

# Université de Corse

## 2025-2026

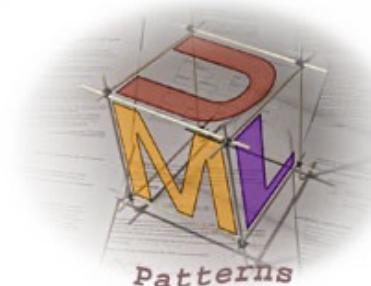
### MASTER DFS-DE 1ère année

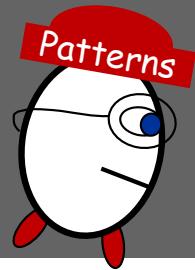
## Cours PATTERNS

### CH 2.4 - Patterns GOF Comportementaux



Evelyne VITTORI  
Université de Corse  
CORTE  
[vittori@univ-corse.fr](mailto:vittori@univ-corse.fr)





# CH 2 – PATTERNS GangOfFour

## 1 – Introduction

## 2 – Patterns créationnels

## 3 - Patterns structurels

## 4 - Patterns comportementaux

■ Strategy

■ Observer  
■ State  
■ Command

■ Chaîne de responsabilités

■ Interpréteur

■ Médiateur

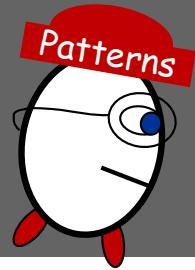
■ Memento

■ Patron de méthode

■ Itérateur

■ Visiteur





# Patterns de Conception

## Plan du Cours

### CH1 – Fondements de l'approche « Patterns »

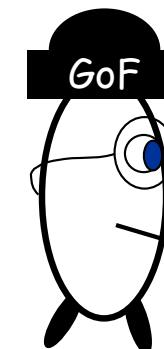
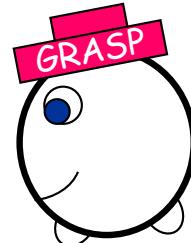
### CH2 – Patterns Gof

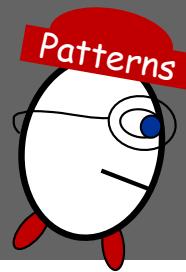
Introduction

2.1 – Patterns créatifs

2.2 - Patterns structurels

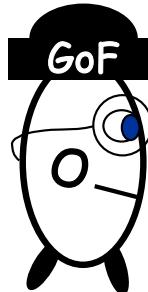
2.3 - Patterns comportementaux





# Observateur «Observer»

## Problème

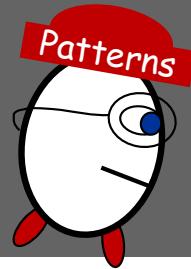


Comment modéliser une application où des **objets souscripteurs** doivent réagir à leur manière aux changements d'état et évènements d'un **objet diffuseur**?

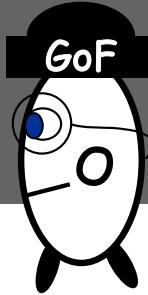
## Exemple

Application d'affichage des données collectées par une station météo

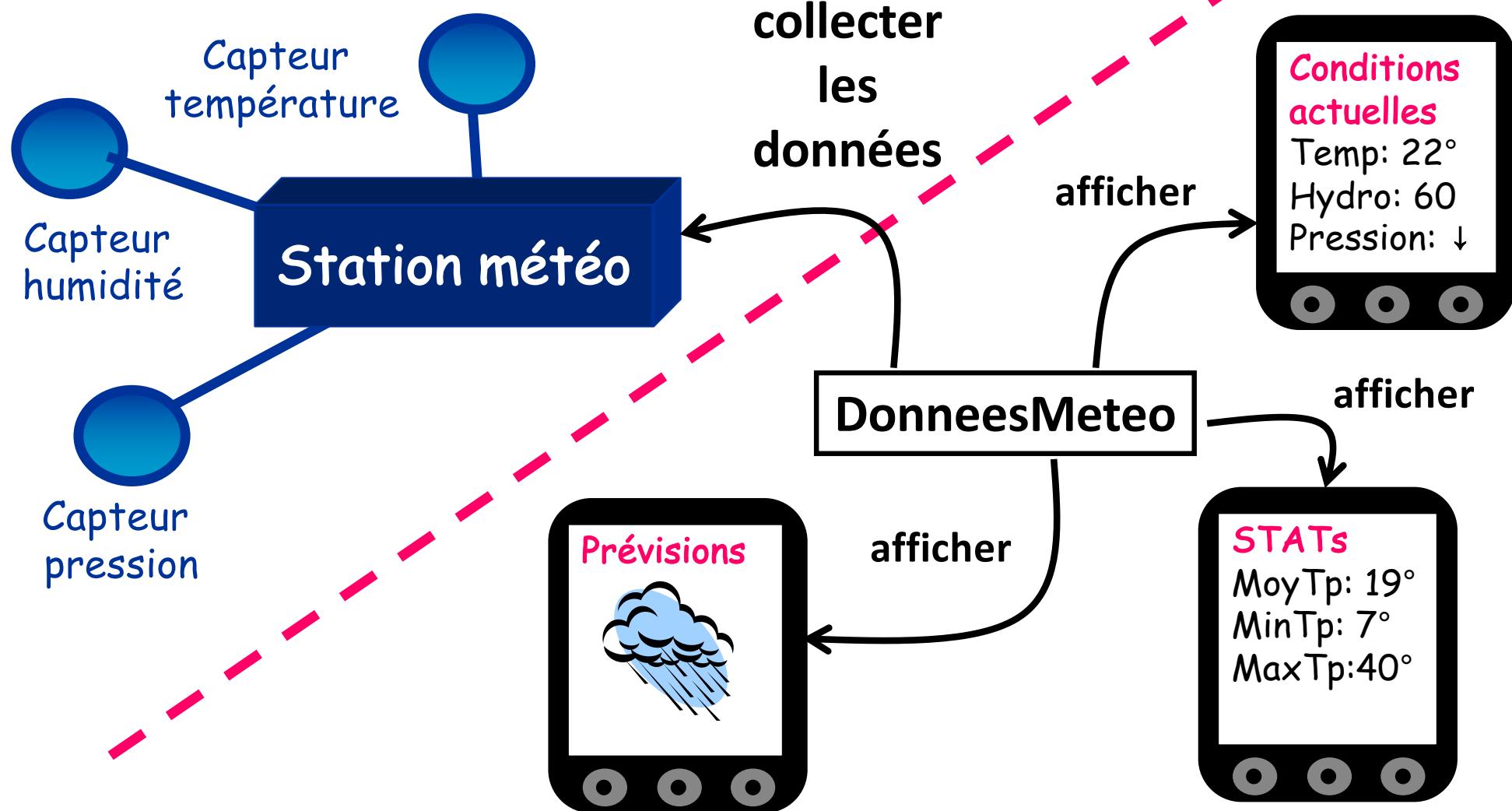


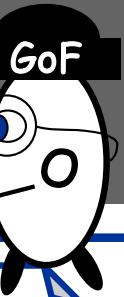
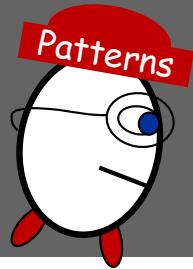


# Observer



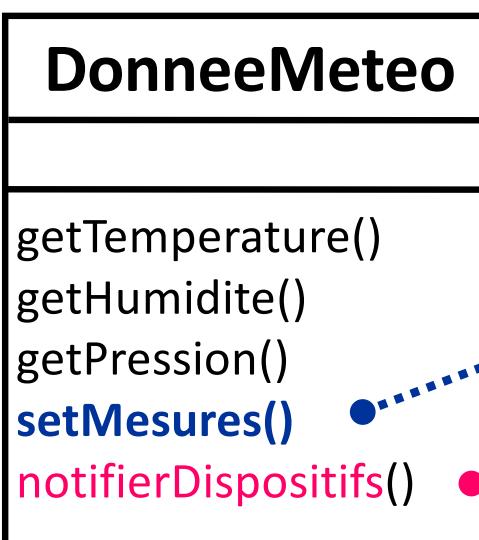
## Exemple (problème)





# Observateur «Observer»

## Exemple



Méthode de collecte des informations auprès de la station météo  
(temp, humidité et pression)

**Méthode à implémenter**  
Méthode informant les dispositifs d'affichage de la modification des données

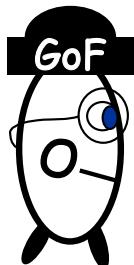
## Problèmes à résoudre

- Implémenter les trois dispositifs d'affichage en assurant leur mise à jour dynamique
- S'assurer de l'extensibilité de l'application (nouveaux dispositifs d'affichage)

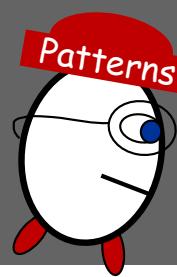


# Observer

## Solution



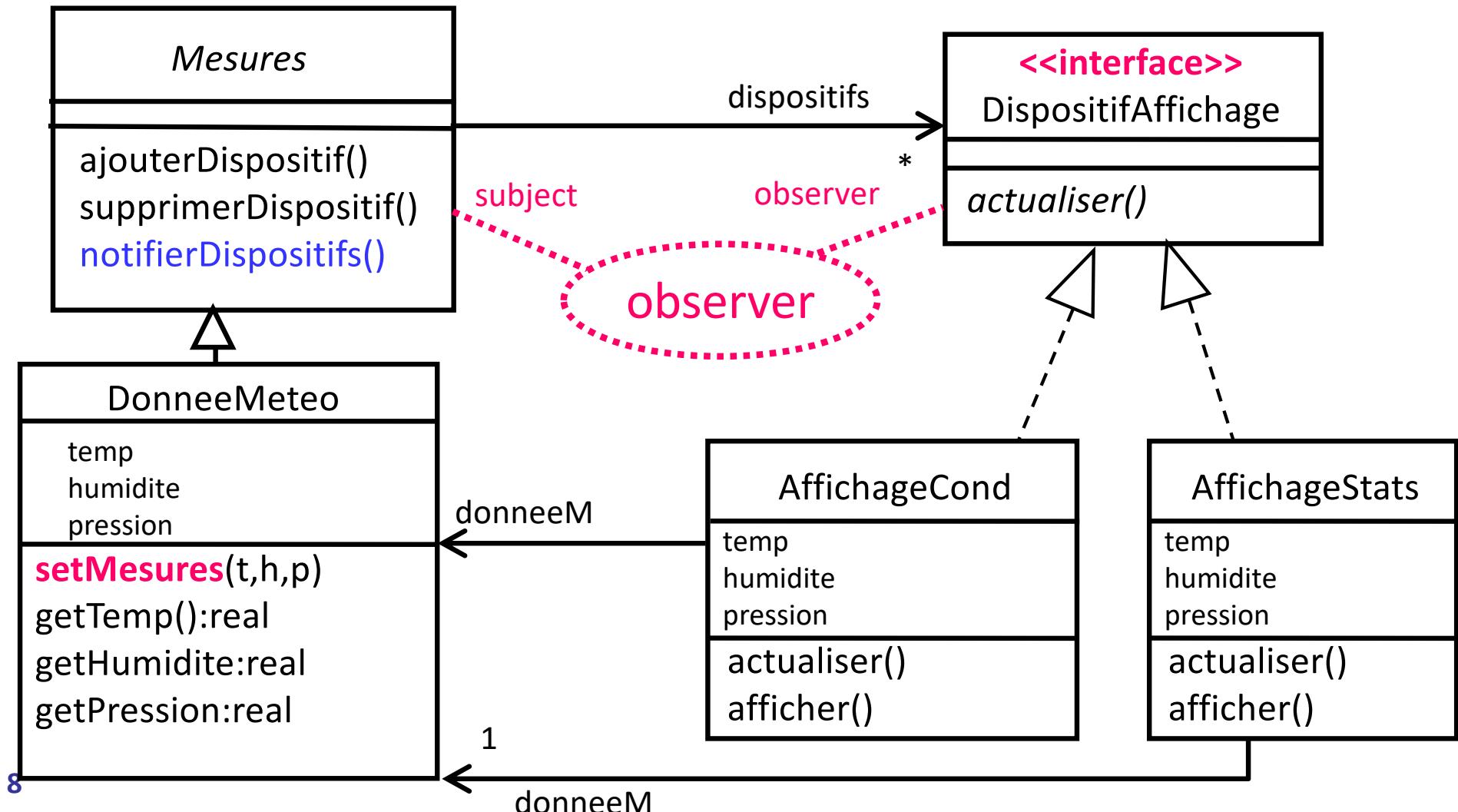
- L'objet diffuseur (**sujet**) met à jour les objets souscripteurs (**observateurs**) via une interface commune.
  
- Le pattern O définit une relation 1-\* de façon à ce que dès que l'état du sujet change, tous les observateurs en soient notifiés.

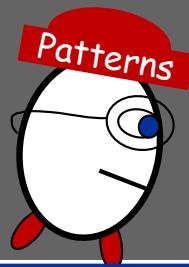


# Observer



## Solution (exemple)

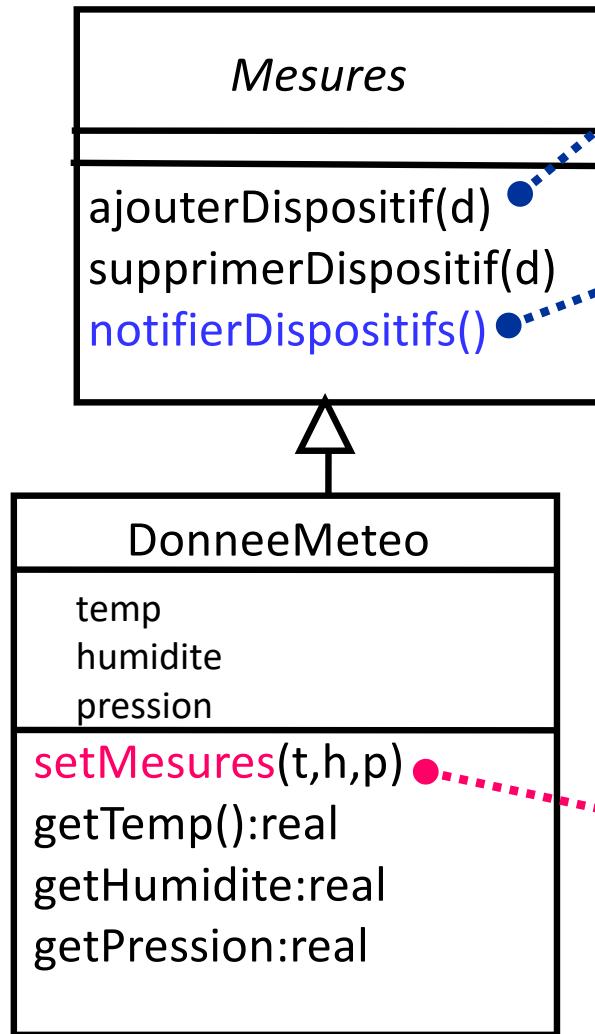




# Observer



## Solution (exemple)



*dispositifs.add(d)*

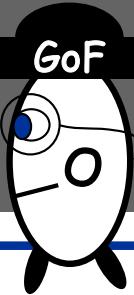
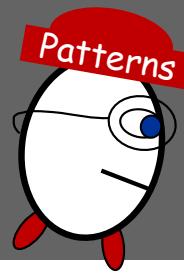
*Pour chaque élément d de dispositifs  
d.actualiser()*

*dispositifs*

**<<interface>>**  
**DispositifAffichage**  
*actualiser()*

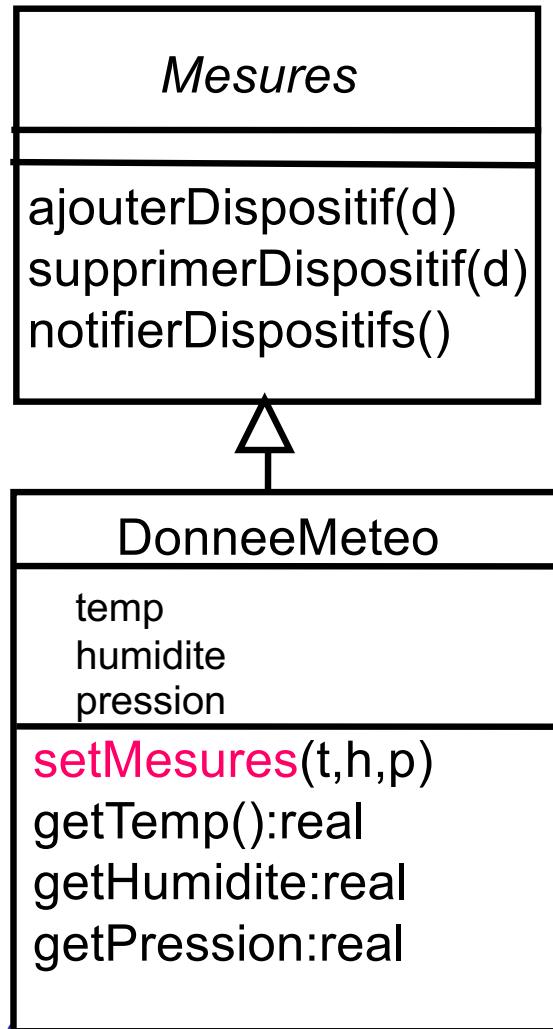
\*

*//Collecte des données sur la station*  
*this.temp= t;*  
*this.humidite=h;*  
*this.pression=p;*  
*notifierDispositifs();*

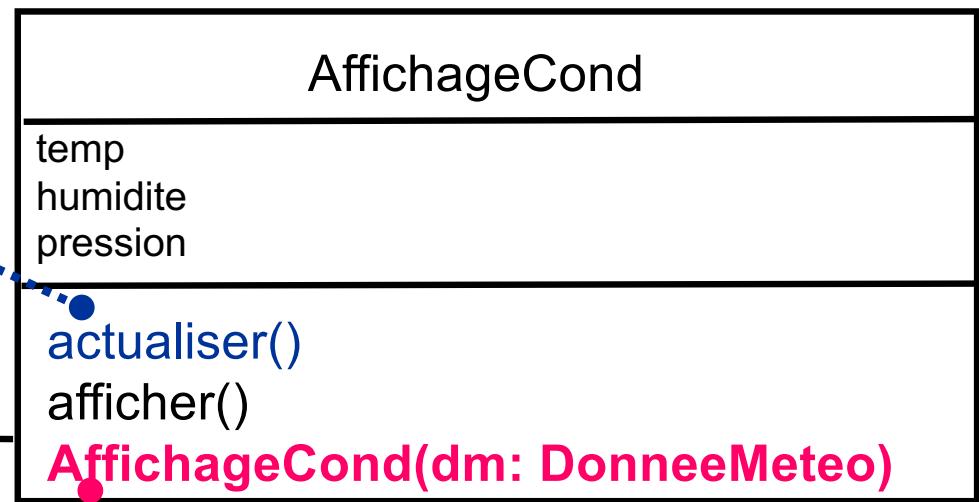


# Observer

## Solution (exemple)

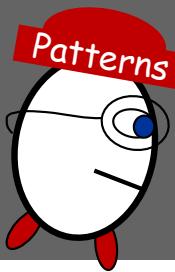


```
this.temp=donneeM.getTemp();
this.humidite=donneeM.getHumidite();
this.pression=donneeM.getPression();
afficher();
```



constructeur

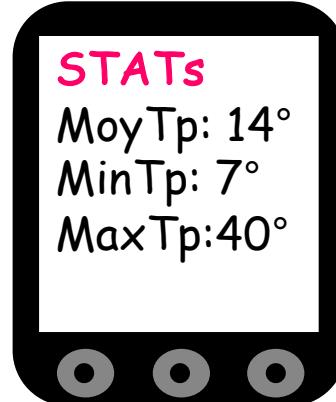
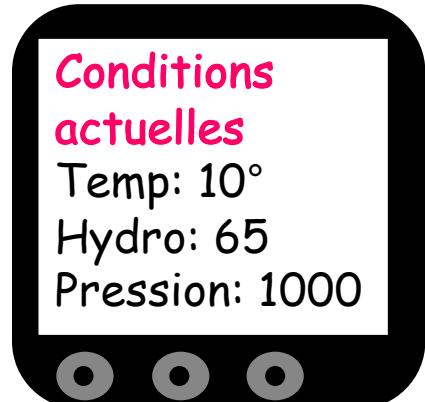
```
this.donneeM=dm;
donneeM.ajouterDispositif(this);
```

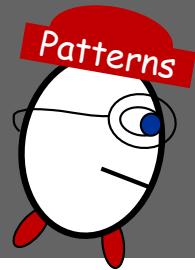


# Observer

→ Exemple de programme de test

```
class StationMeteo {  
public void test() {  
    DonneesMeteo donnees = new DonneesMeteo();  
  
    DispositifAffichage affichCond =new AffichageCond(donnees);  
    DispositifAffichage affichStat =new AffichageStat(donnees);  
    donnees.setMesures(10, 65, 1000);  
}
```

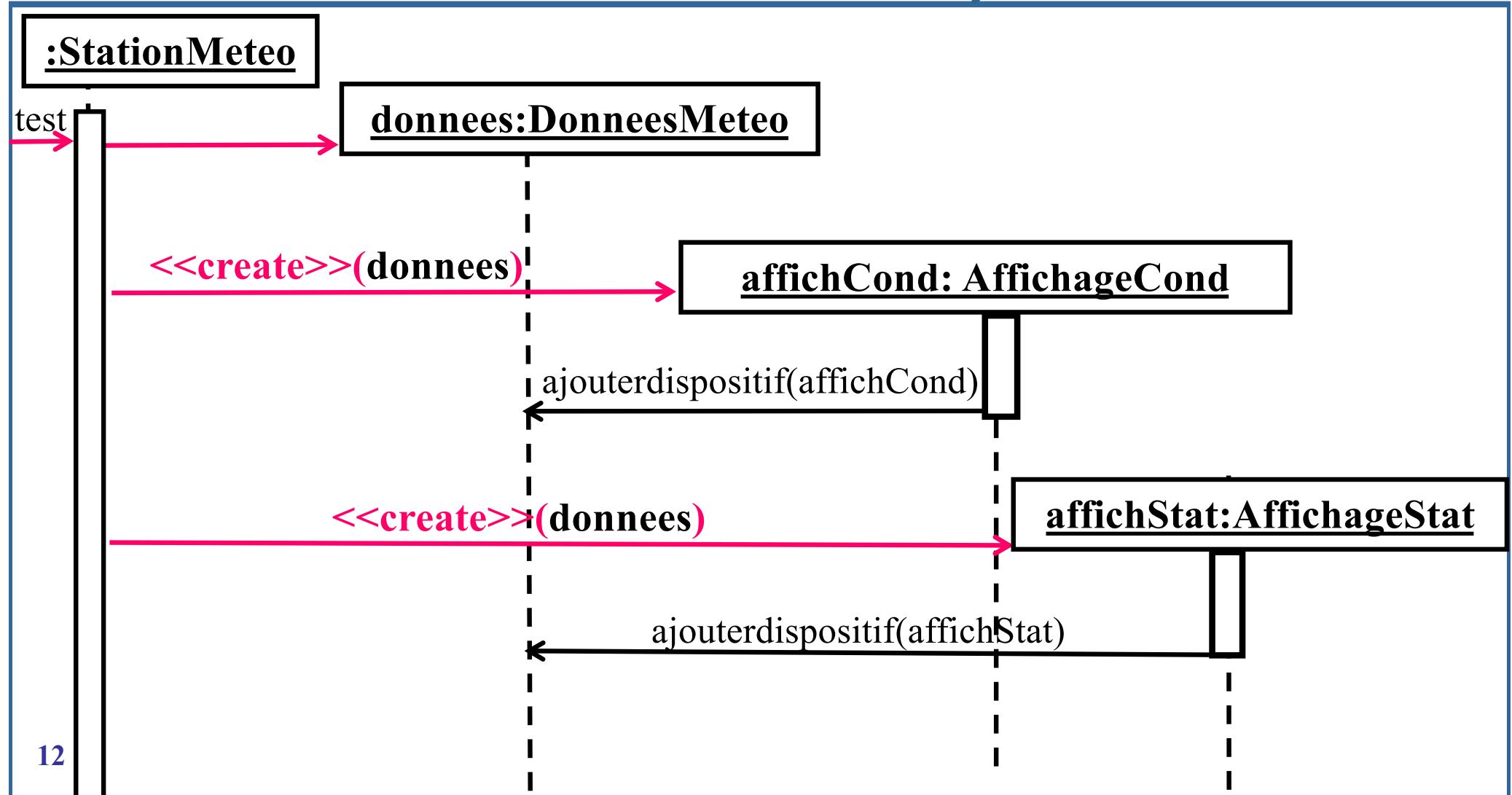


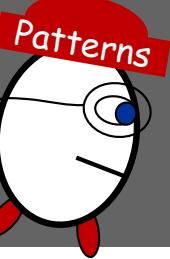


# Observer

## Diagramme de séquence

Exécution de la méthode test sur un objet StationMeteo

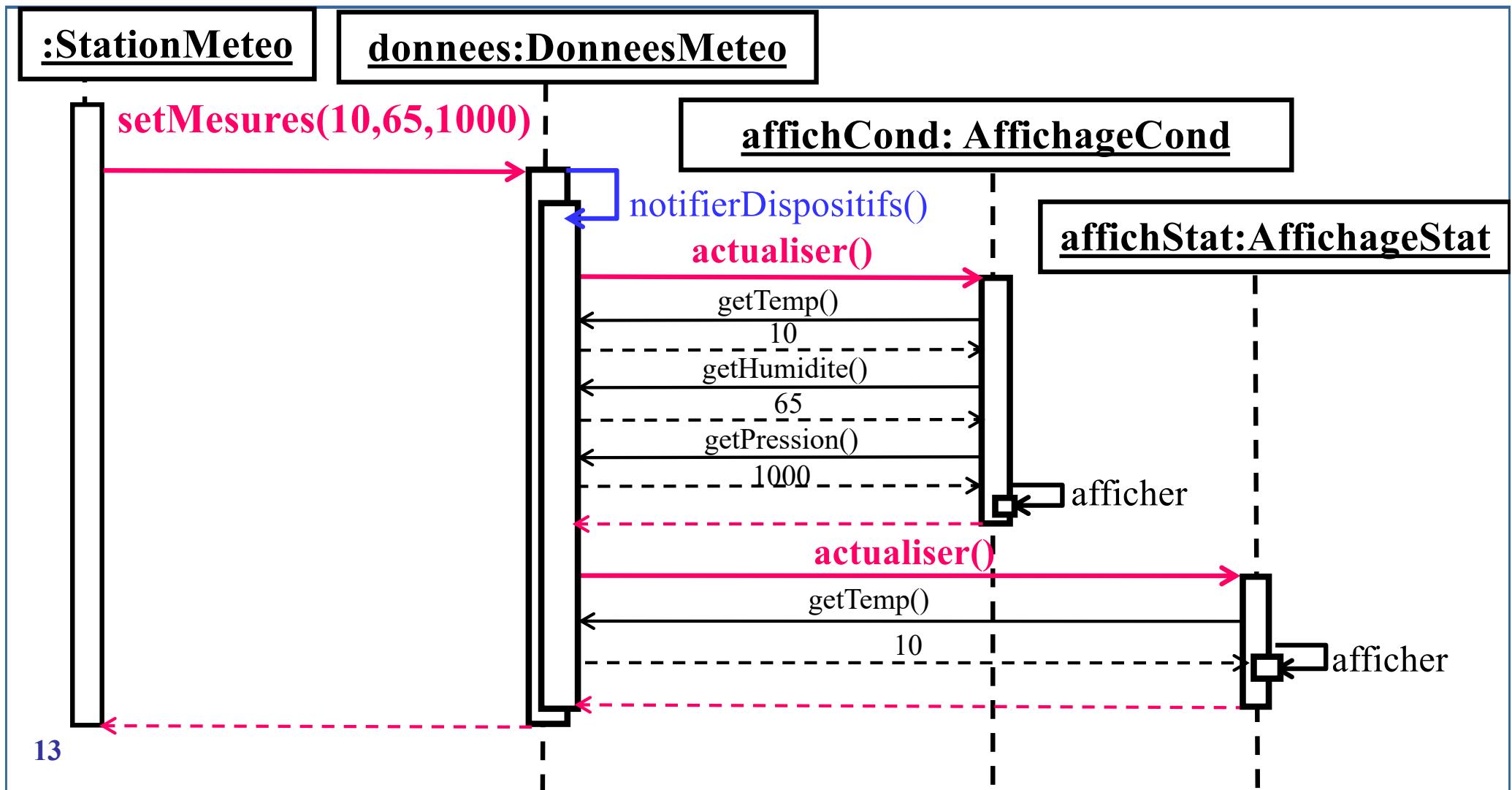


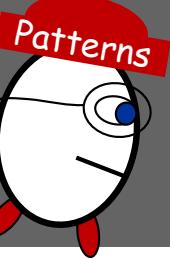


# Observer

## Diagramme de séquence (suite)

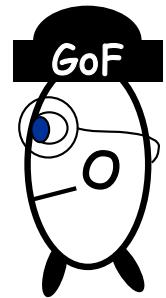
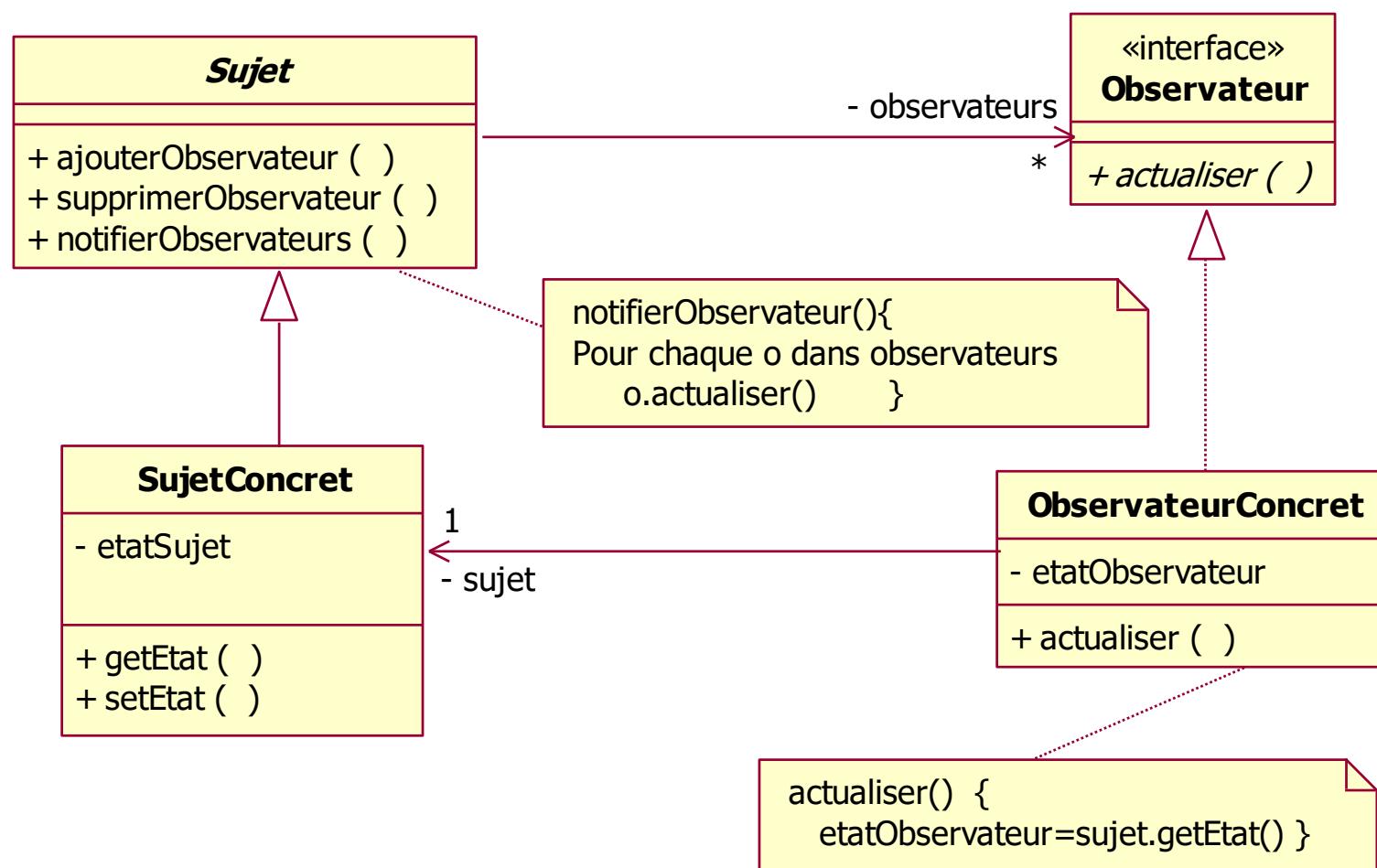
Exécution de la méthode test sur un objet StationMeteo

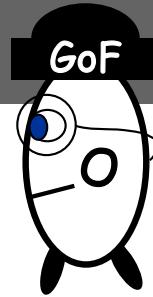
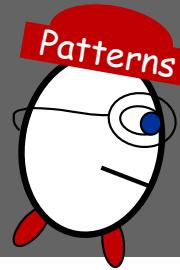




# Observer

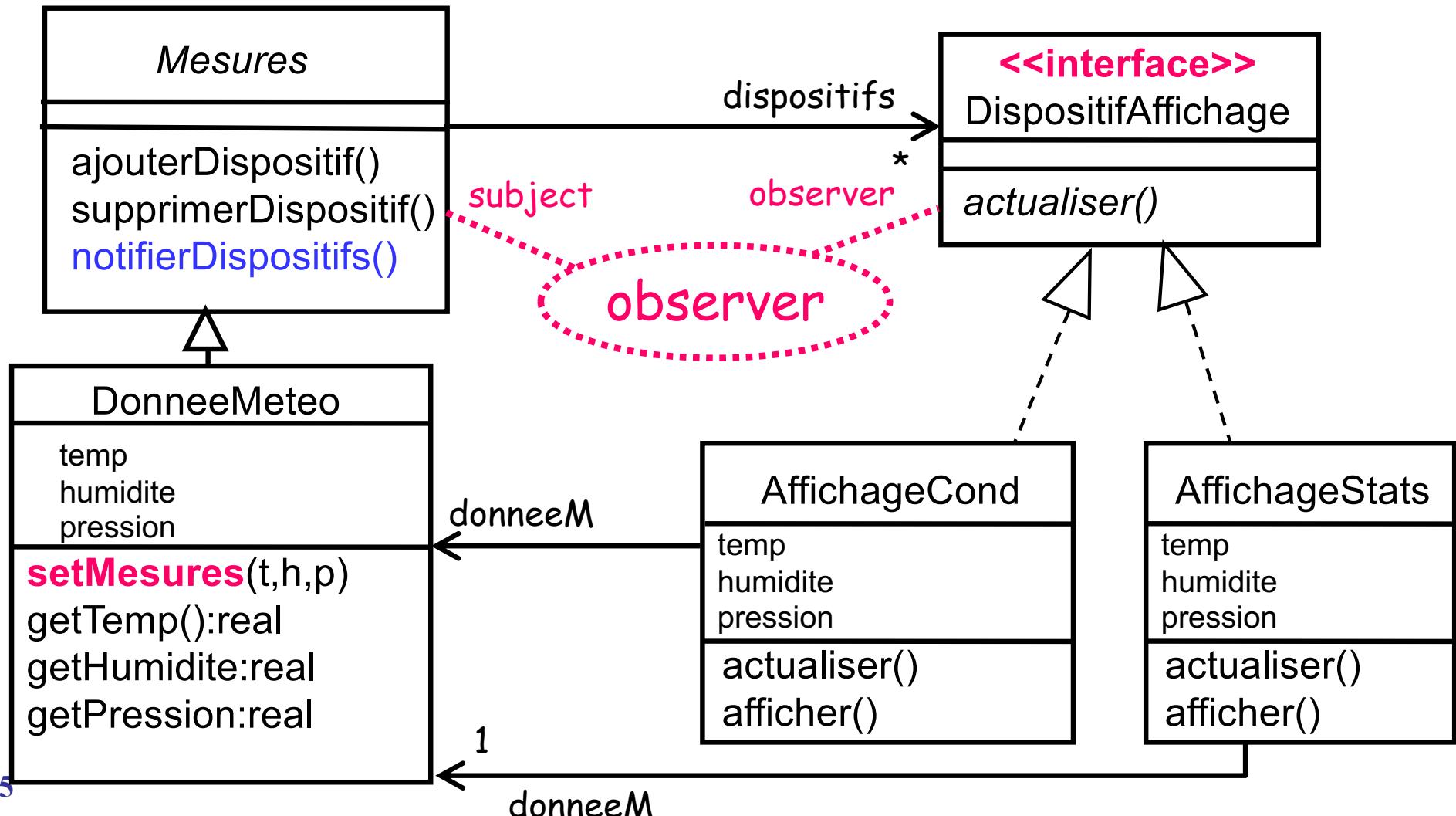
## Solution (cas général)

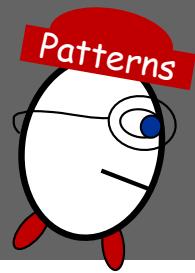




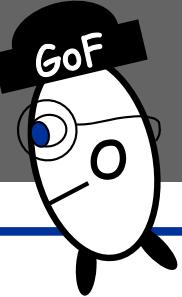
# Observer

## Solution (exemple)





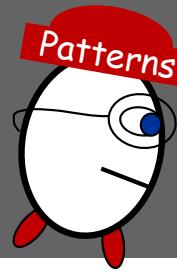
# Observer



## Conséquences

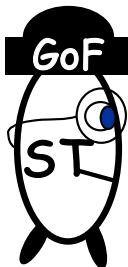
- Nous pouvons ajouter de nouveaux observateurs (ex: nouveaux dispositifs) sans toucher au sujet.
- Les sujets et les observateurs peuvent être réutilisés dans d'autres contextes.
- Les modifications des sujets n'affectent pas les observateurs et réciproquement.
- La structure peut entraîner des difficultés de maintenance





# Etat «State»

## Problème



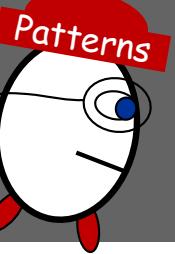
Comment permettre à un objet de modifier son comportement lorsque son état interne change?

## Exemple



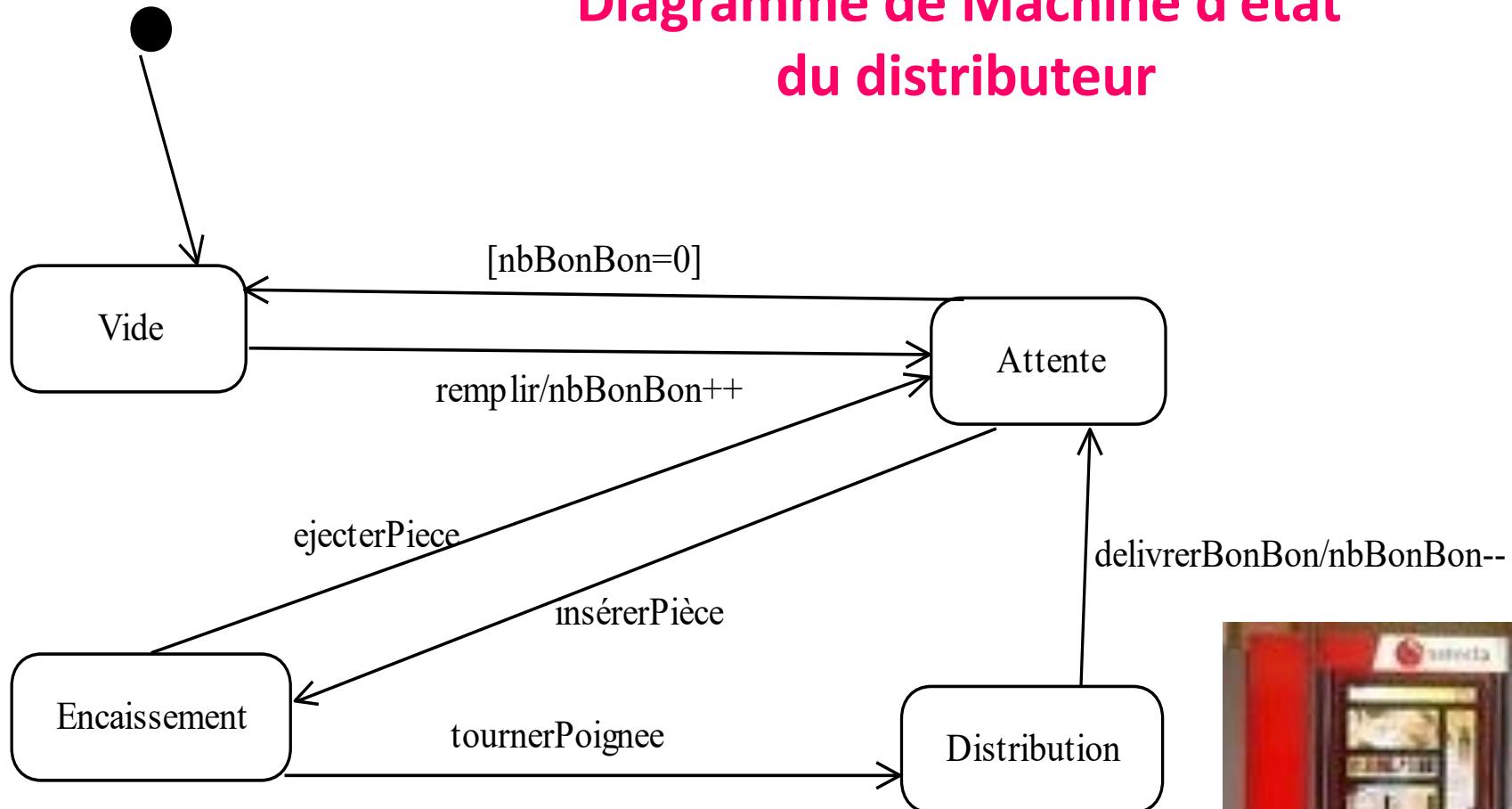
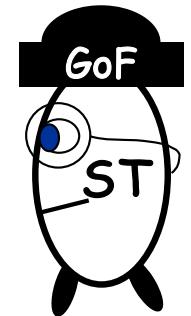
Modélisation d'un distributeur de bonbons  
4 états possibles

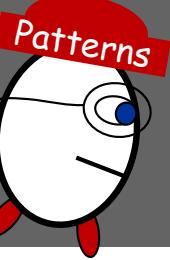
«Vide»  
«Attente»  
«Encaissement»  
«Distribution»



# State

## Diagramme de Machine d'état du distributeur

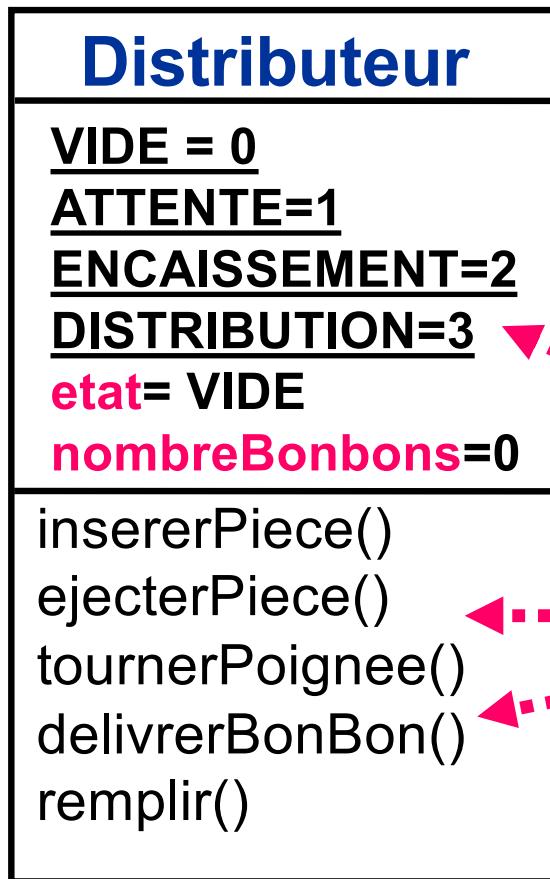




# State

## Exemple

### Solution 1



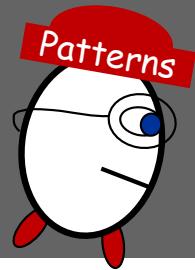
*Modification:* 10% du temps lorsque le client tourne la poignée, il est gagnant et obtient 2 bonbons au lieu d'un!!

Il faut définir un 5ème état  
« Gagnant »

- Ajout constante
- Modifications méthodes

J'ai une meilleure solution!

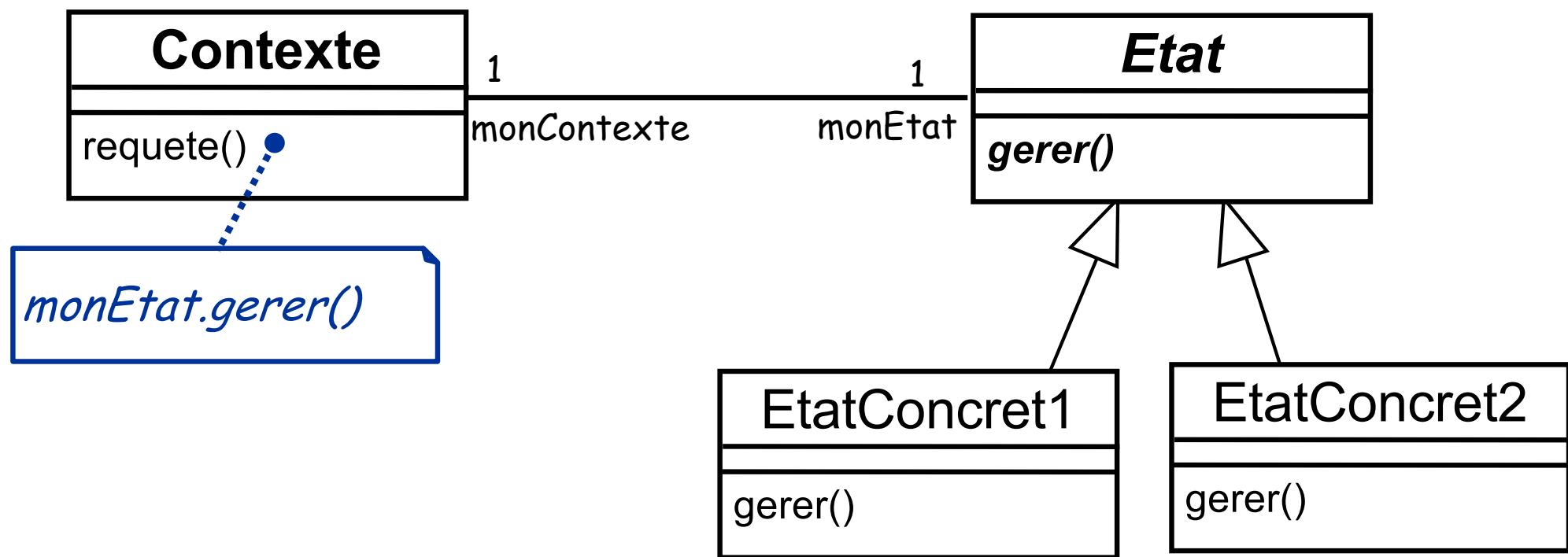
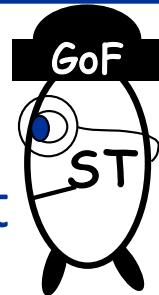


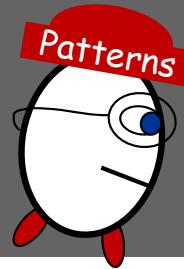


# State

## Solution (cas général)

Utiliser la délégation: attribuer la gestion de l'état à un objet spécifique (*Etat*) lié à l'objet par un lien d'association (agrégation)

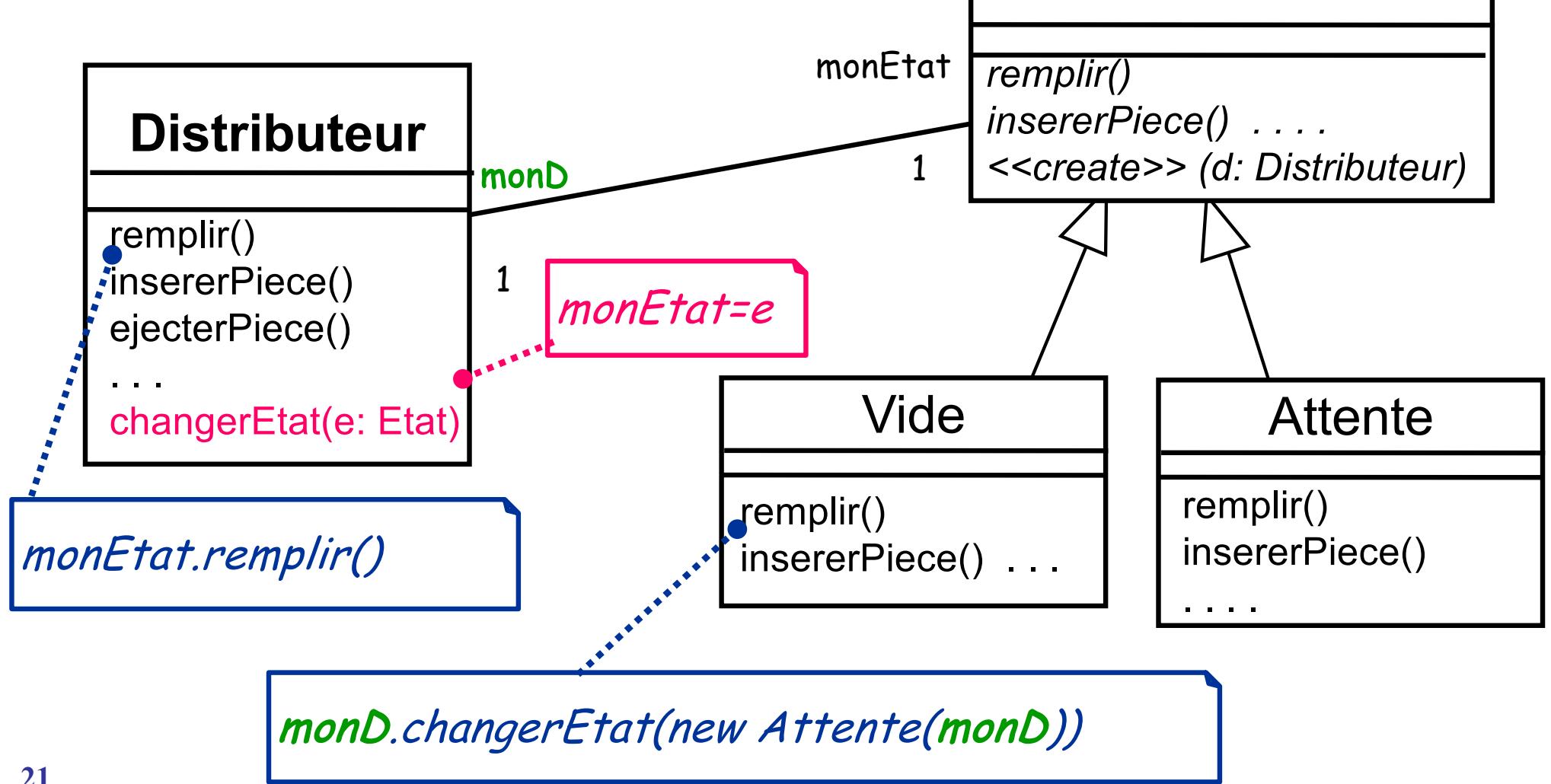


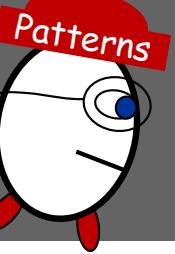


# State

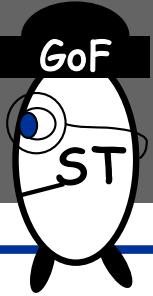


## Solution (exemple)

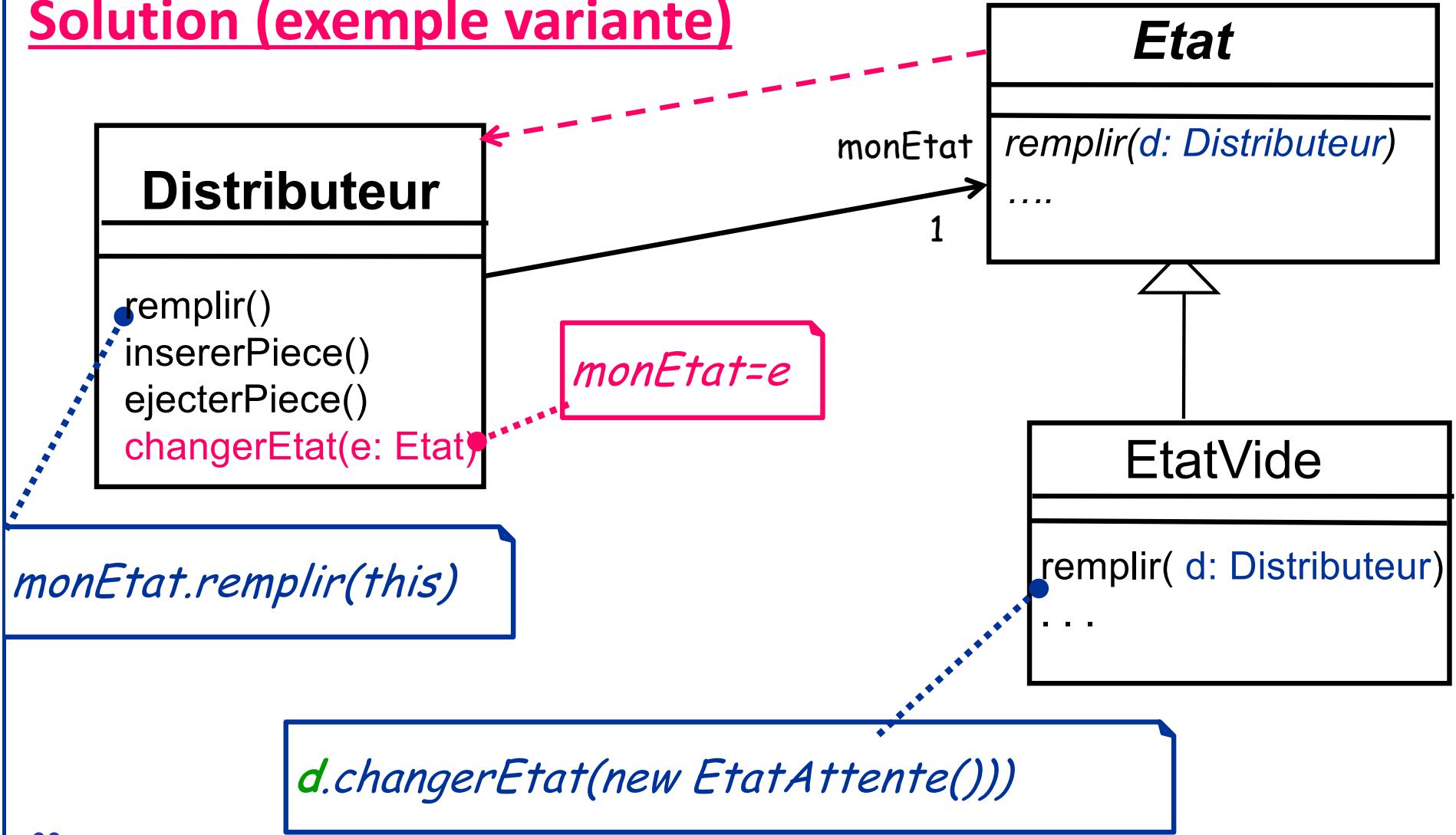


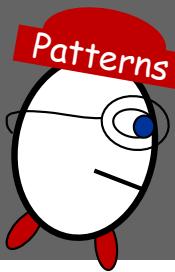


# State



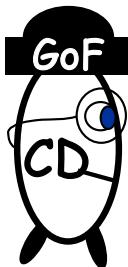
## Solution (exemple variante)



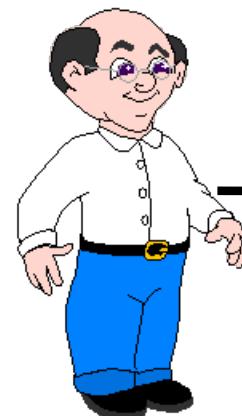


# Pattern COMMAND

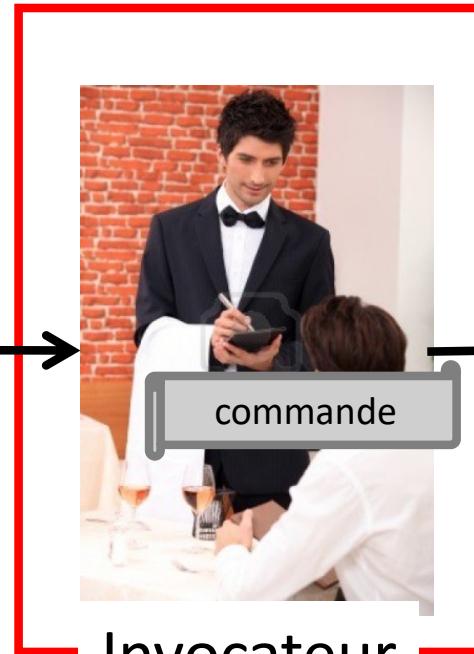
## Problème



Comment demander à un objet d'exécuter une action en permettant au client de définir le récepteur et la nature de l'action à exécuter?

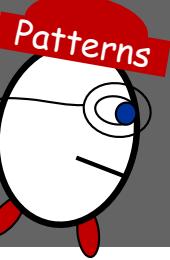


Client



Receveur

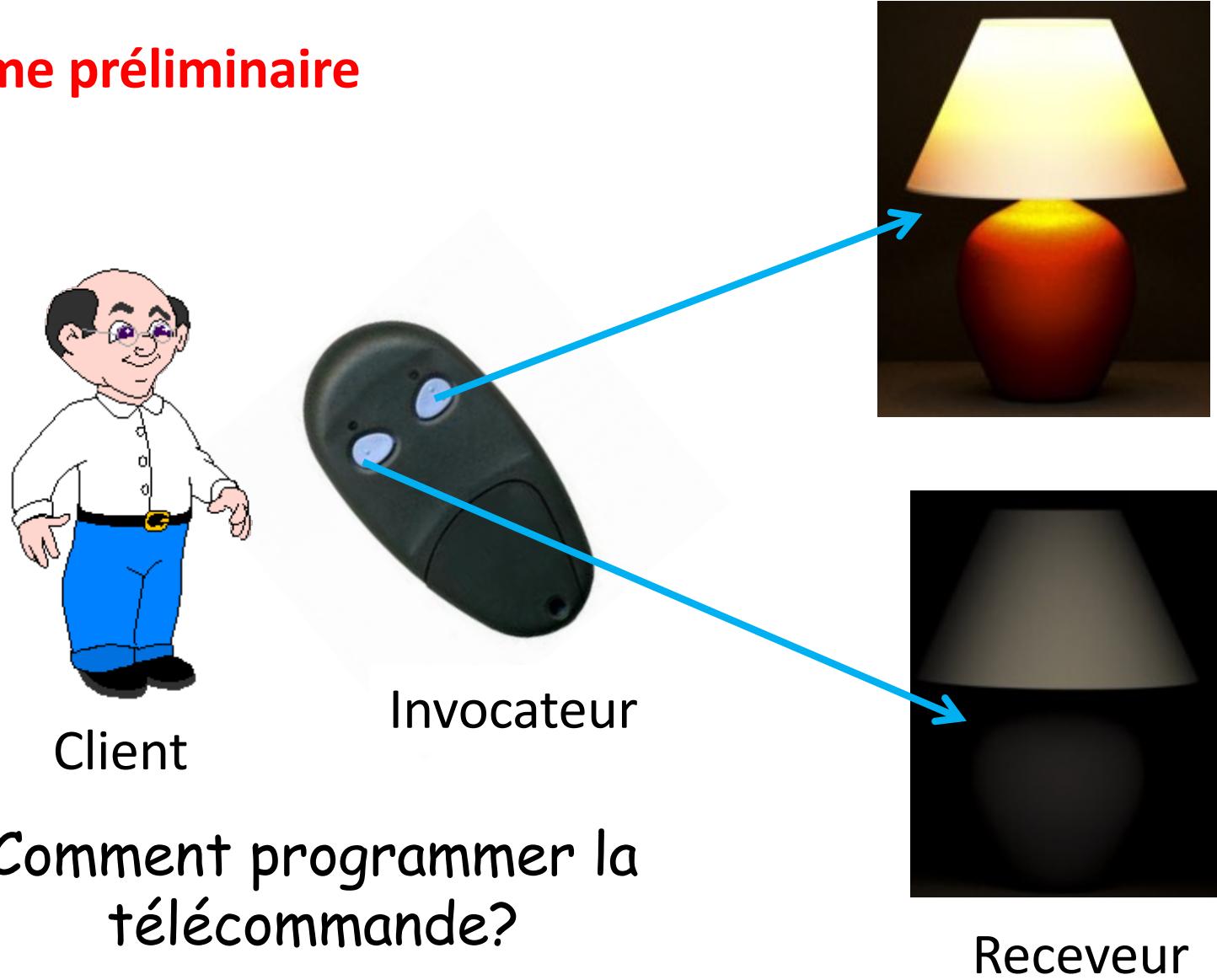


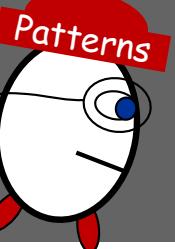


# Command

## Exemple: télécommande

Problème préliminaire

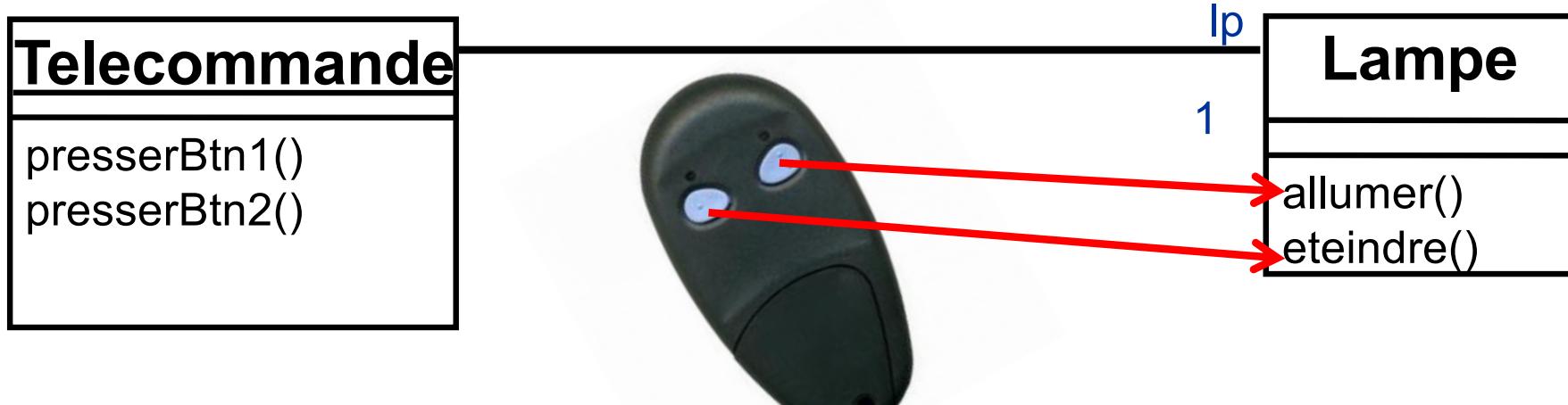




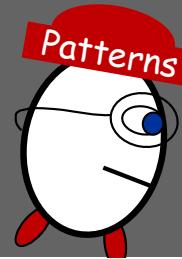
# Command

## Exemple: télécommande

### Solution initiale



```
public class TeleCommande {
    Lampe lp;
    public TeleCommande(Lampe lp) {
        this.lp=lp;}
    public void presserBtn1() {
        lp.allumer();}
    public void presserBtn2() {
        lp.eteindre();}
```

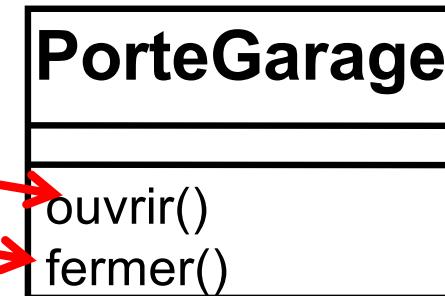


# Command

## Exemple: télécommande

### Evolution

Comment faire en sorte que la télécommande pilote à présent une porte de garage?

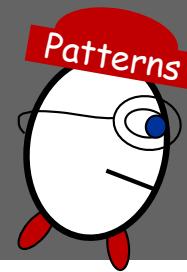


Seule solution:  
réécrire la  
classe



```

public class TeleCommande {
    PorteGarage g;
    public TeleCommande (PorteGarage g) {
        this.g=g;}
    public void presserBtn1 () {
        g.ouvrir(); }
    public void presserBtn2 () {
        g.fermer(); }
}
  
```

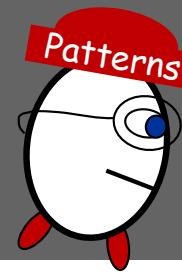


# Command

## Exemple: télécommande

→ **Problème à résoudre** : Comment concevoir la classe Télécommande afin de pouvoir modifier les actions de ses boutons sans modifier son code?



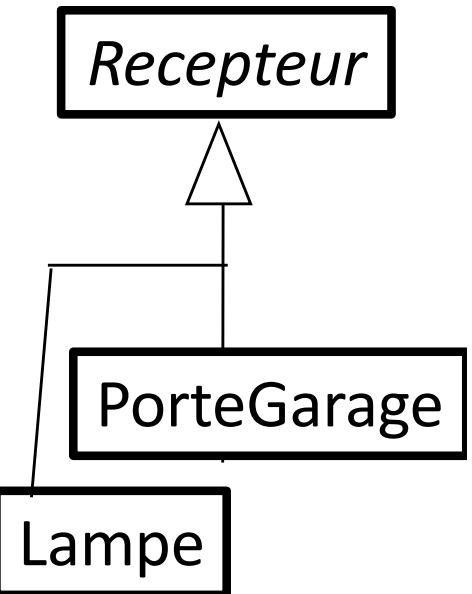


# Command

## Exemple: télécommande

→ **Proposition** : Utiliser une classe abstraite pour le récepteur des actions (Lampe ou PorteGarage)

```
public class TeleCommande {
    Recepteur r;
    public Telecommande(Recepteur r) {
        this.r=r;
    }
    public void presserBtn1() {
        r.???();
    }
    public void presserBtn2() {
        r.???();
    }
}
```



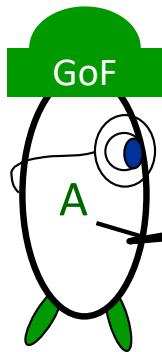
Quelles méthodes invoquer ?

- allumer/éteindre pour Lampe
- ouvrir/fermer pour PorteGarage



# Command

## Exemple: télécommande



Pattern Adapter

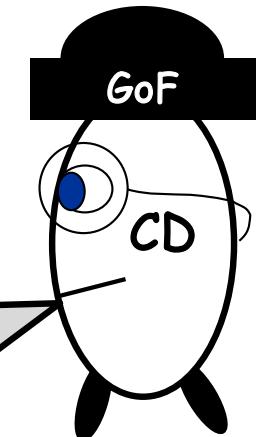
On pourrait créer des Adaptateurs pour que toutes les classes Recepteur implémentent la même interface

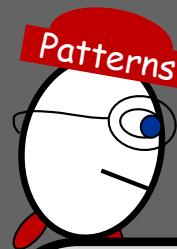


Oui mais cela risque d'être très très lourd!  
Et il ne pourra y avoir qu'un seul récepteur par télécommande.

J'ai une meilleure idée!!

Il faut créer des objets **Commandes** et les affecter aux boutons de la télécommande.



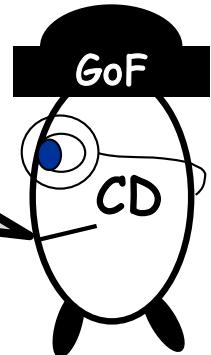


# Command

## Exemple: télécommande

Voilà mon idée!

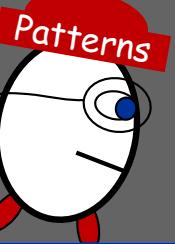
- On peut modifier les actions affectées aux boutons sans reprogrammer
- Les boutons peuvent même avoir des récepteurs différents!



```
public class TeleCommande {  
    Commande c1,c2;  
    public void setBtn1(Commande c) {  
        this.c1=c; }  
    public void setBtn2(Commande c) {  
        this.c2=c; }  
    public void presserBtn1() {  
        c1.executer(); }  
    public void presserBtn2() {  
        c2.executer(); }  
}
```

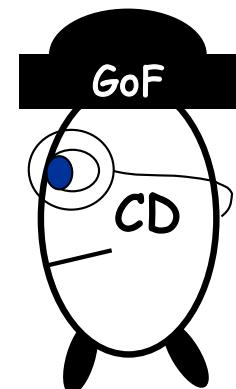
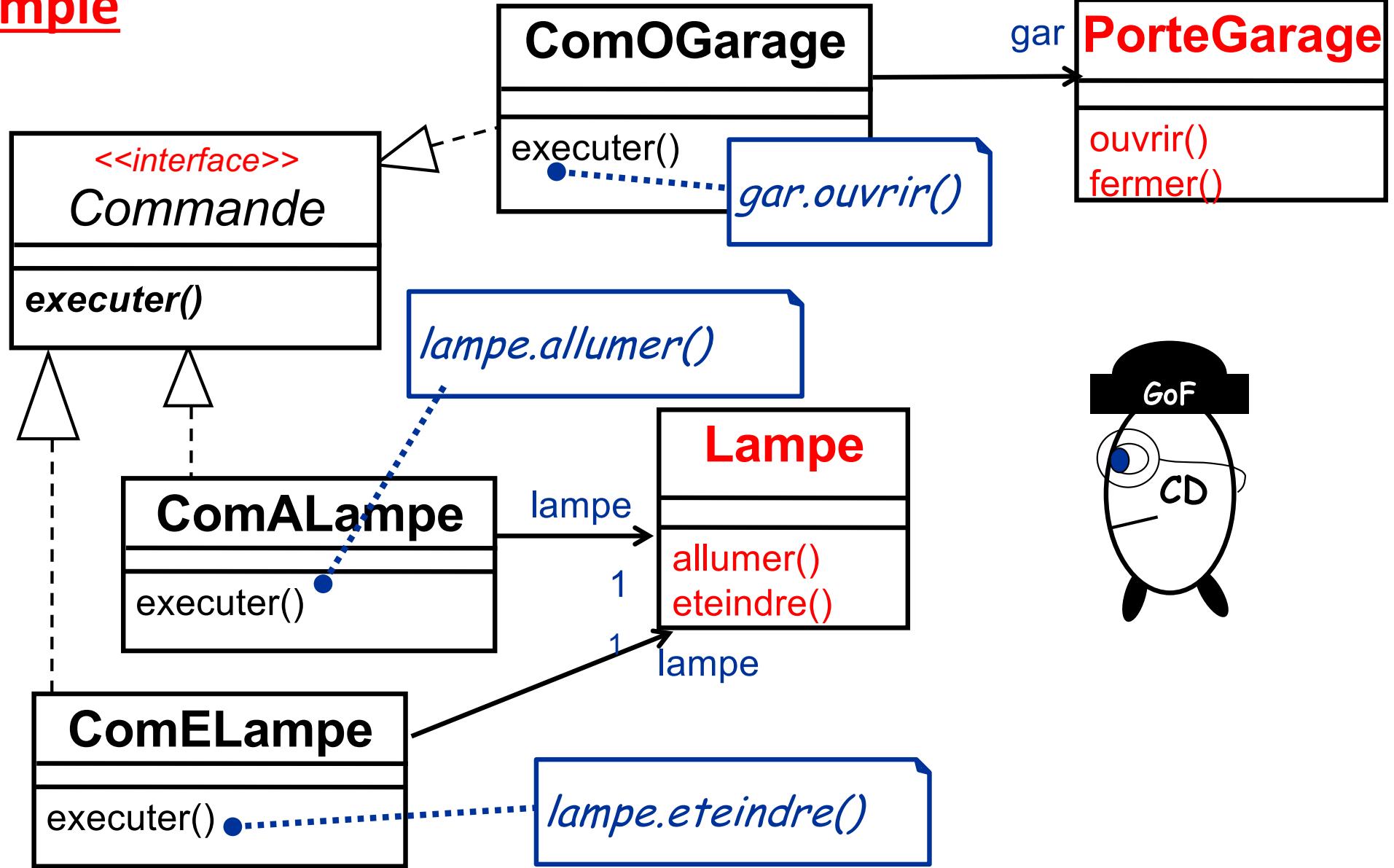
Oui mais bon, la classe  
Commande???  
Je ne vois pas.....

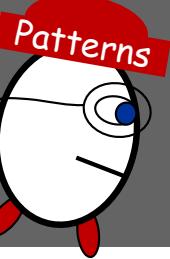




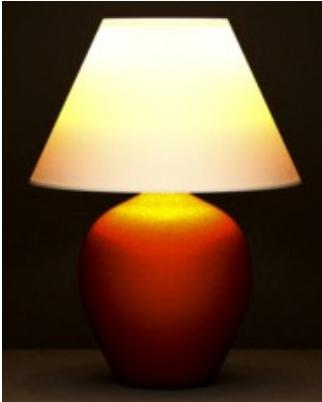
# Command

## Exemple



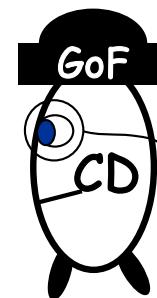
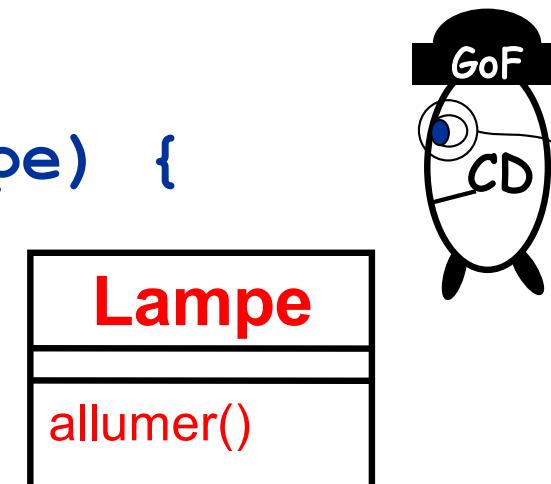


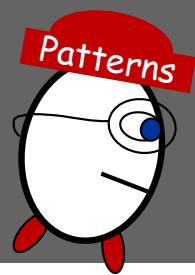
# Command



```
public interface Commande {  
    public void executer();  
}
```

```
public class ComALampe implements Commande  
{  
    Lampe lampe;  
    public ComALampe(Lampe lampe) {  
        this.lampe = lampe;  
    }  
    public void executer() {  
        lampe.allumer();  
    }  
}
```



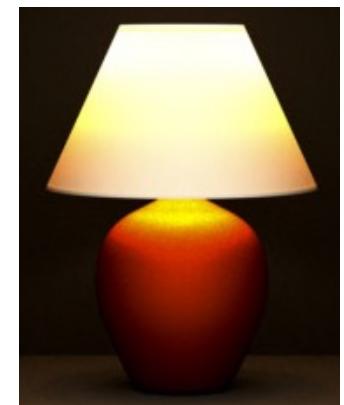


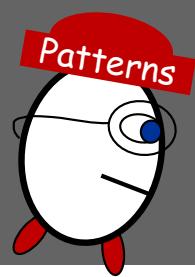
# Command



## Programme de test (Client)

```
public class TestTeleCom{  
    public void test() {  
        //Invocateur  
        TeleCommande telecom=new TeleCommande();  
        //Recepteur  
        Lampe l = new Lampe();  
        //Commandes  
        Commande lal= new ComALampe(l);  
        telecom.setBtn1 (lal);  
        telecom.presserBtn1();  
  
        Commande let= new ComELampe(l);  
        telecom.setBtn2 (let);
```





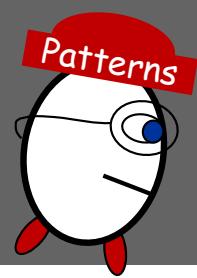
# Command



Et pour piloter une porte de garage !

```
public class TestTeleCom{  
    public void test() {  
        ...//suite //Receveur  
        PorteGarage g = new PorteGarage();  
        //Commandes  
        Commande log= new ComOGarage(g);  
        telecom.setBtn1 (log);  
        telecom.presserBtn1();  
  
        Commande lfg= new ComFGarage(g);  
        telecom.setBtn2 (lfg);  
    }  
}
```





# Command

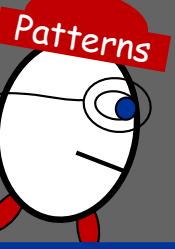
Et pourquoi pas un bouton qui allume la lampe et le deuxième qui ouvre la porte du garage!



```
public class TestTeleCom{
    public void test() {
        ...//suite      //Commandes
        telecom.setbtn1 (lal);
        telecom.presserBtn1 ();

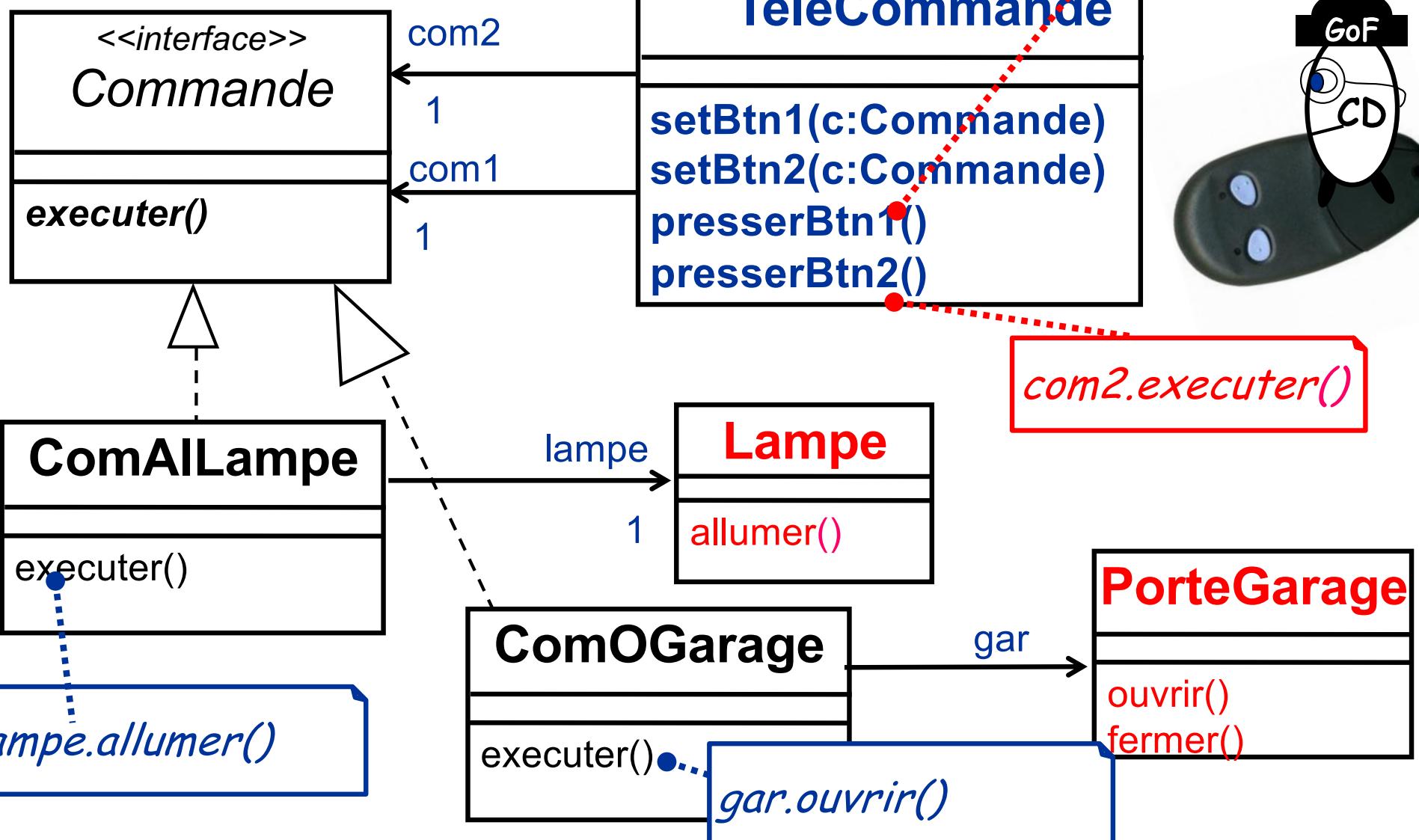
        telecom.setbtn2 (log);
        telecom.presserBtn2 ();
    }
}
```

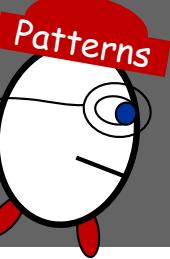




# Command

## Diagramme final

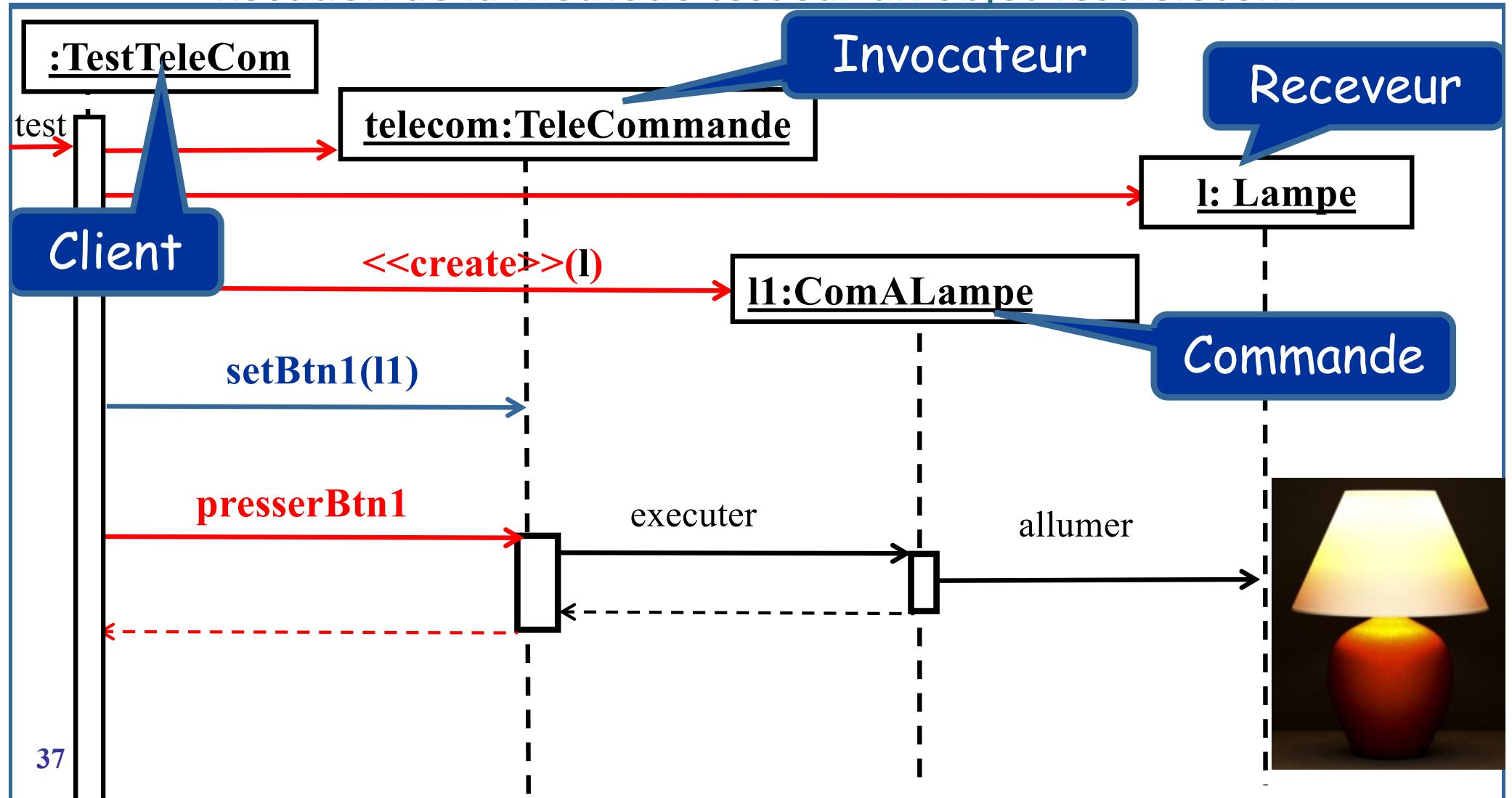


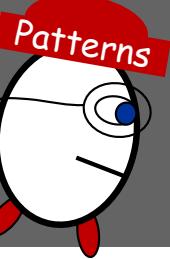


# Command

## Diagramme de séquence

Exécution de la méthode test sur un objet TestTelecom

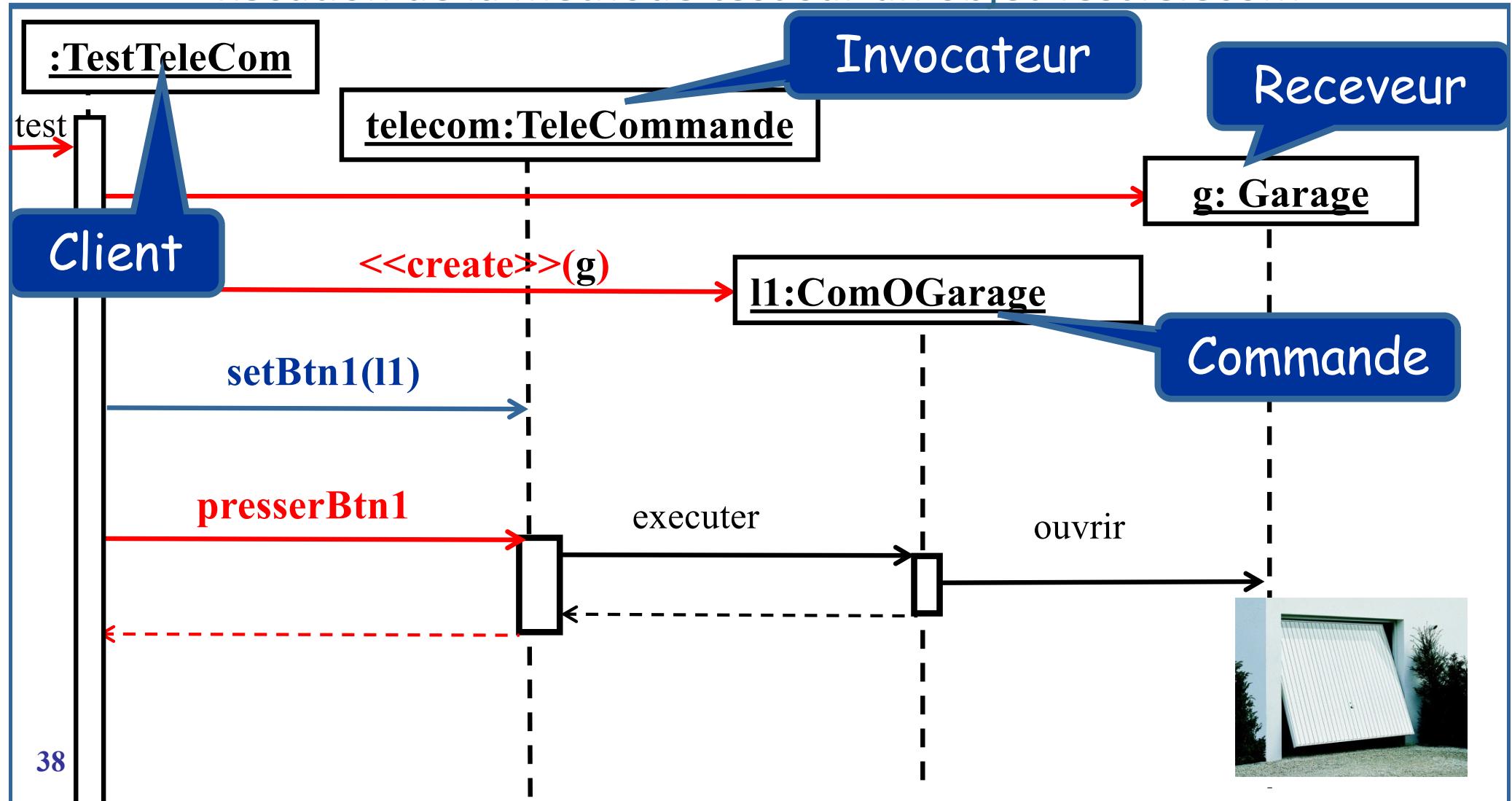


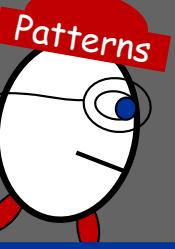


# Command

## Diagramme de séquence (suite)

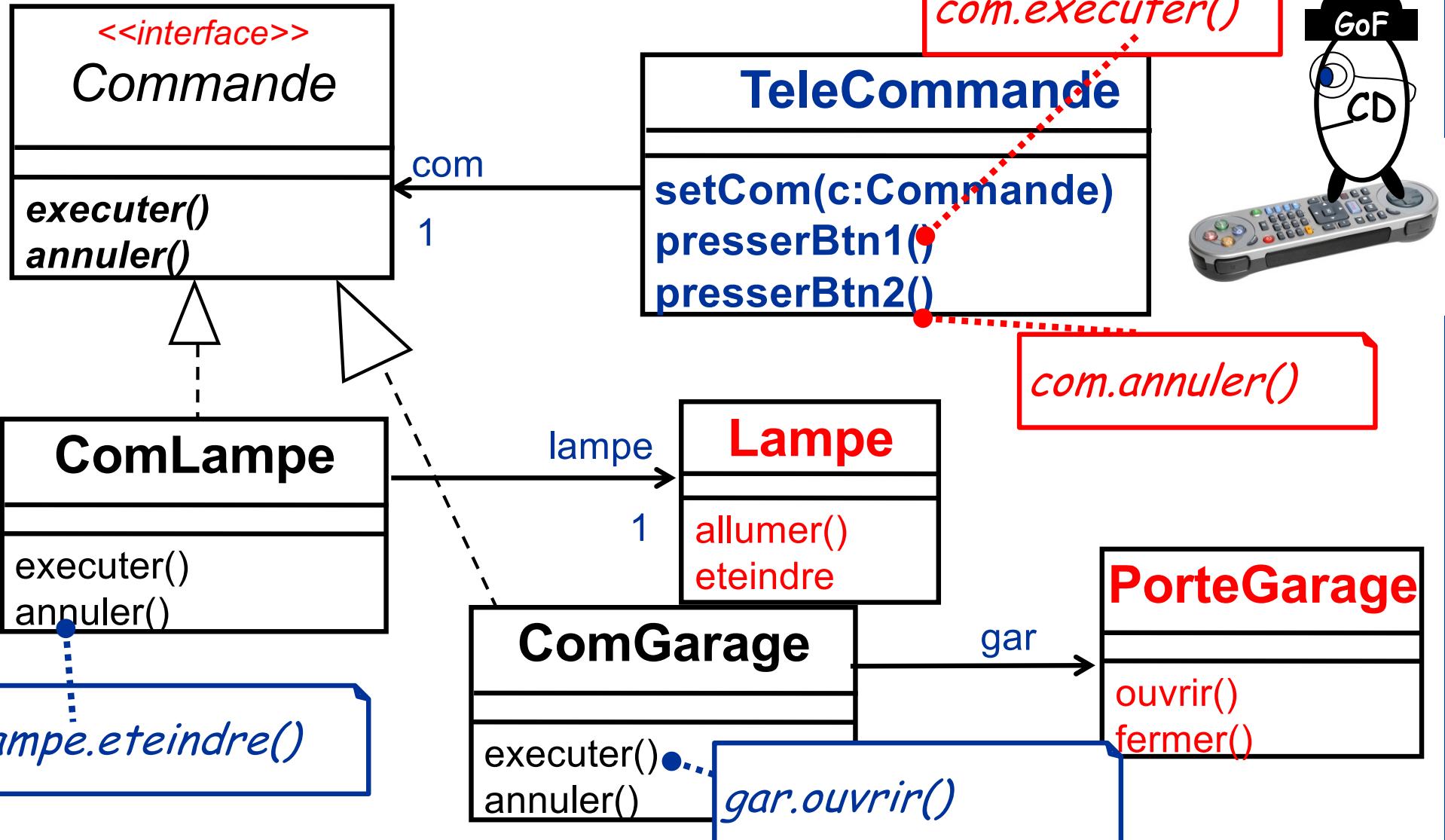
Exécution de la méthode test sur un objet TestTelecom

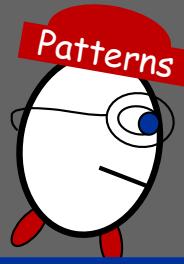




# Command

Une version améliorée ..

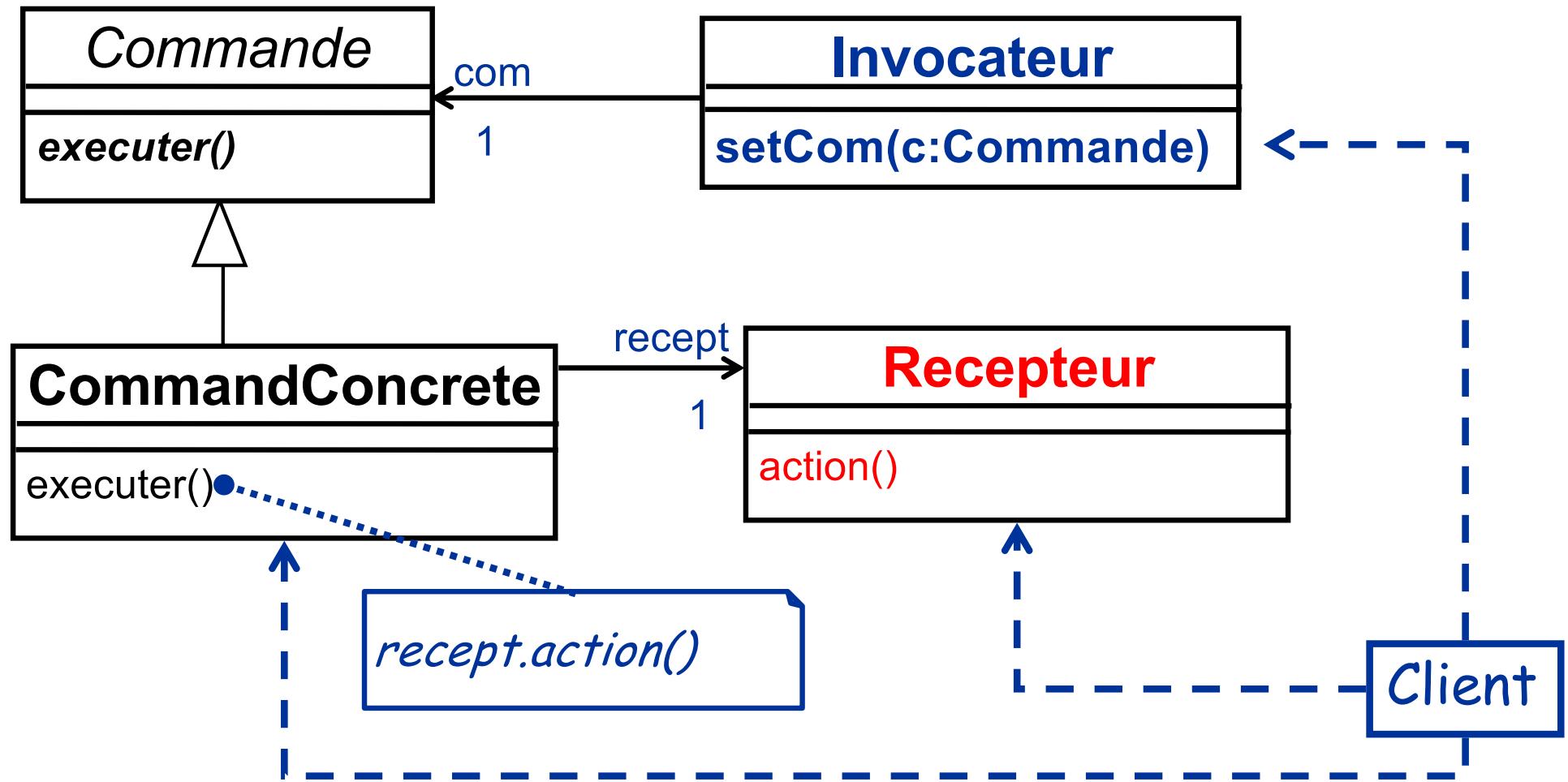


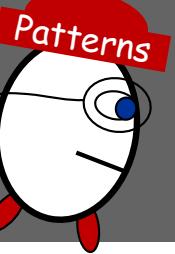


# Command

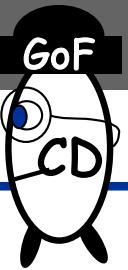


## Solution (cas général)



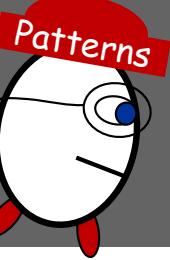


# Command



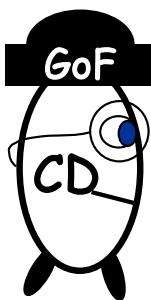
## Solution

- Définir une classe **Commande abstraite** encapsulant la requête et l'associer à une classe **Invocateur** invoquant la requête.
  
- Définir une classe **Commande concrete** et lui associer une classe **Récepteur** exécutant l'action correspondant à la requête.



# Command

## Conséquences



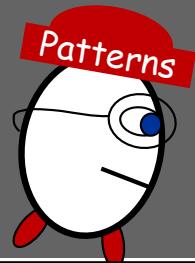
- CD **découple** l'objet invoquant une opération de celui qui sait comment la réaliser.
- Les Commandes sont des objets dits de «1ère classe ». Ils peuvent être manipulés et étendus comme n'importe quel objet.
- Il est facile d'ajouter de nouvelles commandes car il n'est pas nécessaire de modifier les classes existantes.



# Patterns Comportementaux

## Résumé

<b>STRATEGY</b>	Encapsule des comportements interchangeables et utilise la délégation pour décider lequel utiliser
<b>OBSERVER</b>	Permet de notifier des changements d'état à des objets
<b>STATE</b>	Encapsule des comportements basés sur des états et utilise la délégation pour permuter ces comportements
<b>COMMAND</b>	Encapsule une requête sous la forme d'un objet



# FREQUENCES D'UTILISATION DES PATTERNS GOF

Patterns	Fréquence d'utilisation (/5)	Patterns	Fréquence d'utilisation (/5)
Abstract Factory	5	Bridge	3
Facade	5	Decorator	3
Factory method	5	Prototype	3
Iterator	5	State	3
Observer	5	Template Method	3
Adapter	4	Builder	2
Command	4	Chain of Responsability	2
Composite	4	Mediator	2
Proxy	4	Flyweight	1
Singleton	4	Interpreter	1
Strategy	4	Memento	1
		Visitor	1