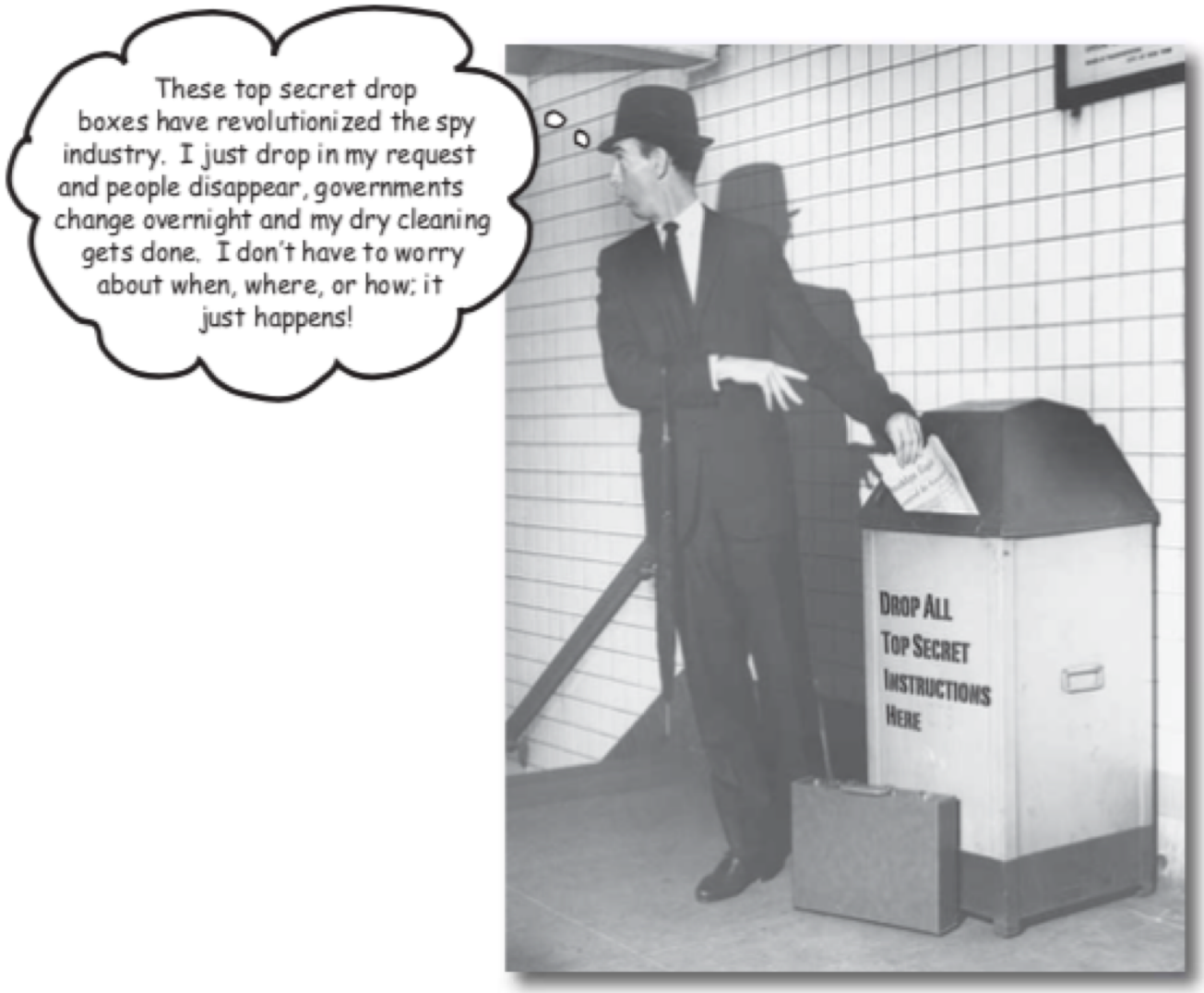
# **TC3003:** Diseño y Arquitectura de Software

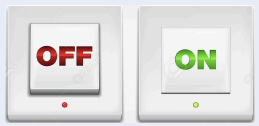
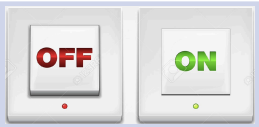
**Dr. Juan Manuel González Calleros**

**Email:** jmgonzale@itesm.mx
**Twitter:** @Juan__Gonzalez
**Facebook:** **Juan Glez Calleros**

Reuniones pedir cita

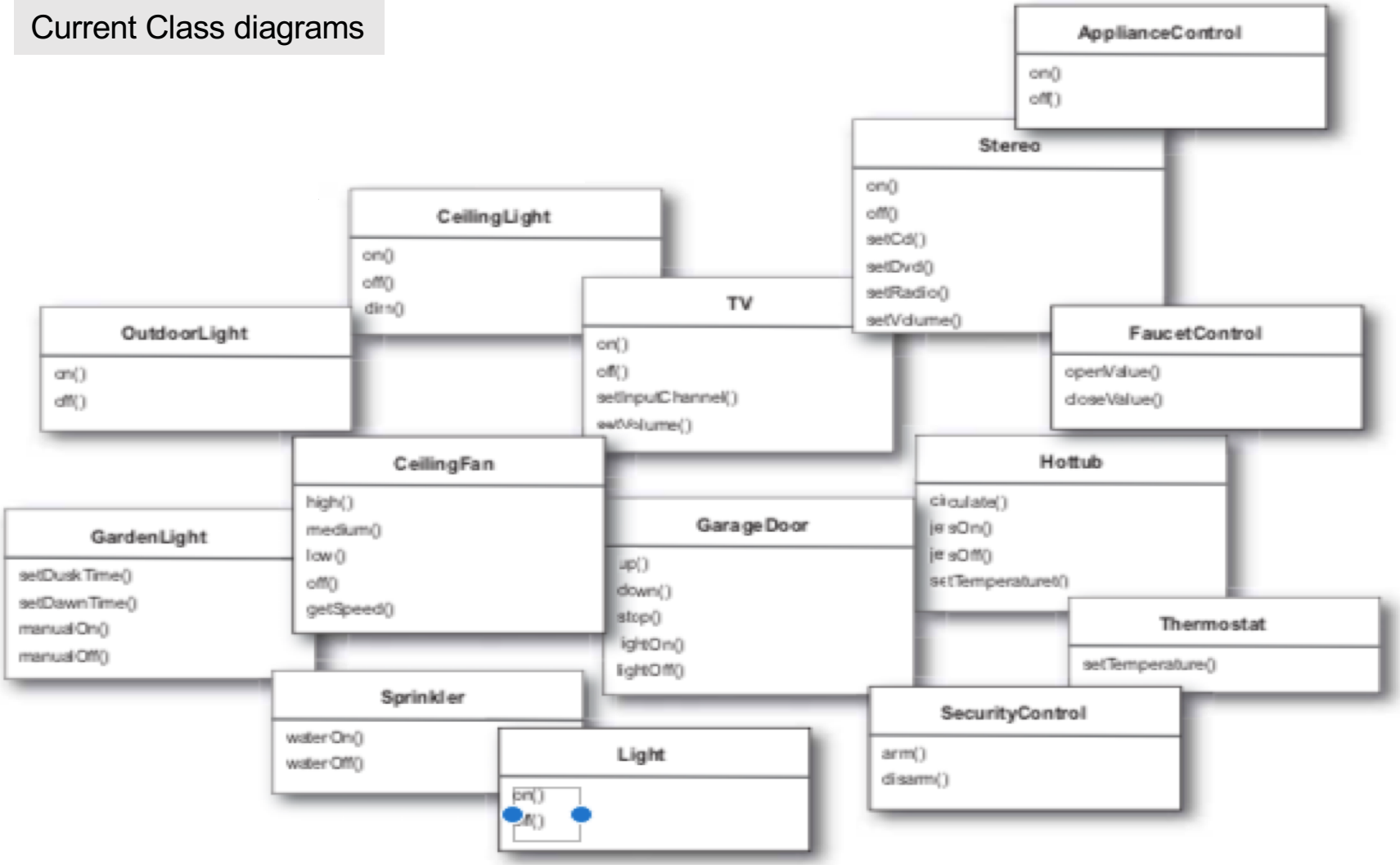

These top secret drop boxes have revolutionized the spy industry. I just drop in my request and people disappear, governments change overnight and my dry cleaning gets done. I don't have to worry about when, where, or how; it just happens!
DROP ALL
TOP SECRET
INSTRUCTIONS
HERE

# CONTROL INTELIGENTE

| Device | Action A |
| --- | --- |
| Living Room Light |   |
| Kitchen Light |   |
| Living Room Celleing |   |
| Garage Door |   |
| Stereo |   |
| All Lights |   |
| Party Mode |   |

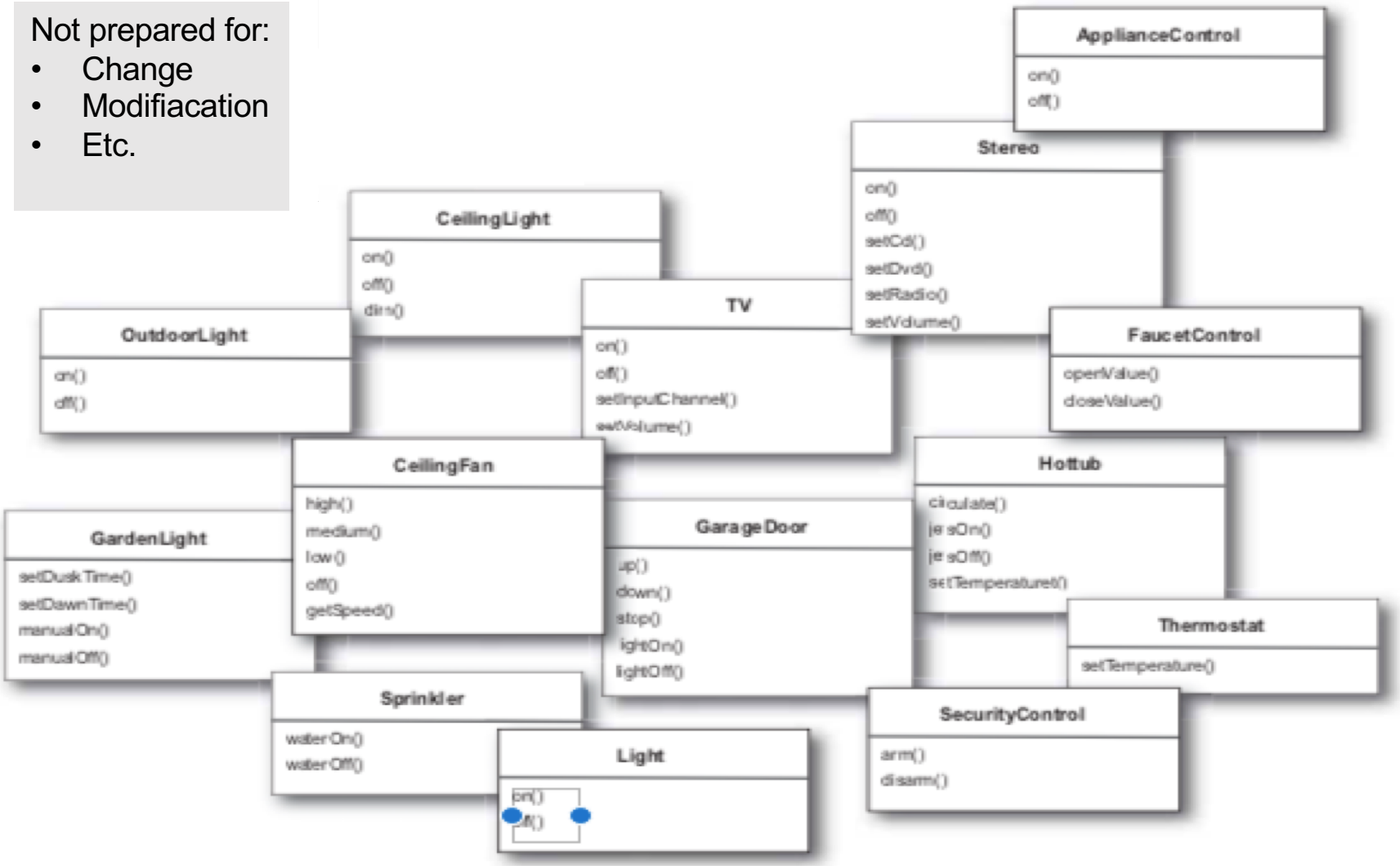
| Device | Action A |
| --- | --- |
| Living Room Light |   |
| Kitchen Light |   |
| Living Room Celleing |     Undo the last action triggered by any button |
| Garage Door |   |
| Stereo |   |
| All Lights |   |
| Party Mode |   |

Current Class diagrams



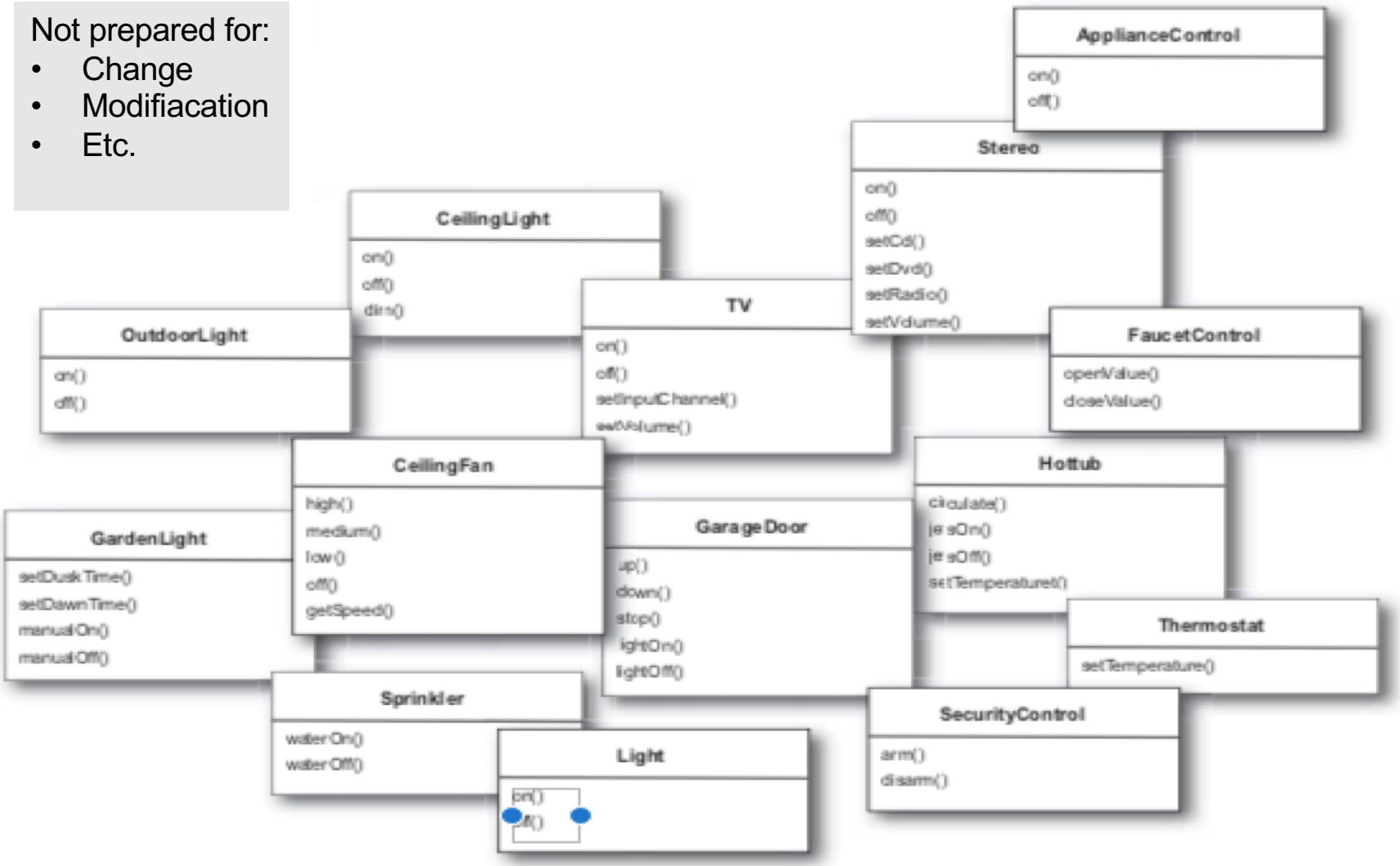
Not prepared for:

- Change
- Modifiacation
- Etc.

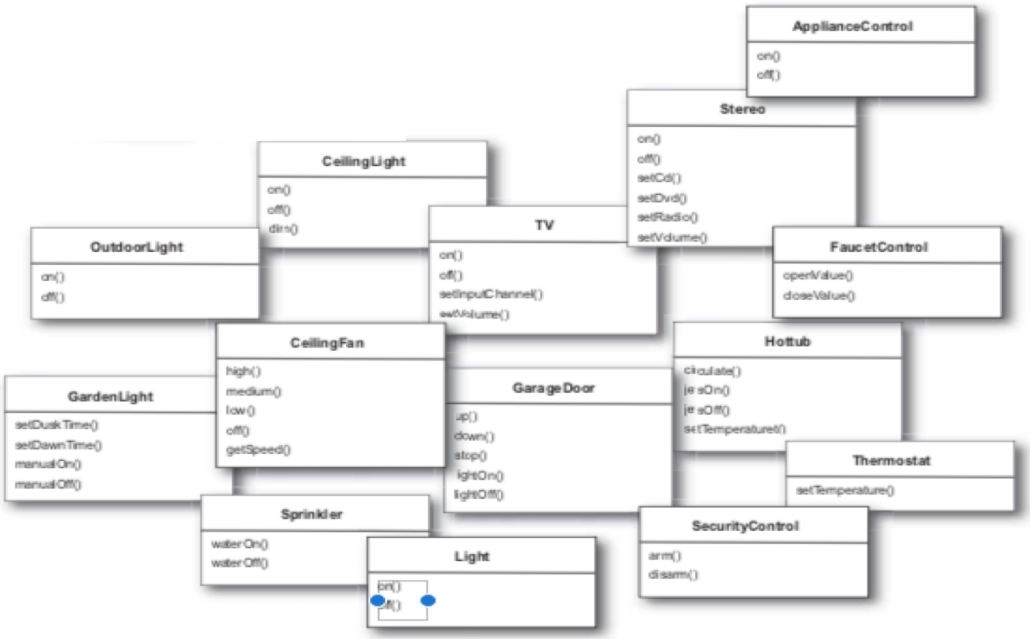


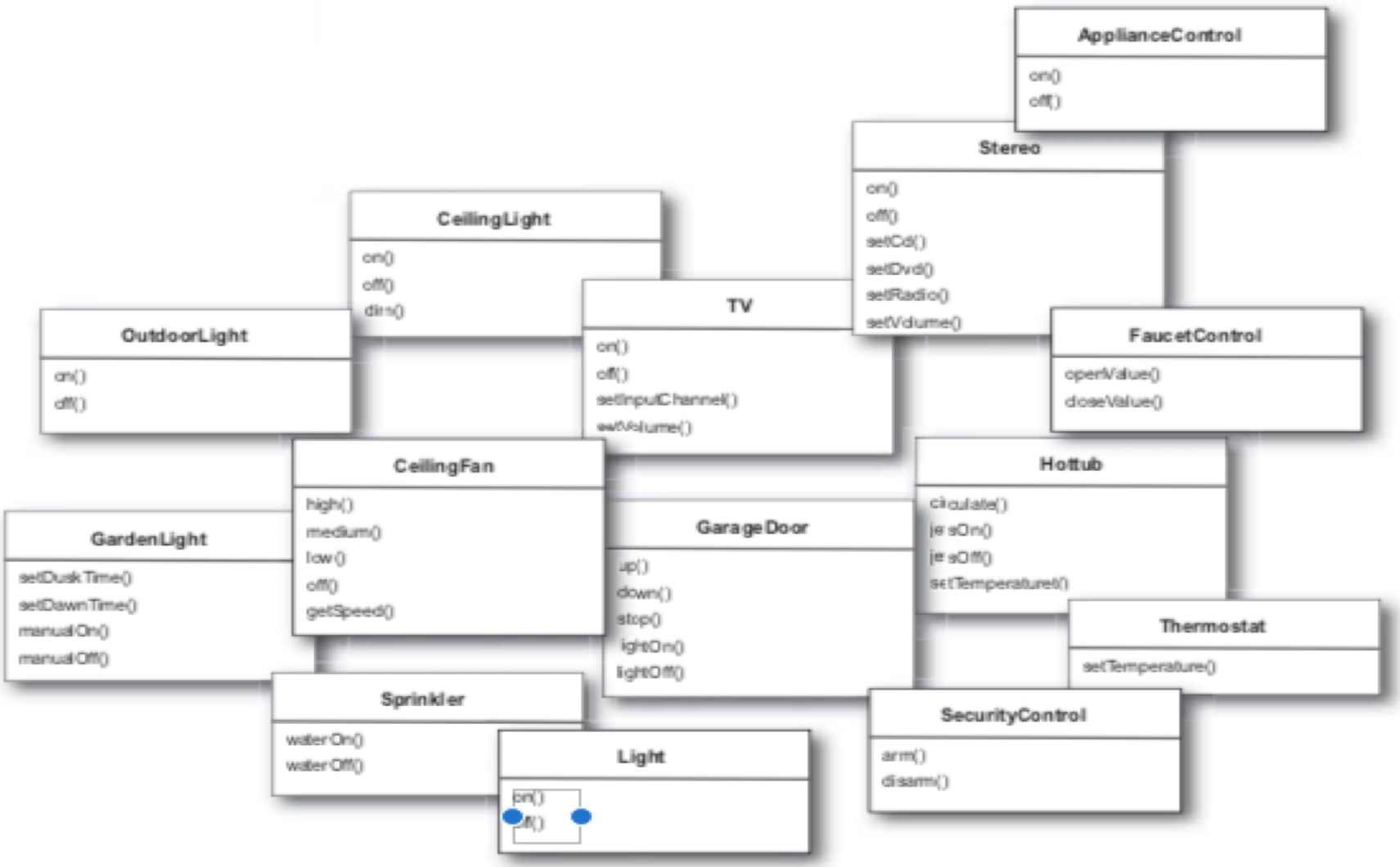
Not prepared for:

- Change
- Modifiacation
- Etc.

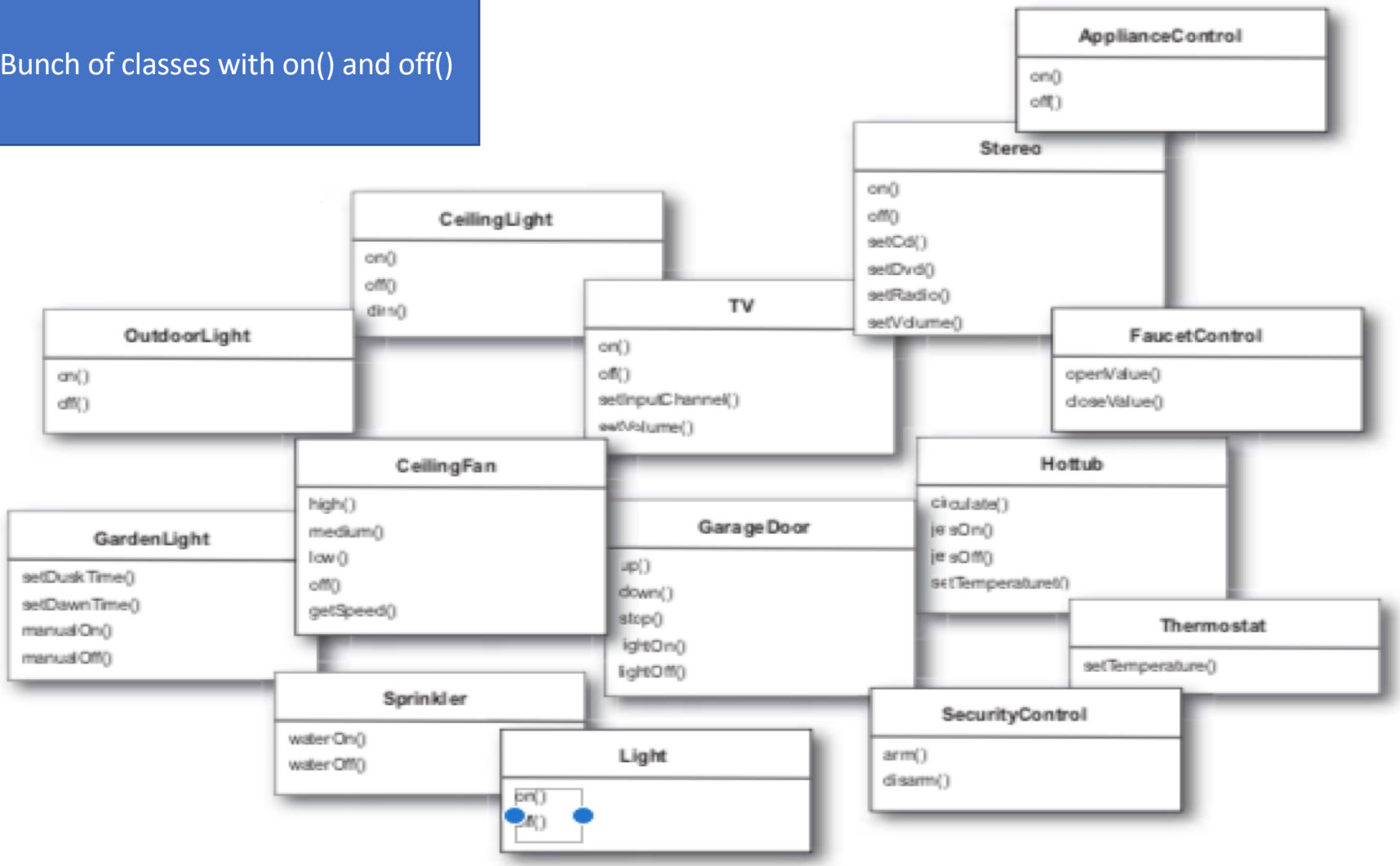



Control API
OFF
ON
OFF
ON
OFF
ON
OFF
ON
OFF
ON
OFF
ON
UNDO

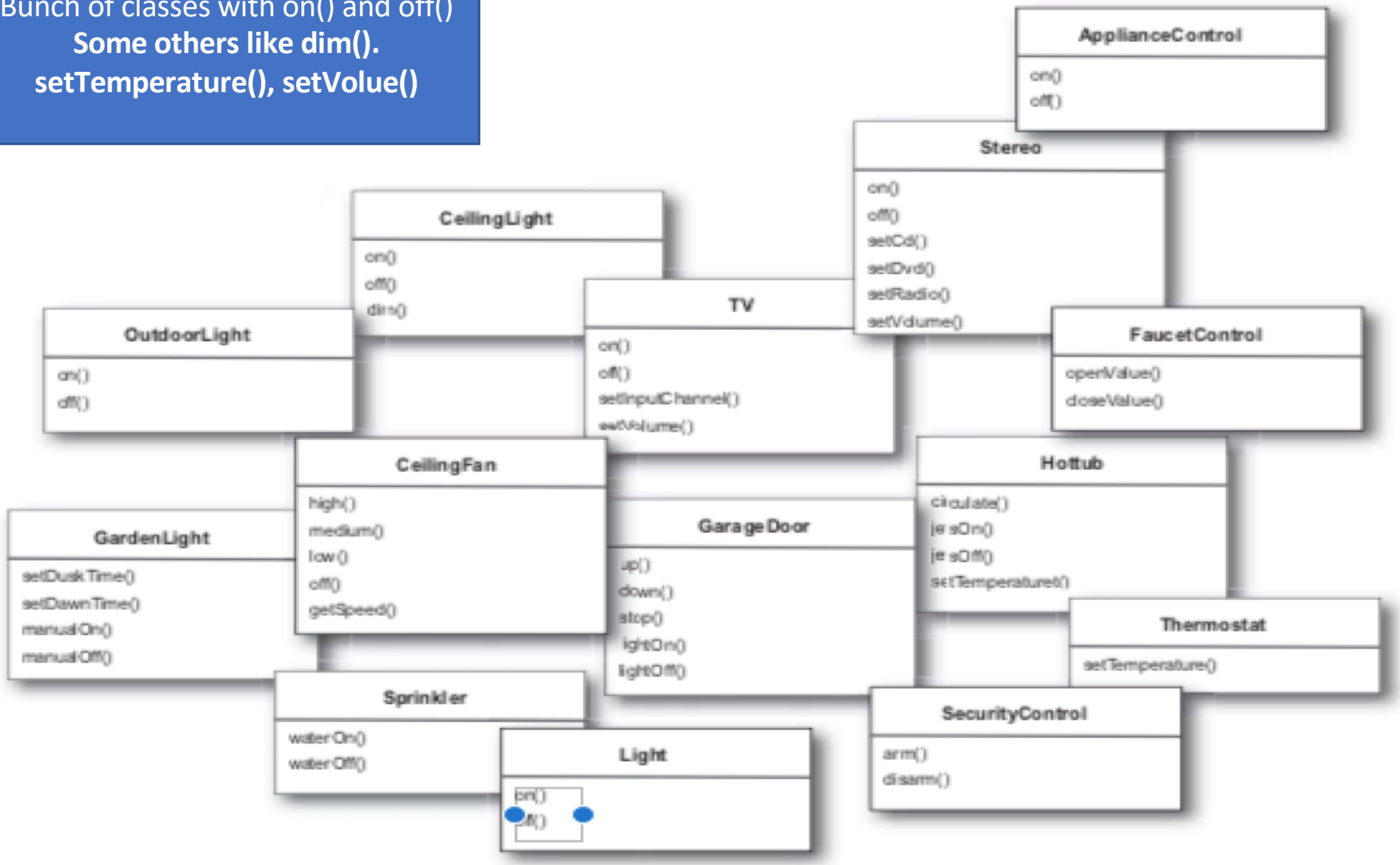

ApplianceControl
on()
off()
Stereo
on()
off()
setCd()
setDvd()
setRadio()
setVolume()
CeilingLight
on()
off()
dim()
TV
on()
off()
setInputChannel()
OutdoorLight
on()
off()
FaucetControl
openValue()
closeValue()
CeilingFan
high()
medium()
low()
off()
getSpeed()
Hottub
circulate()
jetsOn()
jetsOff()
setTemperature()
GardenLight
setDuskTime()
setDawnTime()
manualOn()
manualOff()
GarageDoor
up()
down()
stop()
lightOn()
lightOff()
Thermostat
setTemperature()
Sprinkler
waterOn()
waterOff()
SecurityControl
arm()
disarm()
Light
on()
off()


ApplianceControl
on()
off()
Stereo
on()
off()
setCd()
setDvd()
setRadio()
setVolume()
CeilingLight
on()
off()
dim()
TV
on()
off()
setInputChannel()
setVolume()
OutdoorLight
on()
off()
FaucetControl
openValue()
closeValue()
CeilingFan
high()
medium()
low()
off()
getSpeed()
Hottub
circulate()
jetsOn()
jetsOff()
setTemperature()
GardenLight
setDuskTime()
setDawnTime()
manualOn()
manualOff()
GarageDoor
up()
down()
stop()
lightOn()
lightOff()
Thermostat
setTemperature()
Sprinkler
waterOn()
waterOff()
SecurityControl
arm()
disarm()
Light
on()
off()

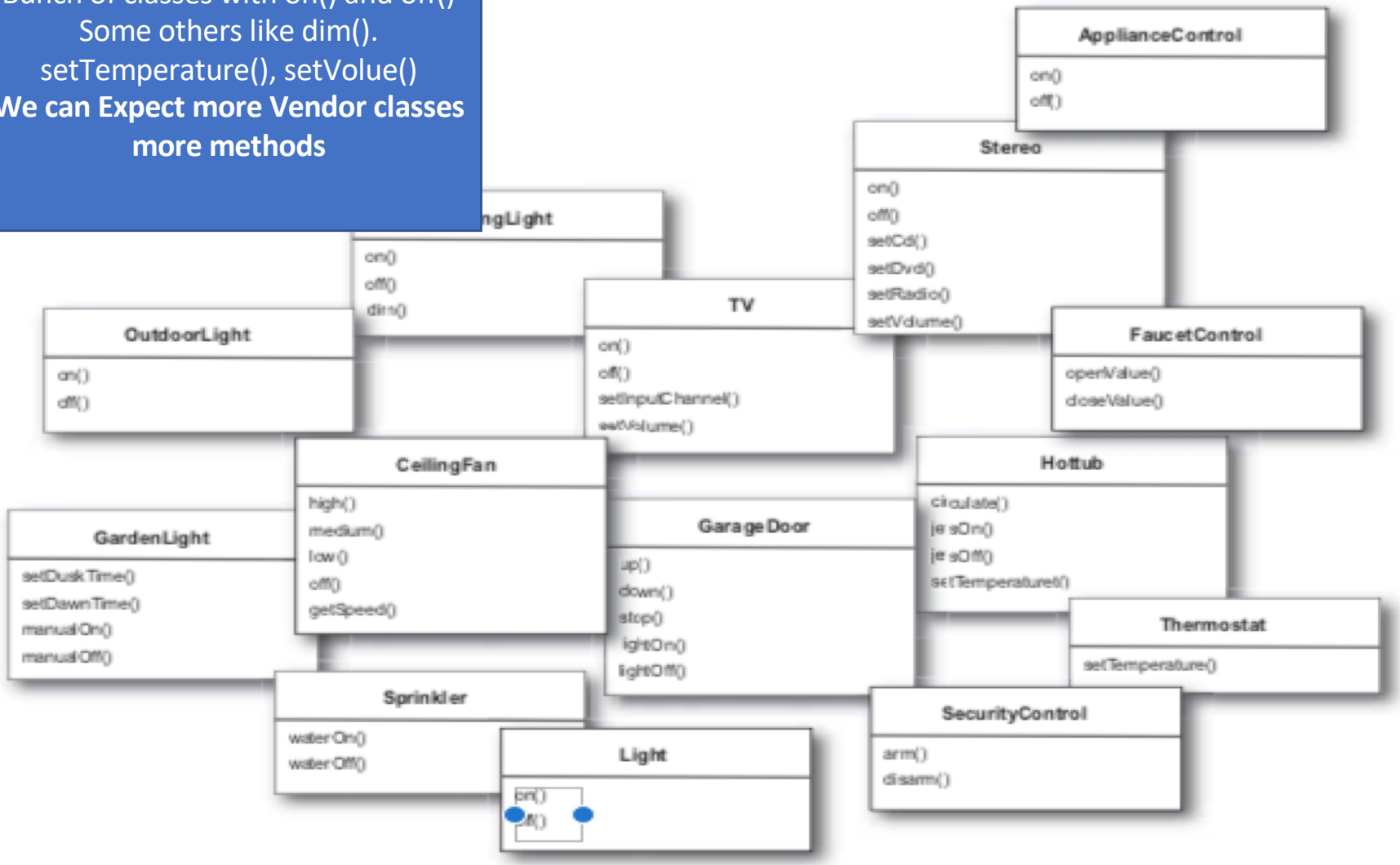
Bunch of classes with on() and off()



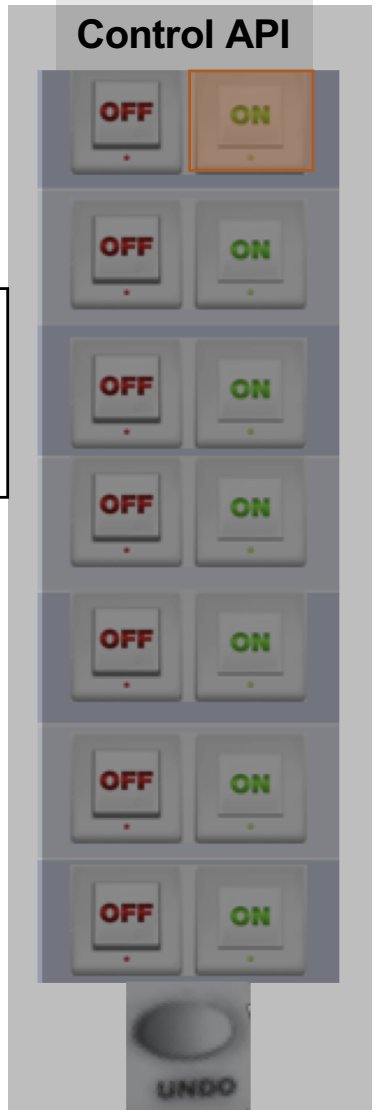
Bunch of classes with on() and off()
**Some others like dim().**
**setTemperature(), setVolue()**

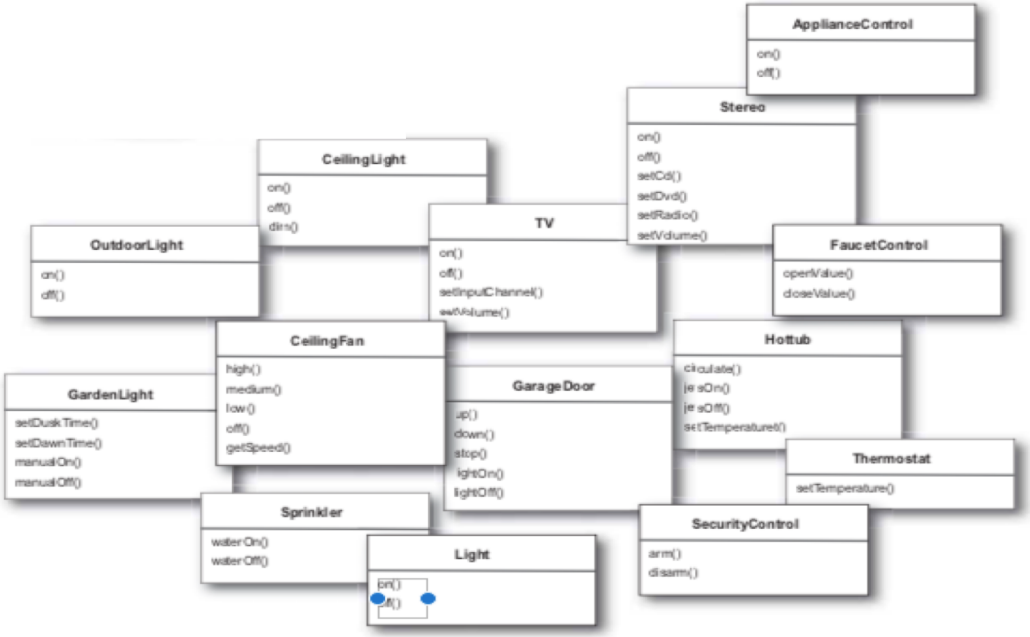


Bunch of classes with on() and off()
Some others like dim().
setTemperature(), setVolue()
**We can Expect more Vendor classes more methods**

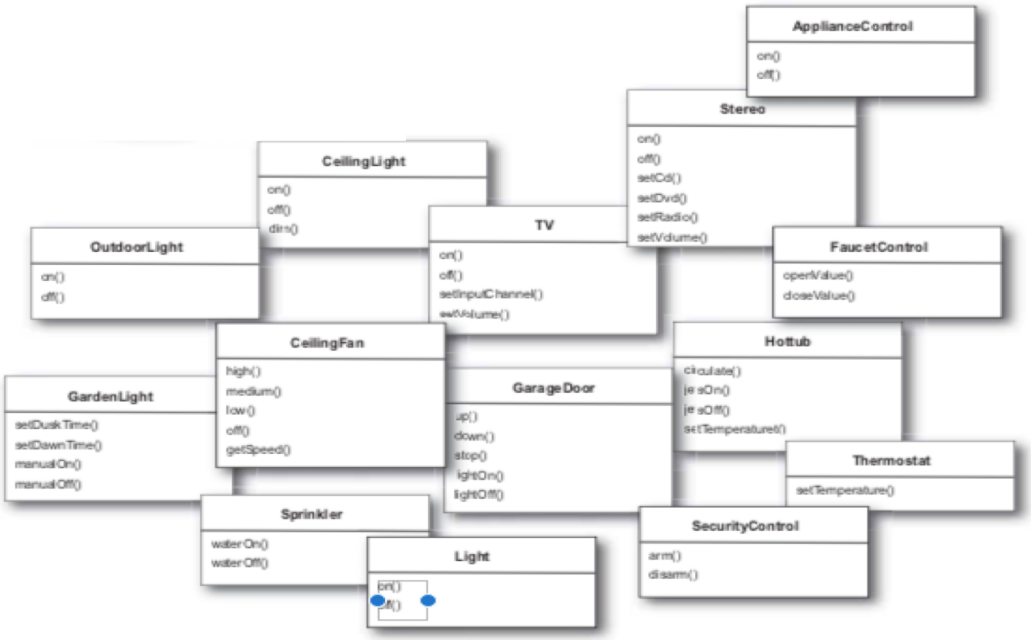


We need to interpret these buttons and what do they do



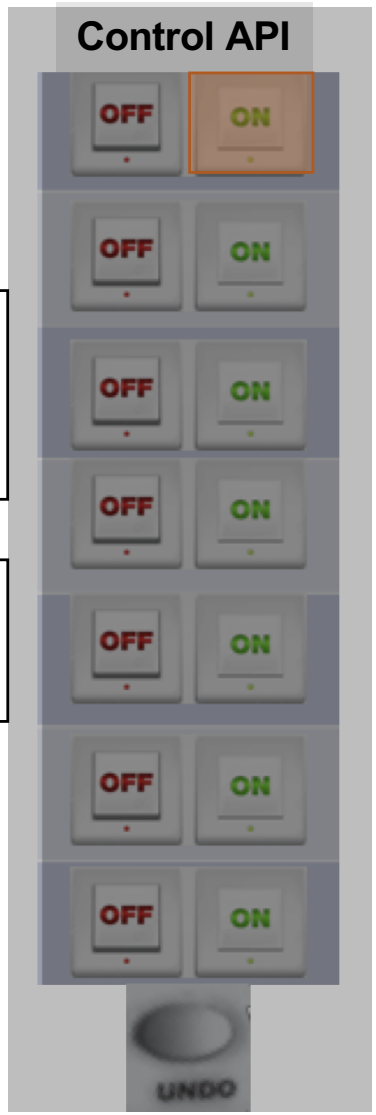
We need to interpret these buttons and what do they do

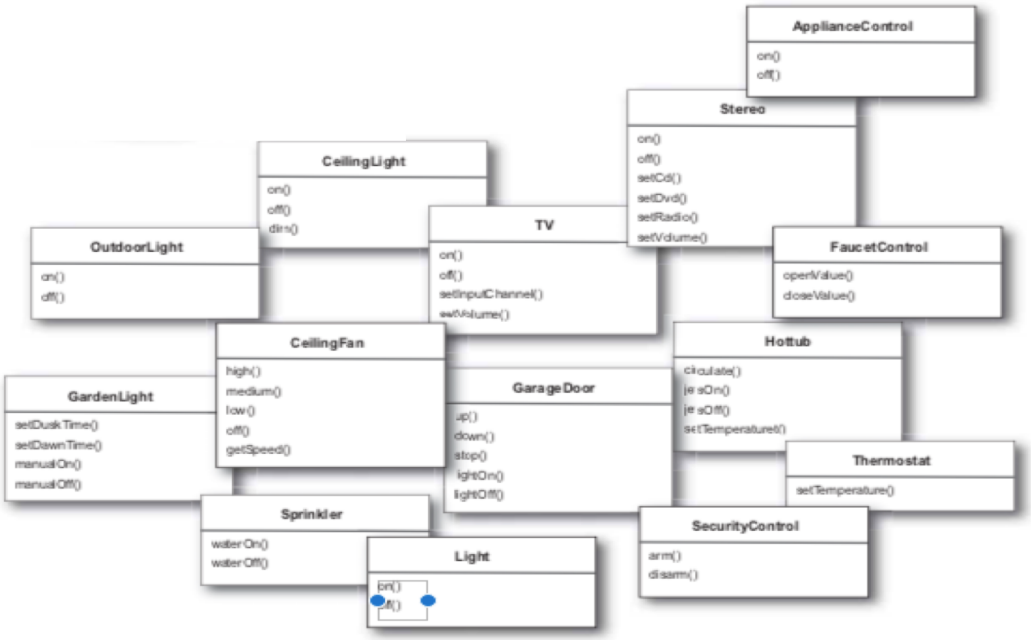




Call the corresponding methods but you do not care about home automation or how to turn on and off a specific device.

We need to interpret these buttons and what do they do

**He just know how to make requests**




Call the corresponding methods but you do not care about home automation or how to turn on and off a specific device.

Customer

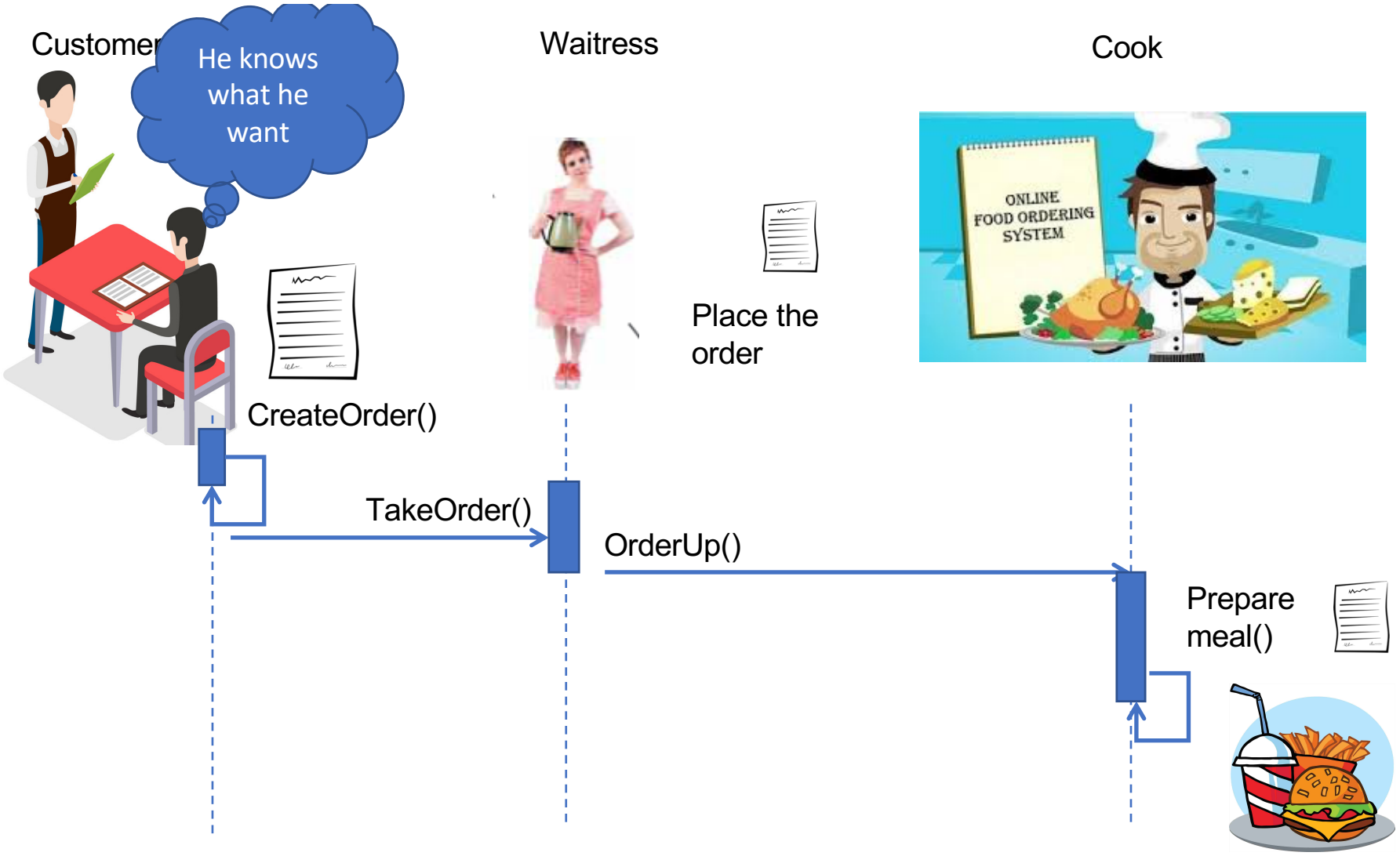
Give the waitress your order

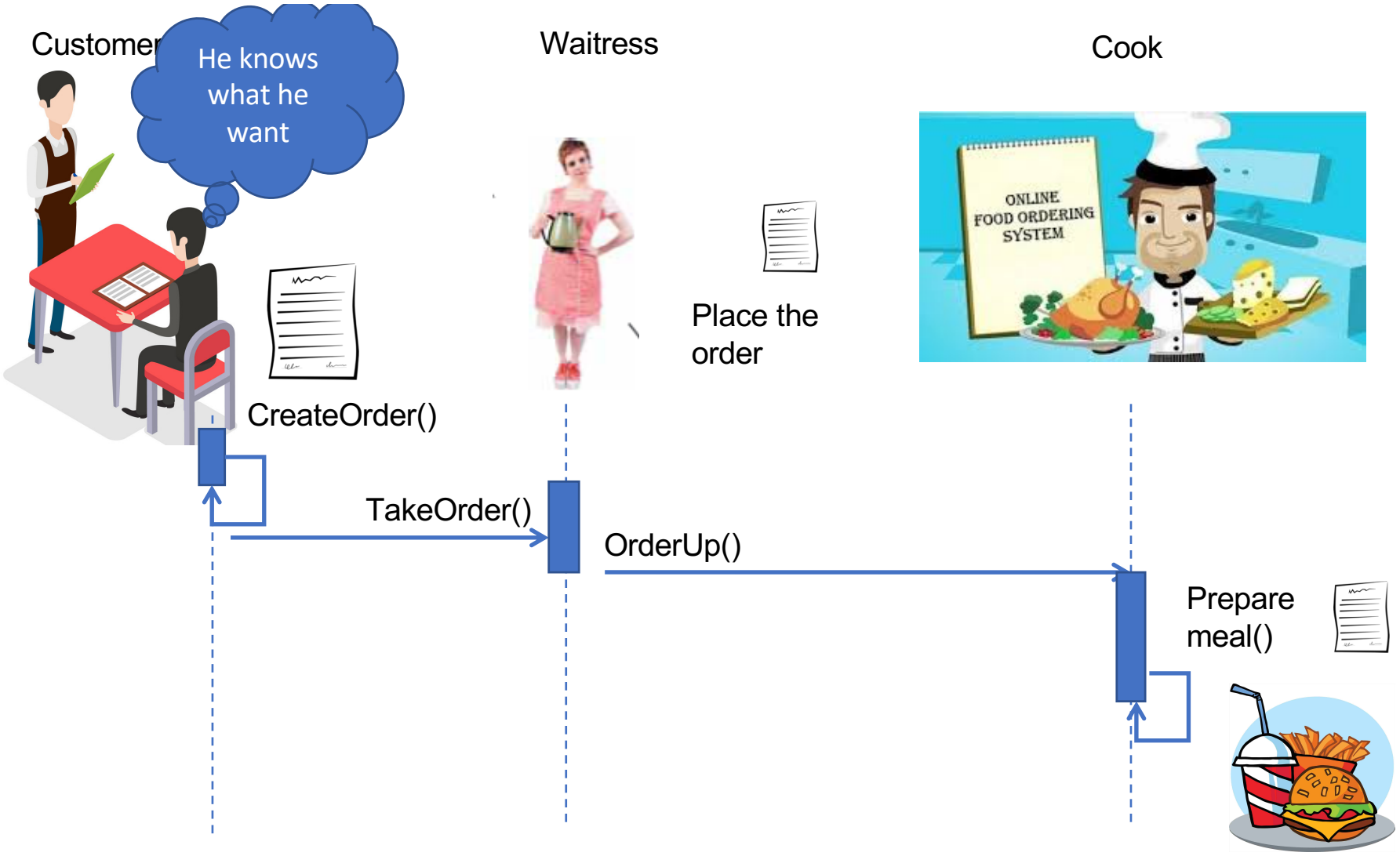
Waitress

Place the order

Cook


Prepare meal


Customer
He knows what he want
Waitress
Cook
ONLINE FOOD ORDERING SYSTEM
Place the order
CreateOrder()
TakeOrder()
OrderUp()
Prepare meal()


Customer
He knows what he want
Waitress
Cook
ONLINE FOOD ORDERING SYSTEM
Place the order
CreateOrder()
TakeOrder()
OrderUp()
Prepare meal()

Client

Invoker

Receiver

command


CreateCommand()

setCommand()

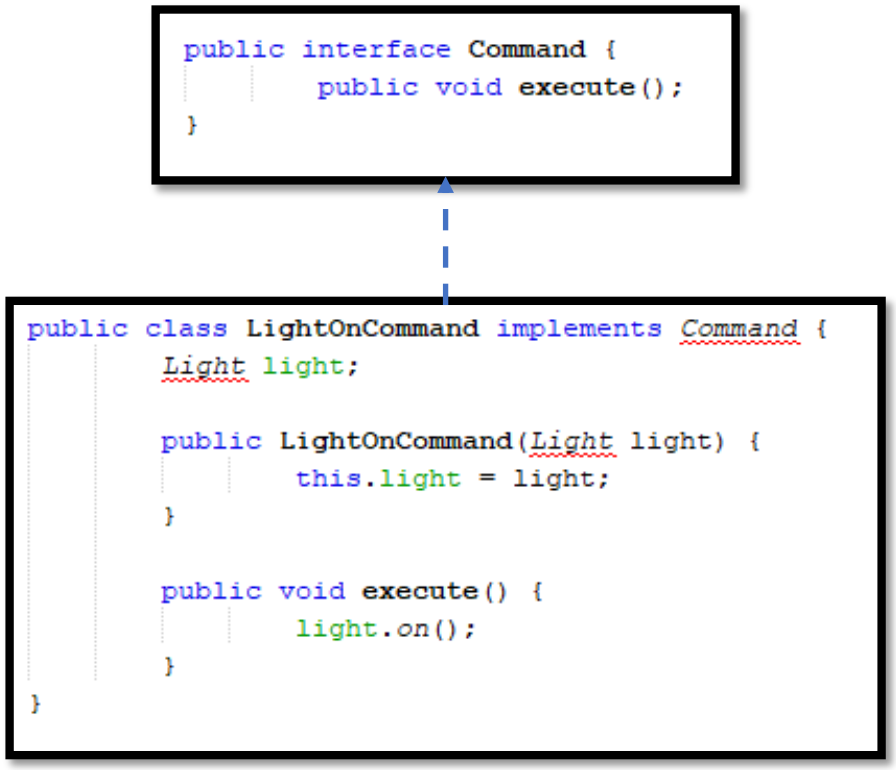
executeCommand()

execute()

Do your
stuff()

command

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


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

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

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

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

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

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

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

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

```java
public class Light {

        public Light() {
        }

        public void on() {
                System.out.println("Light is on");
        }

        public void off() {
                System.out.println("Light is off");
        }
}
```


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

```
public class GarageDoorOpenCommand implements Command {
        GarageDoor garageDoor;

        public GarageDoorOpenCommand(GarageDoor garageDoor) {
                this.garageDoor = garageDoor;
        }

        public void execute() {
                garageDoor.up();
        }
}
```

```
public class GarageDoor {

        public GarageDoor() {
        }

        public void up() {
                System.out.println("Garage Door is Open");
        }

        public void down() {
                System.out.println("Garage Door is Closed");
        }

        public void stop() {
                System.out.println("Garage Door is Stopped");
        }

        public void lightOn() {
                System.out.println("Garage light is on");
        }

        public void lightOff() {
                System.out.println("Garage light is off");
        }
}
```

```
public class SimpleRemoteControl {
        Command slot;

        public SimpleRemoteControl() {}

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

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

```
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();

    }
}
```

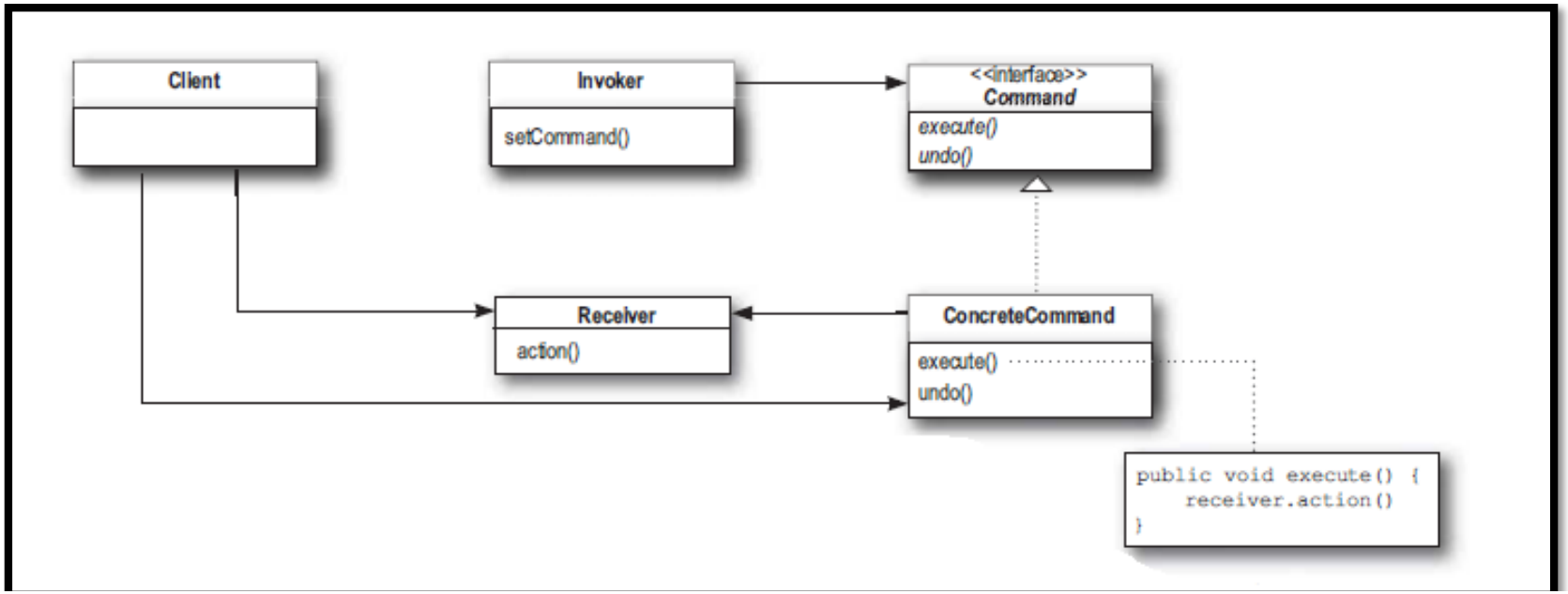
```
public class RemoteControlTest {
        public static void main(String[] args) {
                SimpleRemoteControl remote = new SimpleRemoteControl();
                Light light = new Light();
                LightOnCommand lightOn = new LightOnCommand(light);
                remote.setCommand(lightOn);
                remote.buttonWasPressed();

    }
}
```

# Now do the Garage Open

```
public class RemoteControlTest {
        public static void main(String[] args) {
                SimpleRemoteControl remote = new SimpleRemoteControl();
                Light light = new Light();
                GarageDoor garageDoor = new GarageDoor();
                LightOnCommand lightOn = new LightOnCommand(light);
                GarageDoorOpenCommand garageOpen =
                    new GarageDoorOpenCommand(garageDoor);

                remote.setCommand(lightOn);
                remote.buttonWasPressed();
                remote.setCommand(garageOpen);
                remote.buttonWasPressed();
    }
}
```

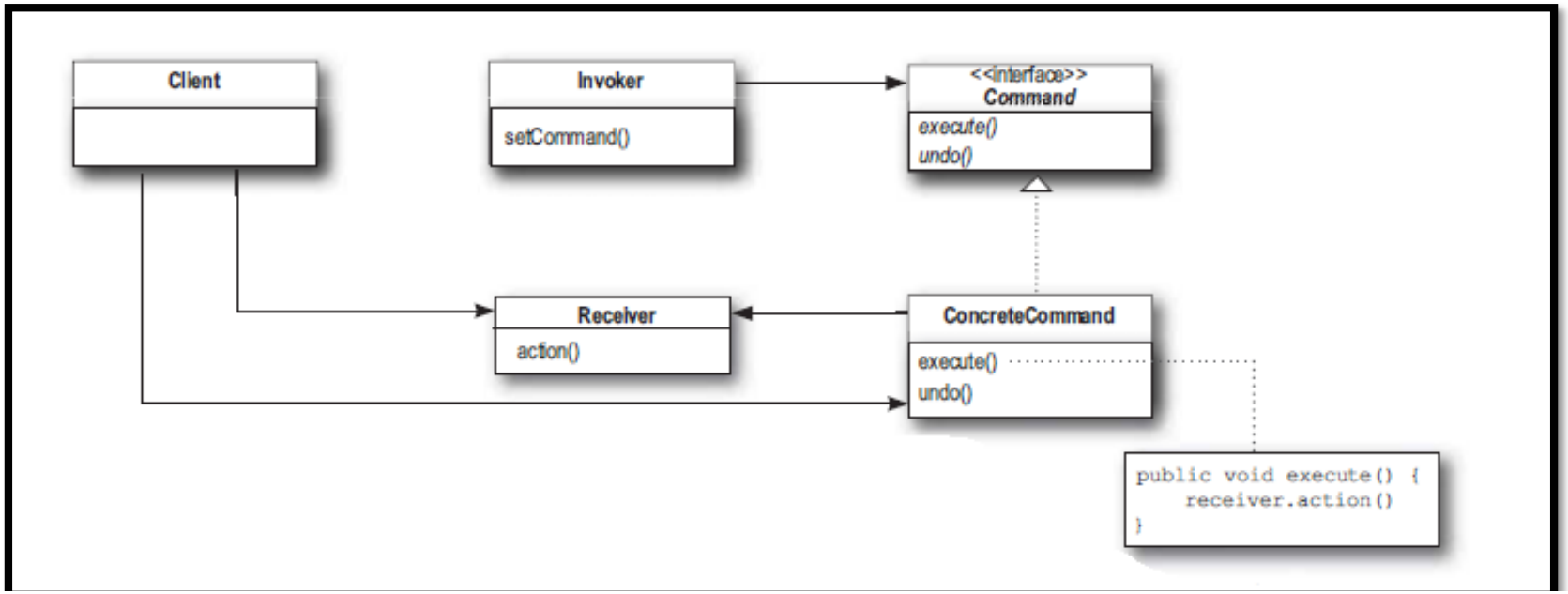
# Command Pattern

Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or long requests, and support undoable operations

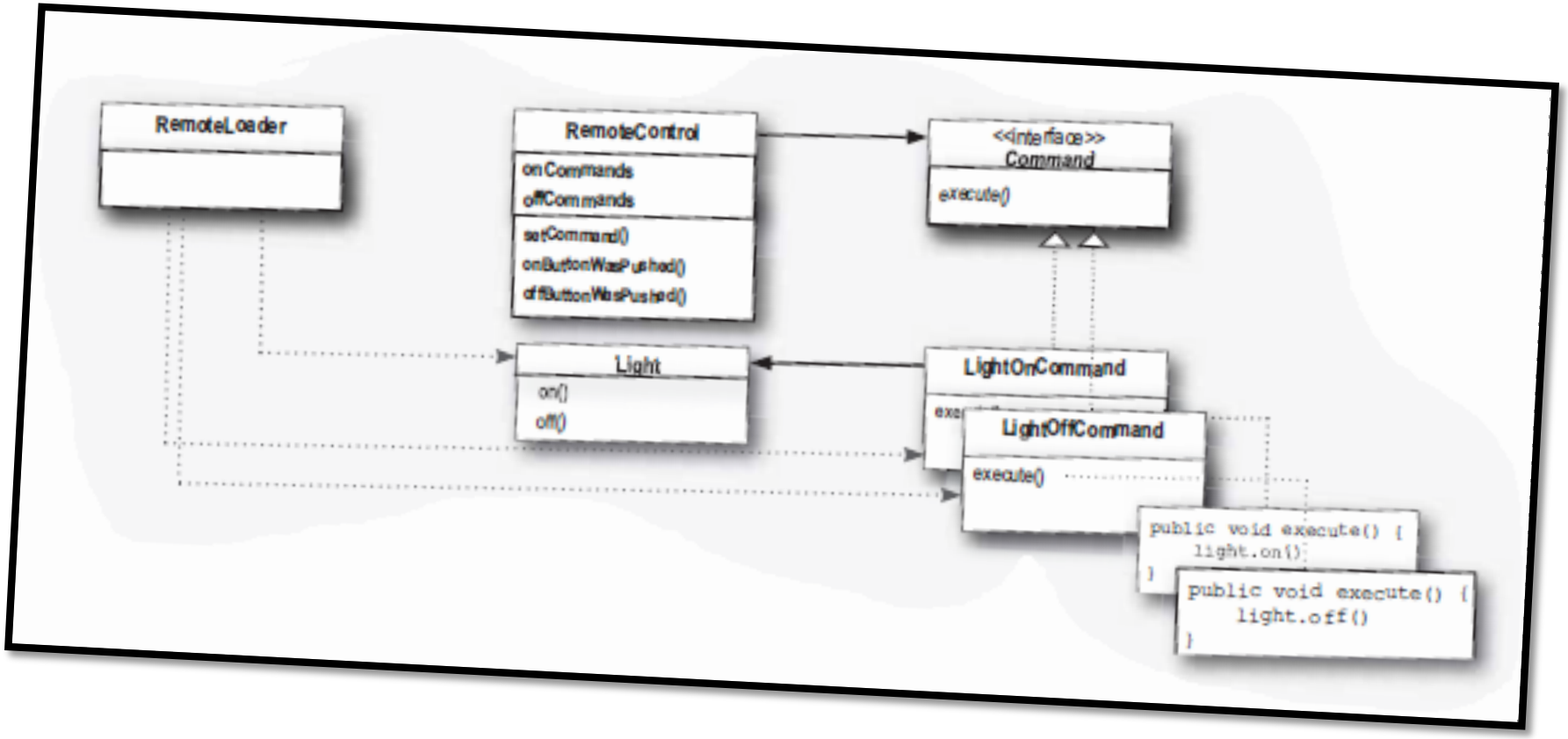


# Command Pattern

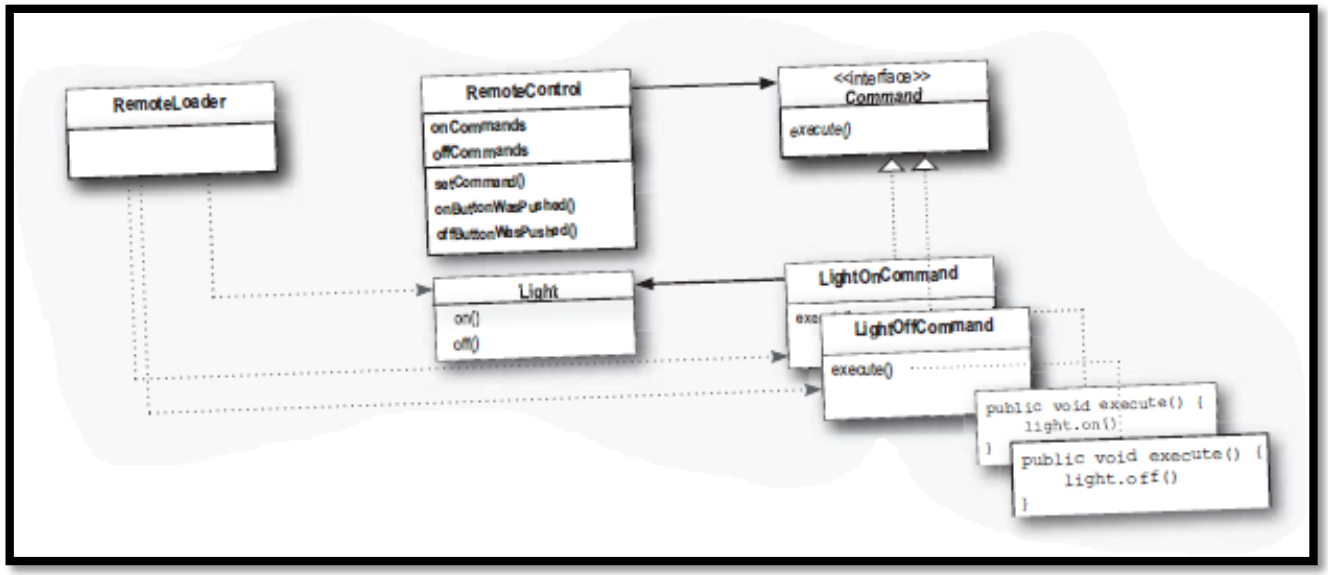
Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or long requests, and support undoable operations



# Command Pattern



# Let´s do the CODE – File available in BB



# Commands

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

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

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

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

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

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

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

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

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

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

        public StereoOffCommand(Stereo stereo) {
                this.stereo = stereo;
        }

        public void execute() {
                stereo.off();
        }
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
public class HottubOnCommand implements Command {
        Hottub hottub;

        public HottubOnCommand(Hottub hottub) {
                this.hottub = hottub;
        }

        public void execute() {
                hottub.on();
                hottub.heat();
                hottub.bubblesOn();
        }
}
```

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

```java
public class HottubOffCommand implements Command {
        Hottub hottub;

        public HottubOffCommand(Hottub hottub) {
                this.hottub = hottub;
        }

        public void execute() {
                hottub.cool();
                hottub.off();
        }
}
```

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

```
public class GarageDoorUpCommand implements Command {
        GarageDoor garageDoor;

        public GarageDoorUpCommand(GarageDoor garageDoor) {
                this.garageDoor = garageDoor;
        }

        public void execute() {
                garageDoor.up();
        }
}
```

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

```
public class GarageDoorDownCommand implements Command {
        GarageDoor garageDoor;

        public GarageDoorDownCommand(GarageDoor garageDoor) {
                this.garageDoor = garageDoor;
        }

        public void execute() {
                garageDoor.up();
        }
}
```

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

```java
public class CeilingFanOnCommand implements Command {
        CeilingFan ceilingFan;

        public CeilingFanOnCommand(CeilingFan ceilingFan) {
                this.ceilingFan = ceilingFan;
        }
        public void execute() {
                ceilingFan.high();
        }
}
```

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

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

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

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

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

```java
public class CeilingFanOffCommand implements Command {
        CeilingFan ceilingFan;

        public CeilingFanOffCommand(CeilingFan ceilingFan) {
                this.ceilingFan = ceilingFan;
        }
        public void execute() {
                ceilingFan.off();
        }
}
```

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

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

# Devices

```
public class Hottub {
        boolean on;
        int temperature;

        public Hottub() {
        }

        public void on() {
                on = true;
        }

        public void off() {
                on = false;
        }

        public void bubblesOn() {
                if (on) {
                        System.out.println("Hottub is bubbling!"
                }
        }

        public void bubblesOff() {
                if (on) {
                        System.out.println("Hottub is not bubbli
                }
        }

        public void jetsOn() {
                if (on) {
                        System.out.println("Hottub jets are on");
                }
        }
        public void jetsOff() {
                if (on) {
                        System.out.println("Hottub jets are off");
                }
        }

        public void setTemperature(int temperature) {
                this.temperature = temperature;
        }

        public void heat() {
                temperature = 105;
                System.out.println("Hottub is heating to a steaming 105 degrees");
        }

        public void cool() {
                temperature = 98;
                System.out.println("Hottub is cooling to 98 degrees");
        }

}
```

```java
public class GarageDoor {
	String location;

	public GarageDoor(String location) {
		this.location = location;
	}

	public void up() {
		System.out.println(location + " garage Door is Up");
	}

	public void down() {
		System.out.println(location + " garage Door is Down");
	}

	public void stop() {
		System.out.println(location + " garage Door is Stopped");
	}

	public void lightOn() {
		System.out.println(location + " garage light is on");
	}

	public void lightOff() {
		System.out.println(location + " garage light is off");
	}
}
```

```java
public class TV {
	String location;
	int channel;

	public TV(String location) {
		this.location = location;
	}

	public void on() {
		System.out.println("TV is on");
	}

	public void off() {
		System.out.println("TV is off");
	}

	public void setInputChannel() {
		this.channel = 3;
		System.out.println("Channel is set for VCR");
	}
}
```

```java
public class Stereo {
        String location;

        public Stereo(String location) {
                this.location = location;
        }

        public void on() {
                System.out.println(location + " stereo is on");
        }

        public void off() {
                System.out.println(location + " stereo is off");
        }

        public void setCD() {
                System.out.println(location + " stereo is set for CD input");
        }

        public void setDVD() {
                System.out.println(location + " stereo is set for DVD input");
        }

        public void setRadio() {
                System.out.println(location + " stereo is set for Radio");
        }

        public void setVolume(int volume) {
                // code to set the volume
                // valid range: 1-11 (after all 11 is better than 10, right?)
                System.out.println(location + " Stereo volume set to " + volume);
        }
}
```

```java
public class Light {
	String location = "";

	public Light(String location) {
		this.location = location;
	}

	public void on() {
		System.out.println(location + " light is on");
	}

	public void off() {
		System.out.println(location + " light is off");
	}
}
```

```java
public class CeilingFan {
        String location = "";
        int level;
        public static final int HIGH = 2;
        public static final int MEDIUM = 1;
        public static final int LOW = 0;

        public CeilingFan(String location) {
                this.location = location;
        }

        public void high() {
                // turns the ceiling fan on to high
                level = HIGH;
                System.out.println(location + " ceiling fan is on high");

        }

        public void medium() {
                // turns the ceiling fan on to medium
                level = MEDIUM;
                System.out.println(location + " ceiling fan is on medium");
        }

        public void low() {
                // turns the ceiling fan on to low
                level = LOW;
                System.out.println(location + " ceiling fan is on low");
        }

        public void off() {
                // turns the ceiling fan off
                level = 0;
                System.out.println(location + " ceiling fan is off");
        }

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

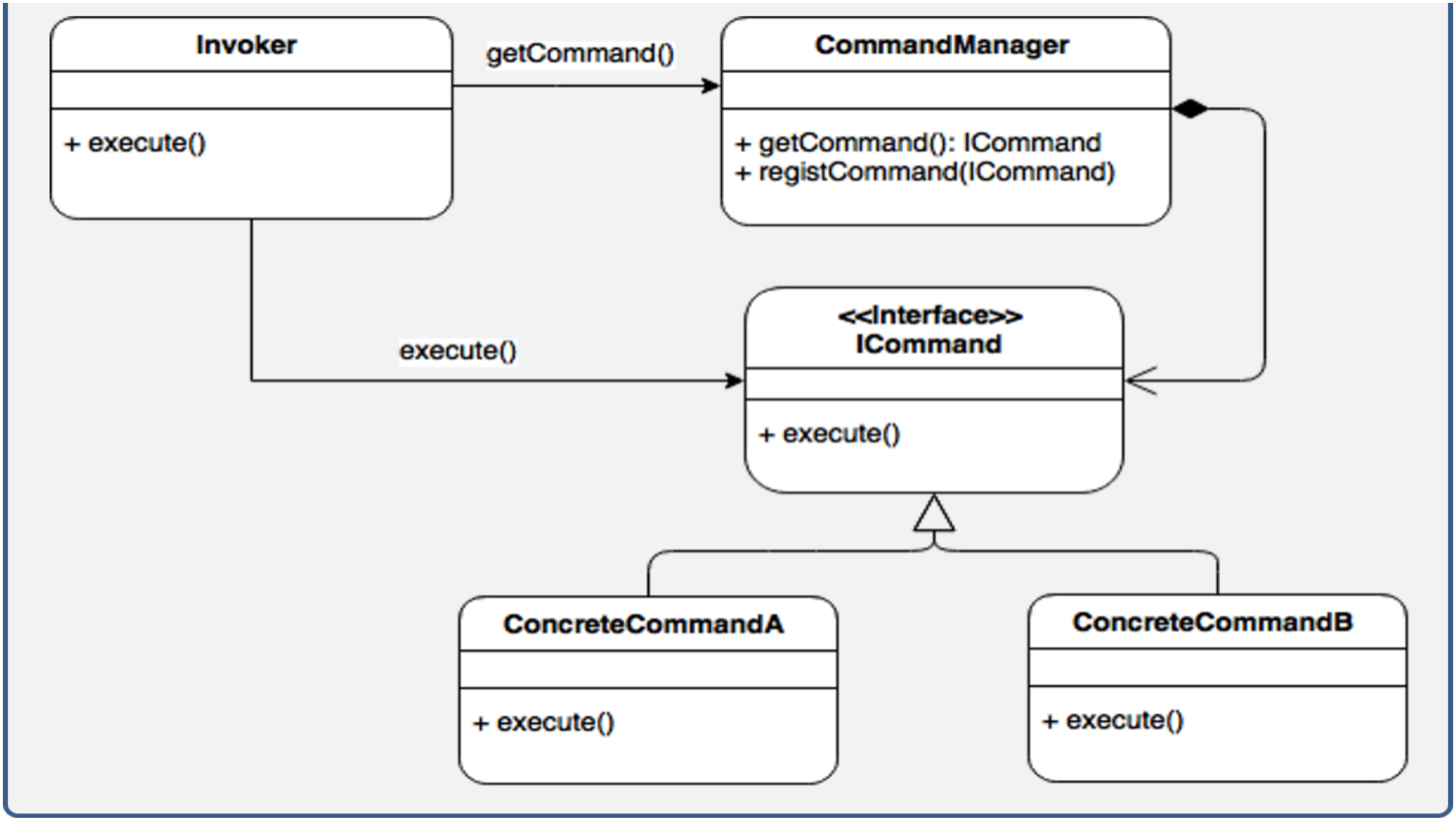
# Remote Loader

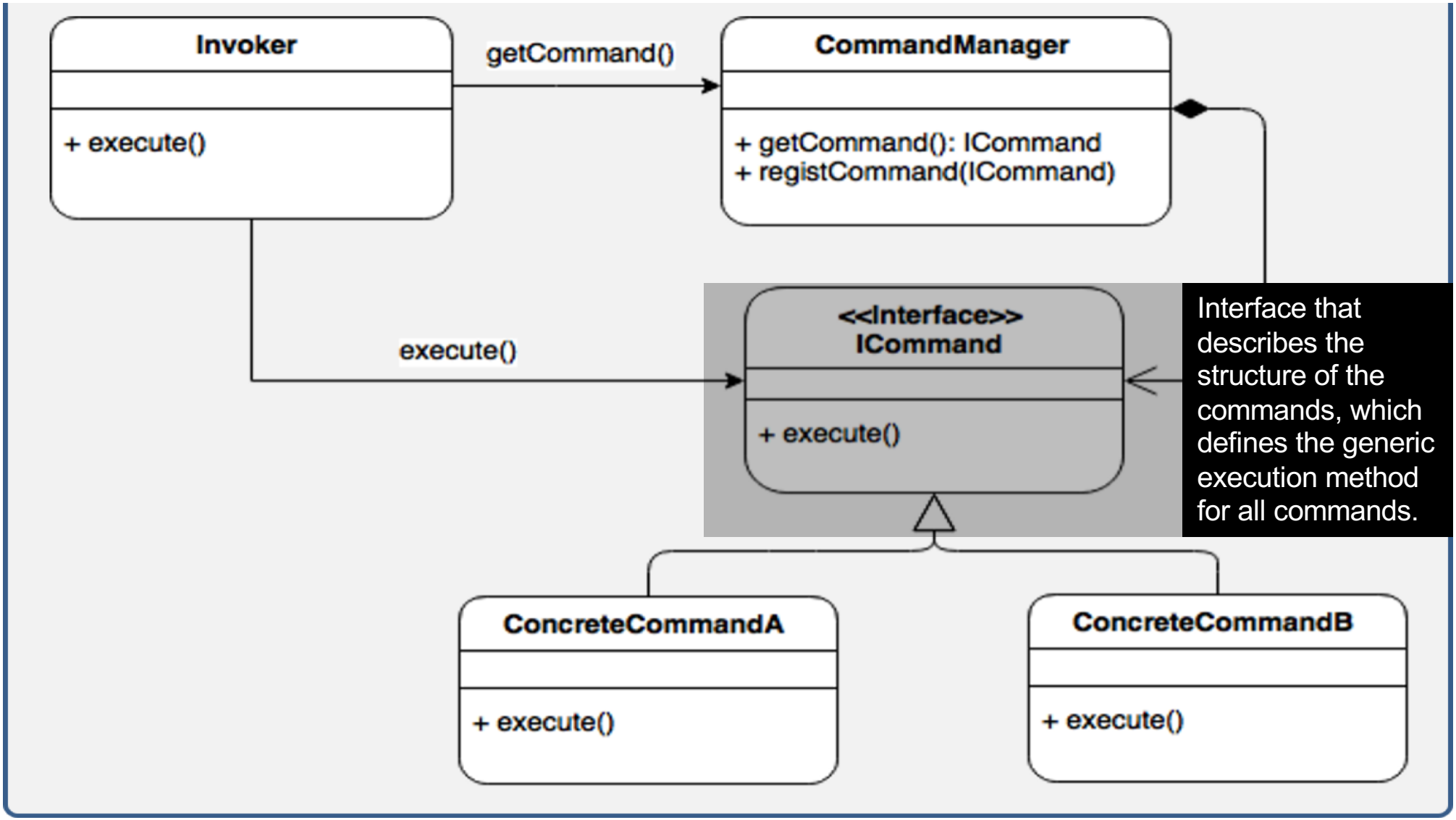
```
public class RemoteLoader {

        public static void main(String[] args) {
                RemoteControl remoteControl = new RemoteControl();

                Light livingRoomLight = new Light("Living Room");
                Light kitchenLight = new Light("Kitchen");
                CeilingFan ceilingFan= new CeilingFan("Living Room");
                GarageDoor garageDoor = new GarageDoor("");
                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);

                CeilingFanOnCommand ceilingFanOn =
                                new CeilingFanOnCommand(ceilingFan);
                CeilingFanOffCommand ceilingFanOff =
                                new CeilingFanOffCommand(ceilingFan);

                GarageDoorUpCommand garageDoorUp =
                                new GarageDoorUpCommand(garageDoor);
                GarageDoorDownCommand garageDoorDown =
                                new GarageDoorDownCommand(garageDoor);

                StereoOnWithCDCommand stereoOnWithCD =
                                new StereoOnWithCDCommand(stereo);
                StereoOffCommand  stereoOff =
                                new StereoOffCommand(stereo);
```

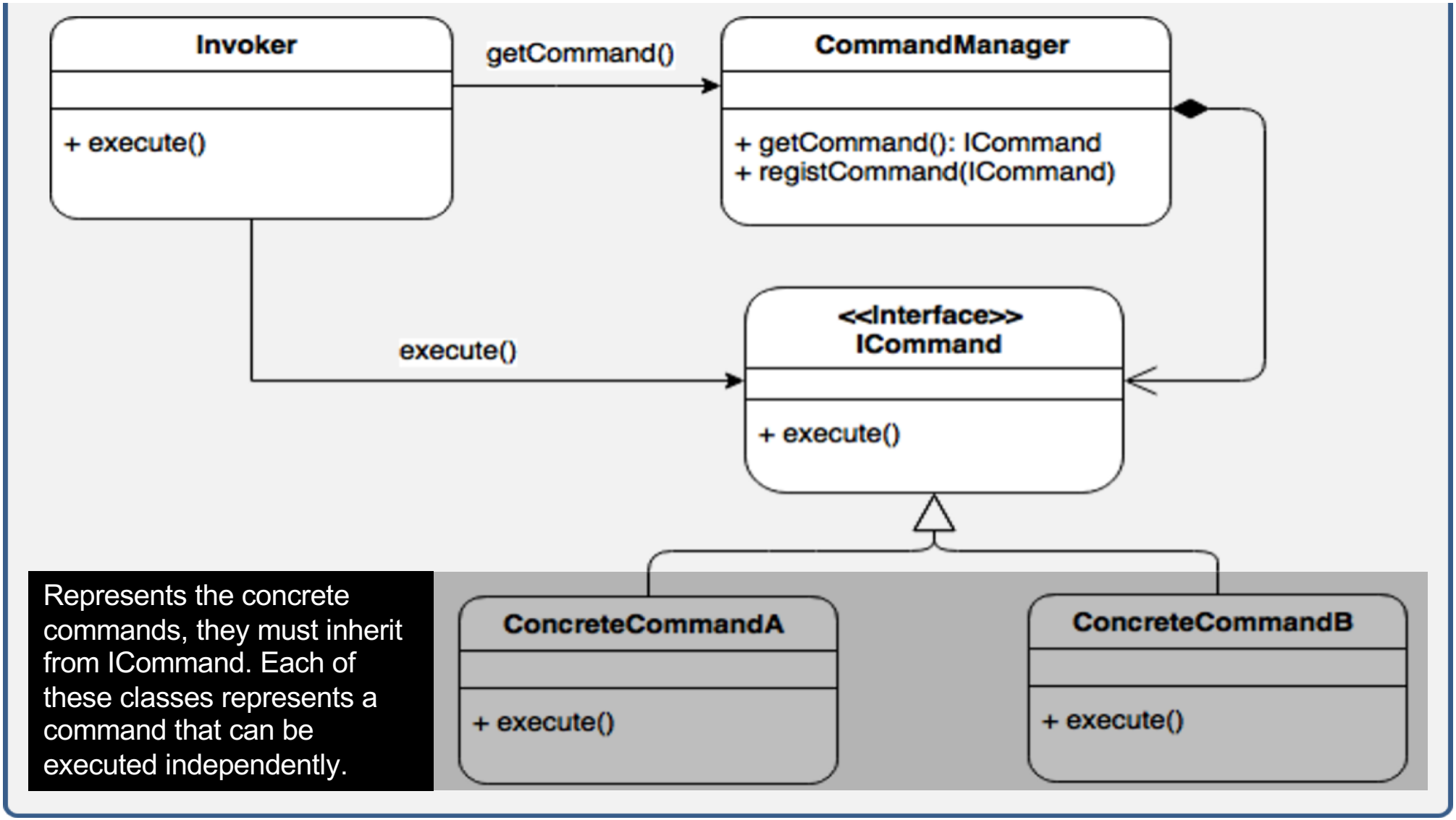
```
                remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
                remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
                remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
                remoteControl.setCommand(3, stereoOnWithCD, stereoOff);

                System.out.println(remoteControl);

                remoteControl.onButtonWasPushed(0);
                remoteControl.offButtonWasPushed(0);
                remoteControl.onButtonWasPushed(1);
                remoteControl.offButtonWasPushed(1);
                remoteControl.onButtonWasPushed(2);
                remoteControl.offButtonWasPushed(2);
                remoteControl.onButtonWasPushed(3);
                remoteControl.offButtonWasPushed(3);
}
```

# Remote Control

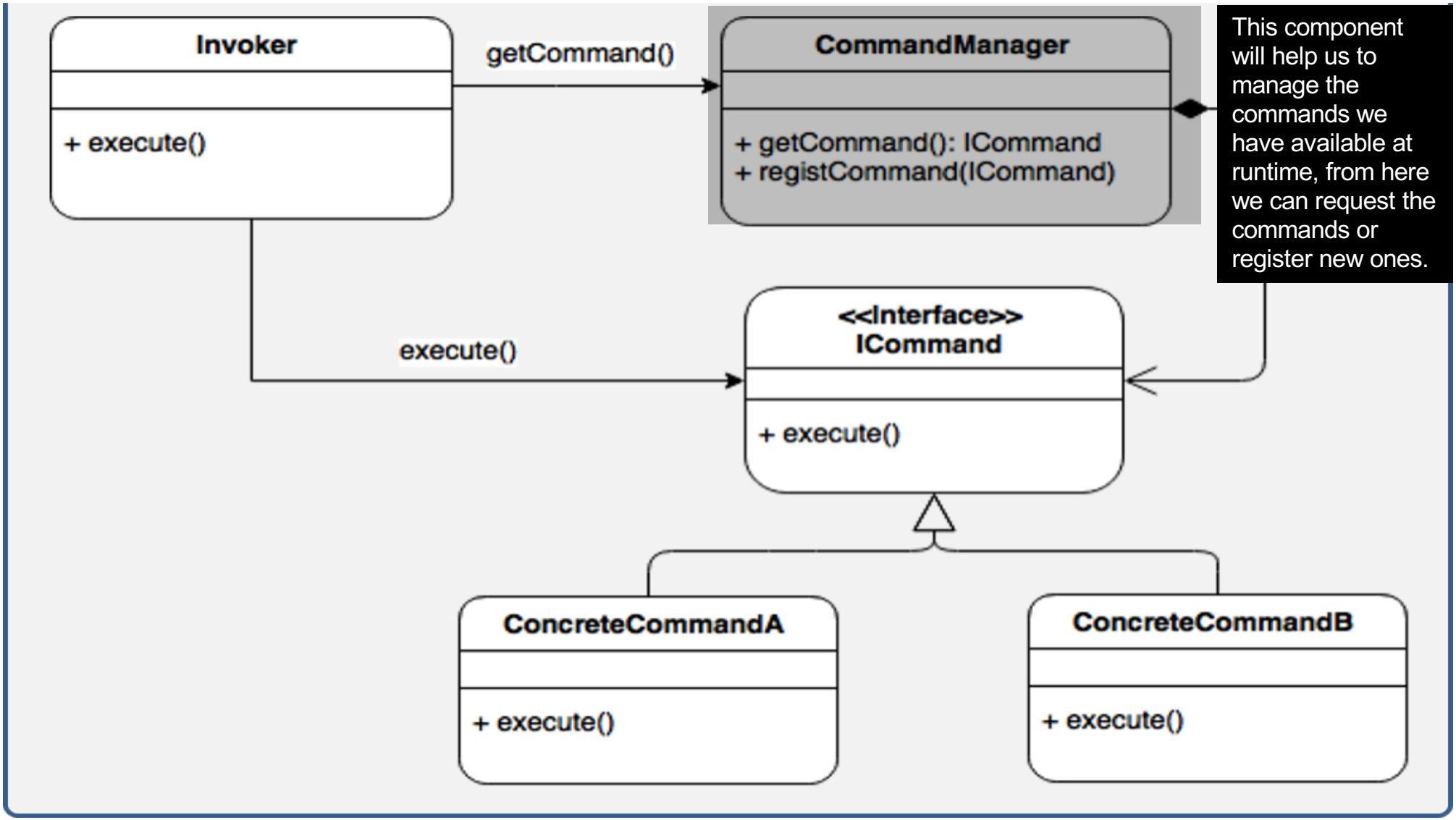
```
public class RemoteControl {
        Command[] onCommands;
        Command[] offCommands;

        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;
        }

        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();
        }
}
```

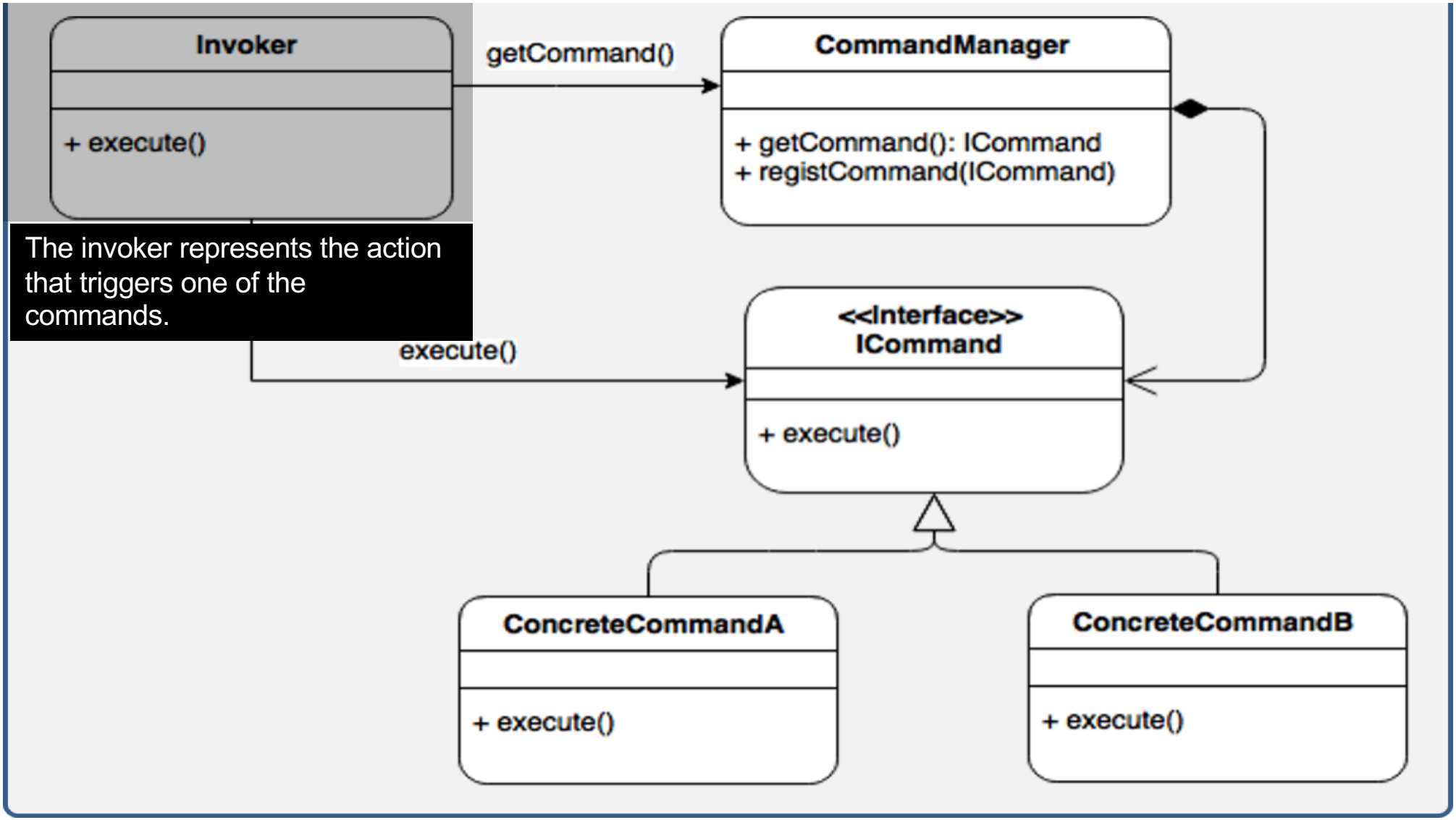

Invoker
+ execute()
getCommand()
CommandManager
+ getCommand(): ICommand
+ registCommand(ICommand)
execute()
<<Interface>>
ICommand
+ execute()
ConcreteCommandA
+ execute()
ConcreteCommandB
+ execute()

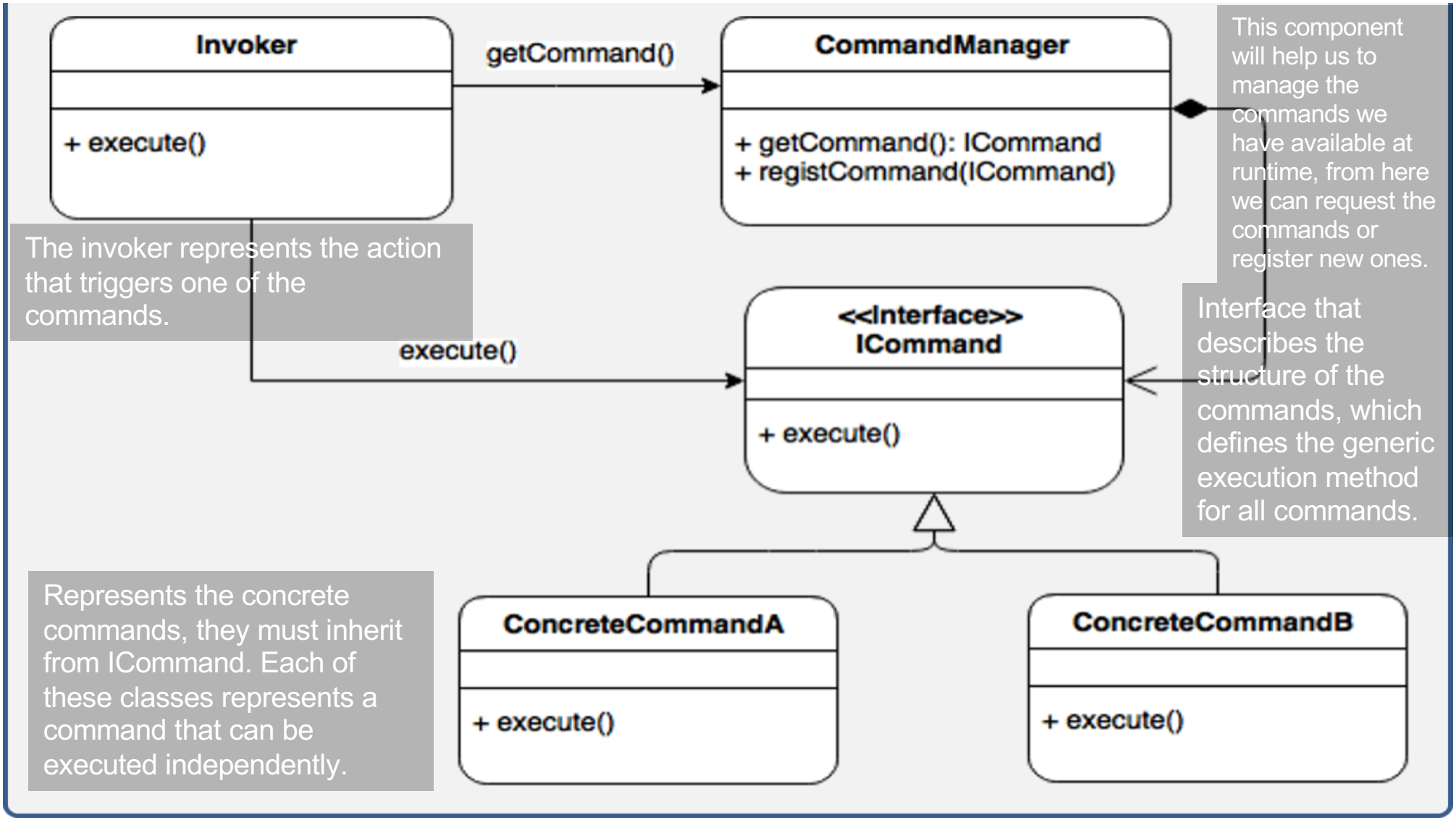

Invoker
+ execute()
getCommand()
CommandManager
+ getCommand(): ICommand
+ registCommand(ICommand)
execute()
<<Interface>>
ICommand
+ execute()
Interface that describes the structure of the commands, which defines the generic execution method for all commands.
ConcreteCommandA
+ execute()
ConcreteCommandB
+ execute()



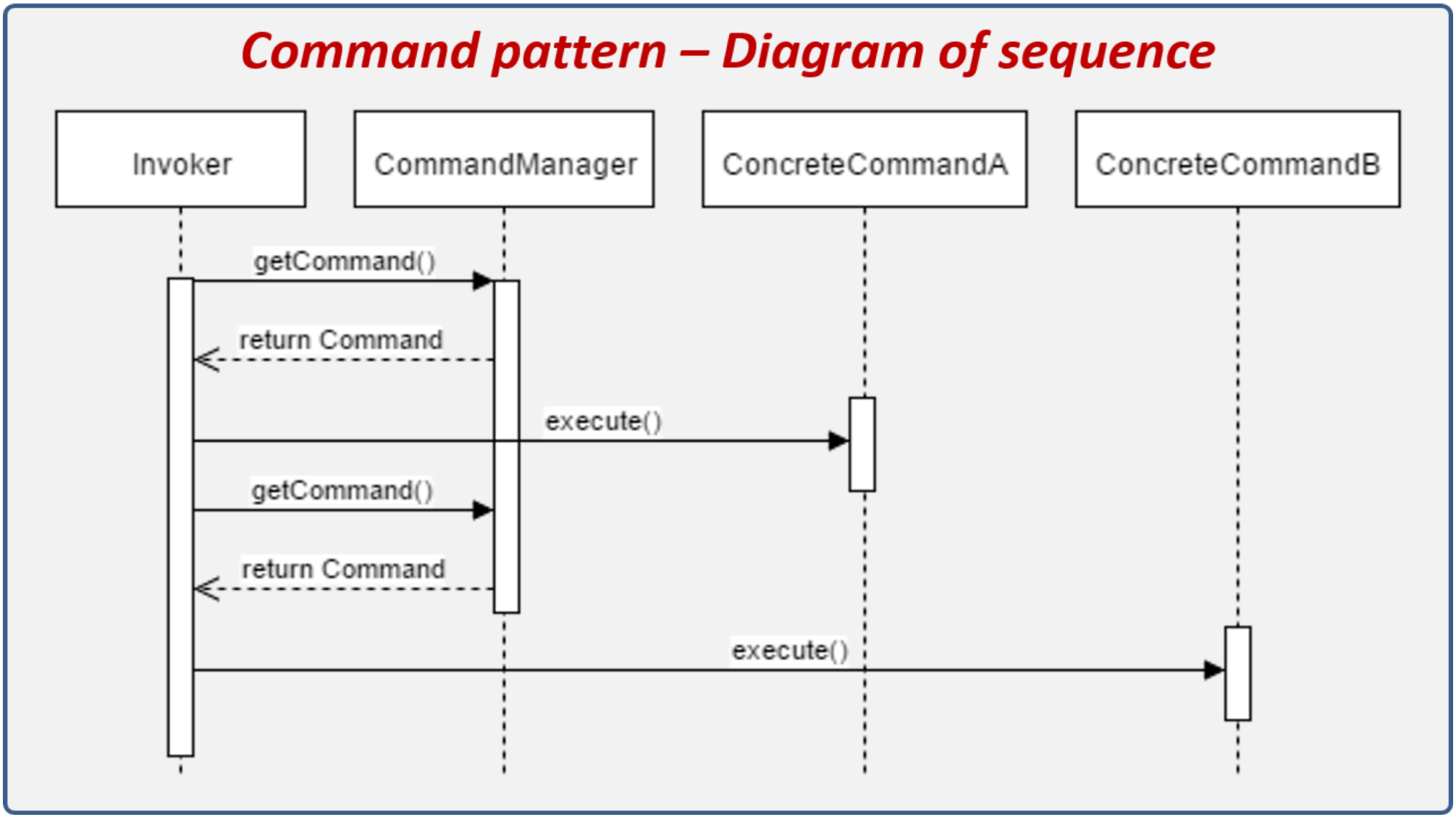

Represents the concrete commands, they must inherit from ICommand. Each of these classes represents a command that can be executed independently.


Invoker
+ execute()
getCommand()
CommandManager
+ getCommand(): ICommand
+ registCommand(ICommand)
This component will help us to manage the commands we have available at runtime, from here we can request the commands or register new ones.
execute()
<<Interface>>
ICommand
+ execute()
ConcreteCommandA
+ execute()
ConcreteCommandB
+ execute()


Invoker
+ execute()
getCommand()
CommandManager
+ getCommand(): ICommand
+ registCommand(ICommand)
The invoker represents the action that triggers one of the commands.
execute()
<<Interface>>
ICommand
+ execute()
ConcreteCommandA
+ execute()
ConcreteCommandB
+ execute()


Invoker
+ execute()
getCommand()
CommandManager
+ getCommand(): ICommand
+ registCommand(ICommand)
This component will help us to manage the commands we have available at runtime, from here we can request the commands or register new ones.
The invoker represents the action that triggers one of the commands.
execute()
<<Interface>>
ICommand
+ execute()
Interface that describes the structure of the commands, which defines the generic execution method for all commands.
Represents the concrete commands, they must inherit from ICommand. Each of these classes represents a command that can be executed independently.
ConcreteCommandA
+ execute()
ConcreteCommandB
+ execute()

# Command pattern – Diagram of sequence



```
 6  public class RemoteControl {
 7      Command[] onCommands;
 8      Command[] offCommands;
 9
10      public RemoteControl() {
11          onCommands = new Command[7];
12          offCommands = new Command[7];
13
14          for (int i = 0; i < 7; i++) {
15              onCommands[i] = () -> { };
16              offCommands[i] = () -> { };
17          }
18      }
19
20      public void setCommand(int slot, Command onCommand, Command offCommand) {
21          onCommands[slot] = onCommand;
22          offCommands[slot] = offCommand;
23      }
24
25      public void onButtonWasPushed(int slot) {
26          onCommands[slot].execute();
27      }
28
29      public void offButtonWasPushed(int slot) {
30          offCommands[slot].execute();
31      }
32
```

```
32
33      public String toString() {
34          StringBuffer stringBuff = new StringBuffer();
35          stringBuff.append("\n------ Remote Control -------\n");
36          for (int i = 0; i < onCommands.length; i++) {
37              stringBuff.append("[slot " + i + "] " + onCommands[i].getClass().getName()
38                  + "    " + offCommands[i].getClass().getName() + "\n");
39          }
40          return stringBuff.toString();
41      }
42
43  }
44
```

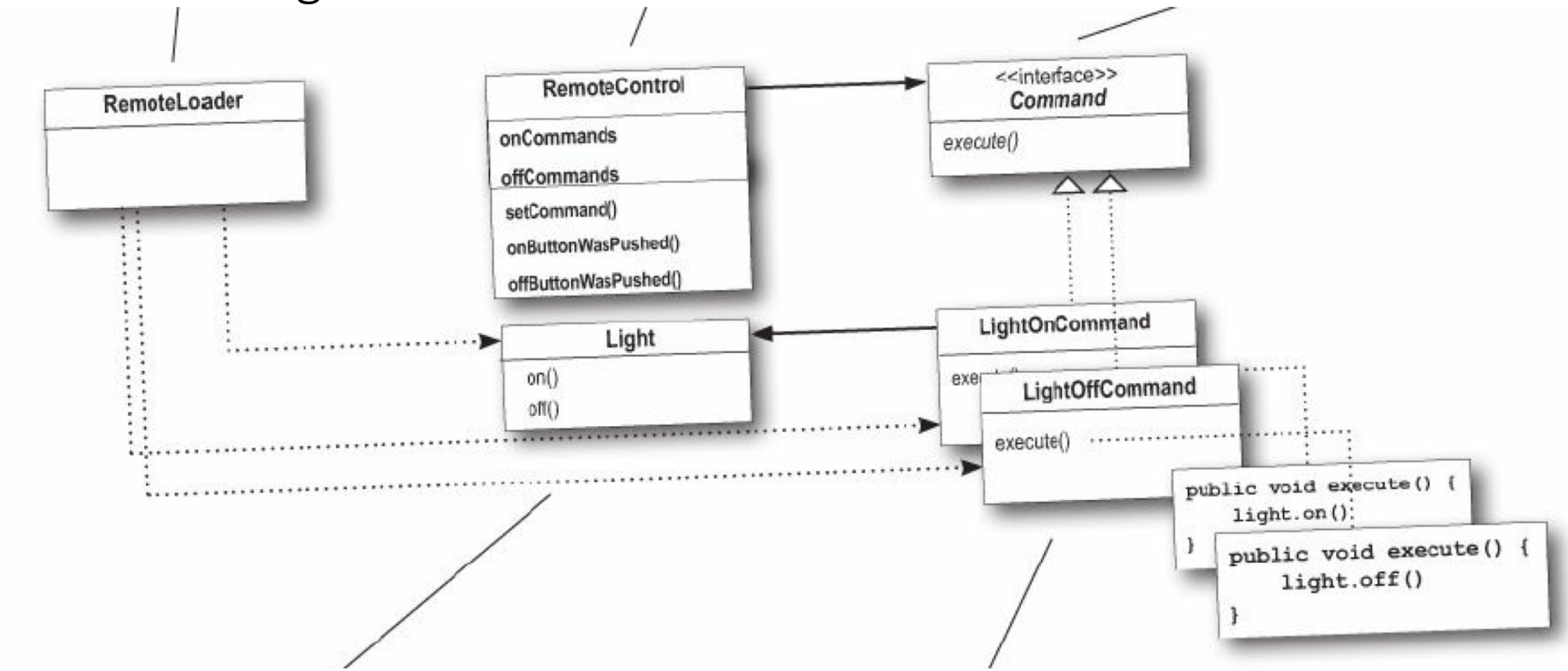
```
 2
 3  public class LightOffCommand implements Command {
 4      Light light;
 5
 6      public LightOffCommand(Light light) {
 7          this.light = light;
 8      }
 9
10      public void execute() {
11          light.off();
12      }
13  }
14
```

```
2
3  public class StereoOnWithCDCommand implements Command {
4      Stereo stereo;
5
6      public StereoOnWithCDCommand(Stereo stereo) {
7          this.stereo = stereo;
8      }
9
10     public void execute() {
11         stereo.on();
12         stereo.setCD();
13         stereo.setVolume(11);
14     }
15 }
16
```

```
2
3  public class RemoteLoader {
4
5      public static void main(String[] args) {
6          RemoteControl remoteControl = new RemoteControl();
7
8          Light livingRoomLight = new Light("Living Room");
9          Light kitchenLight = new Light("Kitchen");
10         CeilingFan ceilingFan= new CeilingFan("Living Room");
11         GarageDoor garageDoor = new GarageDoor("");
12         Stereo stereo = new Stereo("Living Room");
13
14         LightOnCommand livingRoomLightOn =
15             new LightOnCommand(livingRoomLight);
16         LightOffCommand livingRoomLightOff =
17             new LightOffCommand(livingRoomLight);
18         LightOnCommand kitchenLightOn =
19             new LightOnCommand(kitchenLight);
20         LightOffCommand kitchenLightOff =
21             new LightOffCommand(kitchenLight);
22
23         CeilingFanOnCommand ceilingFanOn =
24             new CeilingFanOnCommand(ceilingFan);
25         CeilingFanOffCommand ceilingFanOff =
26             new CeilingFanOffCommand(ceilingFan);
27
28         GarageDoorUpCommand garageDoorUp =
29             new GarageDoorUpCommand(garageDoor);
30         GarageDoorDownCommand garageDoorDown =
31             new GarageDoorDownCommand(garageDoor);
32
```

```
32
33          StereoOnWithCDCommand stereoOnWithCD =
34                  new StereoOnWithCDCommand(stereo);
35          StereoOffCommand  stereoOff =
36                  new StereoOffCommand(stereo);
37
38          remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
39          remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
40          remoteControl.setCommand(2, ceilingFanOn, ceilingFanOff);
41          remoteControl.setCommand(3, stereoOnWithCD, stereoOff);
42
43          System.out.println(remoteControl);
44
45          remoteControl.onButtonWasPushed(0);
46          remoteControl.offButtonWasPushed(0);
47          remoteControl.onButtonWasPushed(1);
48          remoteControl.offButtonWasPushed(1);
49          remoteControl.onButtonWasPushed(2);
50          remoteControl.offButtonWasPushed(2);
51          remoteControl.onButtonWasPushed(3);
52          remoteControl.offButtonWasPushed(3);
53      }
54  }
55
```

# Diseño API de control remoto para la automatización del hogar



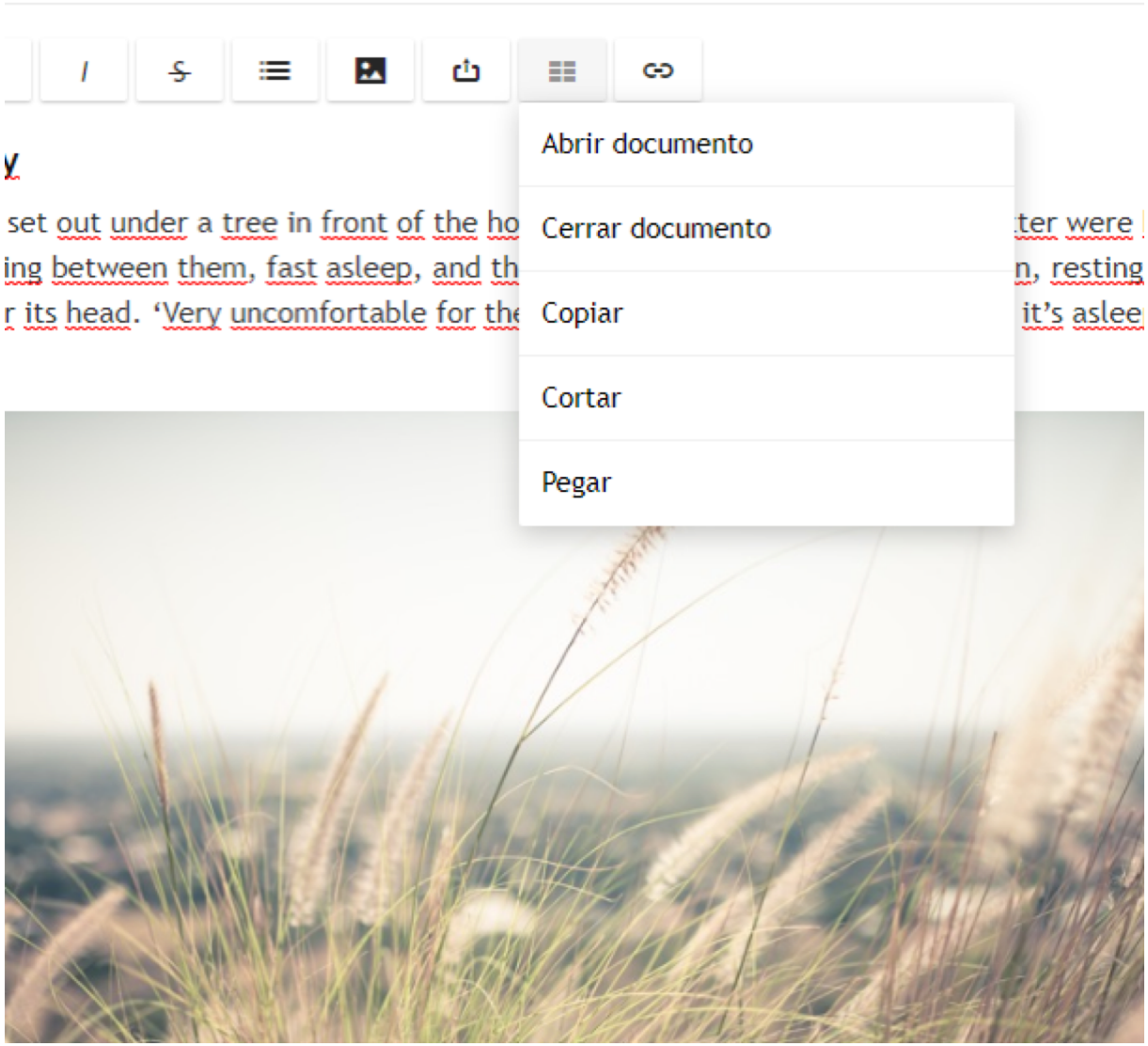
# Usa el patrón de comando cuando quieras:

- Parametrizar objetos por una acción a realizar
- Especificar, poner en cola y ejecutar solicitudes en diferentes momentos.
- Necesitas deshacer cambios.
- Admitir cambios de login para que puedan volver a iniciar en caso de un fallo del sistema.
- Estructurar un sistema alrededor de operaciones de alto nivel basadas en operaciones primitivas.
- Cuando necesita desacoplar un objeto que realiza una solicitud de los objetos que saben cómo realizar la solicitud, utiliza el Patrón de comando.

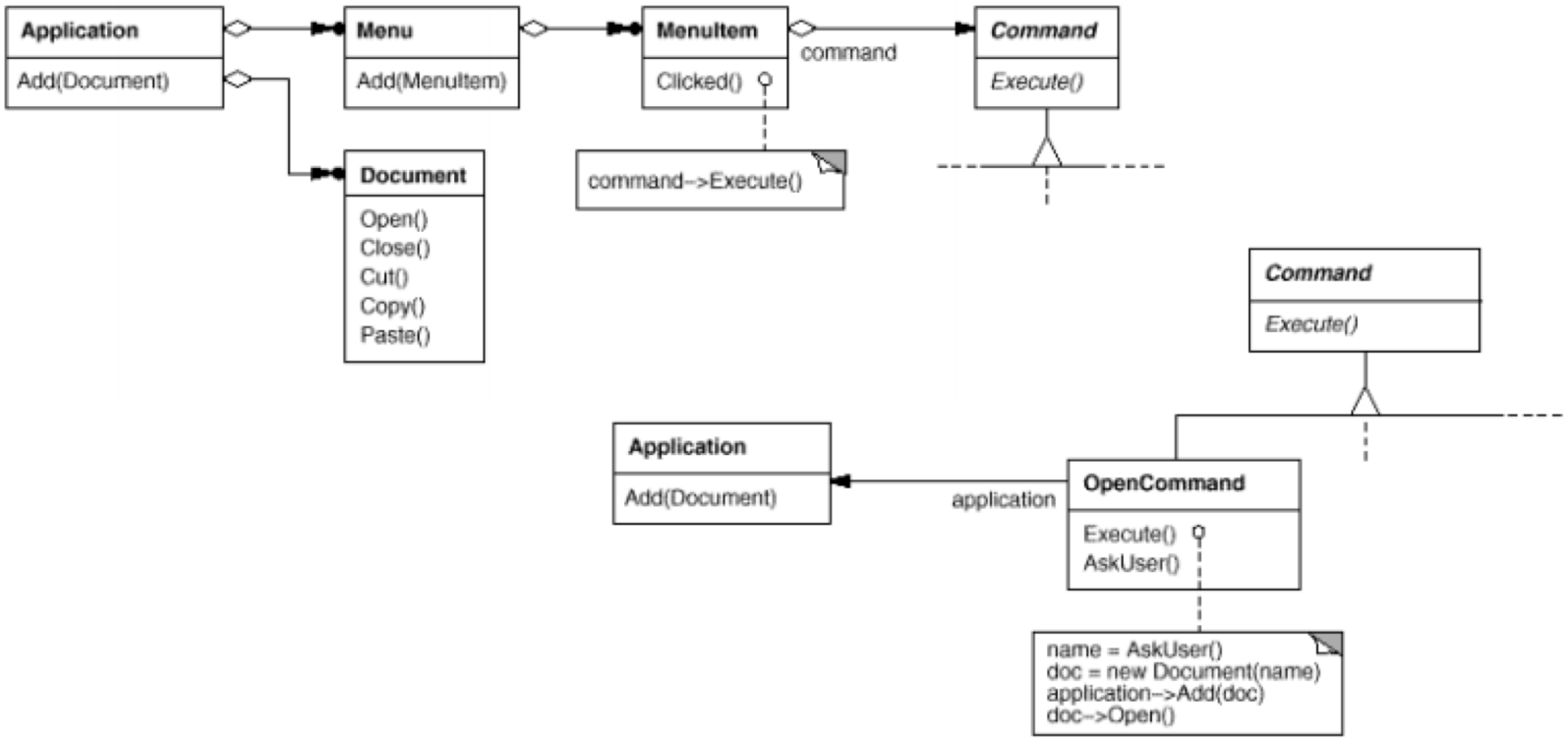
- El patrón de comando desacopla un objeto que realiza una solicitud desde el que sabe cómo realizarlo.
- Un objeto Command está en el centro de este desacoplamiento y encapsula un receptor con una acción (o conjunto de acciones).
- Un invocador realiza una solicitud de un objeto Command llamando a su método execute(), que invoca esas acciones en el receptor.
- Los invocadores se pueden parametrizar con comandos, incluso dinámicamente en tiempo de ejecución.
- Los comandos pueden admitir deshacer mediante la implementación de un método de deshacer que restaura

- Los comandos macro son una extensión simple de comando que permite múltiples comandos a invocar. Del mismo modo, los comandos de macro pueden admitir fácilmente deshacer().
- En la práctica, no es raro que los objetos de comando "inteligentes" implementen solicitarse a sí mismos en lugar de delegar en un receptor.
- Los comandos también pueden usarse para implementar sistemas de registro y transaccionales.

Una aplicación para editar un documento proporciona un menú en una interfaz, cada opción del menú debe ejecutar la acción indicada sobre el documento. Las operaciones que se desean realizar en el documento son: abrir documento, cerrar documento, copiar, cortar y pegar. Realice el diagrama para esta aplicación.



# Solución diagrama



# Ejemplo código

```
class Command {
public:
virtual ~Command();
virtual void Execute() = 0;
protected:
Command();
};
```

```
template <class Receiver>
class SimpleCommand : public Command {
public:
typedef void (Receiver::* Action)();
SimpleCommand(Receiver* r, Action a) :
_receiver(r), _action(a) { }
virtual void Execute();
private:
Action _action;
Receiver* _receiver;
};
```

```
class OpenCommand : public Command {
public:
OpenCommand(Application*);
virtual void Execute();
protected:
virtual const char* AskUser();
private:
Application* _application;
char* _response;
};

OpenCommand::OpenCommand (Application* a) {
_application = a;
}

void OpenCommand::Execute () {
const char* name = AskUser();
if (name != 0) {
Document* document = new Document(name);
_application->Add(document);
document->Open();
}
}
```

```
class PasteCommand : public Command {
public:
PasteCommand(Document*);
virtual void Execute();
private:
Document* _document;
};

PasteCommand::PasteCommand (Document* doc) {       _document = doc;    }       void
PasteCommand::Execute () {       _document->Paste();    }
```