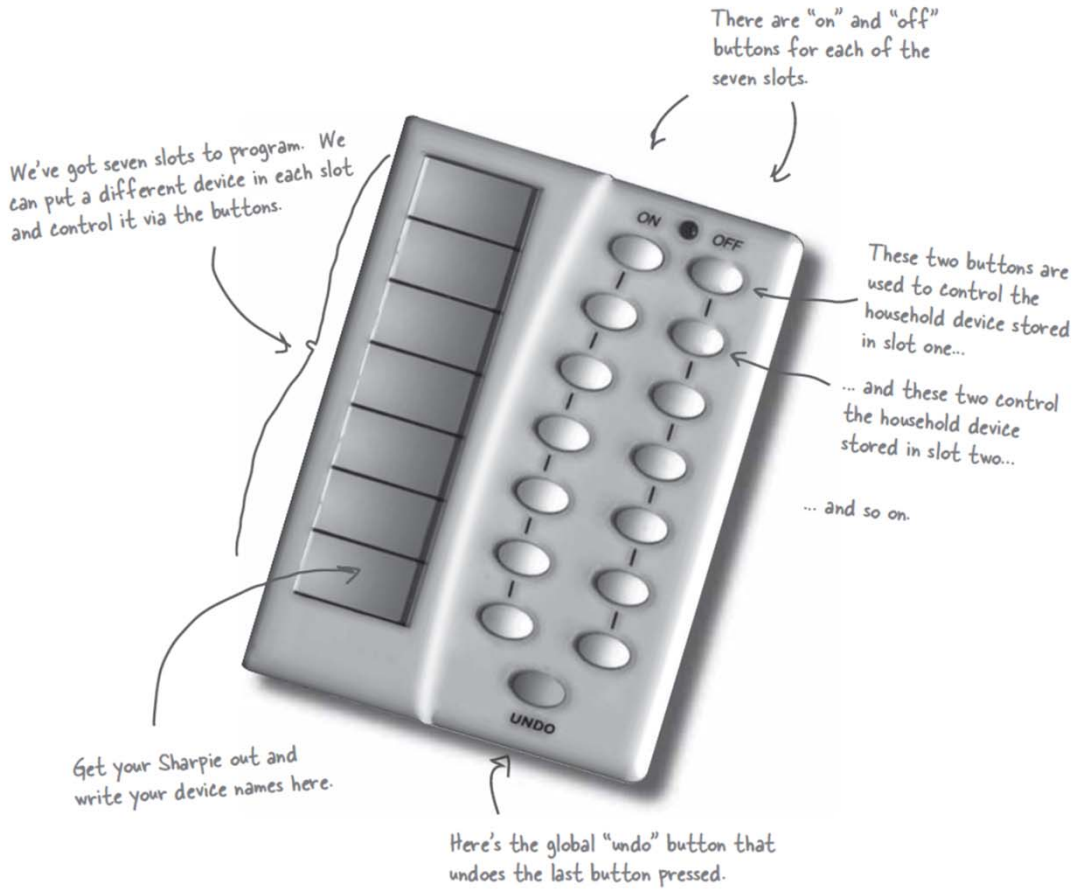
# Command Pattern

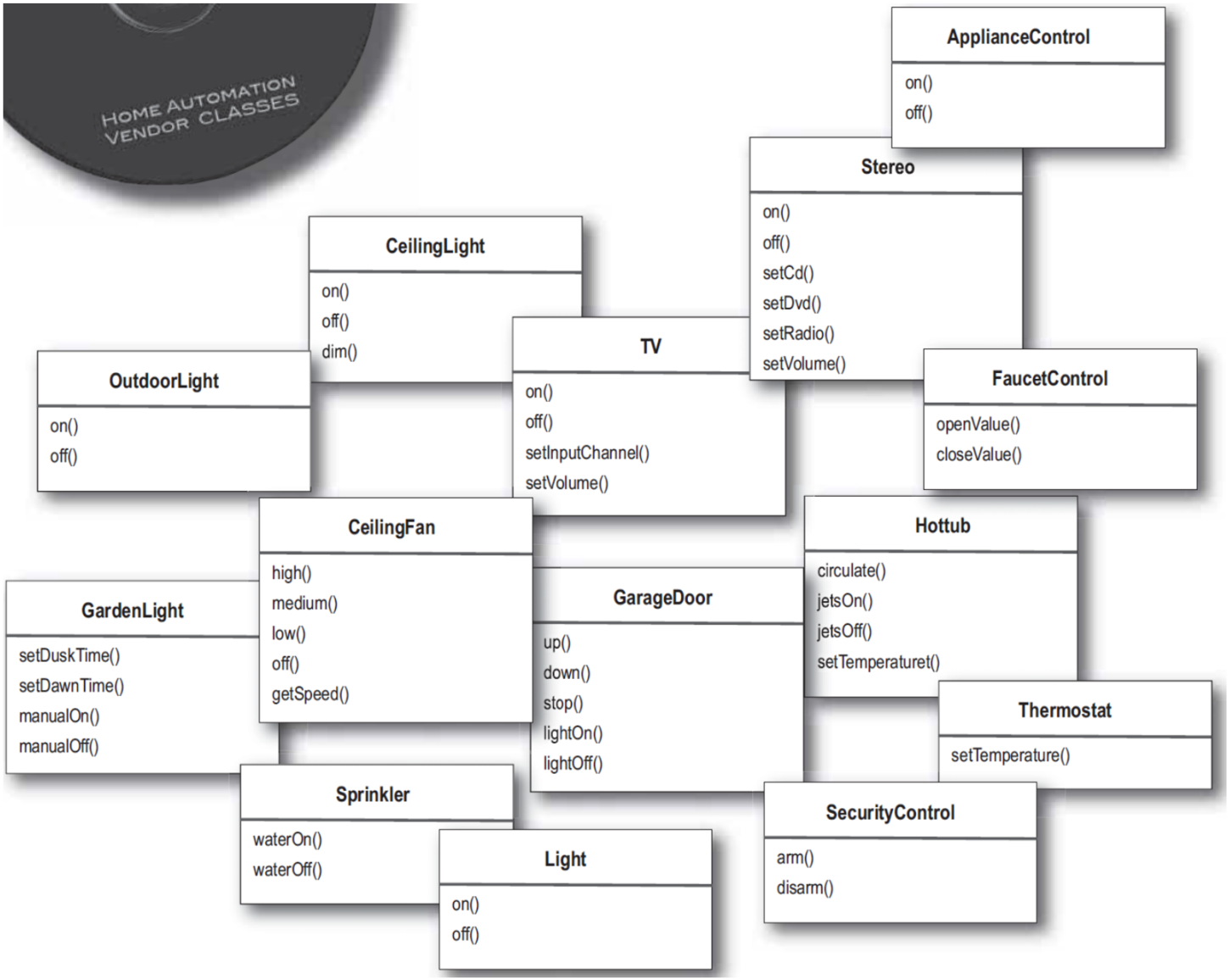
# Command Pattern

- Purpose
  - Encapsulates a request as an object.
  - This allows the request to be handled in traditionally **object based relationships** such as queuing and callbacks.
- Use When
  - Requests need to be specified, queued, and executed at variant times or in variant orders.
  - A history of requests is needed.
  - The invoker should be decoupled from the object handling the invocation.

# Given Hardware



# Our Vendor Classes



# Without Using Command Pattern

- Never do this!

```
if(command == Slot1On)
    light.on();

else if(command == Slot1Off)
    light.off();
```

- Every time you want to modify the behavior of a button, you need to touch the client code!

# Implementing Command interface

- Command interface

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

- You may want to include undo() for undo operation later.

# Implementing a command

```
public class LightOnCommand implements Command {
     Light light; // Stores info. about its receiver

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

# Using the command object (Building Invoker)

```
Public class SimpleRemoteControl{
     Command slot;

     Public simpleRemoteControl() { }

     Public void setcommand( Command command ) {
        slot = command;
     }

     Public void buttonWaspressed() {
        slot.execute();
     }

}
```

# Client Program

```
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();
    }
}
```

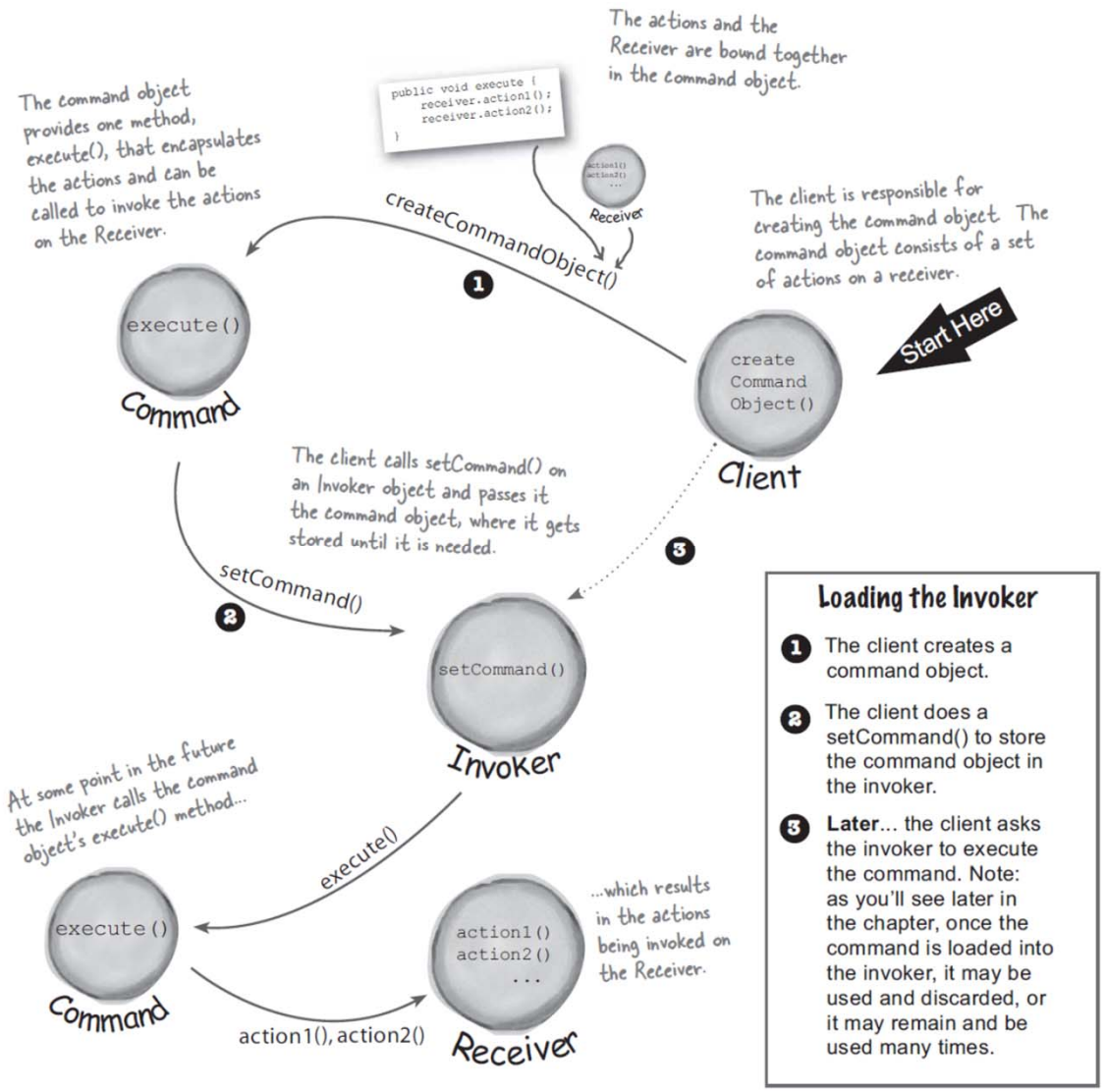
# Command Pattern

- LightOn, LightOff, TVOn, TVOff, etc.

```
public class LightOffCommand implements Command {
    Light light; // Stores info. about its receiver

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

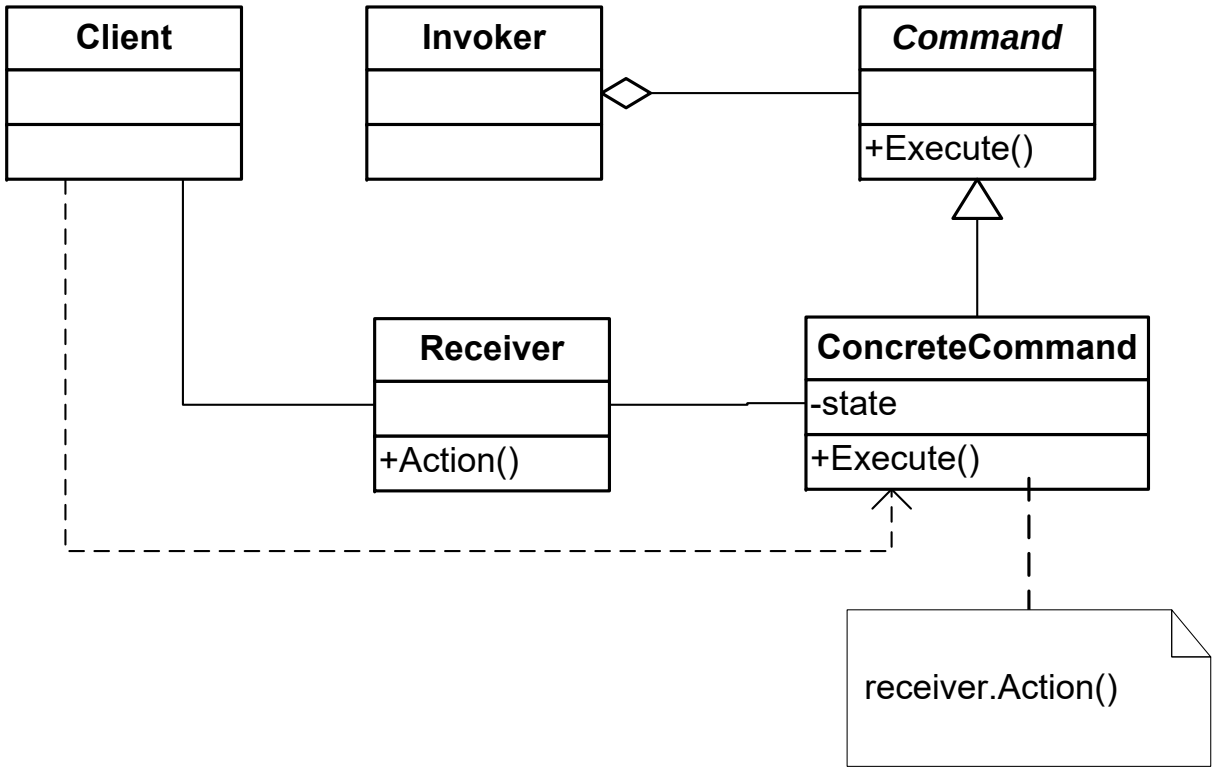
- Command Pattern encapsulates a request as an object:
  - Each command object exposes only execute (and undo) method

# Our Approach

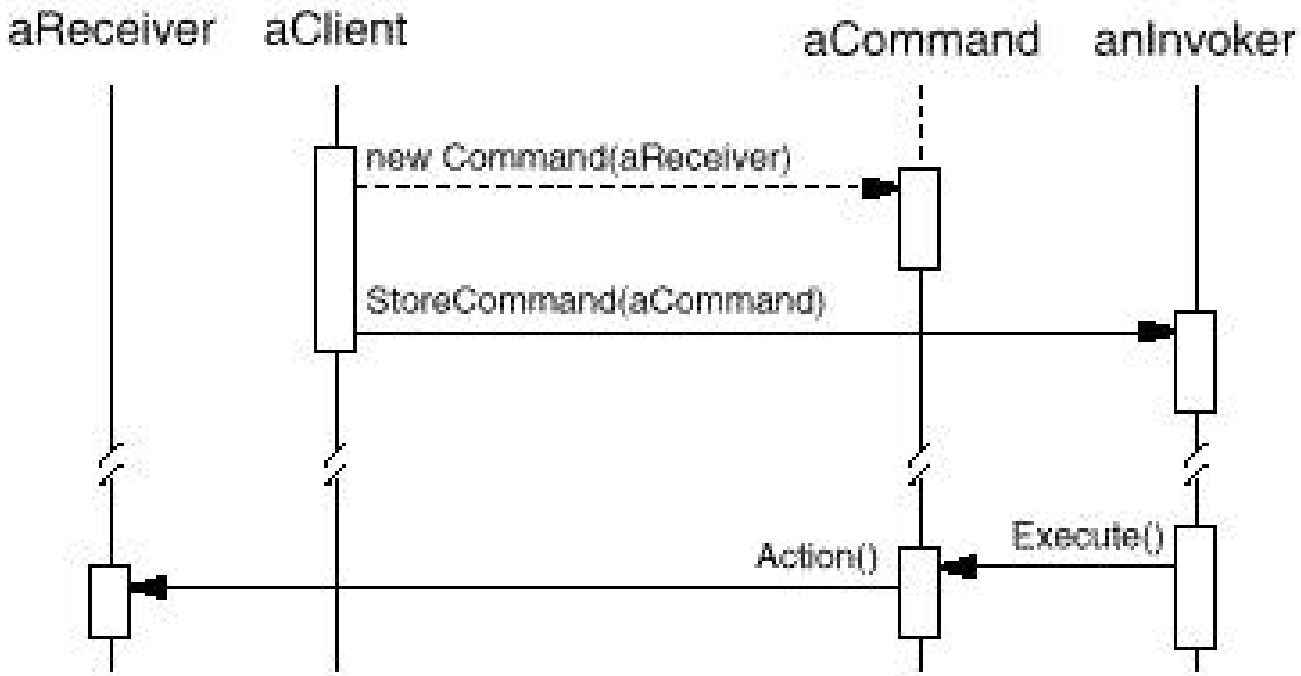


## Encapsulates a request as an object

thus, let you parameterize clients with different requests, queue or log requests, and support undoable operations

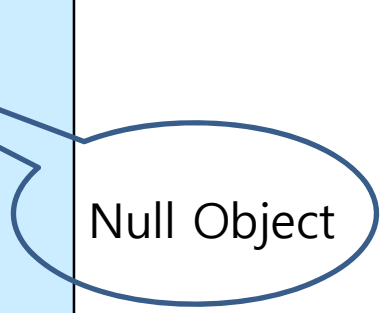


# Collaborations



# Extending the Remote Control

```
public class RemoteControl {

    Command[] oncommands;
    Command[] offcommands;

    RemoteControl() {
        oncommands = new Command[7];
        offcommands = new Command[7];

        Command noCommand = new NoCommand();

        for(int i = 0; i< 7; i++) {
            onCommand[i] = noCommand;
            offCommand[i] = noCommand;
        }
    }
}

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



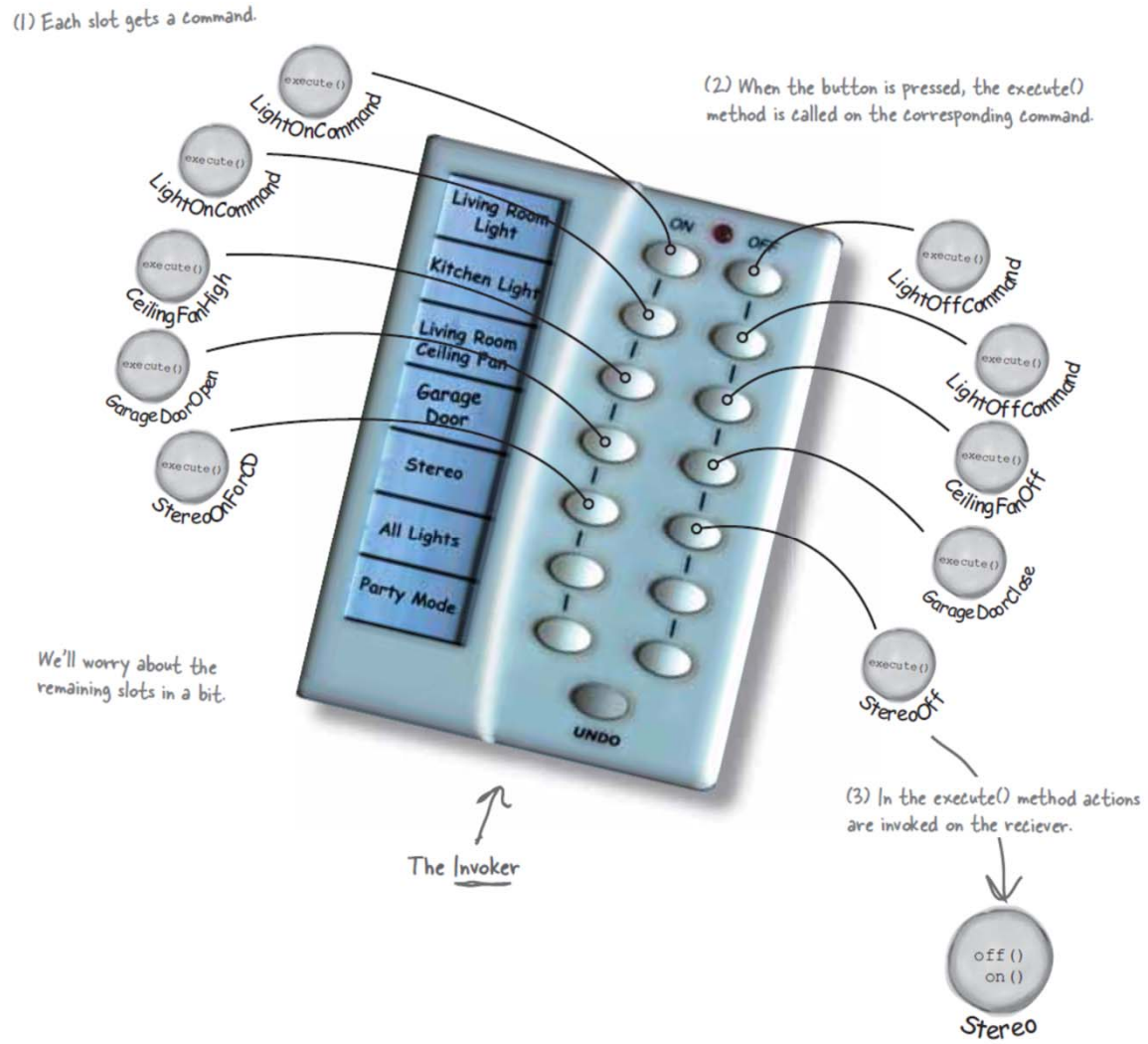
# Command Pattern

```
public void setCommand(int slot, Command oncommand, Command offcommand){
    oncommand[slot] = oncommand;
    offcommand[slot] = offcommand;
}

public void onButtonWasPushed(int slot) {
    oncommand[slot].execute();
}

public void offButtonWasPushed(int slot){
    offcommand[slot].execute();
}
```

# Client Program

```
public class RemoteLoader {
    public static void main(String[] args) {
          // create invoker
          RemoteControl remotecontrol = new RemoteControl();

          // create receivers
          Light livingRoomLight = new Light("Living Room");
          Light kitchenLight = new Light("Kitchen");

          // create commands
          Command livingRoomLightOn = new LightOnCommand(livingRoomLight);
          Command livingRoomLightOff = new LightOffCommand(livingRoomLight);
          Command kitchenlightOn = new LightOnCommand(kitchenLight);
          Command kitchenlightOff = new LightOffCommand(kitchenLight);

          // linking the invoker with the commands
          remotecontrol.setCommand(0,livingRoomLightOn,livingRoomLightOff);
          remotecontrol.setCommand(1,kitchenlightOn,kitchenlightOff);

          remotecoltrol.onButtonWasPushed(0);
          remotecoltrol.offButtonWasPushed(0);
          remotecoltrol.onButtonWasPushed(1);
          remotecoltrol.offButtonWasPushed(1);
    }
}
```

# After Setting the Commands



# Supporting 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();
    }
}

public void onButtonWaspressd(int slot){
    oncommand[slot].execute();
    undocommand = oncommand[slot];
}
public void offButtonWaspressd(int slot){
    offcommand[slot].execute();
    undocommand = offcommand[slot];
}

public void undoButtonWaspressd(){
    undocommand.undo();
}
```

# Supporting Macro Commands

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

# Command + Iterator

```
public class MacroCommand implements Command {
    ArrayList <Command>commands;

    public MacroCommand(Command[] commands) {
        this.commands = new ArrayList();
        for (int i=0; i< commands.length; i++)
            this.commands.add(commands[i]);
    }
    public void execute() {
        for (Command item: this.commands)
            item.execute();
    }
    // NOTE: commands have to be done backwards to ensure proper undo
    public void undo() {
        ListIterator<Command> i=commands.listIterator(commands.size());
        while (i.hasPrevious())
            i.previous().undo();
    }
}
```

# Active Object Pattern

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

class ActiveObjectEngine {
    LinkedList itsCommands = new LinkedList();

    public void addCommand(Command c) {
        itsCommands.add(c);
    }

    public void run() {
        while (!itsCommands.isEmpty()) {
            Command c = (Command) itsCommands.getFirst();
            itsCommands.removeFirst();
            c.execute();
        }
    }
}
```

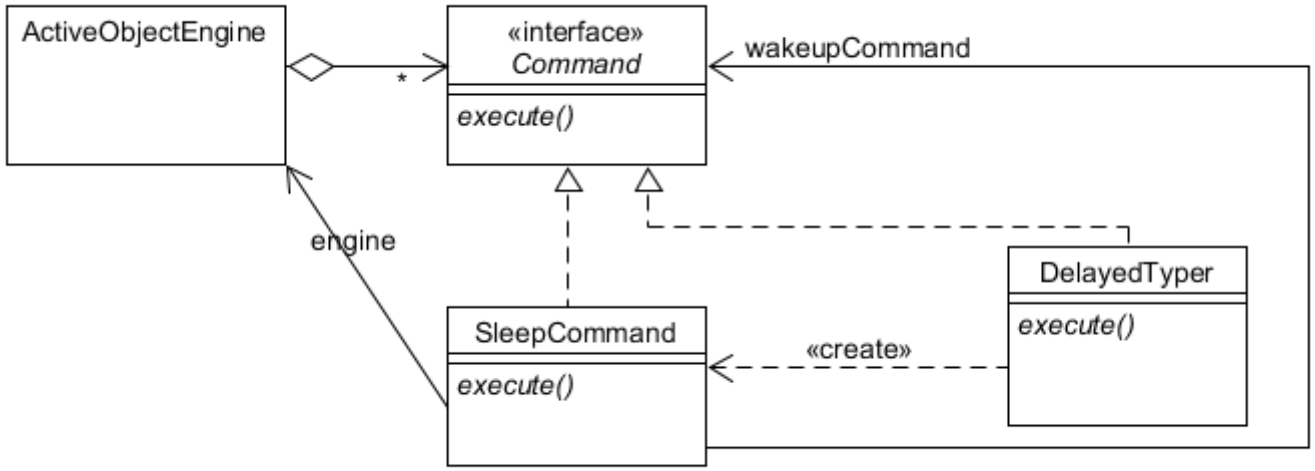
```
class SleepCommand implements Command {
    private Command wakeupCommand = null;
    private ActiveObjectEngine engine = null;
    private long sleepTime = 0; private long startTime = 0;
    private boolean started = false;
    public SleepCommand(long milliseconds, ActiveObjectEngine e,
        Command wakeupCommand) {
        sleepTime = milliseconds;
        engine = e;
        this.wakeupCommand = wakeupCommand;
    }
    public void execute() {
        long currentTime = System.currentTimeMillis();
        if (!started) {
            started = true;
            startTime = currentTime;
            engine.addCommand(this);
        }
        else if ((currentTime - startTime) < sleepTime ) {
            engine.addCommand(this);
        } else
            engine.addCommand(wakeupCommand);
    }
}
```

```
public class DelayedTyper implements Command {
    private long mDelay = 0;
    private char mChar;
    private static boolean stop = false;
    private static ActiveObjectEngine engine = new ActiveObjectEngine();
    public DelayedTyper(long delay, char c) {
        mDelay = delay;
        mChar = c;
    }
    @Override
    public void execute() {
        System.out.print(mChar);
        if (!stop) {
            delayAndRepeat();
        }
    }
    private void delayAndRepeat() {
        engine.addCommand(new SleepCommand(mDelay, engine, this));
    }
    // continued in the next page
```

```
    public static void main(String args[]) throws Exception {
        engine.addCommand(new DelayedTyper(100, '1'));
        engine.addCommand(new DelayedTyper(300, '3'));
        engine.addCommand(new DelayedTyper(500, '5'));
        engine.addCommand(new DelayedTyper(700, '7'));

        Command stopCommand = new Command() {
            @Override public void execute() {
                stop = true;
            }
        };
        engine.addCommand(new SleepCommand(10000, engine, stopCommand));
        engine.run();
    }
} // end of class
```

```
13571113115131711315113117351111315137111131513117135111131571131111315171311135111371151311131571131113511713115131113715113113511711315171311173151357
```

# Class Diagram for DelayedTyper



# Summary

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- Commands can be manipulated and extended like any other object.
- You can assemble commands into a composite command.
  - E.g. MacroCommand class
  - In general, composite commands are an instance of Composite pattern
- It's easy to add new Commands, because you don't have to change existing classes.