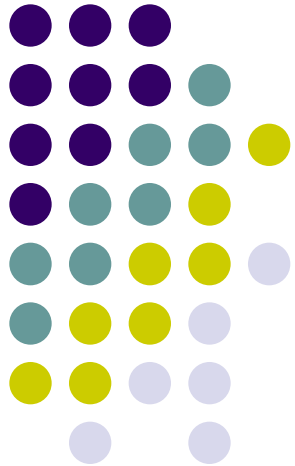
Паттерн Команда

Инкапсуляция вызова



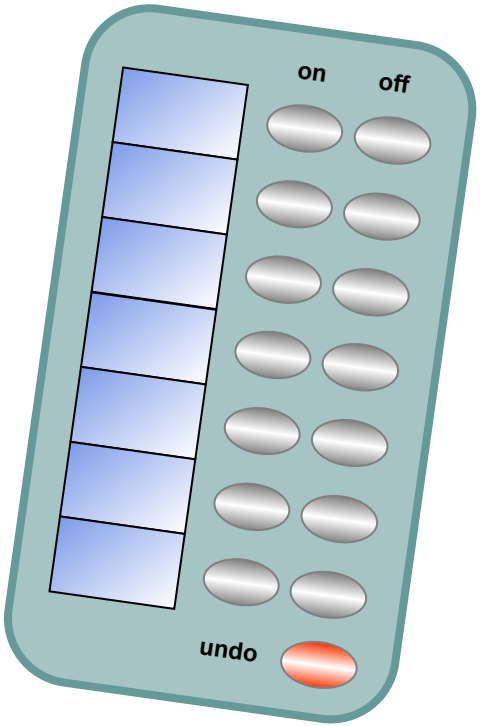
Подготовили: Мотузов Олег, Яхонт Никита

Инкапсуляция вызовов методов

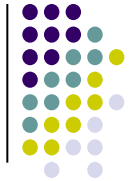
- Вызывающему объекту не нужно беспокоиться о том, как будут выполняться его запросы. Он просто использует инкапсулированный метод для решения своей задачи.
- Инкапсуляция позволяет решать и такие нетривиальные задачи, как регистрация или отмена вызовов.

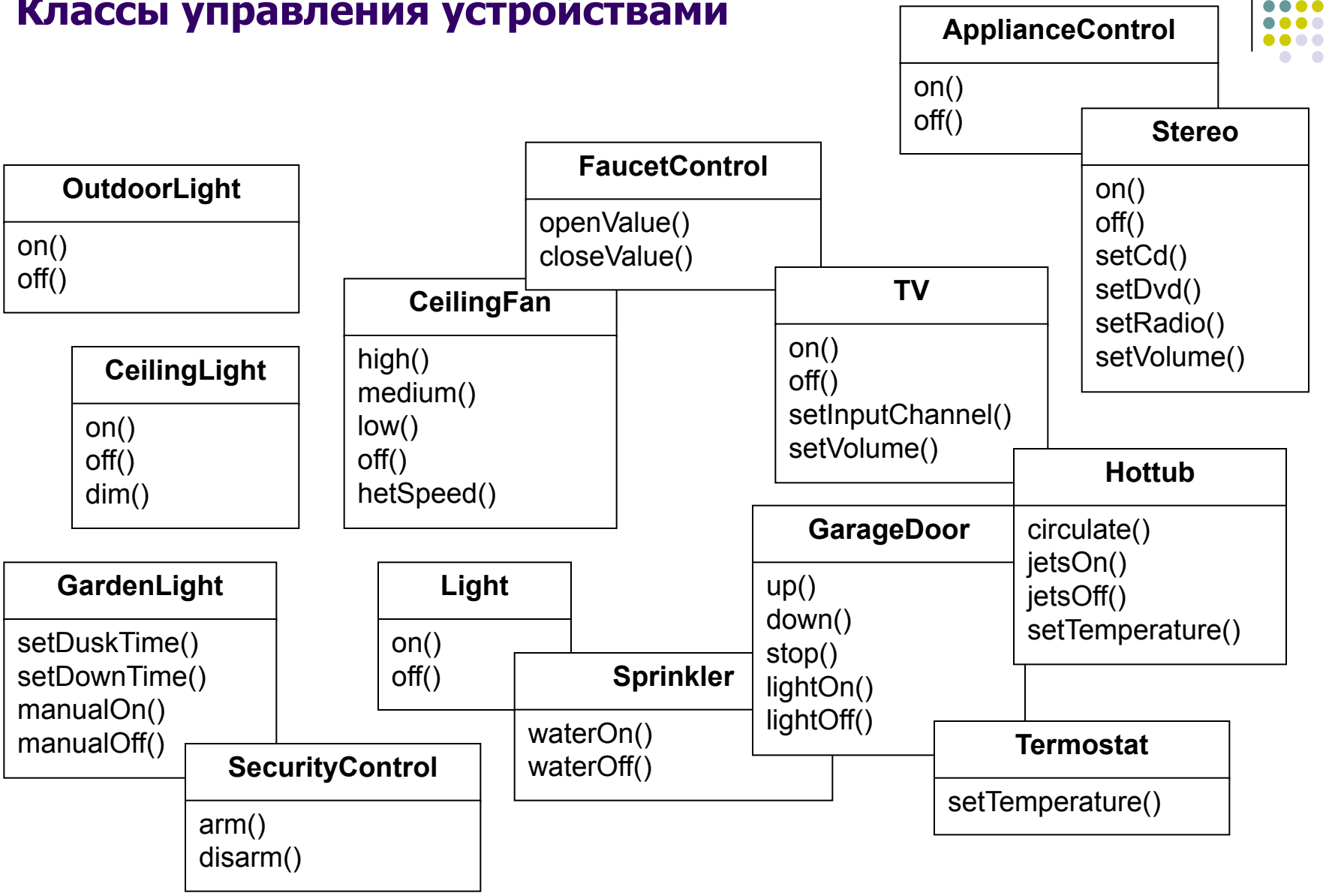
Автоматизация дома

- Требуется разработать API для Пульта Домашней Автоматизации.
- Пульт имеет семь программируемых ячеек (каждая из которых связывается с отдельным домашним устройством) и соответствующую кнопку «вкл/выкл» для каждой ячейки. Кроме того, устройство оснащено кнопкой глобальной отмены.
- Также прилагается диск с набором классов Java, созданных разными фирмами-разработчиками для управления всевозможными домашними устройствами: светильниками, вентиляторами, ваннами-джакузи, акустическим оборудованием и т. д.
- Задача — создать API для программирования пульта, чтобы каждая ячейка могла быть настроена на управление устройством или группой устройств. Также следует учесть, что пульт должен поддерживать как текущий набор устройств, так и все устройства, которые могут быть добавлены в будущем.

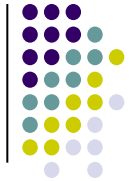


Классы управления устройствами





Обсуждение



Сам пульт устроен просто: всего две кнопки включение/выключение на каждое устройство, но классы устройств очень разные…

кроме on() и off(), классы содержат много других методов: dim(), setTemperature(), setVolume(), setDirection()…

в будущем появятся новые классы устройств с еще более разнообразными методами

Архитектуру необходимо рассматривать с точки зрения *разделения обязанностей*: пульт должен **обрабатывать нажатия кнопок** и **выдавать запросы.** Он *не должен обладать информацией об устройствах*

Пульт в любом случае не должен привязываться к конкретной реализации классов устройств.
Программа **не** должна состоять из цепочки условных команд вида «if slot1 == Light, then light.on(), else if slot1 = Hottub then hottub.jetsOn()»
Это признак плохой архитектуры

Паттерн «Команда»

- Паттерн Команда отделяет сторону, выдающую запрос, от объекта, фактически выполняющего операцию.
- В нашем примере запрос поступает от пульта, а объектом, выполняющим операцию, будет экземпляр одного из классов устройств.
- В архитектуру приложения вводятся «объекты команд». Объект команды инкапсулирует запрос на выполнение некой операции (скажем, включение света) с конкретным объектом (допустим, с осветительной системой).
- Если для каждой кнопки в приложении хранится свой объект команды, при ее нажатии мы обращаемся к объекту команды с запросом на выполнение операции.
- Сам пульт понятия не имеет, что это за операция, — он знает только, как взаимодействовать с нужным объектом для выполнения операции.
- Получается, что пульт полностью отделен от объекта осветительной системы

Взаимодействие объектов, на примере кафе

1

Посетитель передает **официантке** свой **заказ**

Бланк заказа

№	Наименование	выход	цена

2

Официантка получает **заказ**, кладет его на стойку и говорит «У нас заказ!»

3

повар готовит блюда, входящие в **заказ**

Более подробно

посетитель просматривает меню и создает заказ

мне мороженое с фруктами

createOrder()

бланк инкапсулирует запрос на приготовление блюд

Бланк заказа

№	Наименование	выход	цена
	мороженое		
	с фруктами		

takeOrder()

orderUp()

задача официантки – получить заказ и вызвать метод orderUp()

результат

повар выполняет инструкции, содержащиеся в заказе

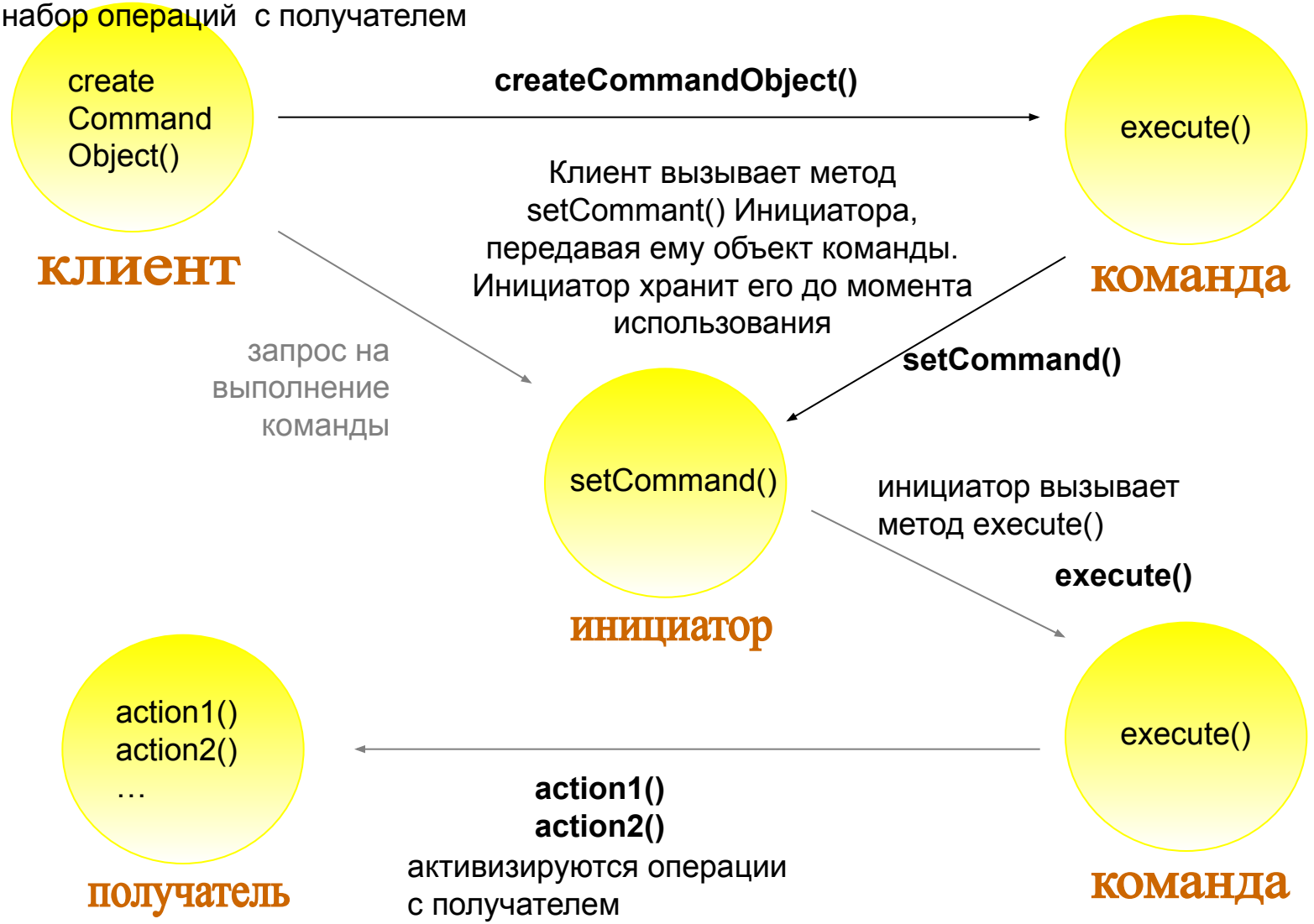
makeIceCream()
makeFrut()

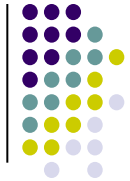
для передачи распоряжений повару используются вызовы методов вида makeIcecream()

От кафе к паттерну Команда

Команда содержит единственный метод execute(), в котором инкапсулированы операции с получателем

Клиент отвечает за создание объекта команды, содержащего набор операций с получателем





Реализация интерфейса Command

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

Очень простой интерфейс: всего один метод execute().

Реализация команды для включения света

Класс команды должен реализовать интерфейс Command.

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

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

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

В переменной light конструктору передается конкретный объект, которым будет управлять команда (допустим, освещение в гостиной). При вызове execute получателем запроса будет объект light.

Метод execute вызывает метод on() объекта-получателя (то есть осветительной системы).

Light
on() off()

GarageDoor
up() down() stop() lightOn() lightOff()

Использование объекта команды

```
public class SimpleRemoteControl {
    Command slot;

    public SimpleRemoteControl() {}

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

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

Всего одна ячейка для хранения команды (и одно управляемое устройство).

Метод для назначения команды. Может вызываться повторно, если клиент кода захочет изменить поведение кнопки.

Метод, вызываемый при нажатии кнопки. Мы просто берем объект команды, связанный с текущей ячейкой, и вызываем его метод execute().

Тестирование

```
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();
    }
}
```

Клиент в терминологии паттерна.

Объект remote — Инициатор; ему будет передаваться объект команды.

Создание объекта Light, который будет Получателем запроса.

Создание команды с указанием Получателя.

Команда передается Инициатору.

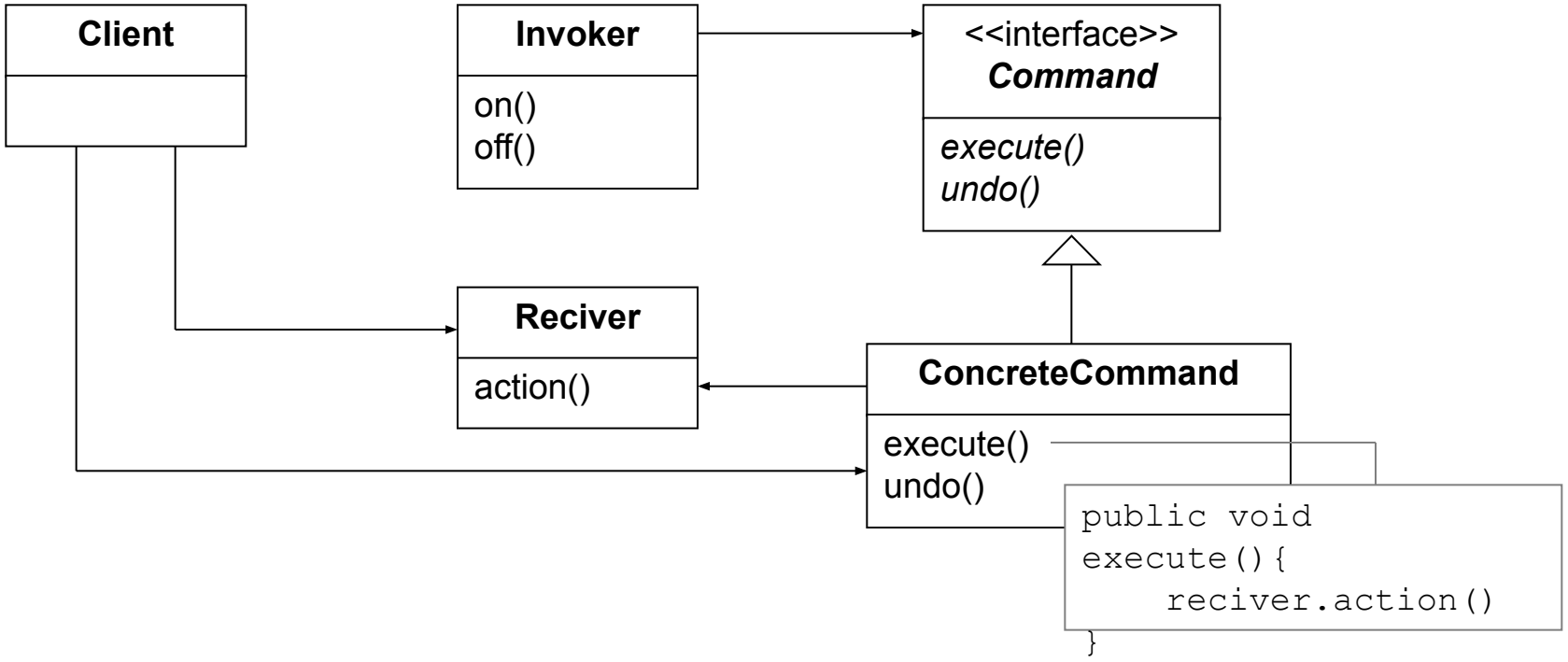
Имитируем нажатие кнопки.

Результат выполнения тестового кода.

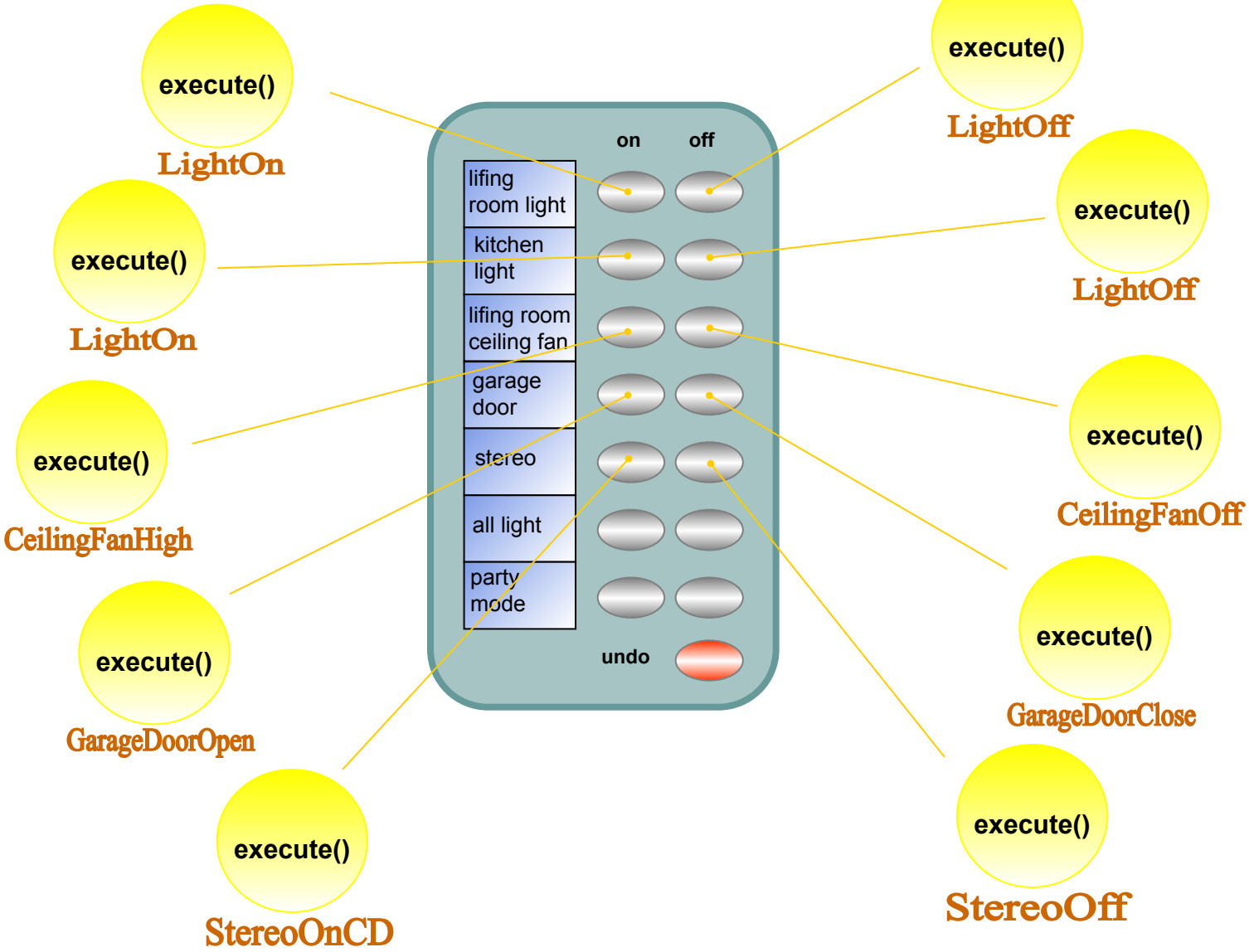
```
File Edit Window Help DinerFoodYum
%java RemoteControlTest
Light is On
%
```

Определение паттерна Команда (Action, Command)

Паттерн Команда инкапсулирует запрос в виде объекта, делая возможной параметризацию клиентских объектов с другими запросами, организацию очереди или регистрацию запросов, а также поддержку отмены операций.



Связывание команд с ячейками



Реализация

```
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();
    }
}
```

В этой версии пульт будет поддерживать все семь команд «вкл/выкл», которые будут храниться в соответствующих массивах.

Конструктор создает экземпляры команд и инициализирует массивы onCommands и offCommands.

Метод setCommand() получает ячейку и команды включения/выключения для этой ячейки. Команды сохраняются в массивах для последующего использования.

При нажатии кнопки «вкл» или «выкл» пульт вызывает соответствующий метод: onButtonWasPushed() или offButtonWasPushed().

Переопределенный метод toString() выводит все ячейки с соответствующими командами. Мы воспользуемся им при тестировании пульта.

Реализация

Как избавится от лишних проверок?

```
public void onButtonWasPushed(int slot){
    if(onCommands[slot]!= null){
        onCommands[slot].execute();
    }
}
```

Объект NoCommand является примером *пустого* (null) *объекта.*

Пустые объекты применяются тогда, когда вернуть «полноценный» объект невозможно, но вам хочется избавить клиента от необходимости проверять null-ссылки.

Пустые объекты используются во многих паттернах проектирования, а некоторые авторы даже считают их самостоятельным паттерном.

Архитектура API пульта

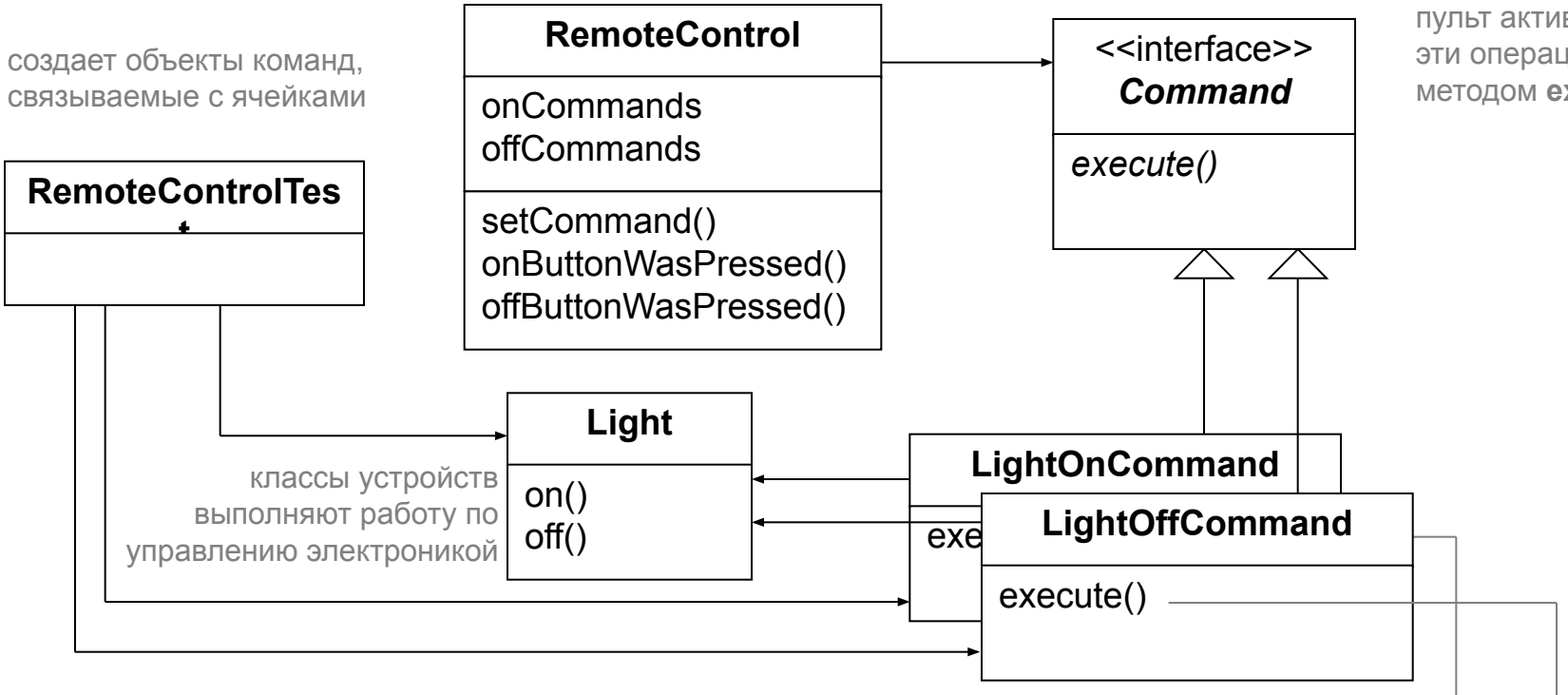
управляет набором объектов-команд (по одному на кнопку)

все команды реализуют интерфейс

команды инкапсулируют набор операций с классом устройства

пульт активизирует эти операции методом **execute()**

создает объекты команд, связываемые с ячейками



классы устройств выполняют работу по управлению электроникой

при нажатии на кнопку пульта вызывается метод execute() соответствующего объекта команды

объект команды хранит ссылку на объект класса устройства, и в своем методе execute() вызывает один или несколько методов объекта устройства

```
public void
execute(){
}
```

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

Кнопка отмены

1. `interface Command` + метод `undo()`
2. в классах команд реализовать этот метод
3. включить в класс пульта `RemoteControl` механизм отслеживания последней нажатой кнопки и нажатия кнопки отмены

Макросы

Нажатием одной кнопки выключить свет, включить телевизор и стереосистему, запустить DVD и наполнить джакузи.

```
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);
```

Создание объектов устройств (свет, телевизор, стерео, джакузи).

Создание команд включения для управления этими устройствами.

Макросы

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

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

...и два объекта макрокоманд, в которых они хранятся.

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

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

Результат.

Резюме

Принципы

Инкапсулируйте, то что изменяется

Отдавайте предпочтение композиции перед наследованием

Программируйте на уровне интерфейсов, а не реализации

Стремитесь к слабой связности взаимодействующих объектов

Классы должны быть открыты для расширения, но закрыты для изменения

Код должен зависеть от абстракций, а не от конкретных классов.

Команда инкапсулирует запрос в виде объекта, делая возможной параметризацию клиентских объектов с другими запросами, организацию очереди или регистрацию запросов, поддержку отмены операций

Ключевые моменты

- Паттерн Команда отделяет объект, выдающий запросы, от объекта, который умеет эти запросы выполнять.
- Объект команды инкапсулирует получателя с операцией (или набором операций).
- Инициатор вызывает метод execute() объекта команды, что приводит к выполнению соответствующих операций с получателем.
- Возможна параметризация инициаторов командами (даже динамическая во время выполнения).
- Команды могут поддерживать механизм отмены, восстанавливающий объект в состоянии до последнего вызова метода execute().
- Макрокоманды — простое расширение паттерна Команда, позволяющее выполнять цепочки из нескольких команд. В них также легко реализуется механизм отмены.
- На практике нередко встречаются «умные» объекты команд, которые реализуют запрос самостоятельно вместо его делегирования получателю.
- Команды также могут использоваться для реализации систем регистрации команд и поддержки транзакций.