



PATTERN **OBSERVATEUR**

Par : Bastide Rémi, Billaud Nathan, Bardet--Techer Jordan,
Chevaldonnet Hubert

SOMMAIRE :

1. Introduction
2. Le besoin et première modélisation
3. Première Solution avec le Pattern Observateur
4. Mécanisme du pattern Observateur
5. L'intention du Pattern Observateur
6. Diagramme de classe et séquence
7. Principes SOLID
8. Limite du pattern Observateur
9. Lien avec l'architecture MVC
10. Deuxième exemple
11. Live Coding
12. Bibliographie
13. QCM (Kahoot)

INTRODUCTION :

Créationnels

Structurels

Comportementaux

Pattern Observer

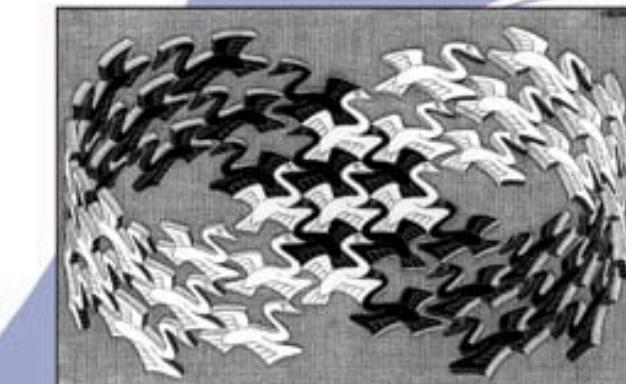
But :

- Relation "un-à-plusieurs" entre objets.

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



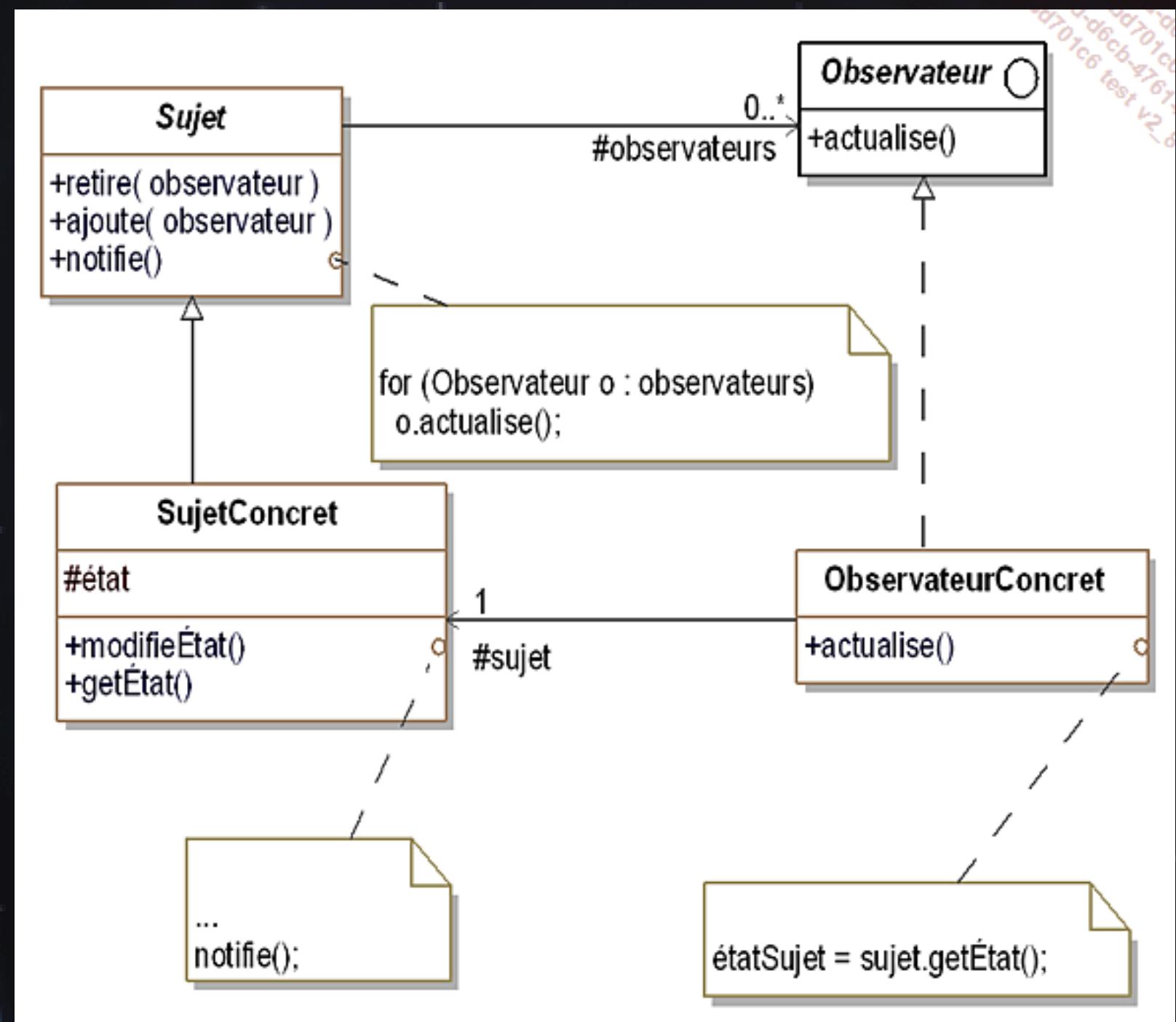
Cover art © 1994 M.C. Escher / Cordon Art - Baam - Holland. All rights reserved.

Foreword by Grady Booch



Le besoin

Le pattern Observer a pour objectif de construire une dépendance entre un sujet et des observateurs de sorte que chaque modification du sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état.



Modélisation sans pattern

Un client nous demande d'implémenter un système de comptabilité de cancanement

Ajout de la classe ComptableDeCancanement

```
public class ComptableDeCancans {  
  
    private int compteurCancan = 0;  
    private int compteurCoincoin = 0;  
  
    public void incrementerCancan() {  
        compteurCancan++;  
        System.out.println("Total de 'Can-can': " + compteurCancan);  
    }  
  
    public void incrementerCoincoin() {  
        compteurCoincoin++;  
        System.out.println(" Total de 'Coin-Coin': " + compteurCoincoin);  
    }  
}
```

Modélisation sans pattern

Un client nous demande d'implémenter un système de comptabilité de cancanement
Modification de la classe Canard

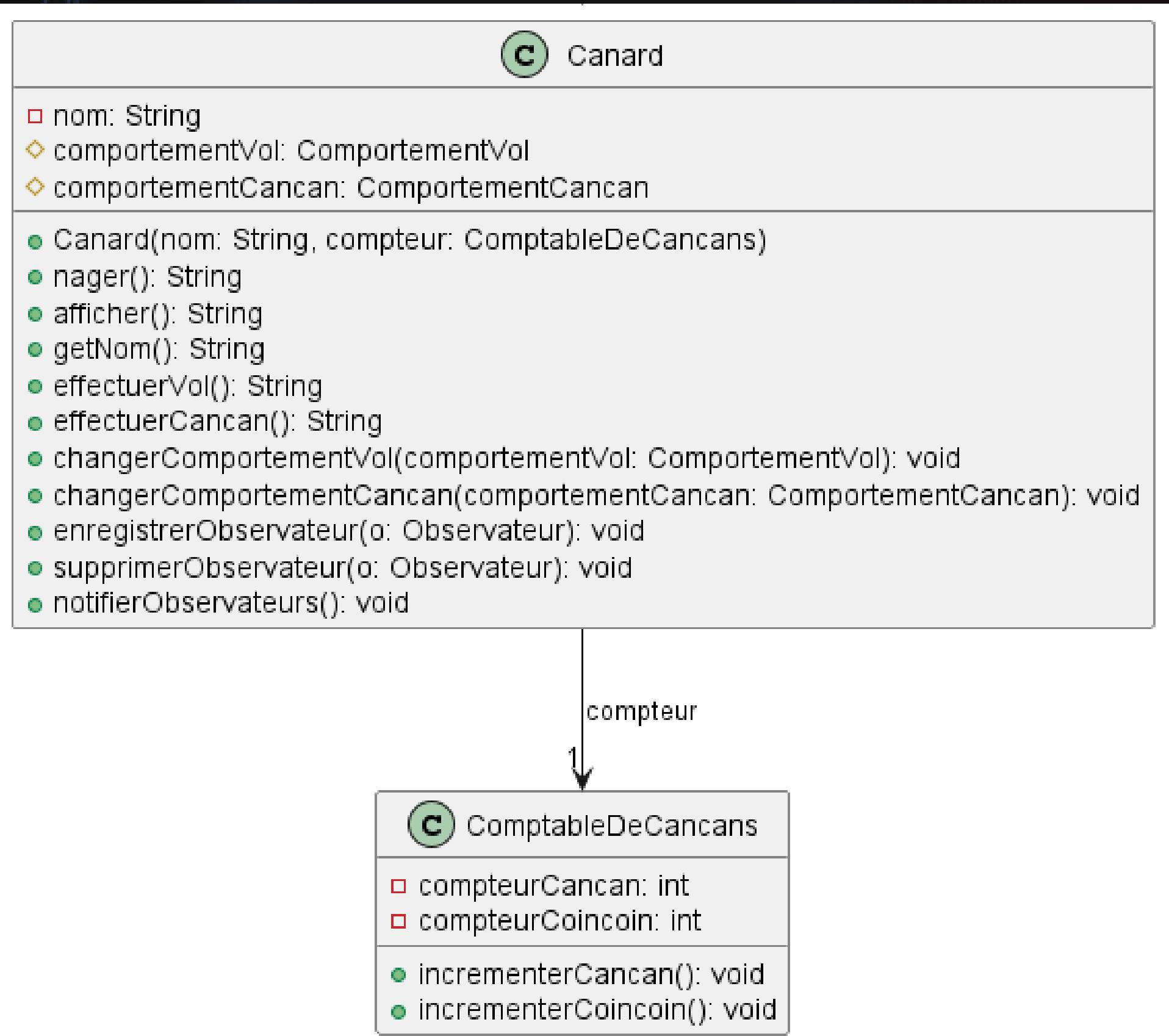
```
private ComptableDeCancans compteur;

public Canard(String nom,ComptableDeCancans compteur) {
    this.nom = nom;
    this.compteur=compteur;
}

public String effectuerCancan() {
    String cancanement = comportementCancan.cancaner();

    if (this.compteur != null) {
        if (comportementCancan instanceof Cancan) {
            compteur.incrementerCancan();
        } else if (comportementCancan instanceof Coincoin) {
            compteur.incrementerCoincoin();
        }
    }
    return cancanement;
}
```

Modélisation sans pattern



Modélisation avec pattern

Etape 1 : Créer les rôles d'observateurs et de sujets.

```
public interface Sujet {  
    void enregistrerObservateur(Observateur o);  
    void supprimerObservateur(Observateur o);  
    void notifierObservateurs();  
}
```

```
public interface Observateur {  
    void actualiser(ComportementCancan comportement);  
}
```

Modélisation avec pattern

Etape 2 : Implémentation de l'interface

```
private final List<Observateur> observateurs = new ArrayList<>();
```

```
@Override  
public void enregistrerObservateur(Observateur o) {  
    observateurs.add(o);  
}  
  
@Override  
public void supprimerObservateur(Observateur o) {  
    observateurs.remove(o);  
}  
  
@Override  
public void notifierObservateurs() {  
    for (Observateur observateur : observateurs) {  
        observateur.actualiser(comportementCancan);  
    }  
}
```

```
public String effectuerCancan() {  
    notifierObservateurs();  
    return comportementCancan.cancaner();  
}
```

Modélisation avec pattern

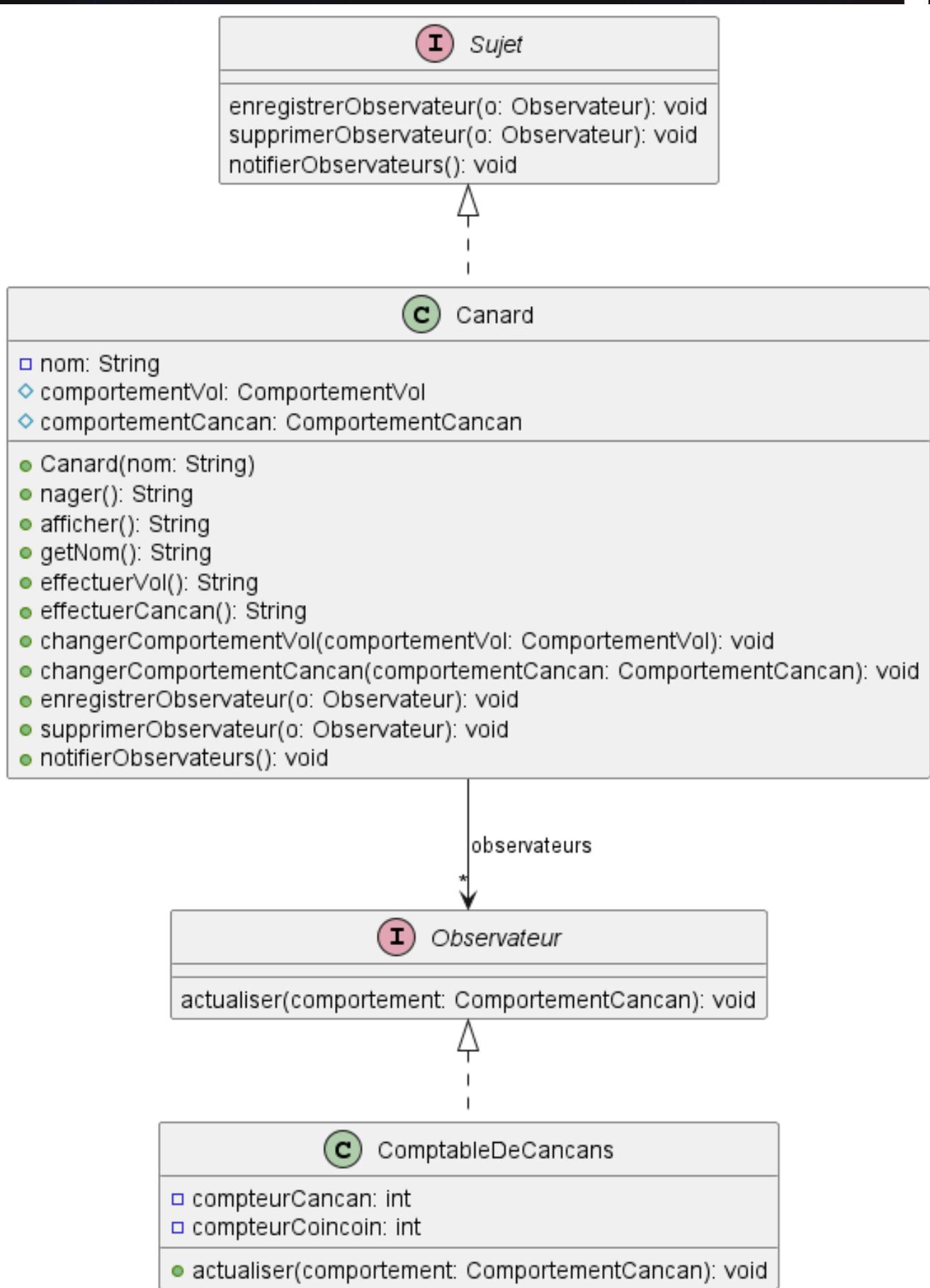
Etape 3 : ComptableDeCancans devient un observateur

```
public class ComptableDeCancans implements Observateur{
    private int compteurCancan = 0;
    private int compteurCoincoin = 0;

    @Override
    public void actualiser(ComportementCancan comportement) {

        if (comportement instanceof Cancan) {
            compteurCancan++;
        } else if (comportement instanceof Coincoin) {
            compteurCoincoin++;
        }
    }
}
```

Modélisation avec pattern



LES MÉCANISMES PUSH / PULL

Sujet Notifie ses Observateurs

Sujet Signale une mise à jour

Sujet Fournit les informations

Observateur Demande des infos

Observateur Les prend en compte

Sujet Envoie les infos

LES MÉCANISMES PUSH / PULL

Plus simple pour les observateurs

Sujet est générique

Moins de couplage temporaire

Augmenter le nombre d'Observateurs

Sujet obligé de connaître ses modifs

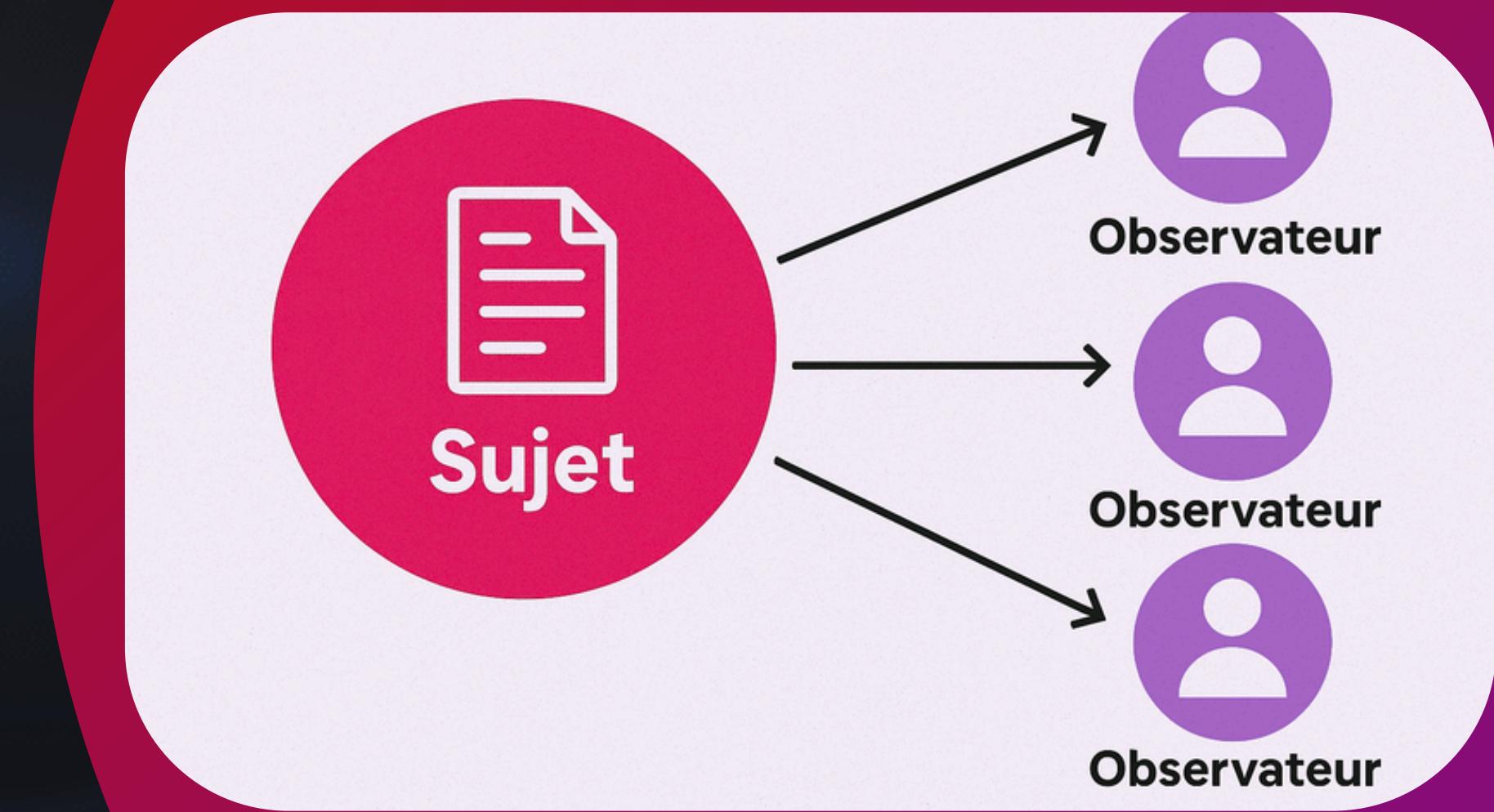
Couplage directionnel

Données transférées inutilement

Coûteux En terme de complexité

INTENTIONS DU PATTERN OBSERVATEUR

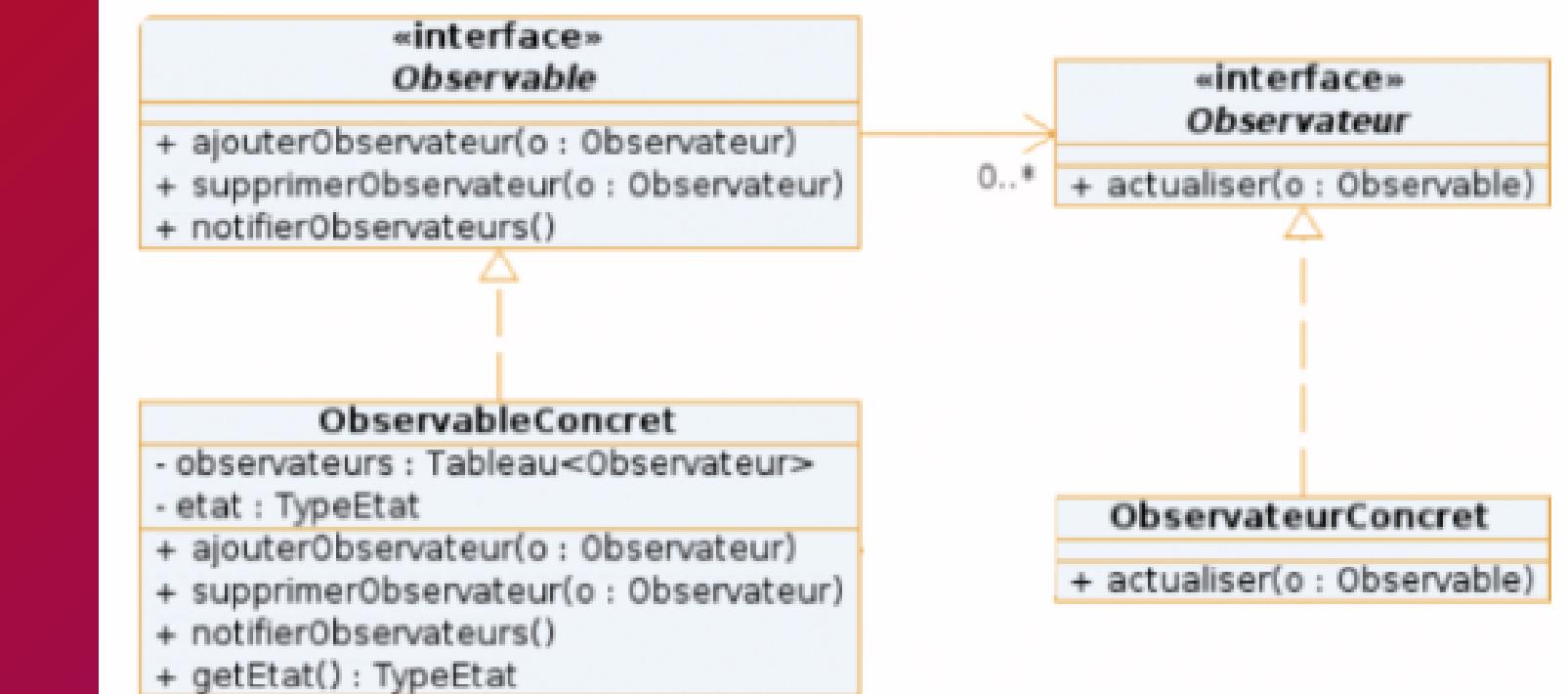
- Répondre à un besoin
- Définir une dépendance "un à plusieurs" entre des objets
- Conserver un couplage faible



DIAGRAMMES GÉNÉRIQUES DU PATERN OBSERVATEUR

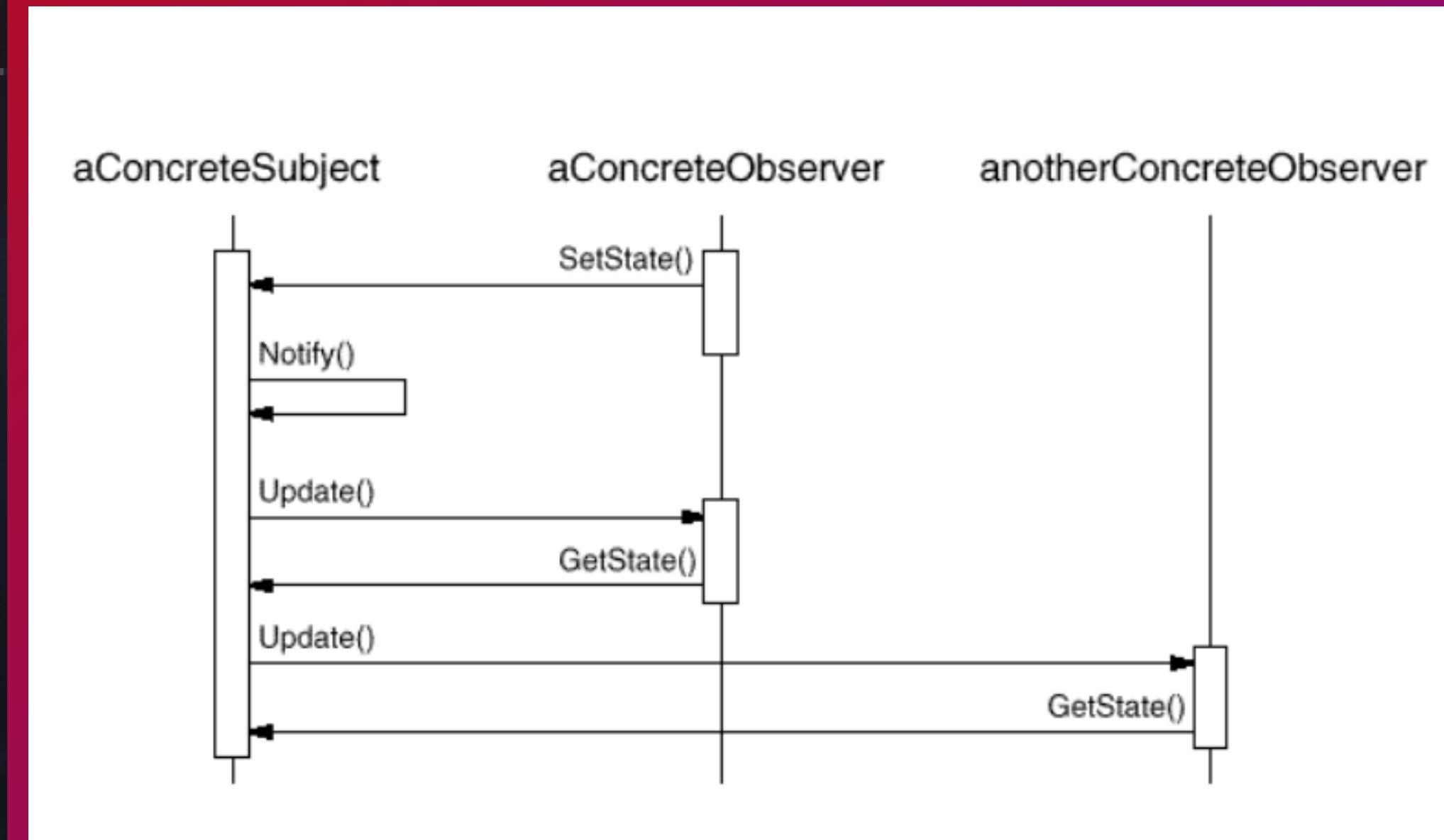
Pattern Comportemental
du Gof (Gang of Four)

Diagramme UML



DIAGRAMMES GÉNÉRIQUES DU PATERN OBSERVATEUR

Pattern Comportemental
du Gof (Gang of Four)



Lien avec Solid

Trois principes respectés :

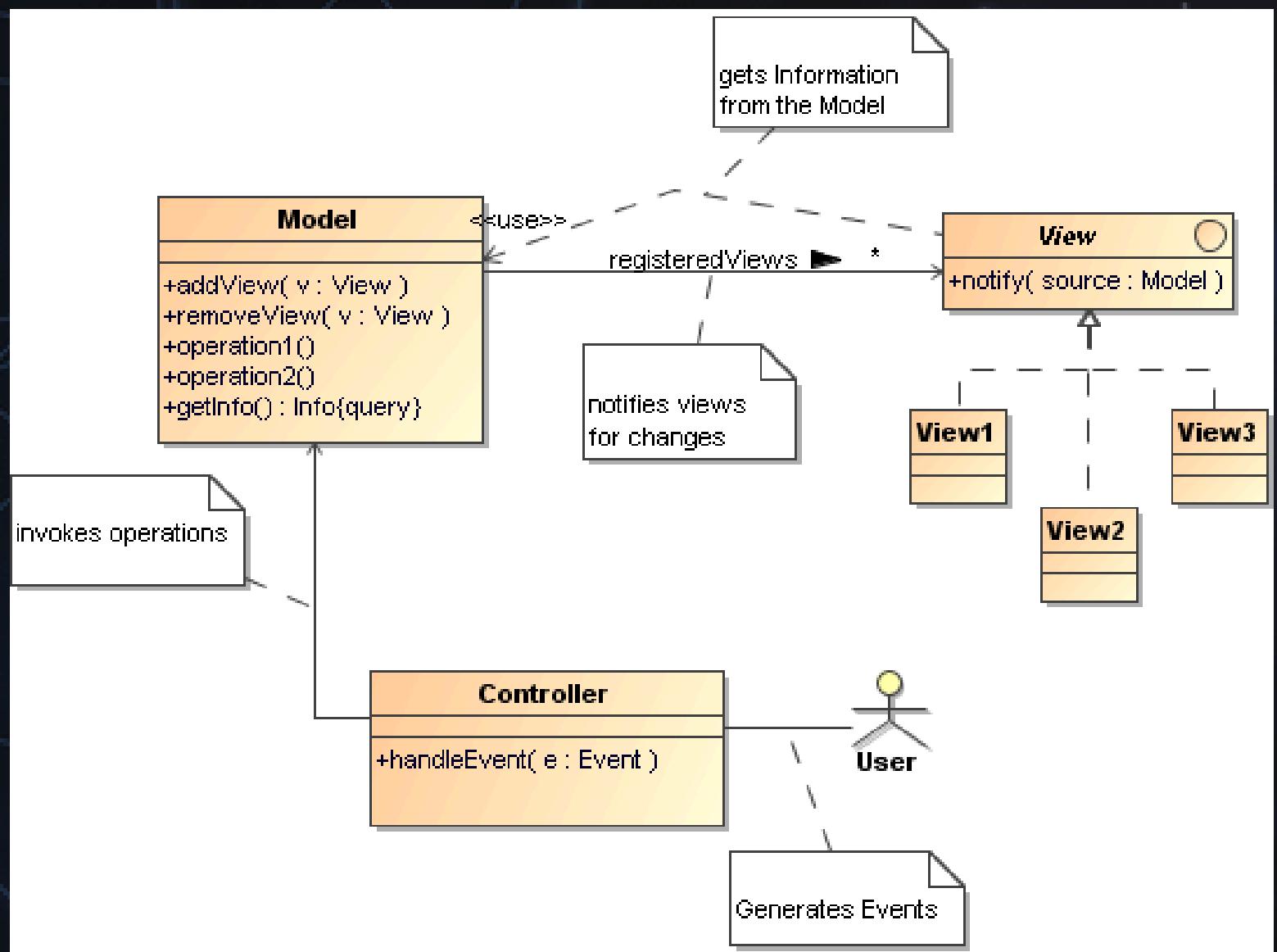
- SRP
- OCP
- ICP

Les limites du pattern Observer

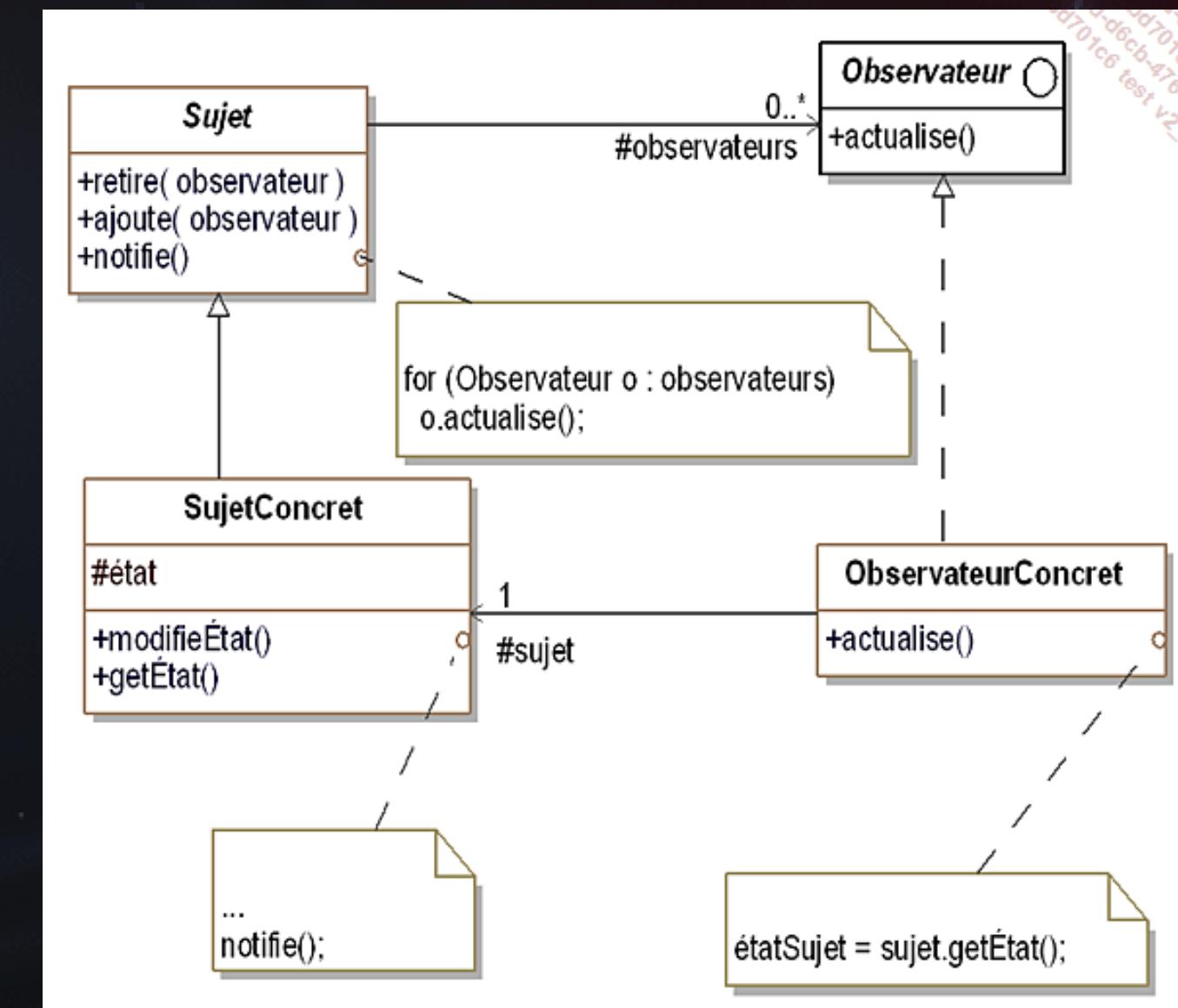
- Le nombre de notifications peut être très important
- Le débogage est difficile
- Risque de fuite de mémoire

Comparaison avec le pattern MVC

Pattern MVC:



Pattern Observer:



Implémentation Client - Magasin

Le magasin :

```
public interface Sujet {  
    void enregistrer(Observateur o);  
    void supprimer(Observateur o);  
    void notifierObservateurs();  
}
```

Les clients :

```
public interface Observateur {  
    void actualiser();  
}
```

Implémentation Client - Magasin

Le magasin :

```
public class Magasin implements Sujet {
    private List<Observateur> clients = new ArrayList<>();
    private boolean produitDisponible;

    public void enregistrer(Observateur o) {
        clients.add(o);
    }

    public void supprimer(Observateur o) {
        clients.remove(o);
    }

    public void notifierObservateurs() {
        for (Observateur o : clients) {
            o.actualiser();
        }
    }

    public void setProduitDisponible(boolean dispo) {
        this.produitDisponible = dispo;
        if (this.produitDisponible) {
            notifierObservateurs();
        }
    }
}
```

Implémentation Client - Magasin

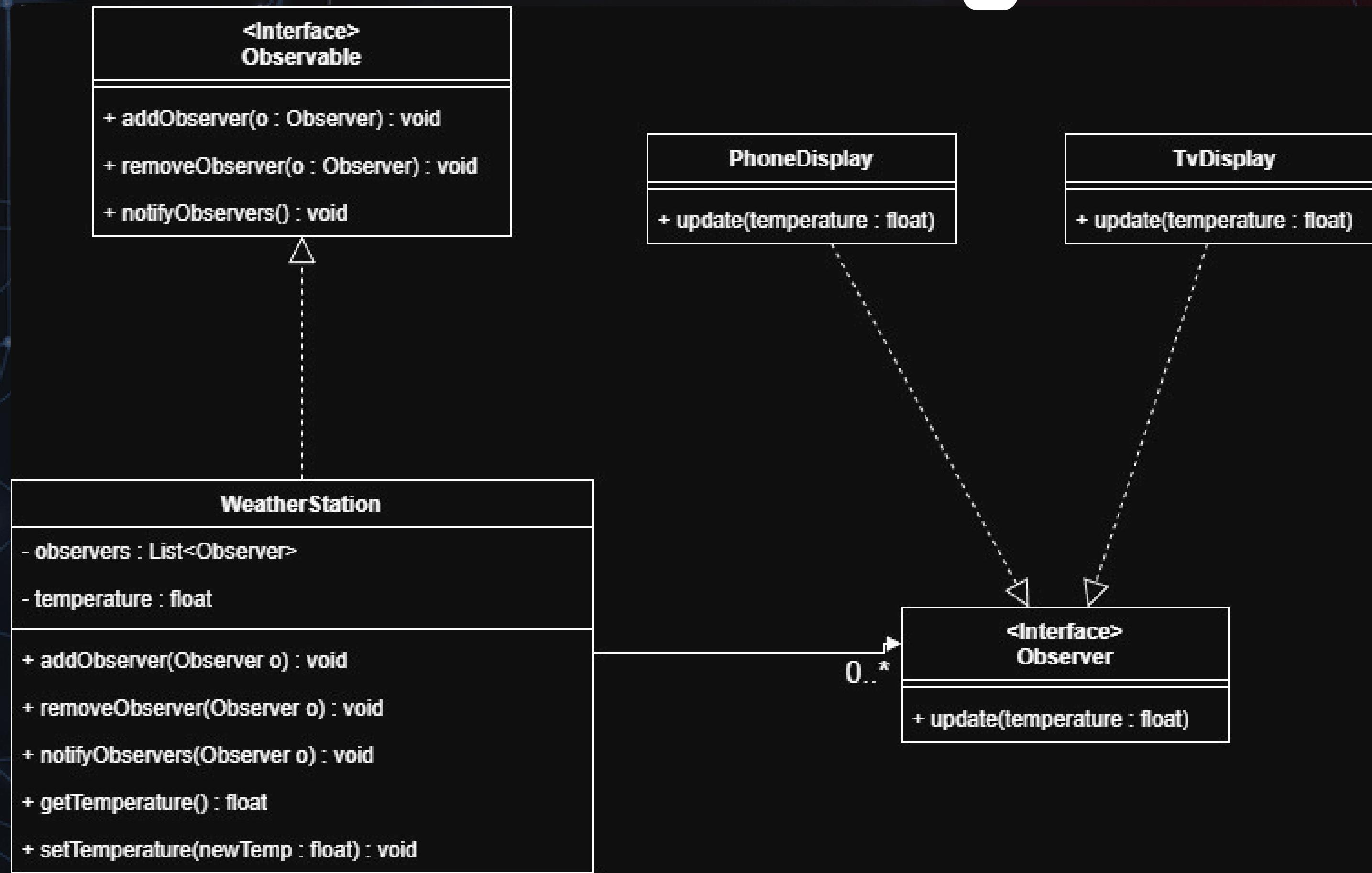
Le client:

```
public class Client implements Observateur {
    private String nom;

    public Client(String nom) {
        this.nom = nom;
    }

    @Override
    public void actualiser() {
        System.out.println("Client " + nom + " : Le produit est de nouveau disponible !");
    }
}
```

Live - Coding



Live - Coding



[Watch video on YouTube](#)

Error 153

Video player configuration error



Live - Coding

rétro-conception

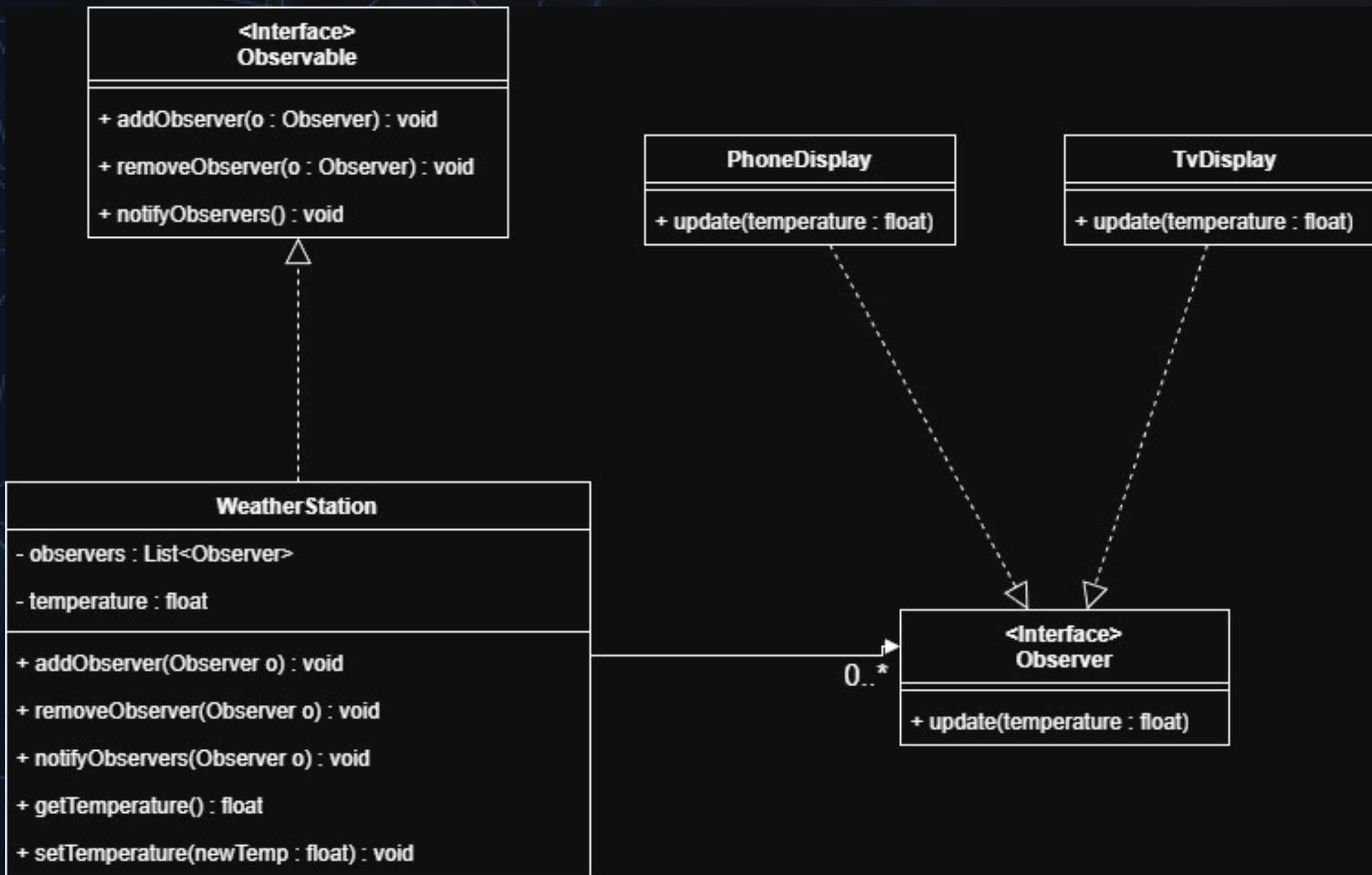
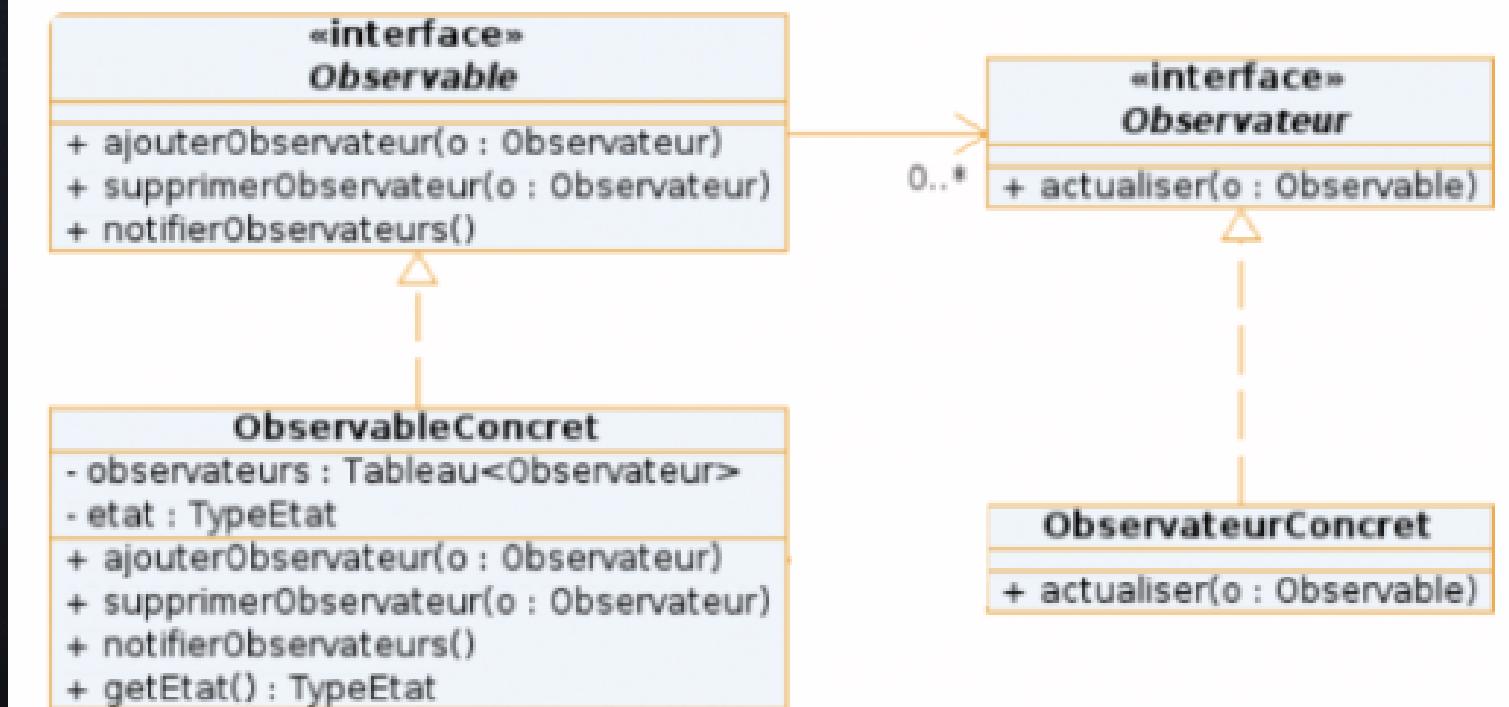


Diagramme UML



Bibliographie

- refactoring.guru
- coderanch.com
- designpattern.fr
- editions-eni.fr

QCM

Kahoot!

Create interactive quizzes, polls, presentations, and more to engage your audience.

 kahoot.it

MERCI !

Par : Bastide Rémi, Billaud Nathan, Bardet--Techer Jordan,
Chevaldonnet Hubert