations.
  - 命令模式把一个请求封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。
- Also Known As
  - Action, Transaction

Motivation

- Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. For example, user interface toolkits include objects like buttons and menus that carry out a request in response to user input. But the toolkit can't implement the request explicitly in the button or menu, because only applications that use the toolkit know what should be done on which object.

Applicability (1/5): Use the Command pattern when:

- parameterize objects by an action to perform.
  - You can express such parameterization with a callback function, that is, a function that's registered somewhere to be called at a later point.
  - Commands are an object-oriented replacement for callbacks.

What is a **callback**?

Xiaoming didn't understand a math problem, so he called his classmate Xiaojun. After listening to the question, Xiaojun felt that he could not answer it immediately. Xiaoming told Xiaojun his phone number, and after Xiaojun finished, he called Xiaoming and told Xiaoming the answer.

Callback function: Pass a function pointer as a parameter to another function. When this pointer is used to call the function it points to, we say that it is a callback function.

Applicability (2/5)

- Specify, queue, and execute requests at different times.
  - A Command object can have a **lifetime independent** of the original request.

Applicability (3/5)

- Support undo
  - The Command's Execute operation can store state for reversing its effects in the command itself.
  - The Command interface must have an added Unexecute operation that reverses the effects of a previous call to Execute.
  - Executed commands are stored in a history list.
  - Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling Unexecute and Execute, respectively.

Applicability (4/5)

- Support logging changes
  - Adding the Command interface with Load and Store operations, you can keep a persistent **log of changes**.
  - They can be reapplied in case of a system crash.
  - Recovering from a crash involves Load logged commands from disk and re-executing them with the Execute operation.

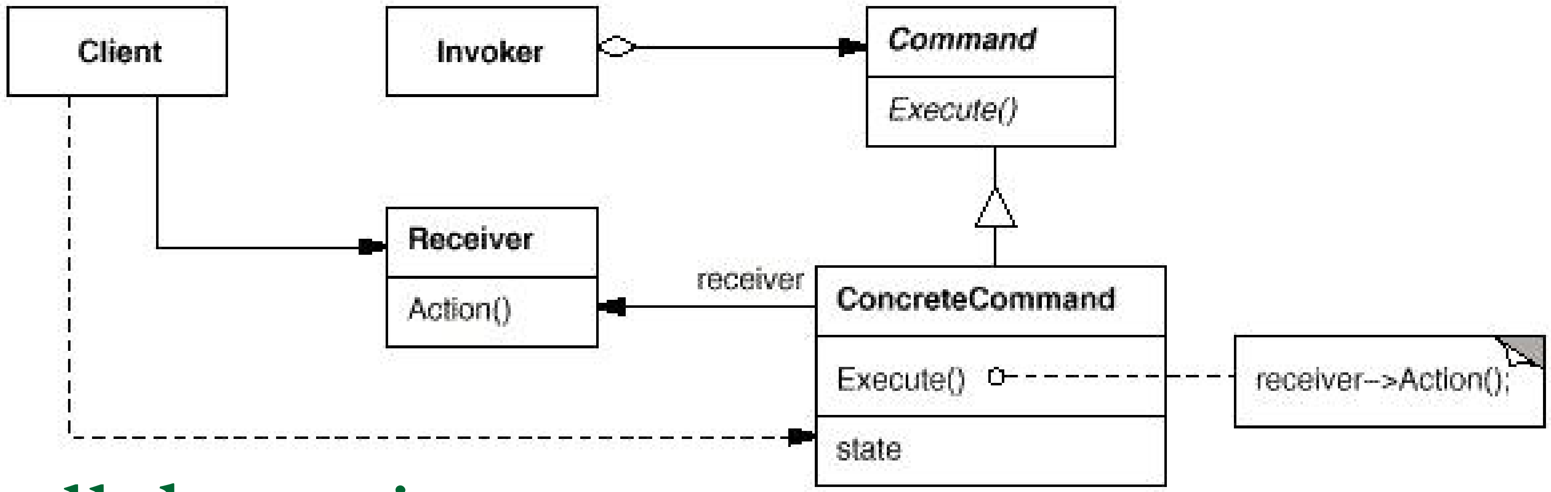
Applicability (5/5)

- Support transactions.
  - Structure a system around high-level operations built on primitives operations.
  - A transaction encapsulates a set of changes to data.
  - Commands have a common interface, letting you invoke all transactions the same way.

Command Pattern

- Command pattern separate the responsibility of sending command and executing command, delegates command to different objects;
- Each command is an operation;
- Invoker send a command as an request of the operation;
- Receiver take a command and execute the operation;
- Invoker is separate from Receiver, and when, where, how the command is executed.

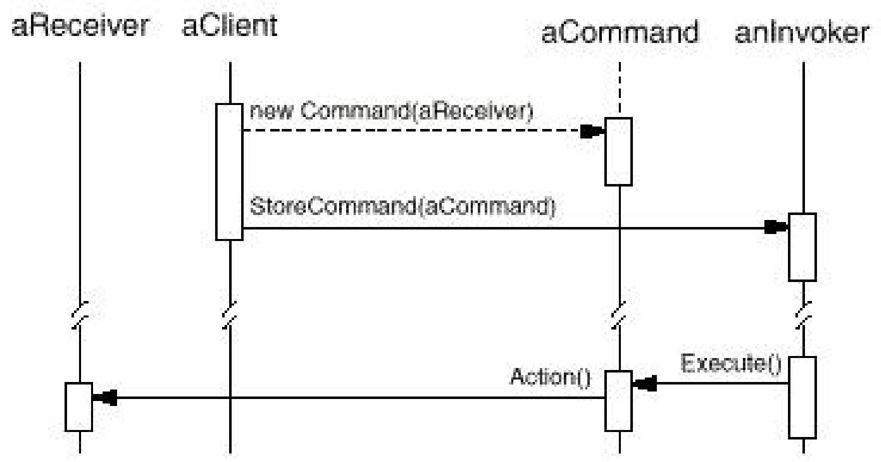
Structure



Collaborations

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.

Sequence Diagram



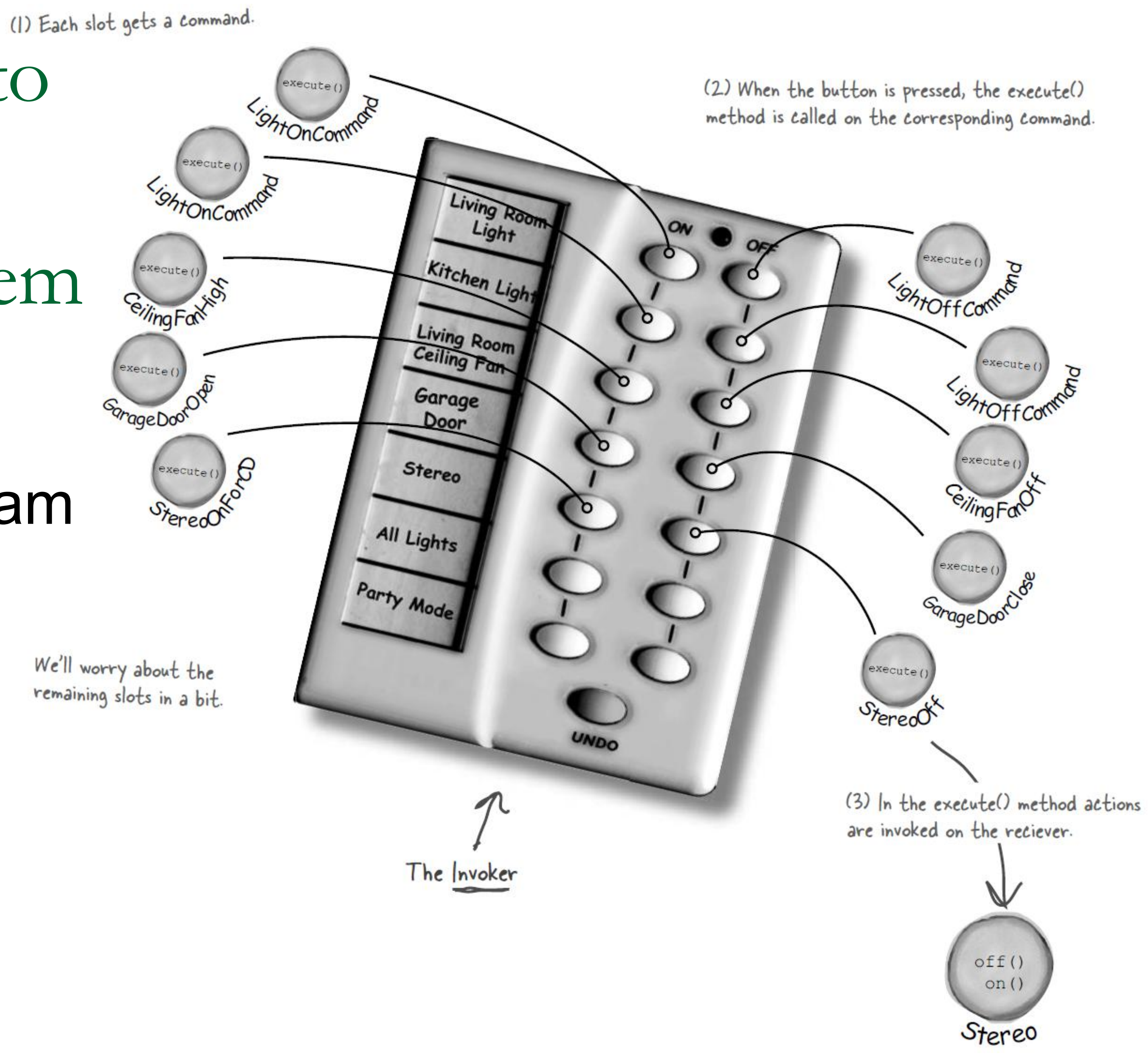
- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.

Consequences

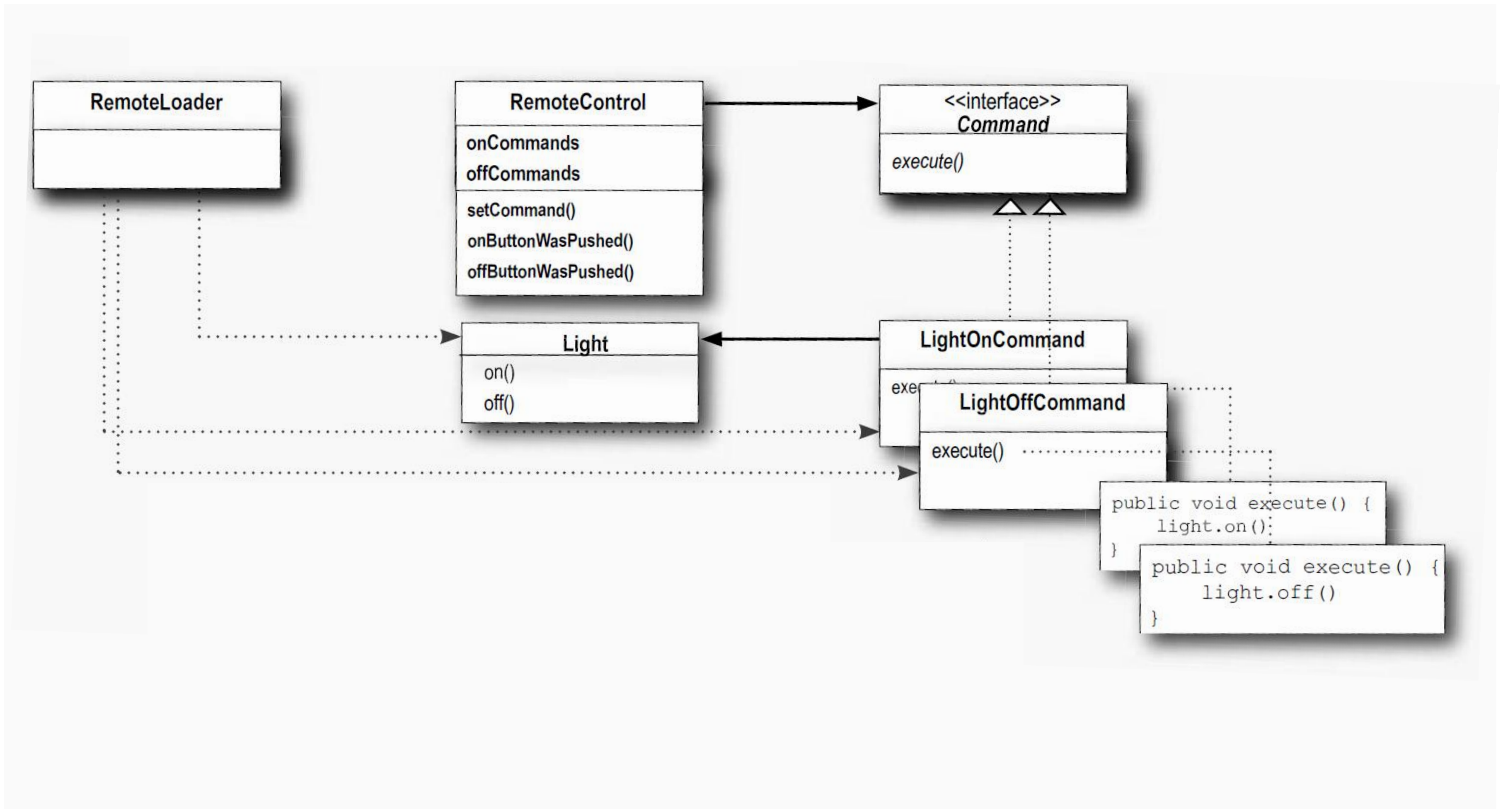
- Command **decouples** the object that invokes the operation from the one that knows how to perform it.
- You can assemble commands into a composite command (Composite pattern).
- It's easy to add **new** Commands, because you don't have to change existing classes.
- It is easy to implement Undo and Redo.

Back to The Problem

- Draw class diagram now



Class Diagram



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

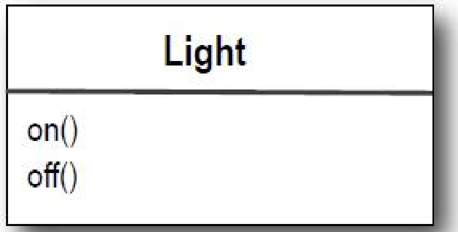
```
public void execute() {
    light.off()
}
```

The **Command** Interface

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

Simple. All we need is one method called execute().

Implementing a **ConcreteCommand** to **turn a light on**



This is a command, so we need to implement the Command interface.

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

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

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

The constructor is passed the specific light that this command is going to control – say the living room light – and stashes it in the light instance variable. When execute gets called, this is the light object that is going to be the Receiver of the request.

The execute method calls the on() method on the receiving object, which is the light we are controlling.

Invoker “has only one” command: RemoteControl: Using the command object

```
public class SimpleRemoteControl {
    Command slot;

    public SimpleRemoteControl() {}

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

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

We have one slot to hold our command, which will control one device.

We have a method for setting the command the slot is going to control. This could be called multiple times if the client of this code wanted to change the behavior of the remote button.

This method is called when the button is pressed. All we do is take the current command bound to the slot and call its execute() method.

Creating a simple test to use the Remote Control

This is our Client in Command Pattern-speak.

```
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();
    }
}
```

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

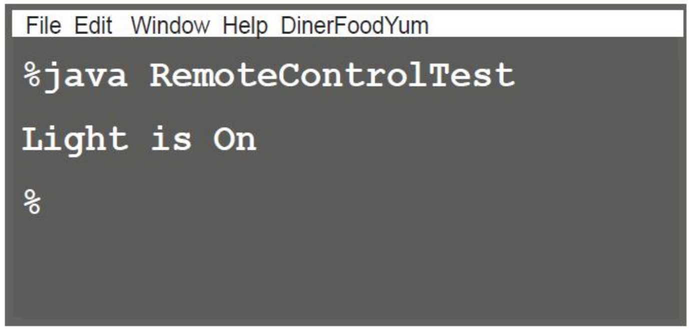
Now we create a Light object, this will be the Receiver of the request.

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!



```
%java RemoteControlTest
Light is On
%
```

Code: net.dp.command.simpleremote.RemoteControlTest

Implement “Undo” Remote Control

Implement “Undo”

- When commands support undo, they have an undo() method that mirrors the execute() method. Whatever execute() last did, undo() reverses.

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

Here's the new undo() method.

Implement ConcreteCommand support "Undo"

```
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.

```
public class LightOffCommand implements Command {
    Light light;

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

    public void execute() {
        light.off();
    }

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

And here, undo() turns the light back on!

```
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() {
        // toString code here...
    }
}
```

This is where we'll stash the last command executed for the undo button.

Just like the other slots, undo starts off with a NoCommand, so pressing undo before any other button won't do anything at all.

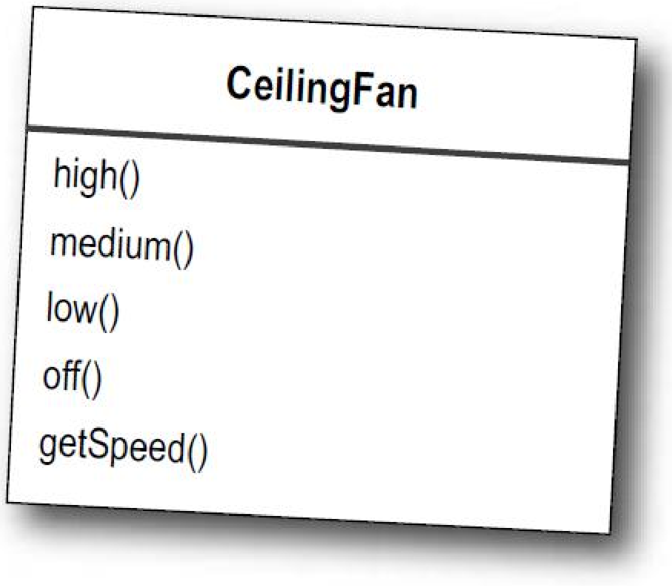
When a button is pressed, we take the command and first execute it; then we save a reference to it in the undoCommand instance variable. We do this for both "on" commands and "off" commands.

When the undo button is pressed, we invoke the undo() method of the command stored in undoCommand. This reverses the operation of the last command executed.

Next, consider the electric fan...

Using state to implement Undo

The Ceiling Fan Has Different States



```
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;
        // code to set fan to high
    }

    public void medium() {
        speed = MEDIUM;
        // code to set fan to medium
    }

    public void low() {
        speed = LOW;
        // code to set fan to low
    }

    public void off() {
        speed = OFF;
        // code to turn fan off
    }

    public int getSpeed() {
        return speed;
    }
}
```

Notice that the CeilingFan class holds local state representing the speed of the ceiling fan.

These methods set the speed of the ceiling fan.

We can get the current speed of the ceiling fan using getSpeed().

Adding **Undo** to the ceiling fan commands

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

    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }

    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();
        }
    }
}
```

We've added local state to keep track of the previous speed of the fan.

In execute, before we change the speed of the fan, we need to first record its previous state, just in case we need to undo our actions.

To undo, we set the speed of the fan back to its previous speed.

Test the ceiling fan

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new Remot

        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);

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

        remoteControl.onButtonWasPushed(1);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

Here we instantiate three commands: high, medium, and off.

Here we put medium in slot zero, and high in slot one. We also load up the off commands.

First, turn the fan on medium.

Then turn it off.

Undo! It should go back to medium...

Turn it on to high this time.

And, one more undo; it should go back to medium.

Code: net.dp.command.undo.RemoteLoader

Using a macro command

1 First we create the set of commands we want to go into the macro:

```
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(hottub);
```

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

Now create all the On commands to control them.

2 Next we create two arrays, one for the On commands and one for the Off commands, and load them with the corresponding commands:

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

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

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

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

3 Then we assign MacroCommand to a button like we always do:

```
remoteControl.setCommand(0, partyOnMacro, partyOffMacro);
```

Assign the macro command to a button as we would any command.

4 Finally, we just need to push some buttons and see if this works.

```
System.out.println(remoteControl);
System.out.println("--- Pushing Macro On---");
remoteControl.onButtonWasPushed(0);
System.out.println("--- Pushing Macro Off---");
remoteControl.offButtonWasPushed(0);
```

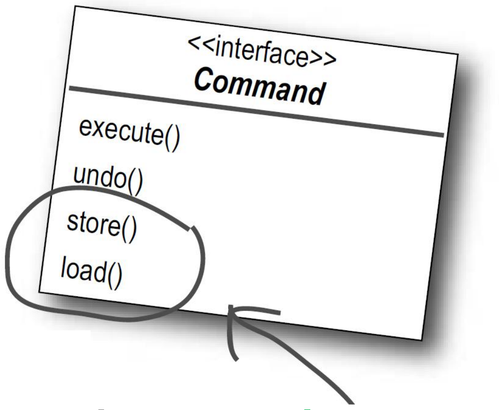
Code: net.dp.command.party.RemoteLoader

More Uses of The Command Pattern: Queuing Requests

- Commands give us a way to package a piece of computation and pass it. The computation itself may be invoked long after some client application creates the command object.
- Useful applications: schedulers, thread pools, job queues.
  - E.g., a job queue: add commands to the queue on one end, and on the other end sit a group of threads. Threads remove a command from the queue, call its execute() method, wait for the call to finish, then discard the command object and retrieve a new one.

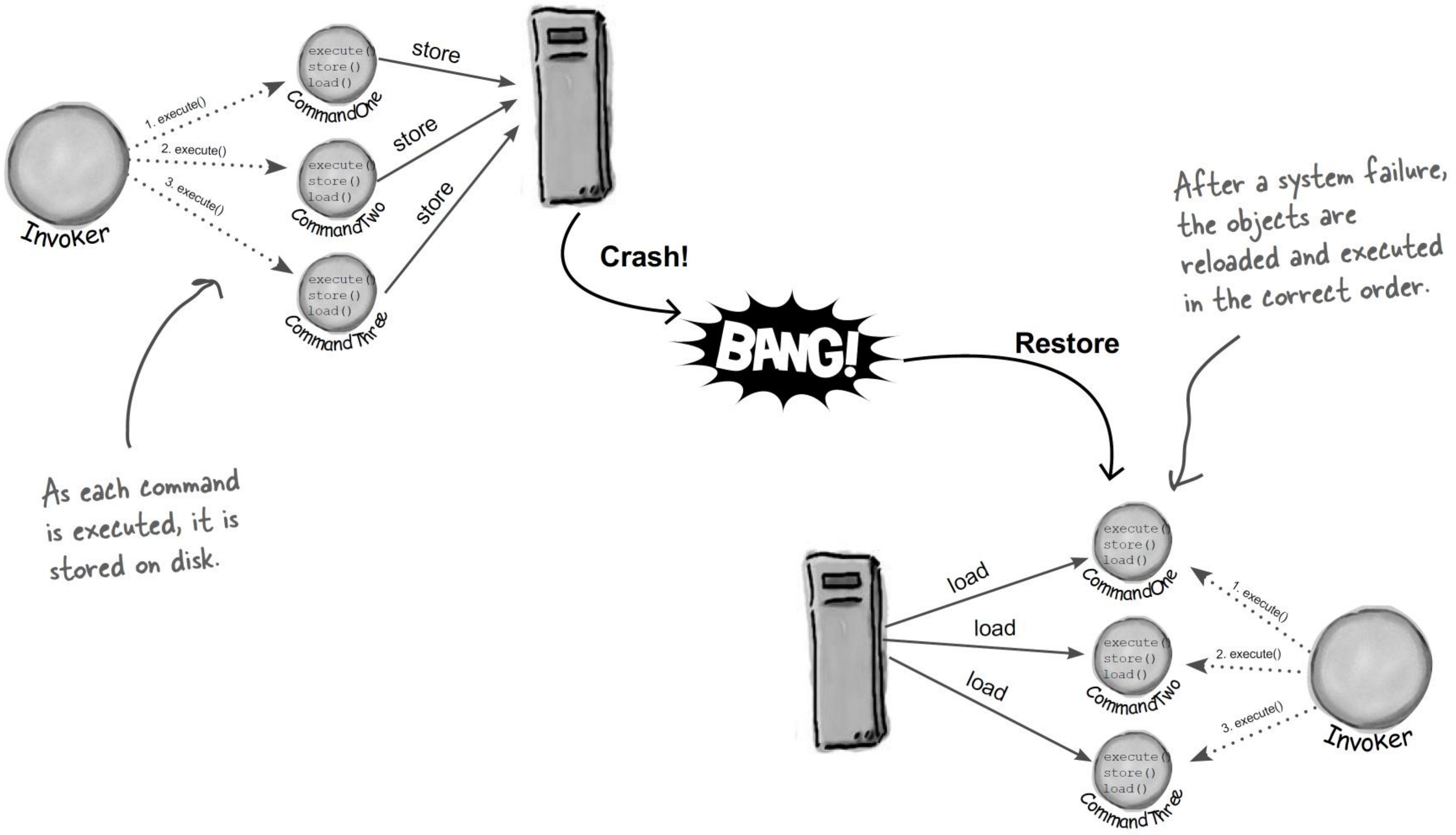
More Uses of The Command Pattern:
Logging Requests

Logging Requests (1/2)



- The semantics of some applications require that we log all actions and be able to recover after a crash by reinvoking those actions.
- The Command Pattern can support these semantics with the addition of two methods: store() and load().
- As we execute commands, we store a history of them on disk. When a crash occurs, we reload the command objects and invoke their execute() methods in batch and in order.
- See pic on **next page**.

Logging Requests (2/2)



Implementation Issues: How intelligent should a command be?

- A command can have a wide range of abilities.
- **At one extreme**, it merely defines a binding between a receiver and the actions that carry out the request.
  - Sometime commands have enough knowledge to find their receiver dynamically.
- **At the other extreme**, it implements everything itself without delegating to a receiver at all.
  - It is useful when (1) you want to define commands that are independent of existing classes, (2) when no suitable receiver exists, or (3) when a command knows its receiver implicitly.