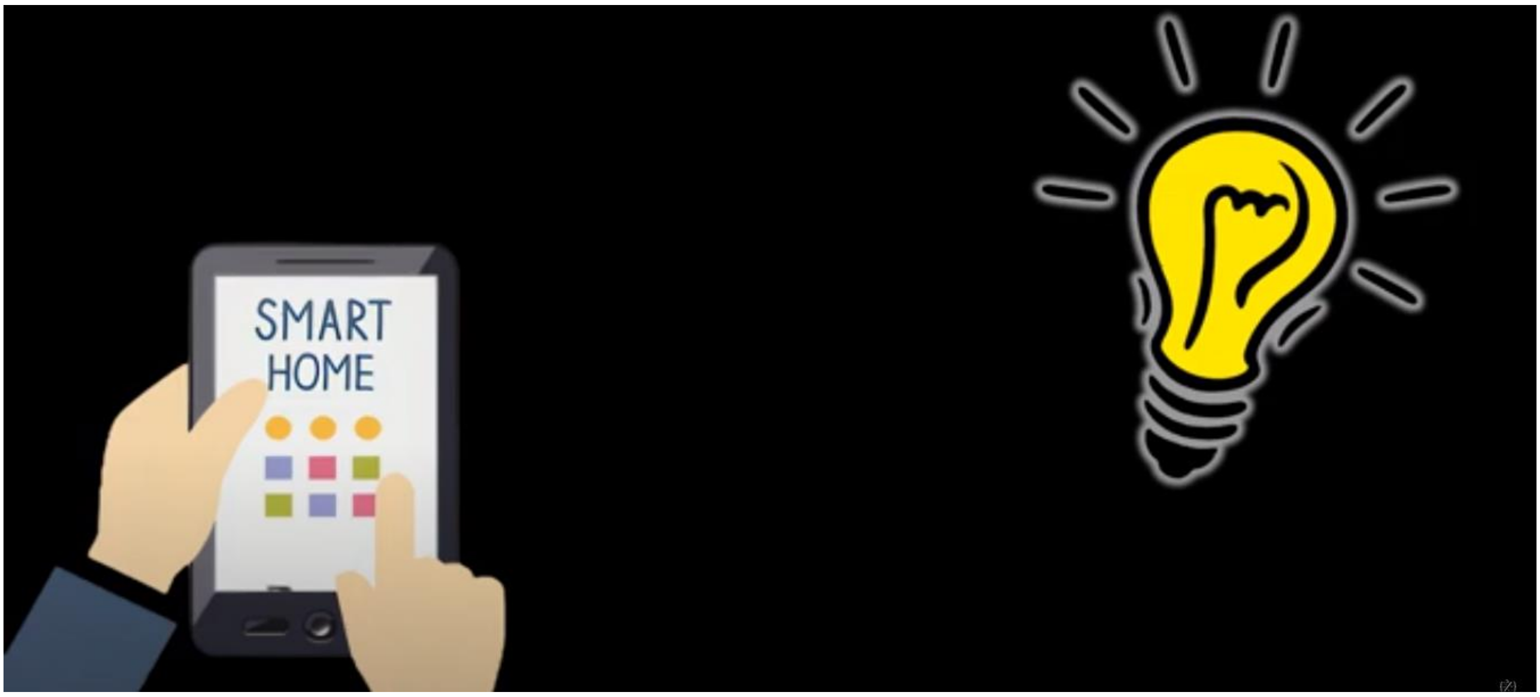
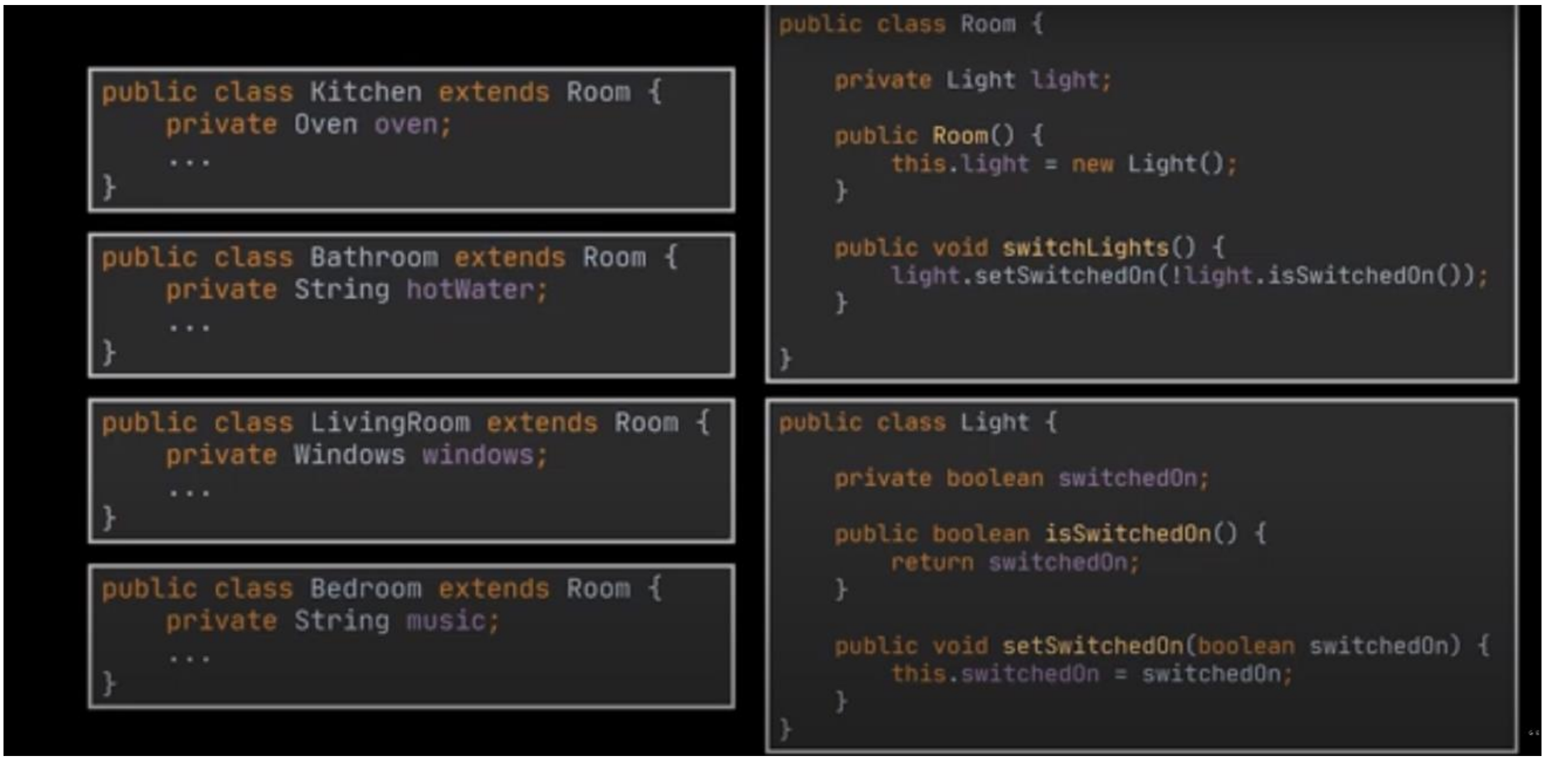
Command Design Pattern





```java
public class Kitchen extends Room {
    private Oven oven;
    ...
}
```

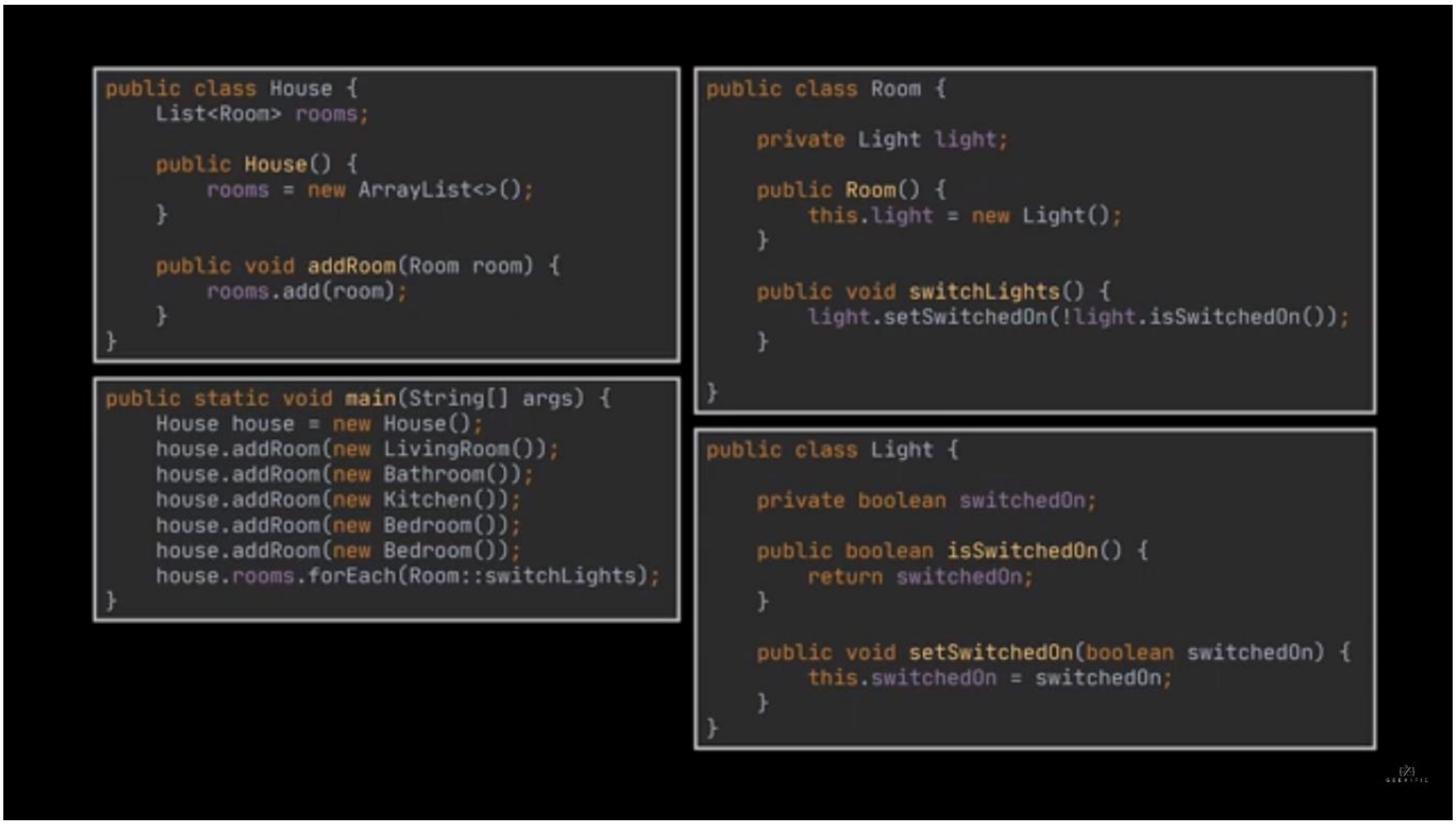
```java
public class Bathroom extends Room {
    private String hotWater;
    ...
}
```

```java
public class LivingRoom extends Room {
    private Windows windows;
    ...
}
```

```java
public class Bedroom extends Room {
    private String music;
    ...
}
```

```java
public class Room {

    private Light light;

    public Room() {
        this.light = new Light();
    }

    public void switchLights() {
        light.setSwitchedOn(!light.isSwitchedOn());
    }

}
```

```java
public class Light {

    private boolean switchedOn;

    public boolean isSwitchedOn() {
        return switchedOn;
    }

    public void setSwitchedOn(boolean switchedOn) {
        this.switchedOn = switchedOn;
    }
}
```



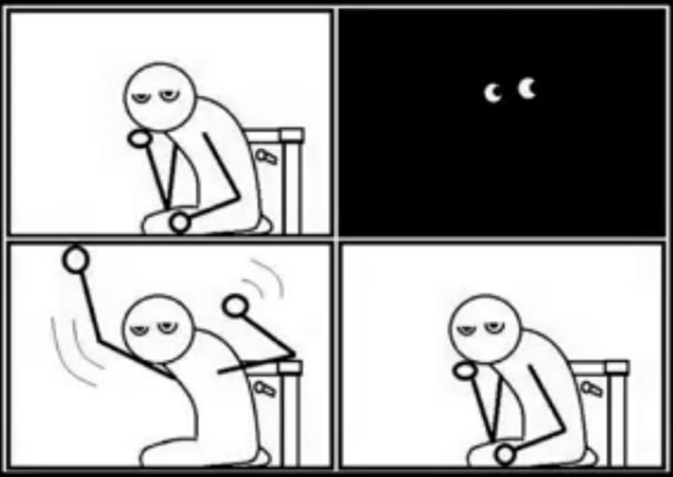
```java
public class House {
    List<Room> rooms;

    public House() {
        rooms = new ArrayList<>();
    }

    public void addRoom(Room room) {
        rooms.add(room);
    }
}
```

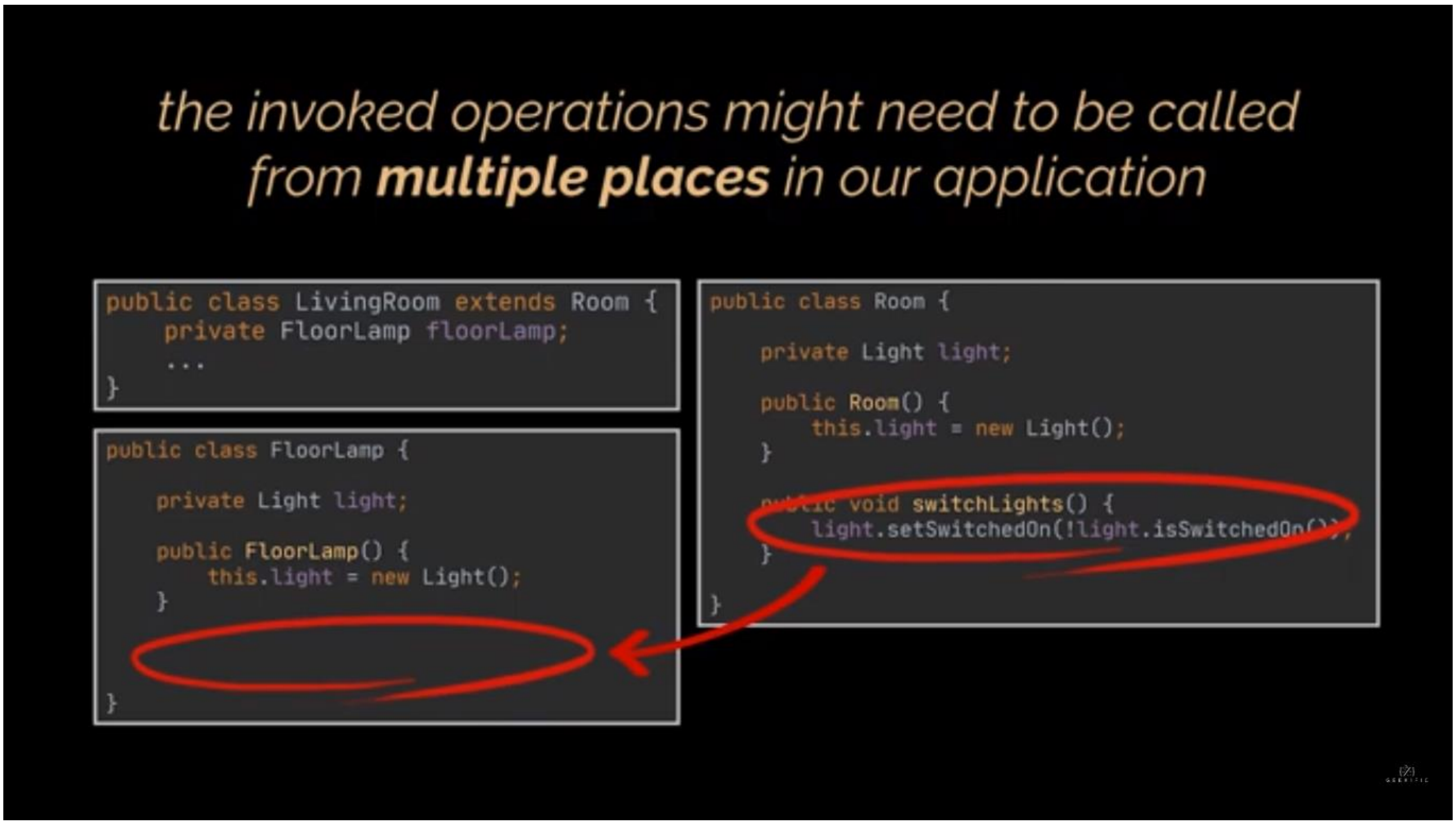
```java
public static void main(String[] args) {
    House house = new House();
    house.addRoom(new LivingRoom());
    house.addRoom(new Bathroom());
    house.addRoom(new Kitchen());
    house.addRoom(new Bedroom());
    house.addRoom(new Bedroom());
    house.rooms.forEach(Room::switchLights);
}
```

```java
public class Room {

    private Light light;

    public Room() {
        this.light = new Light();
    }

    public void switchLights() {
        light.setSwitchedOn(!light.isSwitchedOn());
    }

}
```

```java
public class Light {

    private boolean switchedOn;

    public boolean isSwitchedOn() {
        return switchedOn;
    }

    public void setSwitchedOn(boolean switchedOn) {
        this.switchedOn = switchedOn;
    }
}
```

*having an **enormous number of subclasses** increases the risk of breaking the code in any subclass everytime we modify the parent class*



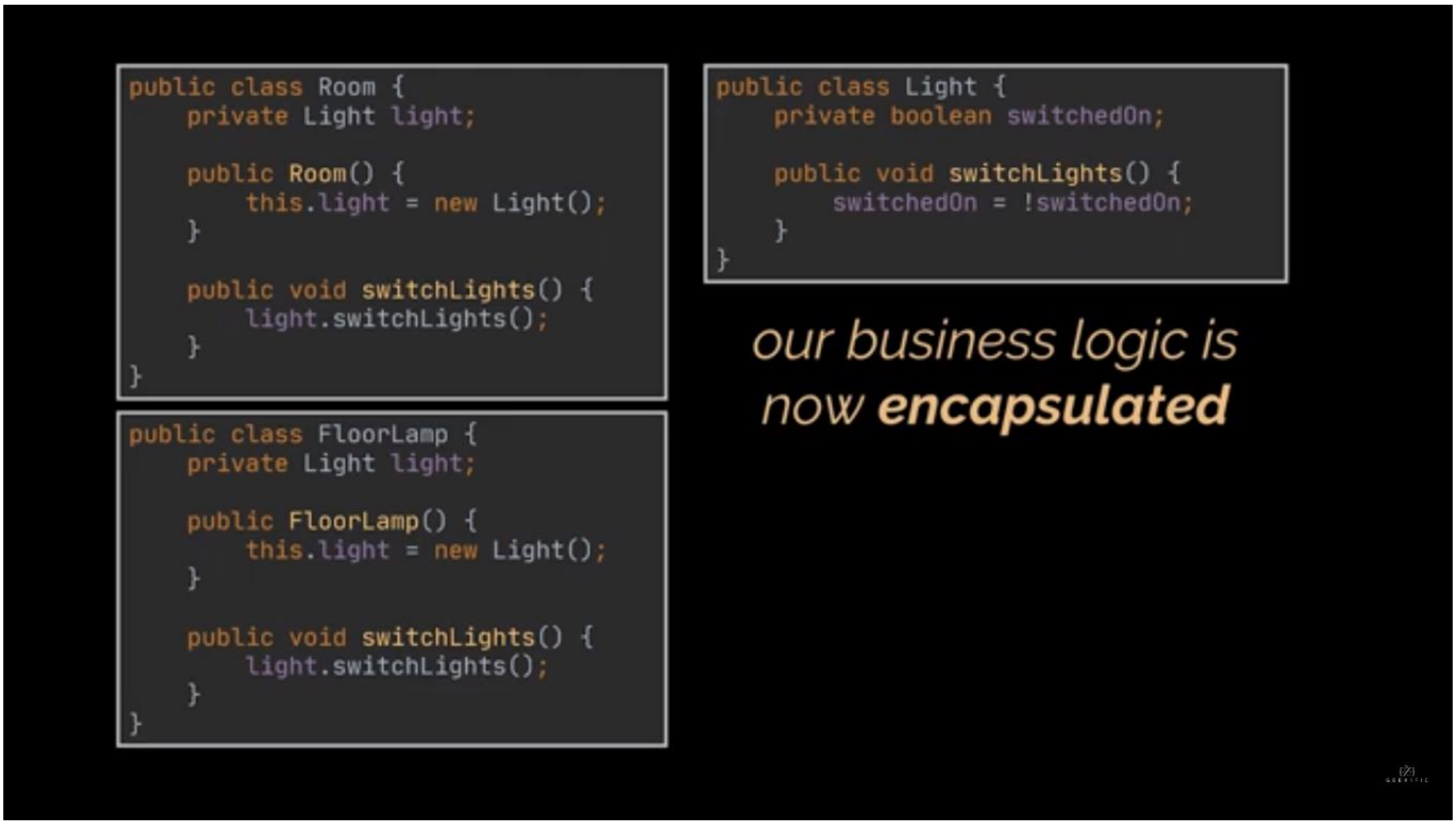


*the invoked operations might need to be called from **multiple places** in our application*

```
public class LivingRoom extends Room {
    private FloorLamp floorLamp;
    ...
}
```

```
public class FloorLamp {

    private Light light;

    public FloorLamp() {
        this.light = new Light();
    }

}
```

```
public class Room {

    private Light light;

    public Room() {
        this.light = new Light();
    }

    public void switchLights() {
        light.setSwitchedOn(!light.isSwitchedOn());
    }

}
```

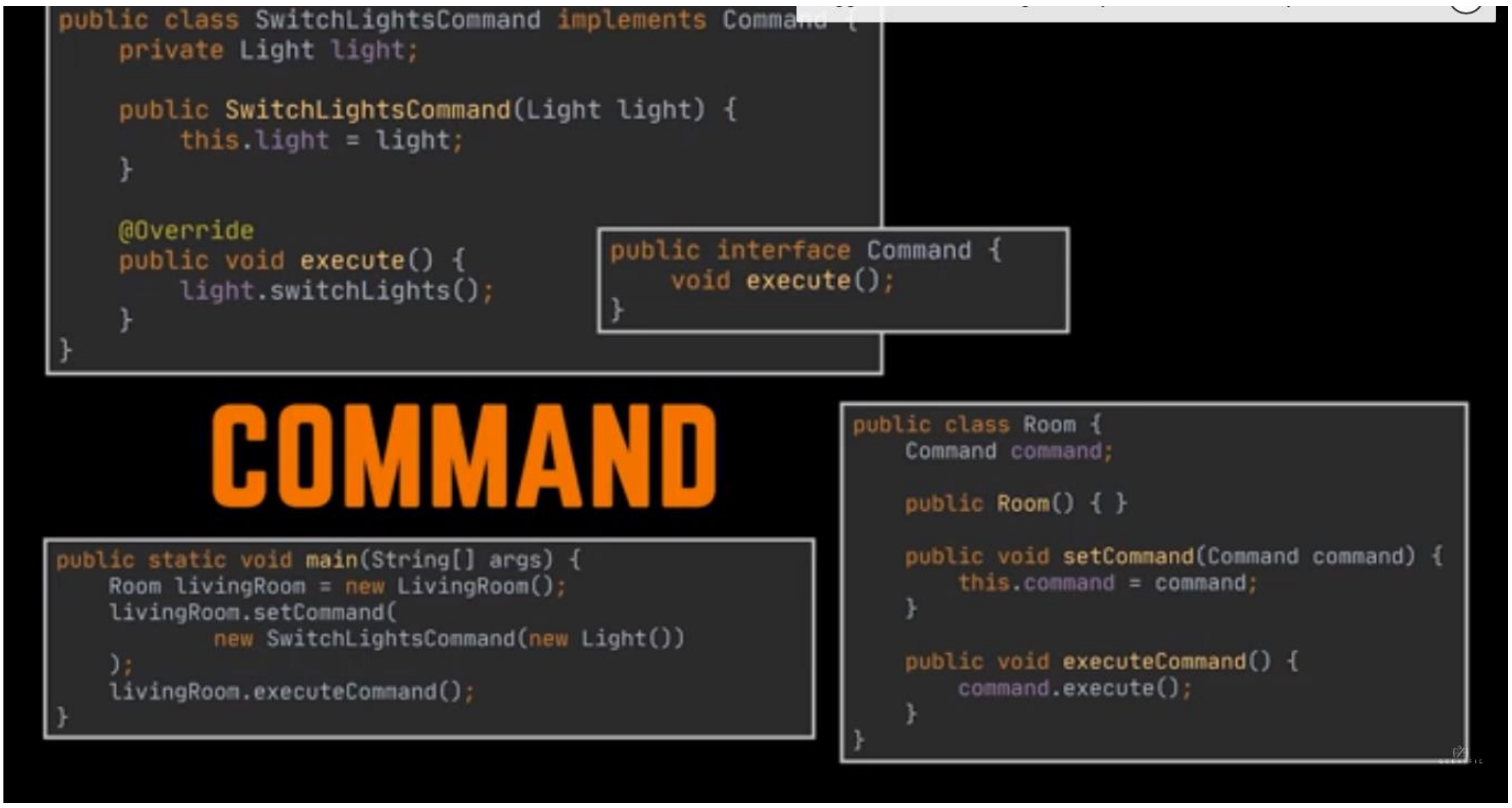


```
public class Room {
    private Light light;

    public Room() {
        this.light = new Light();
    }

    public void switchLights() {
        light.switchLights();
    }
}
```

```
public class FloorLamp {
    private Light light;

    public FloorLamp() {
        this.light = new Light();
    }

    public void switchLights() {
        light.switchLights();
    }
}
```

```
public class Light {
    private boolean switchedOn;

    public void switchLights() {
        switchedOn = !switchedOn;
    }
}
```

*our business logic is now **encapsulated***



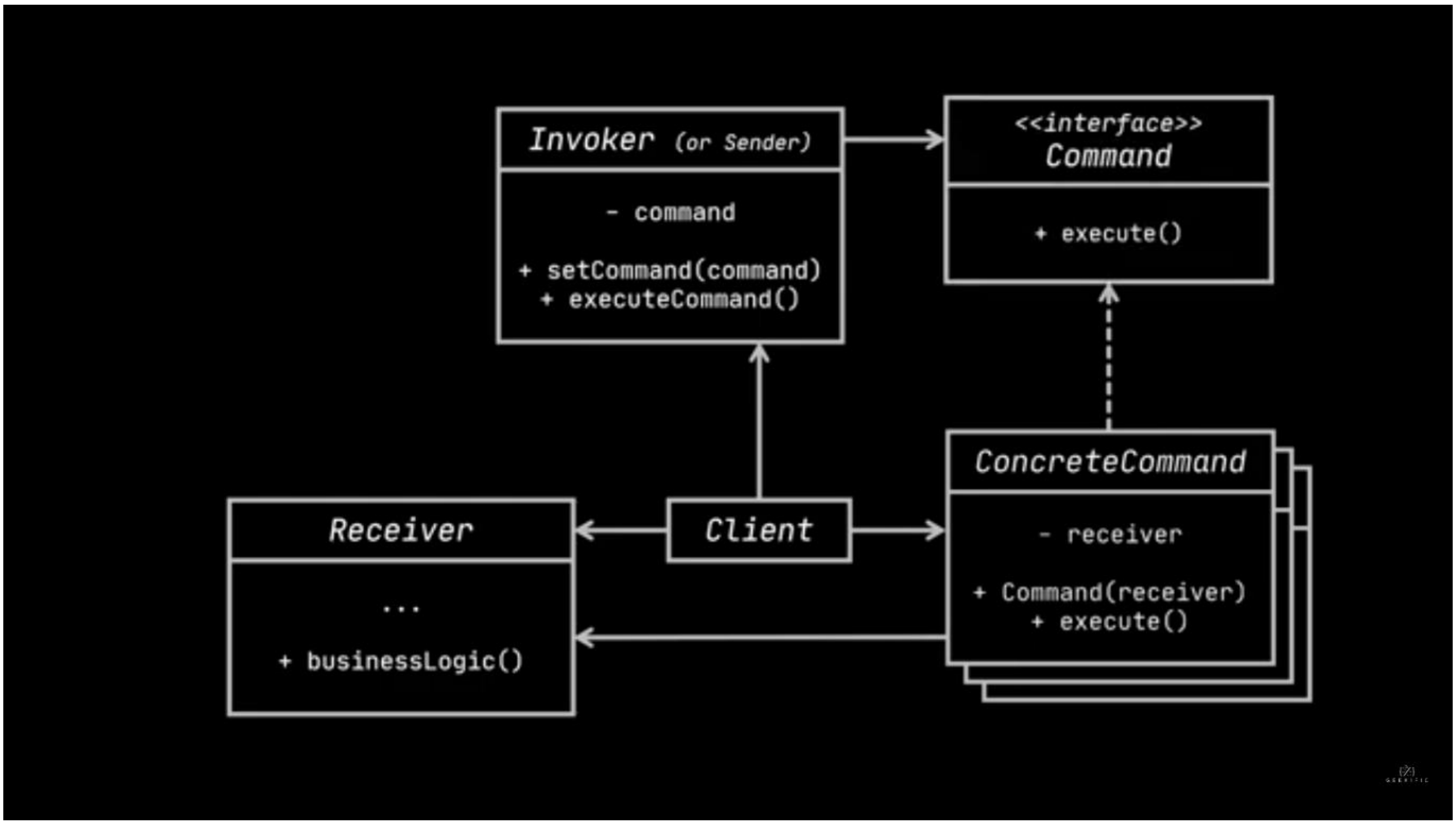
```java
public class SwitchLightsCommand implements Command {
    private Light light;

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

    @Override
    public void execute() {
        light.switchLights();
    }
}
```

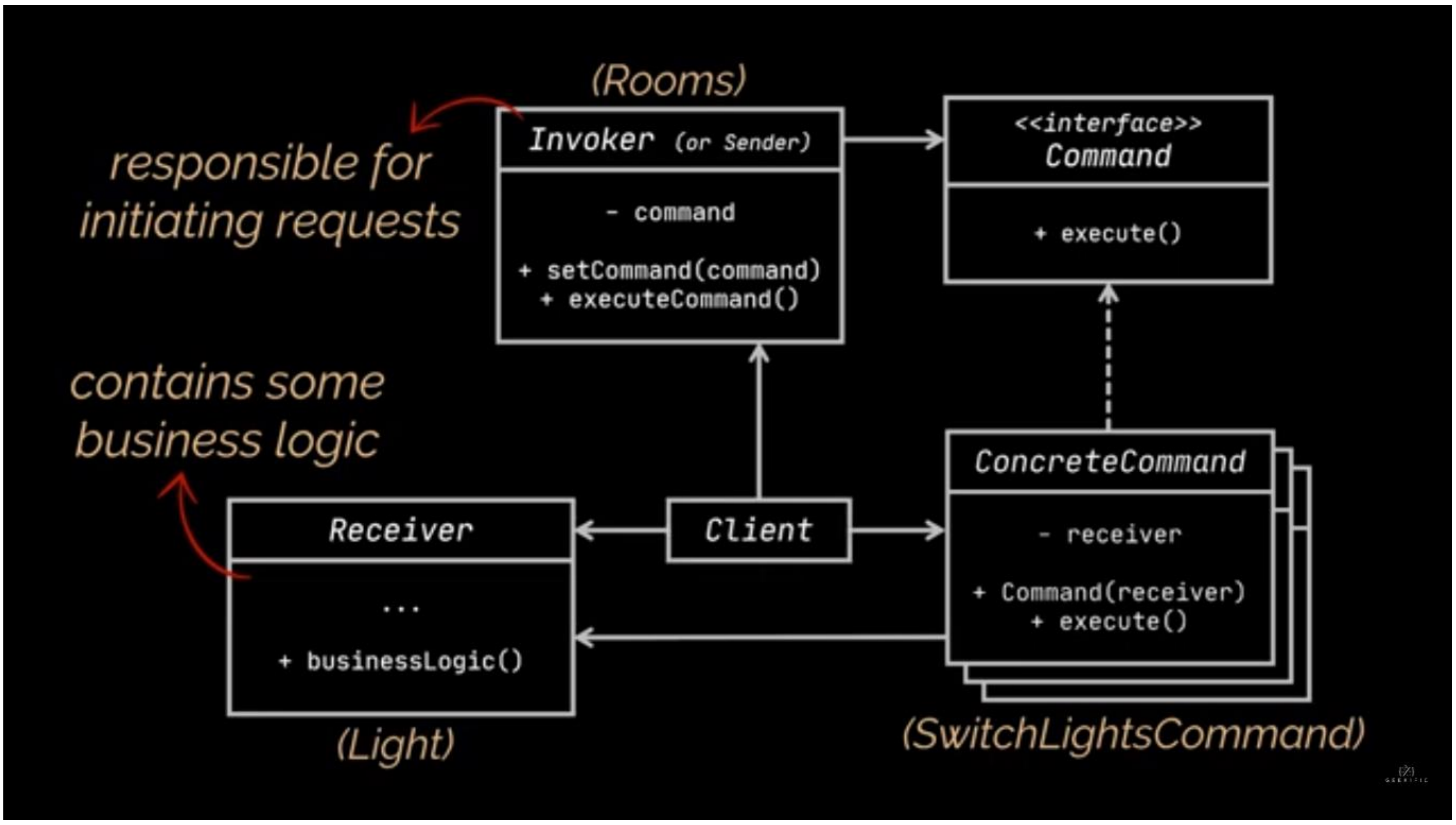
```java
public interface Command {
    void execute();
}
```

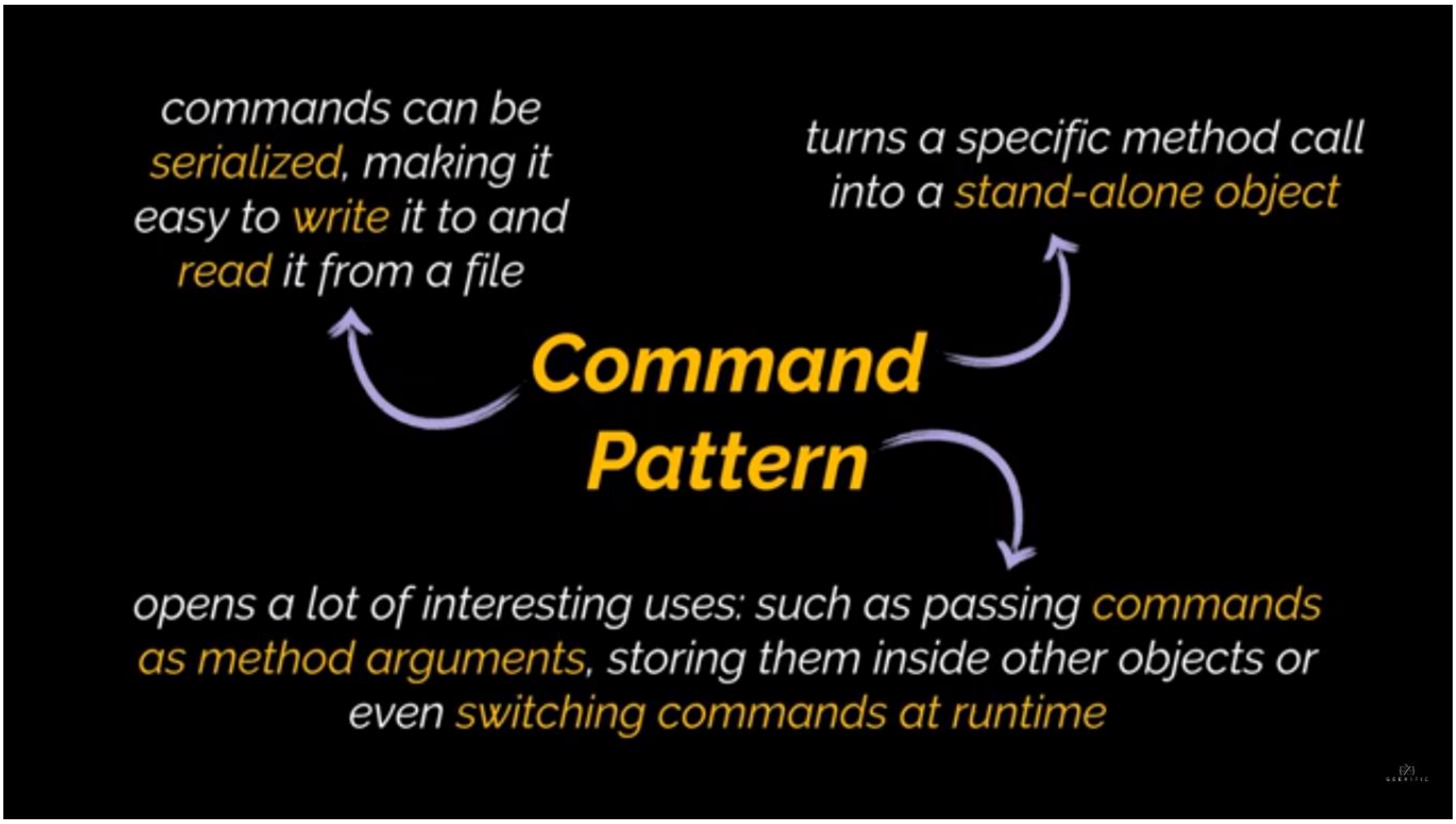
COMMAND

```java
public static void main(String[] args) {
    Room livingRoom = new LivingRoom();
    livingRoom.setCommand(
            new SwitchLightsCommand(new Light())
    );
    livingRoom.executeCommand();
}
```

```java
public class Room {
    Command command;

    public Room() { }

    public void setCommand(Command command) {
        this.command = command;
    }

    public void executeCommand() {
        command.execute();
    }
}
```







commands can be serialized, making it easy to write it to and read it from a file

turns a specific method call into a stand-alone object

Command Pattern

opens a lot of interesting uses: such as passing commands as method arguments, storing them inside other objects or even switching commands at runtime