# Cours M3105 : Conception et programmation objet avancées
# Le patron de conception Observateur

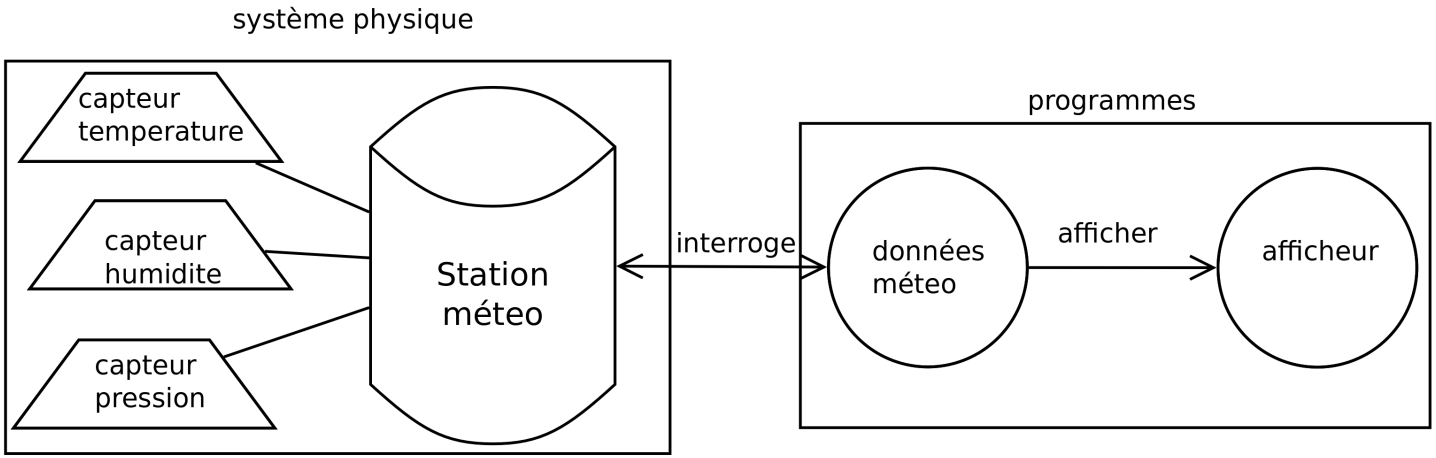
Références : cours de D. Bouthinon, livre *Design patterns — Tête la première*,
E. & E. Freeman, ed. O'Reilly

IUT Villetaneuse

2019-2020

# Plan

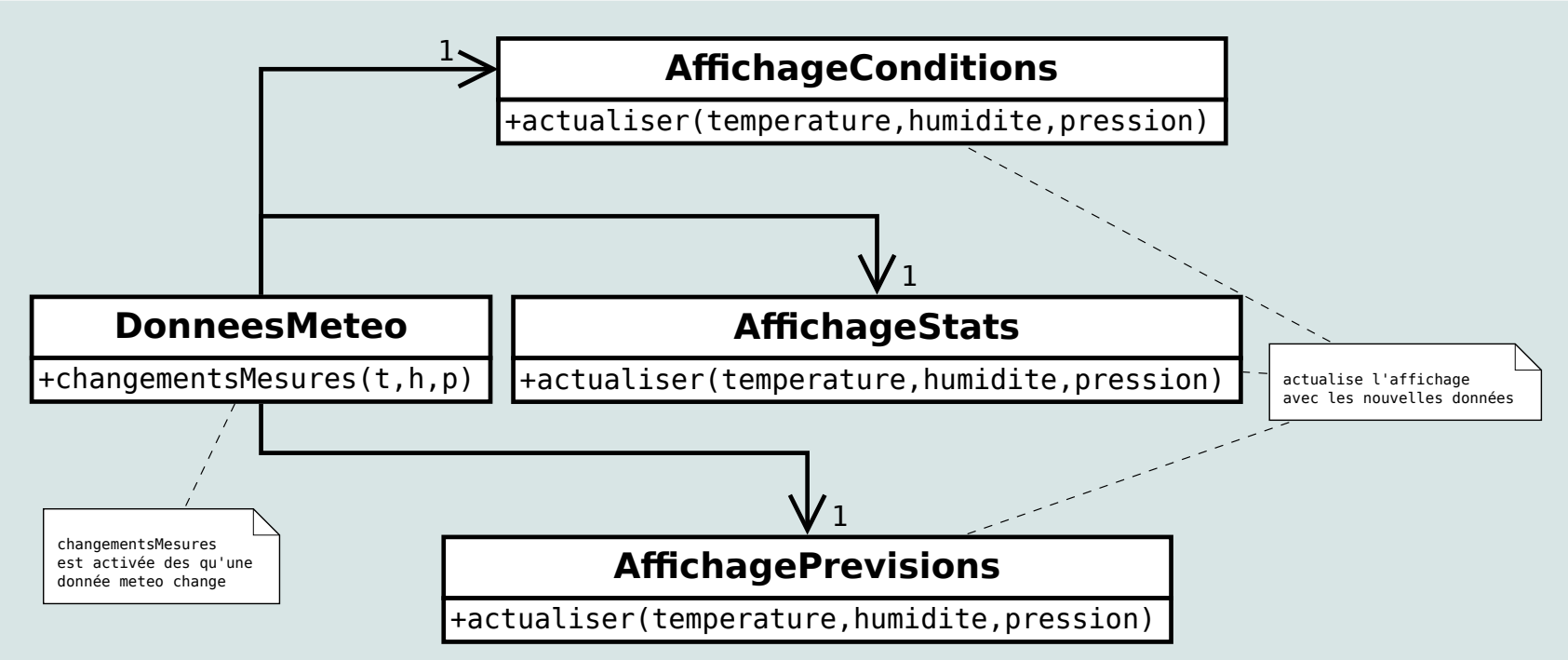
# Répercuter les changements d'un objet aux objets qui en dépendent




L'objet *DonneesMeteo* communique avec la station pour obtenir la température, pression et humidité qu'il communique à l'afficheur.

Il y a trois options d'affichage : conditions actuelles, statistiques et prévisions.

# DonneesMeteo (UML)




Problème : il faudra modifier la classe *DonneesMeteo* dès qu'on ajoute ou retire un affichage.

# DonneesMeteo (java)

```
public class DonneesMeteo
{
    private AffichageConditions ac;
    private AffichageStats as;
    private AffichagePrevisions ap;

    ...
    public void changementMesures(float t, float h, float p)
    {
        ac.actualiser(t, h, p);
        as.actualiser(t, h, p);
        ap.actualiser(t, h, p);
    }
}
```

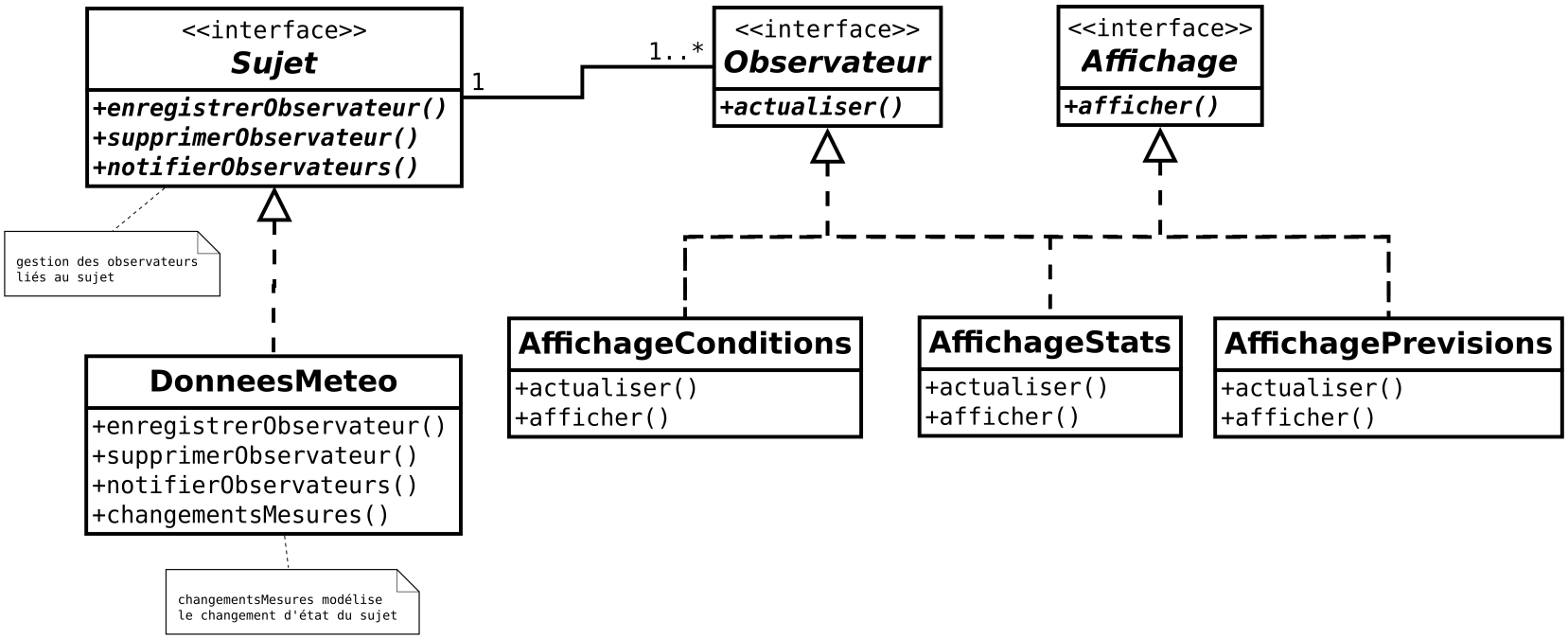
associations avec les afficheurs

Si de nouvelles données sont disponibles...

on actualise les affichages.

Problème : il faudra modifier la classe *DonneesMeteo* dès qu'on ajoute ou retire un affichage.

# On définit une relation à plusieurs entre un sujet (observé) et des observateurs




Un sujet est lié à plusieurs observateurs. Quand son état change, il avertit ses observateurs et leur communique son nouvel état.

# Sujet, observateur et afficheur (java)

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

```
public interface Observateur
{
    public abstract void actualiser(float t, float h, float p);
}
```

```
public interface Affichage
{
    public abstract void afficher();
}
```

# Le sujet « pousse » ses données vers ses observateurs

```
public class DonneesMeteo implements Sujet
{
    private ArrayList<Observateur> observateurs;
    private float temp, humi, pres;

    public DonneesMeteo()
    { this.observateurs = new ArrayList<Observateur>(); }

    public void enregistrerObservateur(Observateur o)
    { this.observateurs.add(o); }

    public void supprimerObservateur(Observateur o)
    { this.observateurs.remove(o); }

    public void changementsMesures(float t, float h, float p)
    {
        this.temp = t; this.humi = h; this.pres = p;
        this.notifierObservateurs(t, h, p);
    }

    public void notifierObservateurs()
    {
        for(int i = 0; i < this.observateurs.size(); i ++)
        { this.observateurs.get(i).actualiser(this.temp, this.humi, this.pres); }
    }
}
```

Quand les mesures changent...

on demande aux observateurs d'actualiser leur affichage.

# L'observateur récupère les données du sujet

```
public class AffichageConditions implements Observateur, Affichage
{
    private float temp, humi, pres ;

    public AffichageConditions(Sujet donnees)
    { donnees.enregistrerObservateur(this) ; }

    public void actualiser(float t, float h, float p)
    {
        this.temp = t ; this.humi = h ; this.pres = p ;
        this.afficher() ;
    }

    public void afficher()
    {
        System.out.println("température = " + this.temp) ;
        System.out.println("humidité =    " + this.humi) ;
        System.out.println("pression =    " + this.pres) ;
    }
}
```

À sa création, l'observateur s'enregistre auprès du sujet

Actualisation de l'affichage.

# Un autre exemple d'observateur :

```
     public enum Condition {BEAU, VARIABLE, MAUVAIS, TEMPÊTE}
    public class AffichagePrévision implements Observateur, Affichage
{
    private Condition prévision;
    public AffichagePrévision( Sujet données) {
        données.enregisterObservateur(this); }
/** 3 paramètres (imposés par l'interface) mais une seule MAJ. */
public void actualiser(float t, float h, float p) {
        if ( p > 1020 ) this.prévision = BEAU;
        else if ( p < 1020  p >= 1010 ) this.prévision = VARIABLE;
        else if ( p < 1010  p >= 1000 ) this.prévision = MAUVAIS;
        else if ( p < 1000 ) this.prévision = TEMPÊTE;
}
public void afficher( ) {
System.out.println(" prévision = ", this.prévision); }
}
```

# Test du schéma sujet-observateur

```
public class TestMeteo
{
    public static void main(String[] args)
    {
        DonneesMeteo d = new DonneesMeteo();

        Affichage ac = new AffichageConditions(d);
        Affichage as = new AffichageStats(d);
        Affichage ap = new AffichagePrevisions(d);

        d.changementsMesures(3, 65, 1200);
        d.changementsMesures(5, 78, 1350);
    }
}
```

On crée le sujet observé

On crée les observateurs et on leur associe le sujet.

Les trois affichages sont automatiquement mis à jour quand l'état du sujet change.

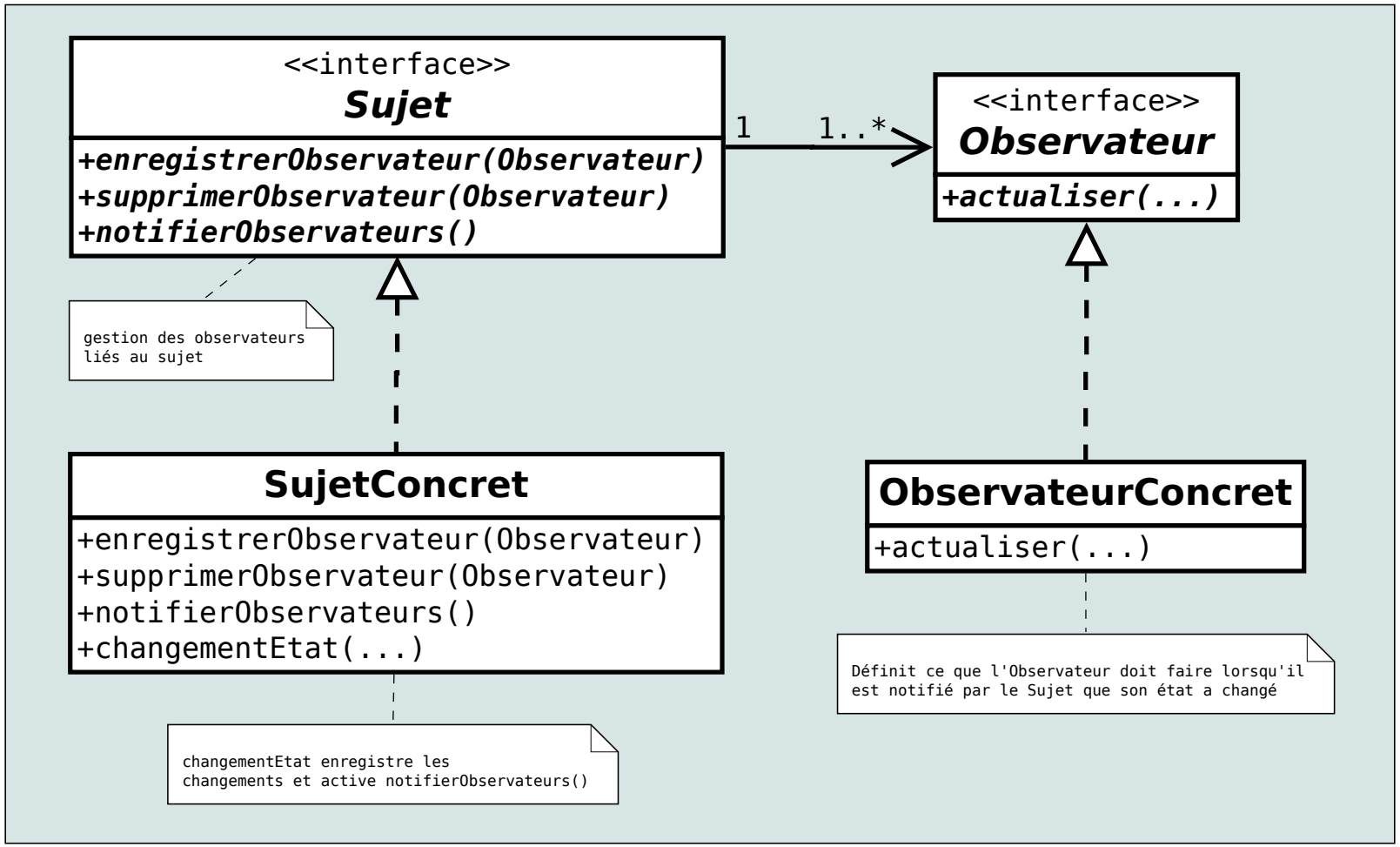
On ne modifiera pas la classe *DonneesMeteo* si on ajoute ou retire un affichage.

# Définition

## Observateur : définition

Le design pattern *Observateur* définit une relation entre objets de type un à plusieurs, de façon à ce que lorsqu'un objet change d'état, tous ceux qui en dépendent soient notifiés et mis à jour automatiquement.

# Structure du design pattern *Observateur*



# Principes généraux mis en œuvre

## Les classes doivent être le plus faiblement couplées

Un *Sujet* ne connaît que l'interface *Observateur*, il ne dépend que d'elle et non des observateurs concrets.

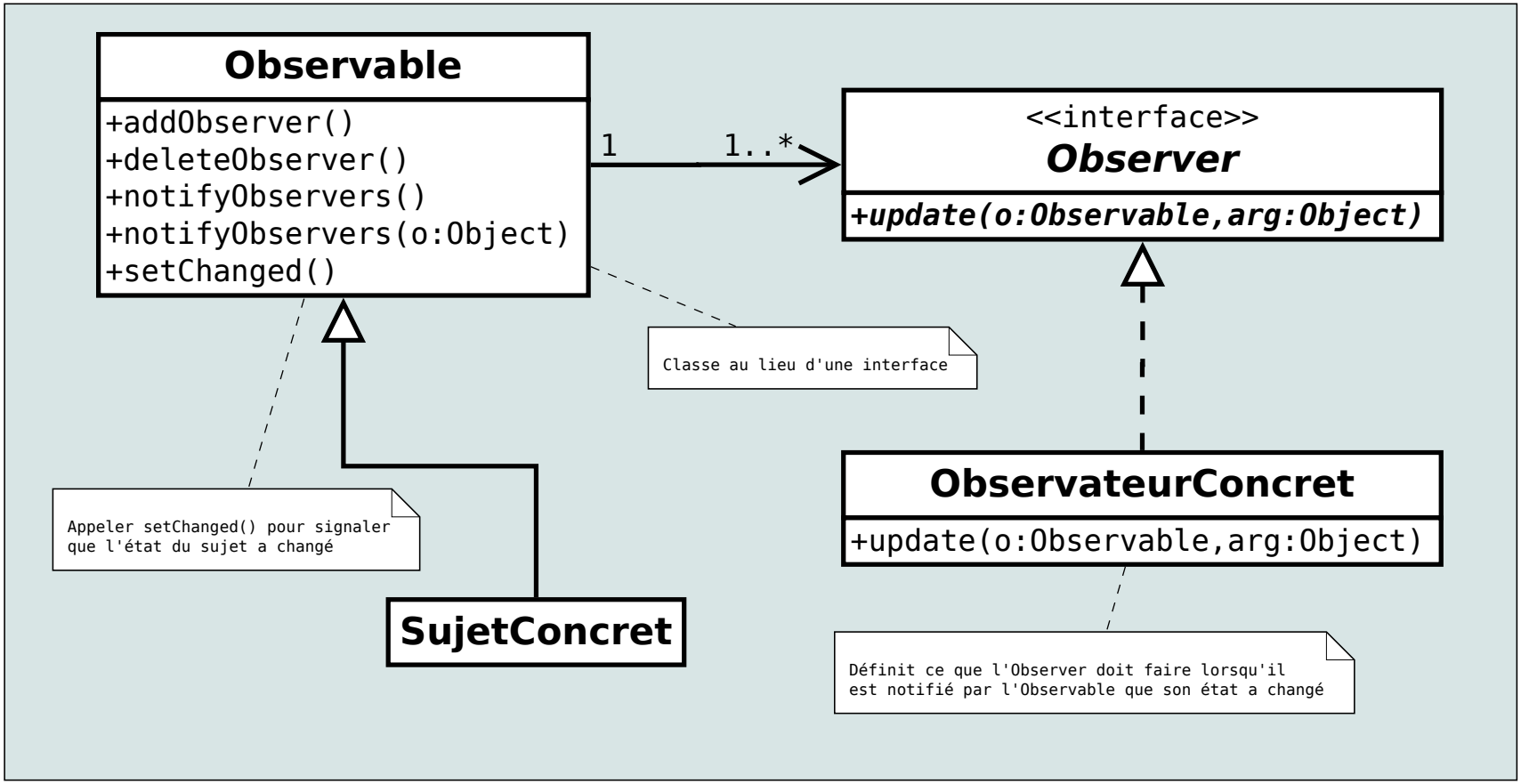
## Principe d'ouverture-fermeture

Les classes concrètes *Sujet* ne sont pas modifiées quand on ajoute/retire des observateurs.
Si on modifie un sujet concret, les observateurs ne sont pas affectés.

## Préférez les interfaces

*Sujet* et *Observateur* sont des interfaces. Une interface se situe au plus haut niveau d'abstraction : c'est une classe abstraite particulière qui n'impose ni donnée ni instruction.

# *Observable* (sujet) et *Observer* (observateur)



# Le sujet (*Observable*)

```
import java.util.Observable;

public class DonneesMeteo extends Observable;
{
    private float temp, humi, pres;

    public DonneesMeteo()          { super(); }
    public float getTemperature() { return this.temp; }
    public float getHumidite()    { return this.humi; }
    public float getPression()    { return this.pres; }

    public void changementMesures(float t, float h, float p)
    {
        this.temp = t;
        this.humi = h;
        this.pres = p;
        this.setChanged();
        this.notifyObservers(null);
    }
}
```

La gestion des observateurs est faite dans *Observable*.

Quand les données sont mises à jour...

On appelle *setChanged()* pour signaler que l'état a changé, puis on notifie les observateurs.

# L'observateur « tire » les données du sujet

```
import java.util.Observable; import java.util.Observer;

public class AffichageConditions implements Observer, Affichage
{
    private float temp, humi, pres;

    public AffichageConditions(Observable o)
    { o.addObserver(this); }

    public void update(Observable o, Object arg)
    {
        DonneesMeteo d = (DonneesMeteo) o;
        this.temp = d.getTemperature();
        this.humi = d.getHumidite();
        this.pres = d.getPression();
        this.afficher();
    }
    public void afficher()
    {System.out.println(this.temp + " " + this.humi + " " + this.pres);}
}
```

À sa création, l'*Observer* s'enregistre auprès de l'*Observable*.

*arg* peut contenir des données supplémentaires passées en argument à *notifyObservers()*.

L'*Observer* demande à l'*Observable* ses données.

# Remarques

L'implémentation java du design pattern *Observateur* ne respecte pas entièrement le design pattern originel : *Observable* est une classe, et non une interface.

- $\Rightarrow$ On ne peut pas ajouter le comportement *Observable* à une classe existante (pas d'héritage multiple en java).
- $\Rightarrow$ On ne peut pas créer sa propre implémentation de *Observable* qui fonctionne avec *Observer*.