
Le patron "Command"

Home-Automation :

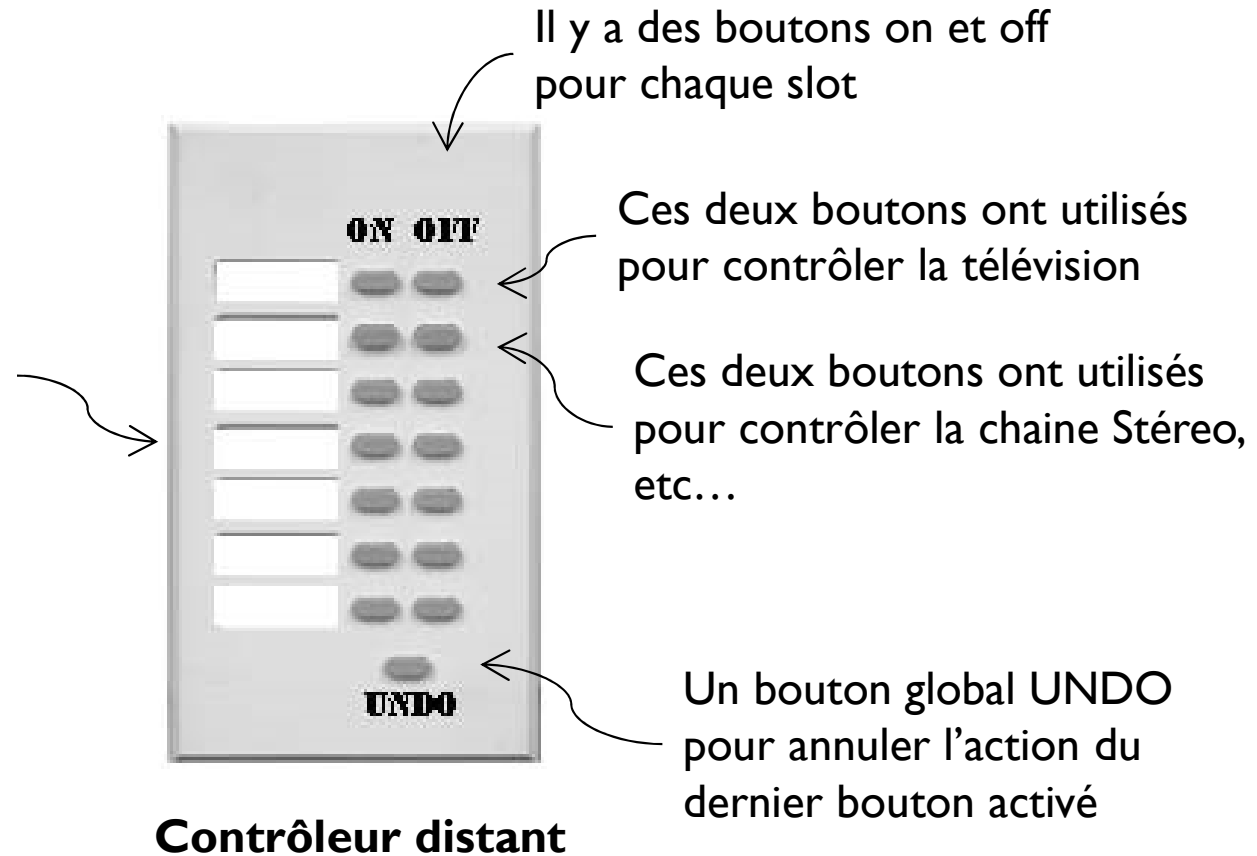
Spécification (1 / 23)

- ▶ Objectif: Mettre en œuvre un système de contrôle distant d'un ensemble d'appareils dans une maison
- ▶ Besoin: programmer les fonctionnalités d'un contrôleur distant (avec 7 slots) selon des classes (prédéfinies par le vendeur) de gestion des appareils installés dans la maison.
- ▶ Conception: OO
 - ▶ Prévoir les relations entre les boutons du contrôleur distant (ON-OFF) avec les fonctionnalités des appareils installés: `setTemperature()`, `setVolume()`, `setDirection()`, etc..

Home-Automation :

Analyse du contrôleur distant (2/23)

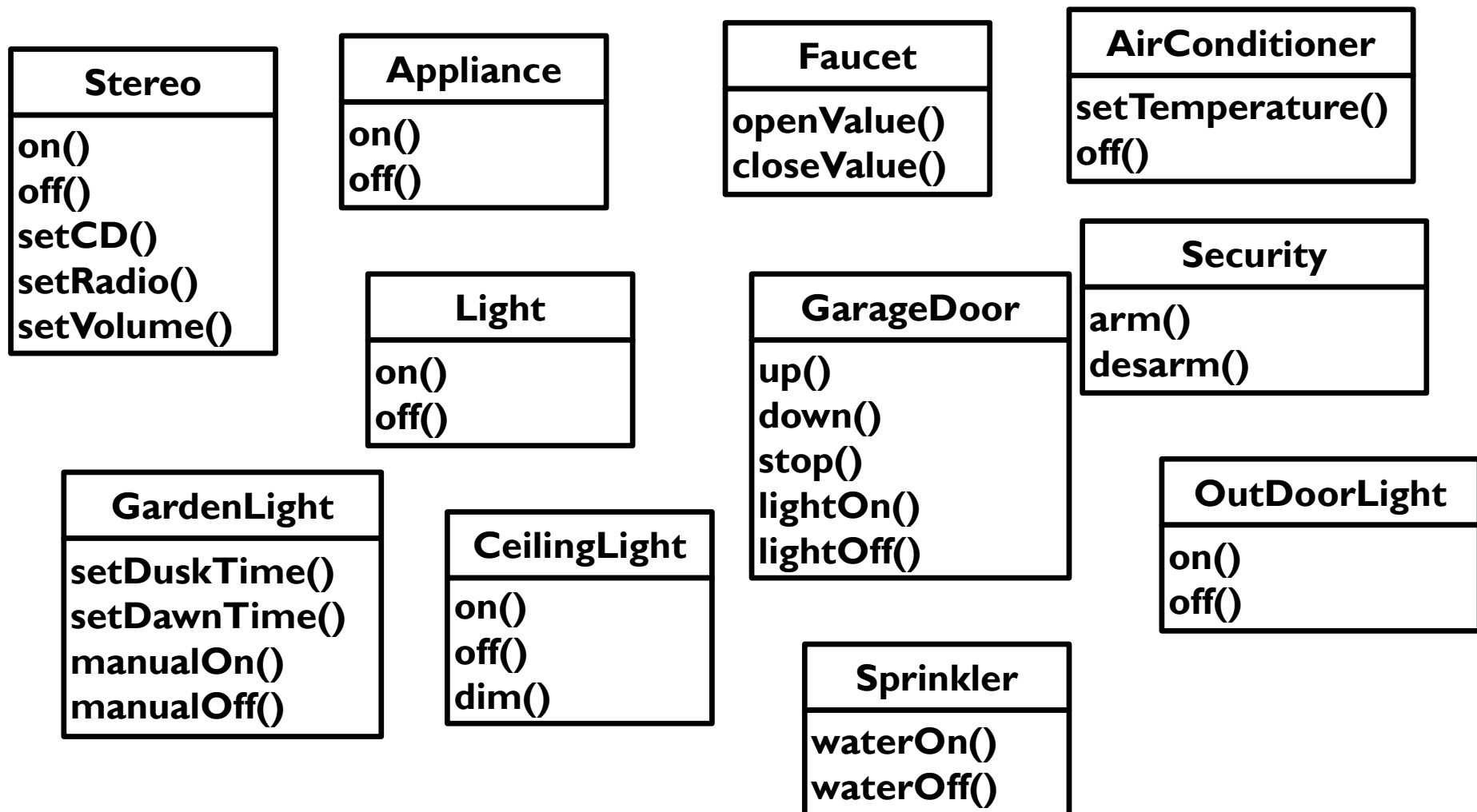
Il y a 7 slots à programmer.
On met un appareil différent
dans chaque slot, et on le
contrôle à travers les boutons



Home-Automation :

Les classes du vendeur (3/23)

- ▶ Les classes du vendeur nous donnent une idée sur les fonctionnalités des appareils installés dans la maison :



Home-Automation :

Discussion de la conception (4/23)

- ▶ On s'attendait à des classes avec des méthodes on()-off() pour bien correspondre avec le contrôleur distant
- ▶ C'est important de voir ça comme séparation des préoccupations : le contrôleur doit savoir comment interpréter l'appui sur le bouton et créer des requêtes, mais il ne doit pas connaître beaucoup sur les appareils et leurs manières de fonctionnement (comment allumer une lampe)
 - ▶ En d'autres termes, le contrôleur émet des requêtes génériques
 - ▶ Une entité prendra en charge la transformation de cette requête en action

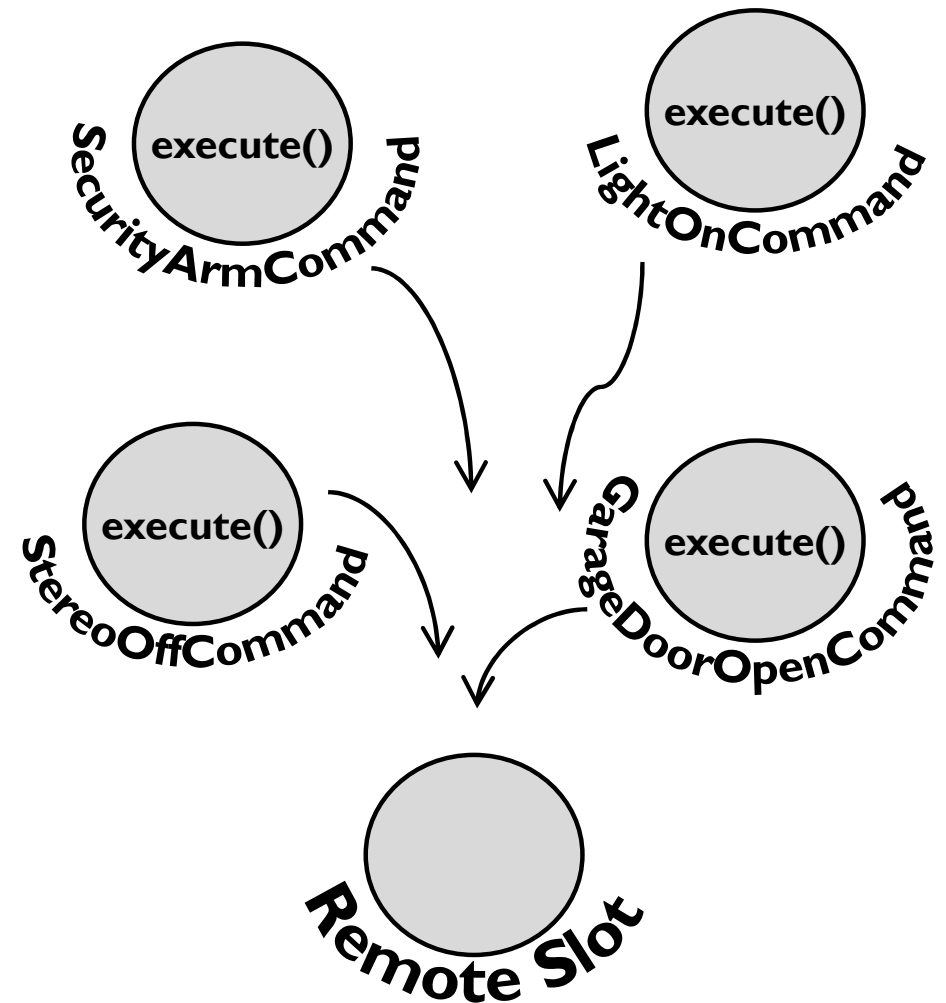
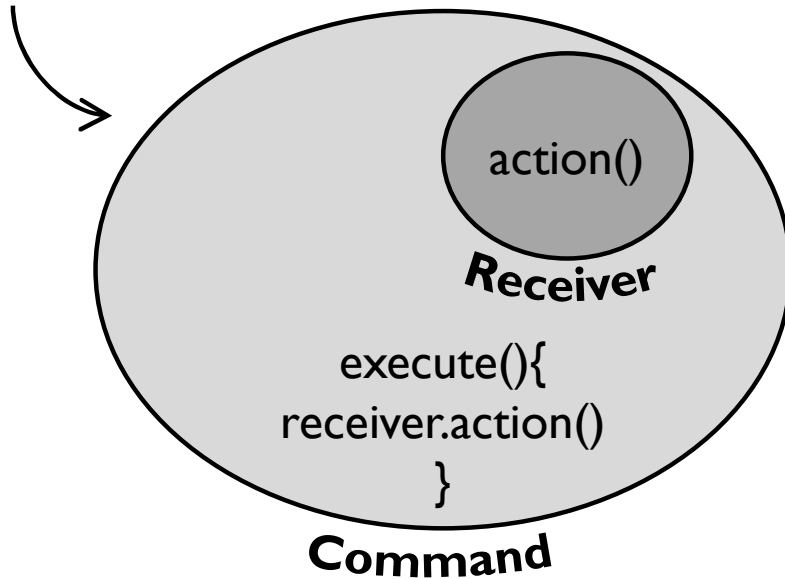
Command :

Le patron Command (5/23)

- ▶ Définition: **Command**
 - ▶ Le **patron Command** encapsule une requête comme un objet, ainsi il nous permet de paramétrer d'autres objets avec différentes requêtes, files d'attente ou longues requêtes, et supporte l'annulation d'une opération

Home-Automation : Des Commandes (6/23)

Une requête encapsulée



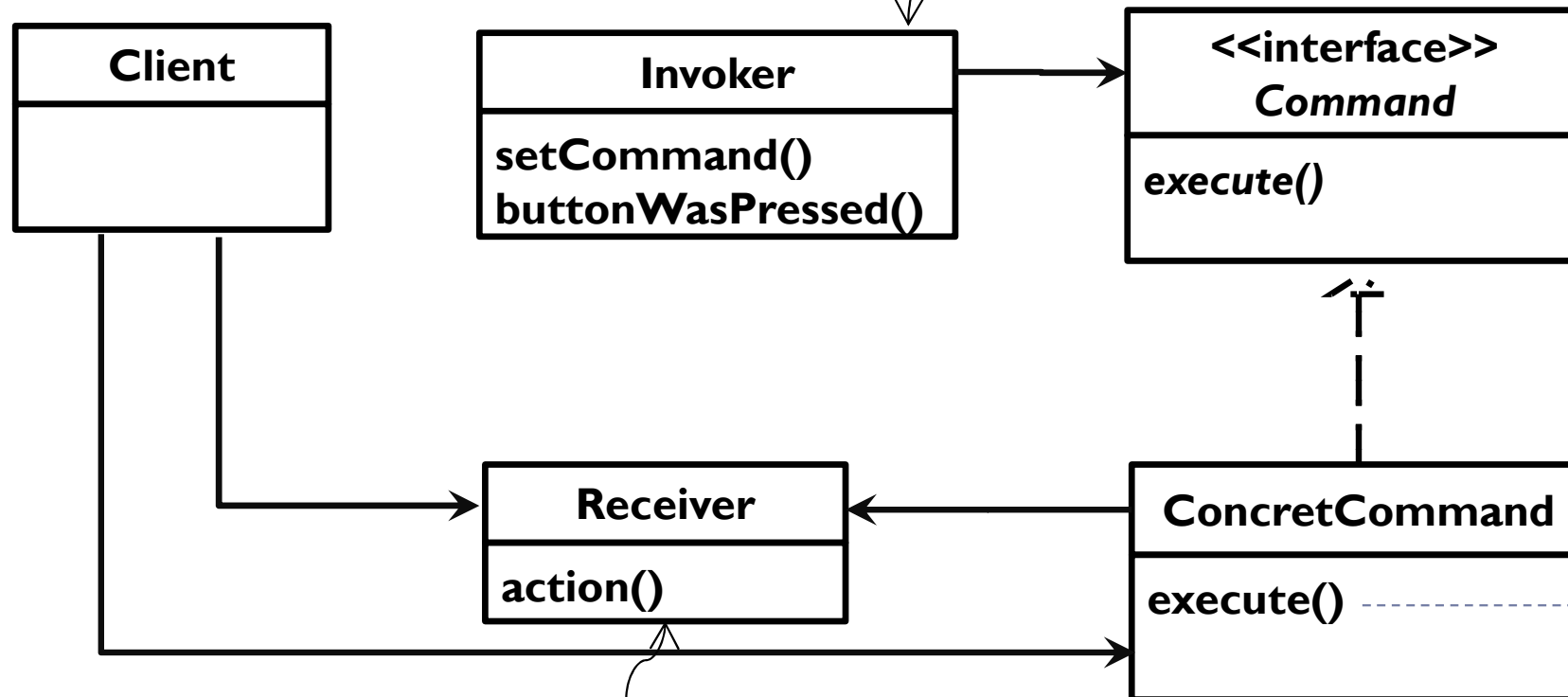
Command :

Le diagramme de classes du patron (7/23)

Le client est responsable de créer une ConcretCommand et affecter son Receiver

L'invoker tient la commande et à un certain moment demande à la commande de réaliser une requête en exécutant sa méthode execute()

Interface de tous les commandes. La commande est invoquée à travers sa méthode execute, qui demande au Receiver de réaliser des actions



Le Receiver connaît comment réaliser le travail demandé afin de satisfaire la requête.

Le ConcretCommand définit une liaison entre une action et un Receiver

```
public void execute(){
    receiver.action();
}
```

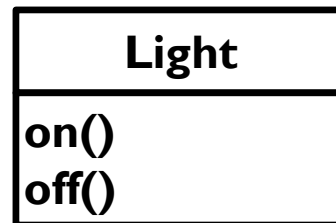

Home-Automation :

Notre premier objet Commande (8/23)

► Implémentons l'interface Command

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

► Implémentons une Commande pour allumer la lumière



```
public class LightOnCommand implements Command {  
    private Light light;  
    public LightOnCommand(Light light){  
        this.light = light;  
    }  
    @Override  
    public void execute(){  
        light.on();  
    }  
}
```

Home-Automation :

Notre premier objet Commande (9/23)

► Utilisons l'objet Commande

```
public class SimpleRemoteControl{  
    private Command slot;  
  
    public SimpleRemoteControl() {}  
  
    public void setCommand(Command command ){  
        slot=command;  
    }  
  
    public void buttonWasPressed(){  
        slot.execute();  
    }  
}
```

Home-Automation :

Notre premier objet Commande (10/23)

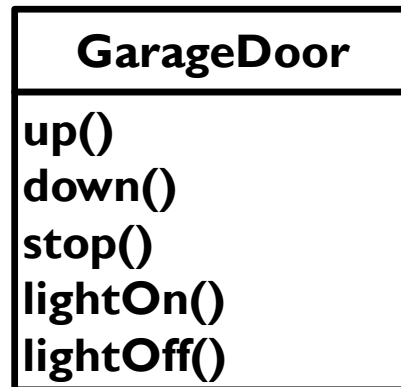
- Testons la fonctionnalité du contrôleur distant

```
public class RemoteControlTest{  
    public static void main(String[] argv) {  
        SimpleRemoteControl remote = new SimpleRemoteControl();  
        Light light=new Light();  
        LightOnCommand lightOn=new LightOnCommand (light);  
  
        remote.setCommand(lightOn); //passer la commande à l'invoker  
        remote.buttonWasPressed(); //simuler l'appui sur le bouton  
  
    }  
}
```

Home-Automation :

2^{ème} Commande (11/23)

- Développer la classe `GarageDoorOpenCommand`



```
public class GarageDoorOpenCommand implements Command {  
    private GarageDoor garageDoor;  
    public GarageDoorOpenCommand (GarageDoor garageDoor) {  
        this.garageDoor = garageDoor;  
    }  
    @Override  
    public void execute() {  
        garageDoor.up();  
        garageDoor.lightOn();  
    }  
}
```

Home-Automation :

Une autre Commande (12/23)

- ▶ Ajoutant cette commande au slot du contrôleur distant

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

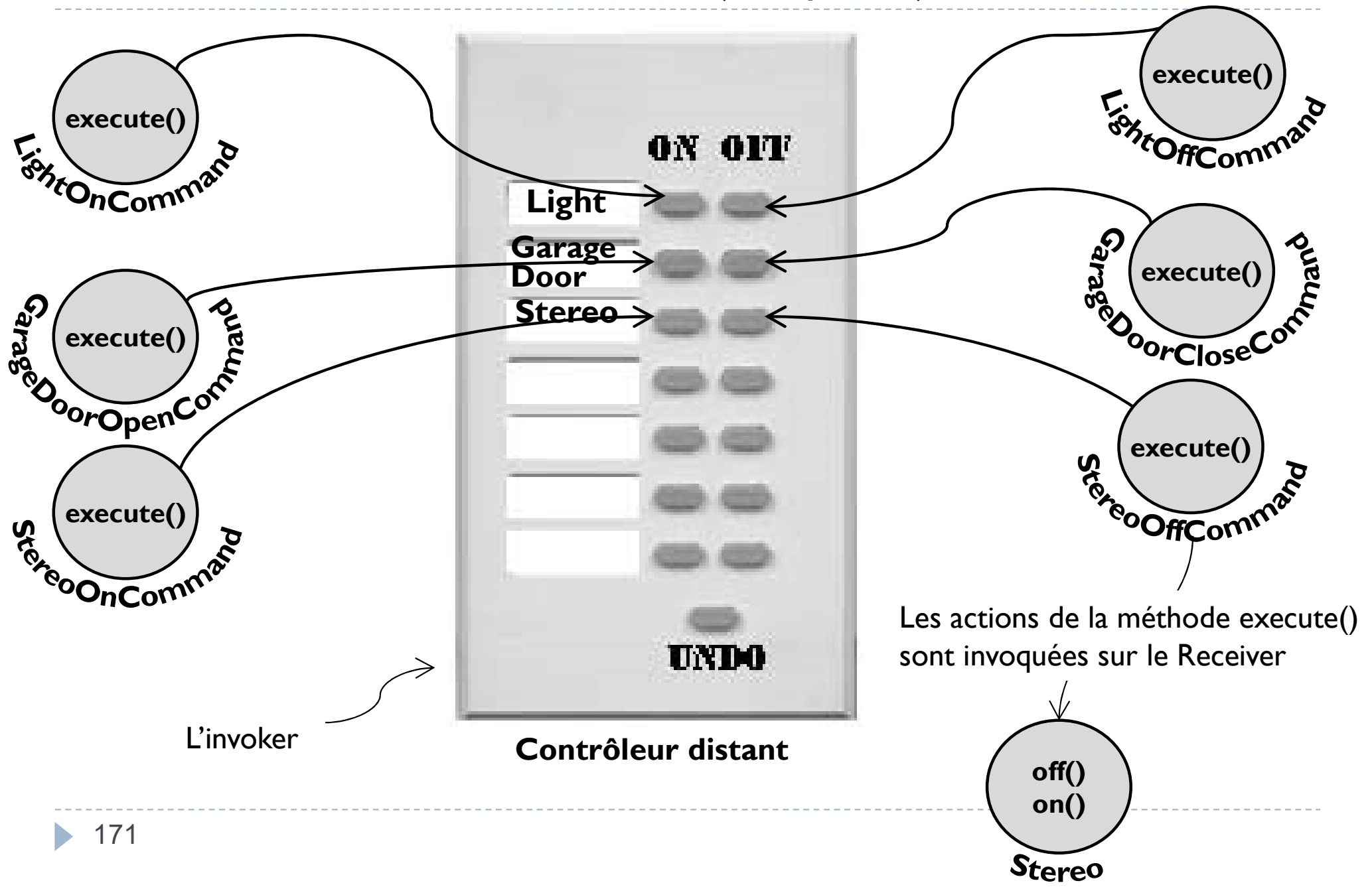
        GarageDoor garageDoor = new GarageDoor();
        GarageDoorOpenCommand garageOpen = new
            GarageDoorOpenCommand(garageDoor);

        remote.setCommand(lightOn); //encapsuler la commande
        remote.buttonWasPressed(); //allumer la lumière

        remote.setCommand(garageOpen); //encapsuler la commande
        remote.buttonWasPressed(); //ouvrir la porte du garage
    }
}
```

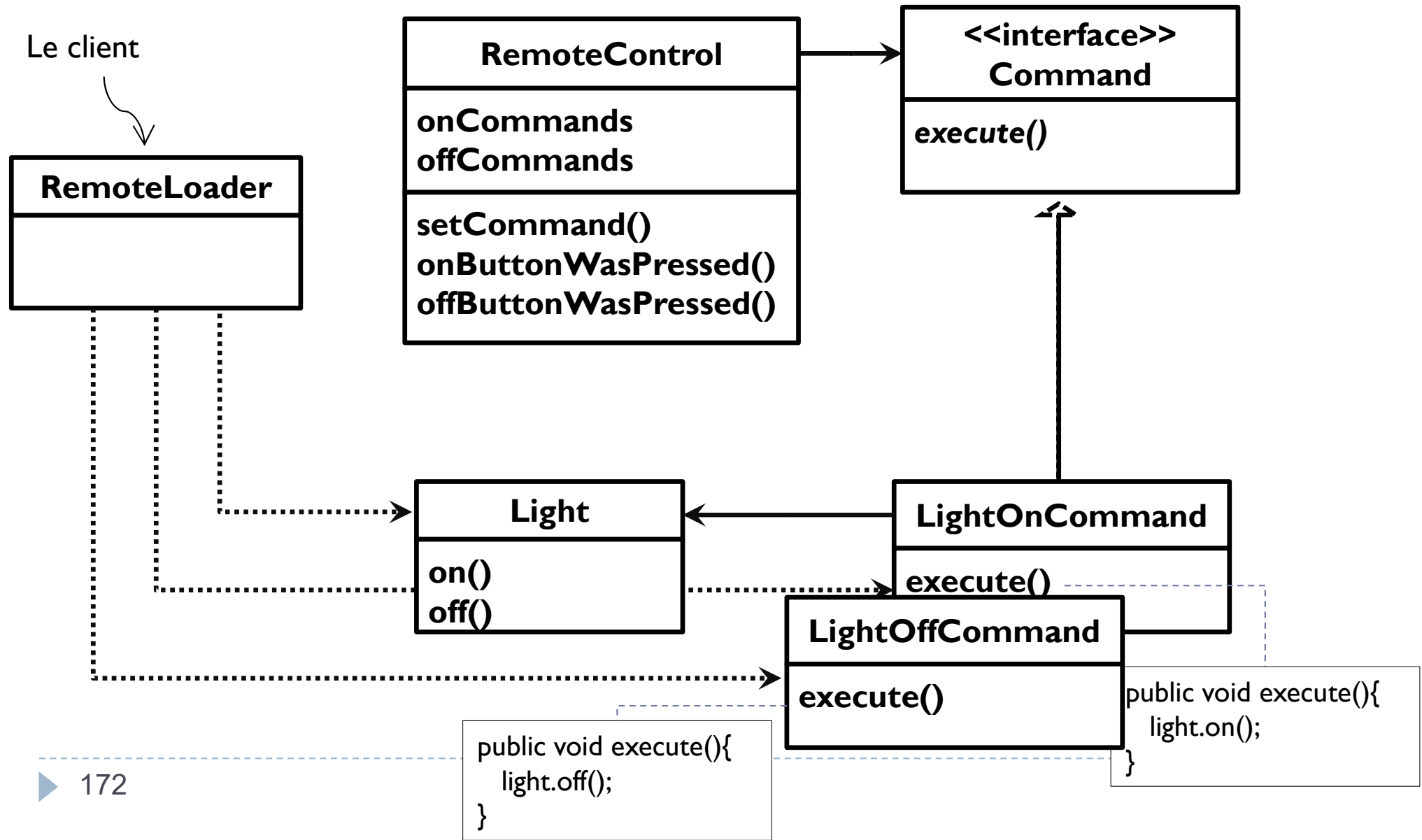
Home-Automation :

Le contrôleur distant (13/23)



Home-Automation :

Le diagramme de classes (14/23)



Home-Automation :

Programmer le contrôleur distant (15/23)

```
public class RemoteControl{
private Command[] onCommands;
private Command[] offCommands;
public RemoteControl() {
onCommands = new Command[7];
offCommands = new Command[7];
Command noCommand = new NoCommand(); ← Eviter la gestion de null
    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 onButtonWasPressed(int slot){
    onCommands[slot].execute();
}
public void offButtonWasPressed(int slot){
    offCommands[slot].execute();
}
@Override
public String toString(){
    String s="";
    for(int i=0;i<7;i++){
        s+="Slot["+i+"] "+onCommands[i].getClass().getName() +"
"+offCommands[i].getClass().getName()+"\n";
    }
    return s;
}}
```

173

Home-Automation :

Programmer les commandes (16/23)

- ▶ Programmer les classes suivantes:
 - ▶ Light.java, Stereo.java
 - ▶ LightOnCommand.java, LightOffCommand.java, StereoOnWithCDCommand.java, StereoOffCommand.java

```
public class LightOnCommand
implements Command {
private Light light;
public LightOnCommand(Light
light){
    this.light = light;
}
@Override
public void execute(){
    light.on();
}
}
```

```
public class StereoOnWithCDCommand
implements Command {
private Stereo stereo;
public StereoOnWithCDCommand(Stereo
stereo){
    this.stereo = stereo;
}
@Override
public void execute(){
    stereo.on();
    stereo.setCD();
    stereo.setVolume(11);
}
}
```

Home-Automation :

Tester le contrôleur (17/23)

```
public class RemoteLoader{
public static void main(String[] arg){
RemoteControl remoteControl = new RemoteControl();
Light livingRoomLight=new Light();
LightOnCommand livingRoomLightOnCommand = new
    LightOnCommand(livingRoomLight);
LightOffCommand livingRoomLightOffCommand = new
    LightOffCommand(livingRoomLight);
remoteControl.setCommand(0, livingRoomLightOnCommand,livingRoomLightOffCommand);
remoteControl.onButtonWasPressed(0);
remoteControl.offButtonWasPressed(0);
Stereo stereo = new Stereo();
StereoOnWithCDCommand stereoOnWithCDCommand = new
StereoOnWithCDCommand(stereo);
StereoOffCommand stereoOffCommand = new StereoOffCommand(stereo);
remoteControl.setCommand(1,stereoOnWithCDCommand,stereoOffCommand);
remoteControl.onButtonWasPressed(1);
remoteControl.offButtonWasPressed(1);
System.out.println(remoteControl.toString());
}
}
```

Home-Automation :

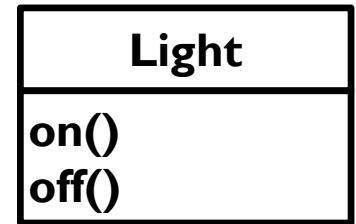
Undo: Annuler la dernière opération (18/23)

► Implémentons l'interface Command

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

► Implémentons une Commande pour allumer la lumière

```
public class LightOnCommand implements Command {  
    private Light light;  
    public LightOnCommand(Light light){  
        this.light = light;  
    }  
    @Override  
    public void execute(){  
        light.on();  
    }  
    @Override  
    public void undo(){  
        light.off();  
    }  
}
```



Opération à exécuter en cas
d'annulation de cette commande

Home-Automation :

Undo: Annuler la dernière opération (19/23)

```
public class RemoteControl{
private Command onCommands[];
private Command offCommands[];
private Command undoCommand;
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;
    }
undoCommand = noCommand; }
public void onButtonWasPressed(int slot){
    onCommands[slot].execute();
    undoCommand = onCommands[slot];
}
public void offButtonWasPressed(int slot){
    offCommands[slot].execute();
    undoCommand = offCommands[slot];
}
public void undoButtonWasPressed(){
    undoCommand.undo();
}
}
```

C'est ou on stockera la dernière commande exécutée pour le bouton undo

Initialisation à NoCommand afin d'éviter le traitement de null

Enregistrer la dernière commande

Lorsqu'on appuie sur le bouton undo, on invoque la méthode undo() pour annuler la dernière commande exécutée

Home-Automation :

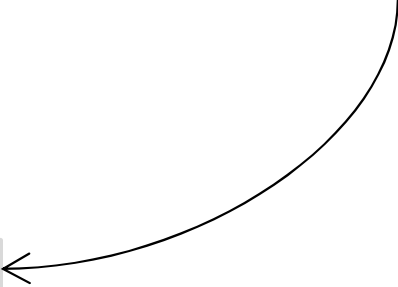
Tester le contrôleur avec UNDO (20/23)

Annuler l'allumage de la lumière

```
class RemoteLoader{
public static void main(String arg[]){
//...
remoteControl.onButtonWasPressed(0);
remoteControl.undoButtonWasPressed();

//...

remoteControl.onButtonWasPressed(1);
remoteControl.offButtonWasPressed(1);
System.out.println(remoteControl.toString());
}
}
```



Home-Automation :

La Macro-Commande (21/23)

- C'est le regroupement de plusieurs commandes en une seule

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

← Stocker les commandes de la macro-commande

Un boucle pour exécuter toutes les commandes de la macro-commande

←

- ❶ Créer les commandes à mettre dans la macro-commande

```
LightOnCommand lightOnCommand = new LightOnCommand(light);  
StereoOnWithCDCommand stereoOnWithCDCommand = new  
    StereoOnWithCDCommand(stereo);  
TVOnCommand tvOnCommand = new TVOnCommand(tv);  
  
//créer aussi les Off-Commandes
```

← Créer les commandes

Home-Automation :

MacroCommand (22/23)

② Créer les macro-commandes

```
Command [] on = {lightOnCommand, stereoOnWithCDCommand, tvOnCommand};  
Command [] off = {lightOffCommand, stereoOffCommand, tvOffCommand};
```



Les commandes sous
formes de tableau

```
MacroCommand onMacro= new MacroCommand(on);  
MacroCommand offMacro= new MacroCommand(off);
```

③ Affecter les macro-commandes à un bouton

```
remoteControl.setCommand(2, onMacro, offMacro);
```



Affecter les macro-commandes
au bouton du slot n°2

④ Exécuter la macro-commande

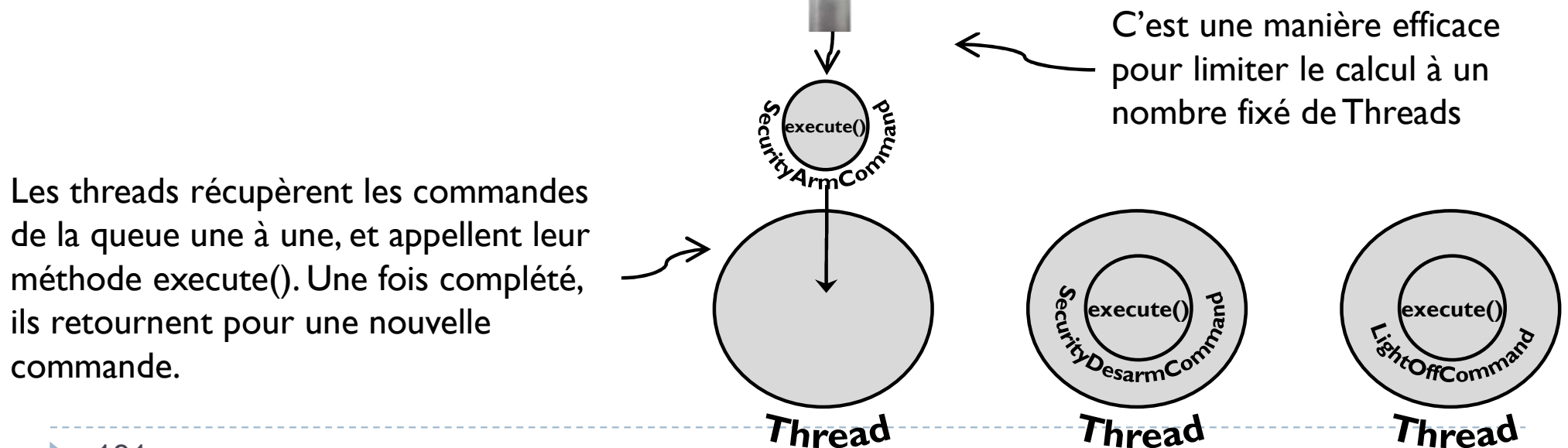
```
System.out.println("Macro On");  
remoteControl.onButtonWasPressed(2);  
System.out.println("Macro Off");  
remoteControl.offButtonWasPressed(2);
```



Tester ces macro-commandes et
donner le résultat de l'exécution

Autre utilisation du patron Command : Organiser les queues de requêtes (23/23)

- Les commandes nous offrent une manière de paquetage les morceaux de calcul (computation). Les calculs (commandes créées par des applications) seront placés dans des queues (queue de jobs) pour être exécutés.



Récapitulatif (1 / 2)

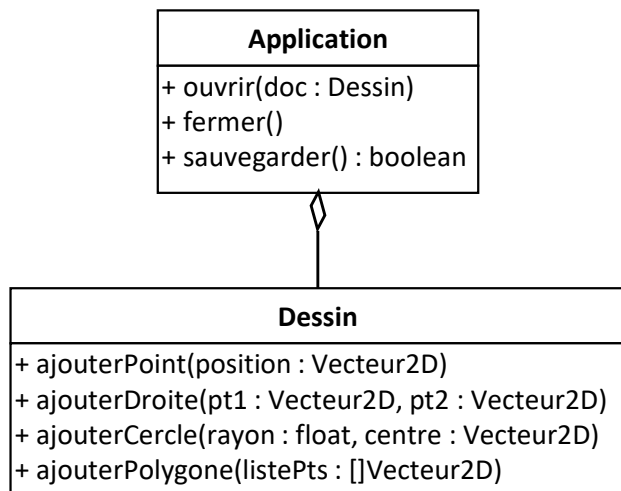
- ▶ Bases de l'OO: Abstraction, Encapsulation, Polymorphisme & Héritage
- ▶ Principes de l'OO
 - ▶ Encapsuler ce qui varie
 - ▶ Favoriser la composition sur l'héritage
 - ▶ Programmer pour des interfaces
 - ▶ Opter pour une conception faiblement couplée
 - ▶ Les classes doivent être ouvertes pour les extensions et fermées pour les modifications
 - ▶ Dépendre des abstractions. Ne jamais dépendre de classes concrètes
- ▶ Patron de l'OO
 - ▶ Strategy: définit une famille d'algorithmes interchangeables
 - ▶ Observer: définit une dépendance 1-à-plusieurs entre objets.
 - ▶ Decorator: attache des responsabilités additionnelles à un objet dynamiquement.
 - ▶ Abstract Factory: offre une interface de création de familles d'objets
 - ▶ Factory Method: définit une interface de création des objets
 - ▶ Singleton: assure à une classe une seule instance
 - ▶ **Command**: encapsule une requête comme un objet

Récapitulatif (2 / 2)

- ▶ Le patron Command découple un objet, faisant des requêtes, de l'objet qui sait comment la réaliser
- ▶ Un objet Command est au centre de ce découplage et encapsule le Receiver avec une action (ou des actions)
- ▶ L'Invoker exécute la requête d'un objet Command en appelant sa méthode `execute()`, qui invoque les actions sur le Receiver
- ▶ Le patron Command supporte l'annulation (undo) par l'implémentation d'une méthode `undo()` -dans la commande- qui restore l'ancien état du système (avant l'exécution de la méthode `execute()`)
- ▶ Les Macro-Commandes sont des simples extensions de Command qui permettent l'invocation de multiple commandes. Pareil, les macro-commandes peuvent supporter les méthodes d'annulation (undo).
- ▶ Le patron Command est utilisé aussi pour implémenter l'organisation des requêtes (Jobs) et la gestion de la journalisation (logging)

Exercice 1 (1/2)

- ▶ On vous demande de participer à la création d'un nouvel outil graphique. Cet outil permettra de créer de manière intuitive des dessins avec un niveau de complexité plus ou moins élevé. Les dessins pourront être composés d'un ensemble de points, droites, arc de cercles ou autres formes simples telles que des cercles ou des polygones. Cet outil sera similaire au programme appelé «Paint» sous l'environnement Windows. La figure suivante présente un diagramme de classes simplifié pour cette application :



- ▶ Vous êtes chargé de concevoir le mécanisme qui permettra de garder une trace des actions de l'utilisateur. Ce dernier pourra ainsi annuler les dernières actions faites.
- ▶ Question: Faites les modifications nécessaires au diagramme de classes pour implanter le patron Commande (voir mail dans la page suivante).

Exercice 1 (2/2)

```
public static void main(String[] args) {  
    Dessin dessin = new Dessin("Dessin1");  
    Application application = new Application();  
    //invoker  
    Palette palette = new Palette();  
    //command ouvrir Application  
    OuvrirApplicationCommand c0 = new OuvrirApplicationCommand(application, dessin);  
    palette.setCommand(0, c0);  
    palette.buttonWasPressed(0);  
    //Command Point  
    Vecteur2D position = new Vecteur2D(1, 2);  
    AjouterPointDessinCommand c1 = new AjouterPointDessinCommand(dessin, position);  
    palette.setCommand(1, c1);  
    palette.buttonWasPressed(1);  
    //Command Polygone  
    Vecteur2D[] listePts = { new Vecteur2D(2, 3), new Vecteur2D(4, 7), new Vecteur2D(1, 5), new Vecteur2D(1, 0) };  
    AjouterPolygoneDessinCommand c2 = new AjouterPolygoneDessinCommand(  
        dessin, listePts);  
    palette.setCommand(2, c2);  
    palette.buttonWasPressed(2);  
    //Command Sauvegarde  
    SauvegarderApplicationCommand c3 = new SauvegarderApplicationCommand(application);  
    palette.setCommand(3, c3);  
    palette.buttonWasPressed(3);  
}
```

Exercice 2

- Le but de cet exercice est de tester la puissance du pattern Command. Pour ce faire, nous disposons d'une calculatrice offrant les opérations arithmétiques de base : +, -, /, * et racine carrée sur 2 réels et nous voulons transformer les actions (des utilisateurs) de calculs sur cette calculatrice en des commandes.

- 1. Donne le digramme de classes décrivant cette transformation avec le pattern Command.

- 2. Implanter ce diagramme tout en respectant le client suivant :

```
public class Client {  
    public static void main(String[] args) {  
        Calculatrice calculatrice = new Calculatrice();  
        Command plusCommand = new PlusCommand(calculatrice, 1, 3);  
        Command sousCommand = new SousCommand(calculatrice, 8, 12);  
        Command divCommand = new DivCommand(calculatrice, 7, 4);  
        Command multCommand = new MultCommand(calculatrice, 4, 5);  
        Command sqrtCommand = new SqrtCommand(calculatrice, 9);  
        CalculatriceControl control = new CalculatriceControl();  
        control.setCommand(0, plusCommand); control.buttonPressed(0);  
        control.setCommand(1, sousCommand); control.buttonPressed(1);  
        control.setCommand(2, divCommand); control.buttonPressed(2);  
        control.setCommand(3, multCommand); control.buttonPressed(3);  
        control.setCommand(4, sqrtCommand); control.buttonPressed(4);    } }  
}
```

Calculatrice
+ plus(x: float, y: float): float
+ mult(x: float, y: float): float
+ sous(x: float, y: float): float
+ div(x: float, y: float): float
+ sqrt(x: float): float

- 3. Implanter la macro commande qui permet de résoudre l'équation : $ax^2 + bx + c = 0$.

Command secondDegre = new MacroCommand(calculatrice, 4, 5, 1); //a = 4, b = 5 et c = 1

Avec : $\Delta = b^2 - 4ac$. $x_1 = \frac{-b + \sqrt{\Delta}}{2a}$ et $x_2 = \frac{-b - \sqrt{\Delta}}{2a}$.