Design Patterns
Part 2

Mohamed Youssfi

Laboratoire Signaux Systèmes Distribués et Intelligence Artificielle (SSDIA)

ENSET, Université Hassan II Casablanca, Maroc

Email : med@youssfi.net

Supports de cours : http://fr.slideshare.net/mohamedyoussfi9

Chaîne vidéo : http://youtube.com/mohamedYoussfi

Recherche : http://www.researchgate.net/profile/Youssfi_Mohamed/publications

Le pattern Observer : Tenez vos objets au courant

Design Patterns du GoF (Gang of Four) (Gamma, Helm, Johnson, Vlissides)

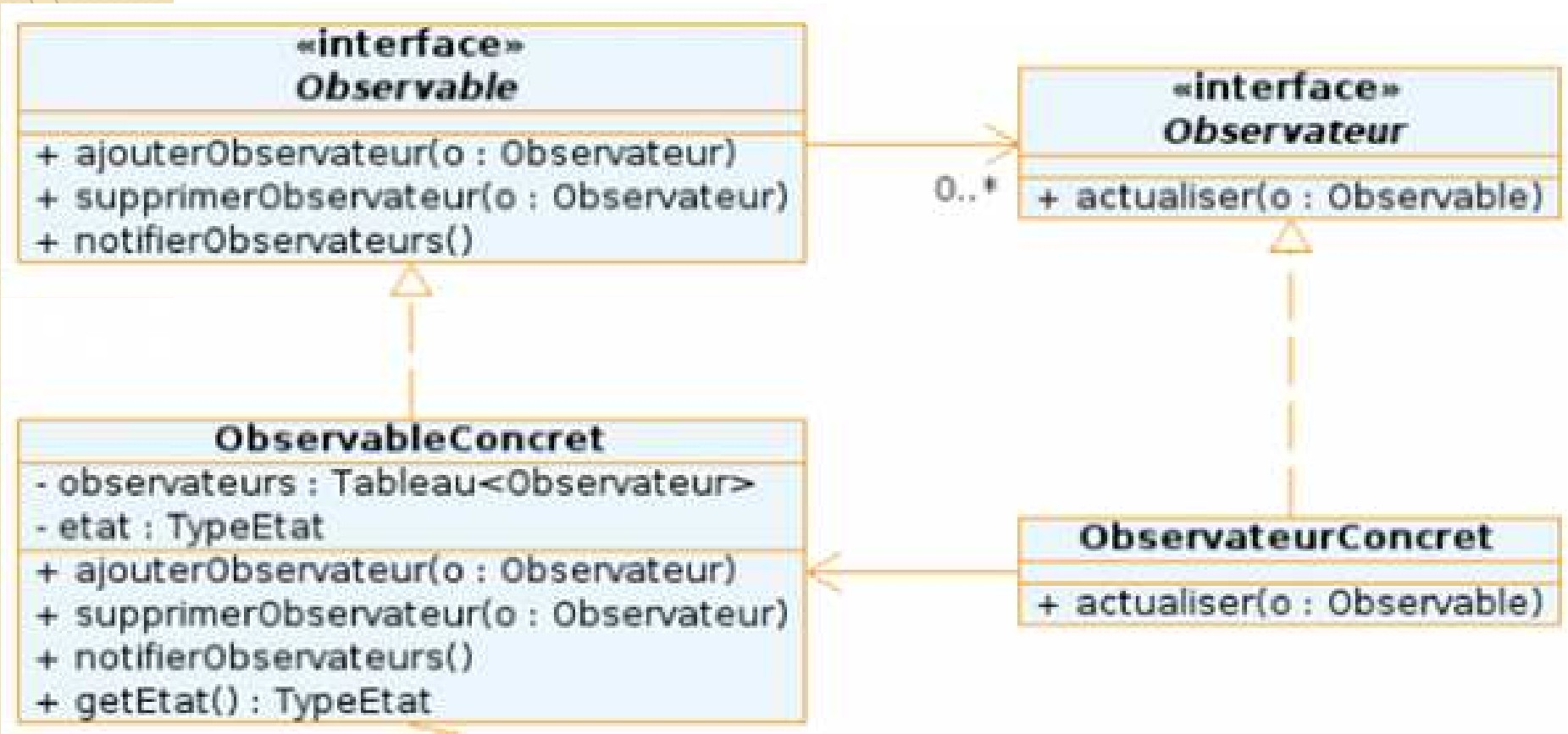
		Catégorie		
		Création	**Structure**	**Comportement**
Portée	**Classe**	Factory Method	Adapter	Interpreter
				Template Method
	Objet	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Pattern Observateur

- *Catégorie : Comportement*
- *Définition :*
  - *Le pattern observateur définit une relation entre les objets de type un à plusieurs, de façon que, lorsqu'un objet change d'état, tous ce qui en dépendent en soient informés et soient mis à jour automatiquement*

Diagramme de classe

Observable **Observer**



Implémentation

- Le diagramme UML du pattern Observateur définit deux interfaces et deux classes.
  - L'interface **Observateur** sera implémenté par toutes classes qui souhaitent avoir le rôle d'observateur.
  - C'est le cas de la classe **ObservateurConcret** qui implémente la méthode actualiser(Observable). Cette méthode sera appelée automatiquement lors d'un changement d'état de la classe observée.
  - On trouve également une interface **Observable** qui devra être implémentée par les classes désireuses de posséder des observateurs.
  - La classe **ObservableConcret** implémente cette interface, ce qui lui permet de tenir informer ses observateurs. Celle-ci possède en attribut un état (ou plusieurs) et un tableau d'observateurs. L'état est un attribut dont les observateurs désirent suivre l'évolution de ses valeurs. Le tableau d'observateurs correspond à la liste des observateurs qui sont à l'écoute.
  - En effet, il ne suffit pas à une classe d'implémenter l'interface Observateur pour être à l'écoute, il faut qu'elle s'abonne à un Observable via la méthode ajouterObservateur(Observateur).

Implémentation

- La classe ObservableConcret dispose de quatre méthodes :
  - ajouterObservateur(Observateur),
  - supprimerObservateur(Observateur),
  - notifierObservateurs()
  - getEtat().
- Les deux premières permettent, respectivement, d'ajouter des observateurs à l'écoute de la classe et d'en supprimer.
- En effet, le pattern Observateur permet de lier dynamiquement (faire une liaison lors de l'exécution du programme par opposition à lier statiquement à la compilation) des observables à des observateurs.
- La méthode notifierObservateurs() est appelée lorsque l'état subit un changement de valeur. Celle-ci avertit tous les observateurs de cette mise à jour.
- La méthode getEtat() est un simple accesseur en lecture pour l'état. En effet, les observateurs récupèrent via la méthode actualiser(Observable) un pointeur vers l'objet observé. Puis, grâce à ce pointeur, et à la méthode getEtat() il est possible d'obtenir la valeur de l'état.

Implémentation

- Il existe une variation possible lors de l'utilisation de ce pattern.
  - **TIRER** : Dans la solution présentée, une référence vers l'objet observable est mis à disposition de chaque observateur. Ainsi les observateurs peuvent l'utiliser pour appeler la méthode getEtat() et ainsi obtenir l'état de l'observable. Cette solution est nommée « TIRER » car c'est aux observateurs, une fois avertis de l'évolution, d'aller chercher l'information sur l'état.
  - **POUSSER** :Mais il existe la solution inverse appelée « POUSSER ». Dans ce cas, on passe directement l'état actuel de l'observable dans la méthode actualiser(TypeEtat). Ainsi les observateurs disposent directement de l'état.
- Qu'elle est la meilleur solution entre les deux ? C'est la première parce qu'elle permet une fois de plus de lier faiblement l'observable à ses observateurs. En effet, si l'observateur dispose d'un pointeur vers l'objet observable et que la classe observable évolue en ajoutant un deuxième état. L'observateur souhaitant se tenir informé de ce deuxième état aura juste à appeler l'accesseur correspondant. Alors que si on « POUSSER » il faudrait changer la signature de la méthode ce qui peut s'avérer plus dommageable.

Application

Cahier des charges

Félicitation! Votre société a été retenue pour construire notre station météorologique de dernière génération consultable en ligne!

La station sera basée sur notre objet DonneesMeteo (brevet en cours),qui enregistre les conditions météorologique à un moment donné (température, hygrométrie et pression atmosphérique).

Nous aimerions que vous nous créiez une application qui fournira d'abord trois affichages: conditions actuelles, statistiques et prévisions simples, tous trois mis à jour en temps réel au fur et à mesure que l'objet DonneesMeteo acquiert les données les plus récentes.

De plus cette station météo doit être extensible. MétéoExpress veut commercialiser une API pour que les autres développeurs puissent réaliser leurs propres affichages et les insérer directement. Nous souhaitons que vous nous fournissiez cette API !

MétéoExpress est convaincu d'avoir un excellent modèle métier: une fois les clients acrochés, nous prévoyons de les facturer pour chaque affichage qu'ils utilisent.

Le meilleur est pour la fin : vous serez payé en stock options.

Nous attendons avec impatience vos documents de conception et votre première version alpha

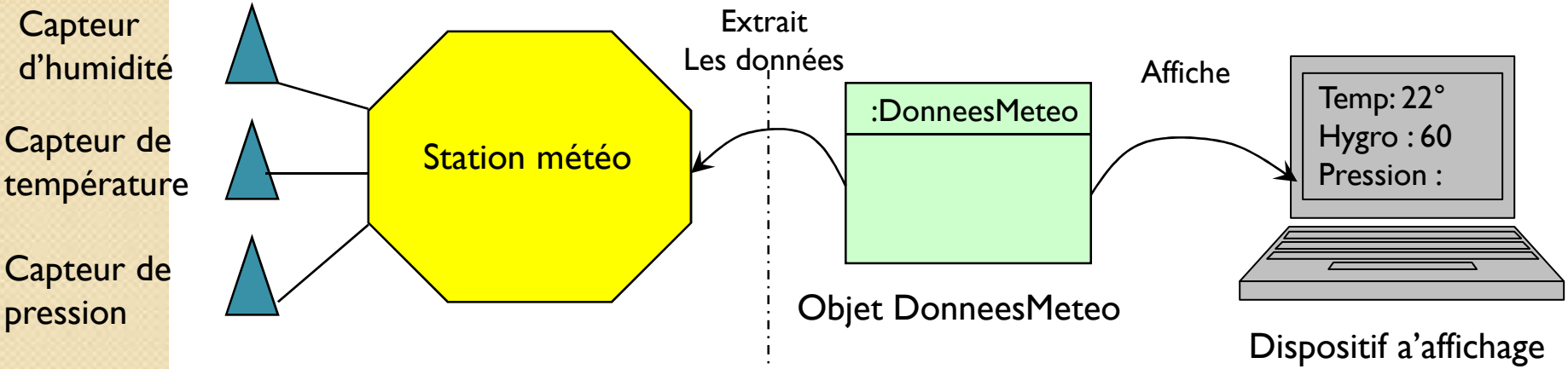
Cordialement,

Jean-Loup Ragan, PDG MétéoExpress

P.S. Nous vous envoyons par chrono les fichiers source de DonnesMeteo

Vue d'ensemble de l'application

- Les trois composants du système sont:
  - La station météo : Équipement physique qui recueille les données météo sur le temps.
  - L'objet DonneesMeteo : qui mémorise les données provenant de la station et actualise l'affichage.
  - L'affichage lui-même que les utilisateurs consultent pour connaître les conditions météorologiques actuelles.

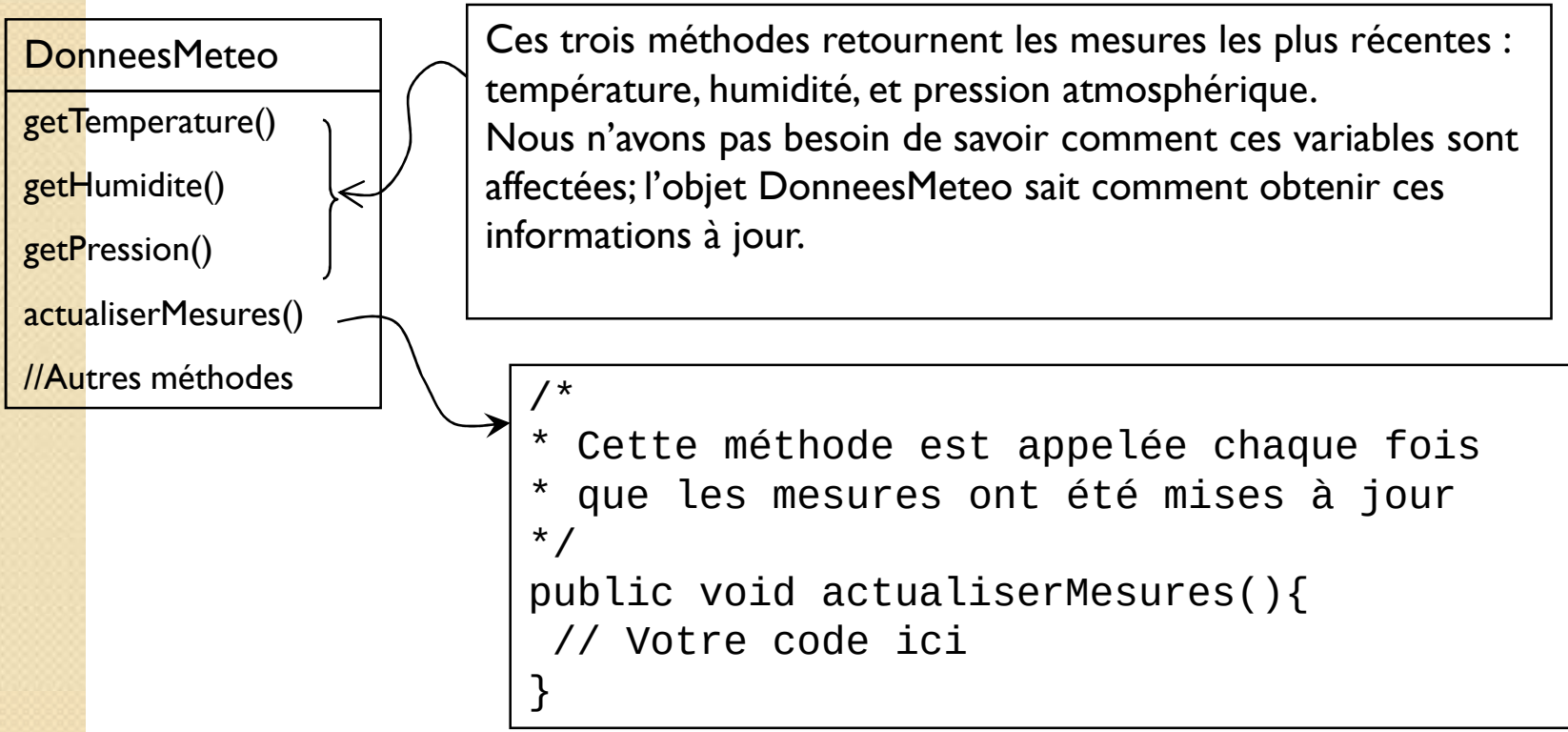


Ce que MétéoExpress Fournit | Ce que nous implémentons

Si nous choisissons de l'accepter, notre tâche consiste à créer une application qui utilise l'objet DonneesMeteo qui actualise ces trois affichages: conditions actuelles, statistiques et prévisions

A l'intérieur de la classe DonneesMeteo

- Le lendemain matin, les fichiers source de DonneesMeteo arrivent comme promis.
- Nous jetons un coup d'œil au code qui semble relativement simple:



Récapitulons….

- La classe DonneesMeteo a des méthodes d'accès pour trois valeurs de mesures: température, hygrométrie, pression atmosphérique.
- La méthode actualiserMesures() est appelée chaque fois qu'une nouvelle mesure est disponible. Nous ne savons pas comment cette méthode est appelée, et peu importe.
- Nous devrons implémenter trois affichages qui utilisent les données météorologiques:
  - Un affichage des conditions actuelles
  - Un affichage des statistiques
  - Un affichage des prévisions.
- Ces trois affichages doivent être mis à jour, chaque fois que DonnesMeteo aquiert de nouvelles données.
- Le système doit être extensible:
  - D'autres développeurs pourrons créer d'autres affichages personnalisés
  - Les utilisateurs pourront ajouter ou retirer autant d 'éléments qu'ils le souhaitaient à l'application
  - Actuellement, nous ne connaissons que les trois types d'affichages initiaux
    - Conditions actuelles
    - Statistiques
    - prévisions

Premier essai de station météo

- Voici une première possibilité d'implémentation: nous suivons l'indication des développeurs de MeteoExpress et nous ajoutons notre code à la méthode actualiderMesures():

```
public class DonneesMeteo{
 //déclarations des variables d'instance
public void actualiserMesures(){
 float t=getTemperature();
 float h=getHumidite();
 float p getPression();
 affichageConditions.actualiser(t,h,p);
 affichageStats.actualiser(t,h,p);
 affichagePrevisions.actualiser(t,h,p);
 }
// Autres méthodes
}
```

A vos crayons

D'après notre première implémentation, quels énoncés suivants sont vrais?

- A. Nous codons des implémentations concrètes, non des interfaces
- B. Nous devons modifier le code pour chaque nouvel élément d'affichage
- C. Nous n'avons aucun moyen d'ajouter (ou de supprimer) des éléments d'affichage dynamiquement
- D. Les éléments d'affichage n'implémentent pas une interface commune
- E. Nous n'avons pas encapsulé les parties qui varient
- F. Nous violons l'encapsulation de la classe DonneesMeteo

Qu'est ce qui cloche dans notre implémentation

- Repenser à tous les concepts et les principes du chapitre précédent.

```
public class DonneesMeteo{
 //déclarations des variables d'instance
public void actualiserMesures(){
 float t=getTemperature();
 float h=getHumidite();
 float p getPression();

 affichageConditions.actualiser(t,h,p);
 affichageStats.actualiser(t,h,p);
 affichagePrevisions.actualiser(t,h,p);
 }
// Autres méthodes
}
```

En codant des implémentation concrètes, nous n'avons aucun moyen d'ajouter ni de supprimer des éléments sans modifier le programme

Au moins, nous semblons utiliser une interface commune pour communiquer avec les affichages.. Ils ont tous une méthode actualier() qui lie les valeurs de temp, humidité et pression

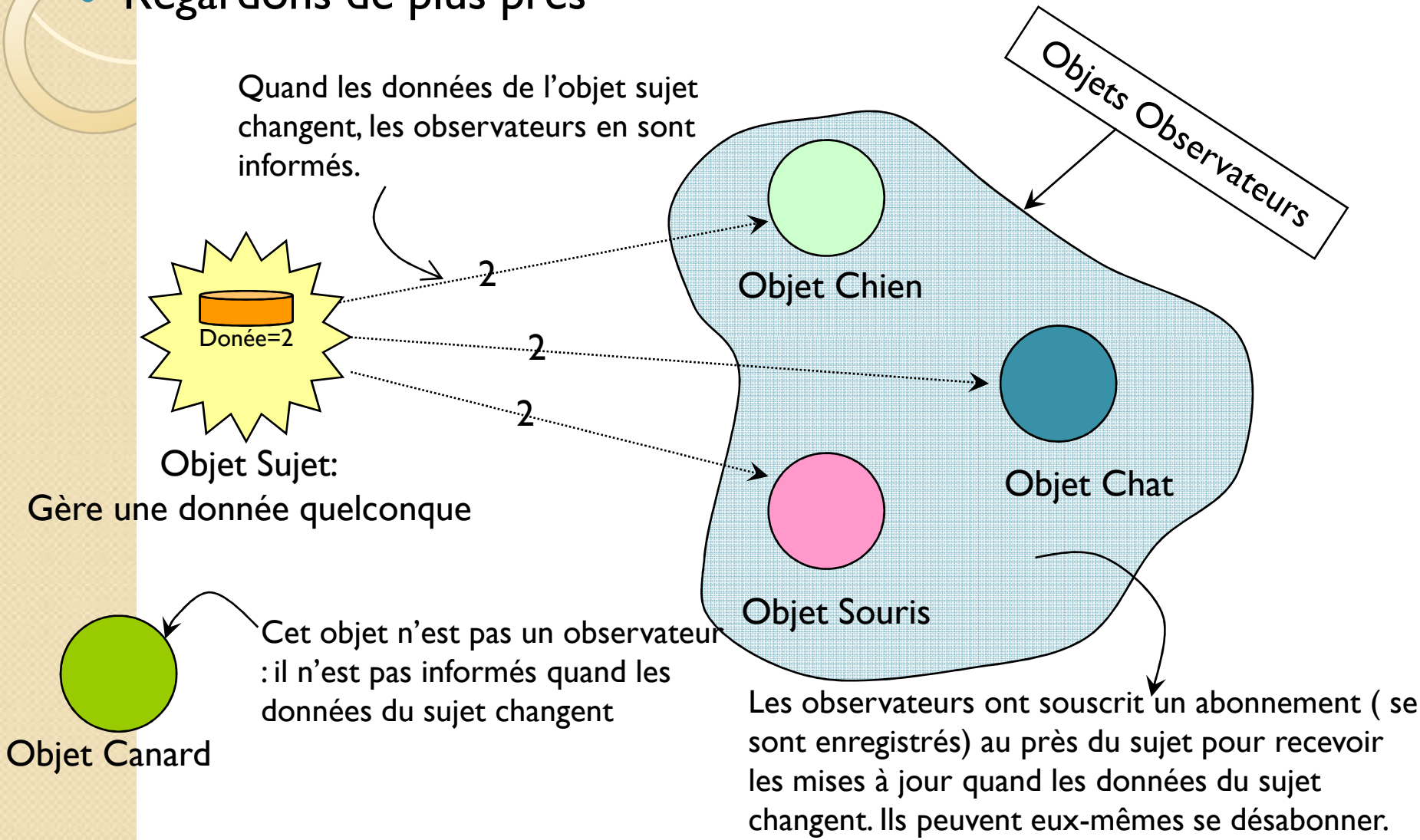
Points de variation : nous devons l'encapsuler

Faites connaissance avec le pattern Observateur

- Vous savez comment fonctionne un abonnement à un magazine:
  1. Un éditeur se lance dans les affaires et commence à diffuser des magazines
  2. vous souscrivez un abonnement
  3. Chaque fois qu'il y'a une nouvelle édition, vous la recevez. Et tant que vous êtes abonné, vous recevez les nouvelles éditions.
  4. Quand vous ne vouliez pas de ces magazines, vous résiliez votre abonnement. On cesse alors de vous les livrer.
  5. Tant que l'éditeur reste en activité, les particuliers, les hôtels, les compagnies aériennes, etc., ne cessent de s'abonner et de se désabonner.
- Si vous avez compris cet exemple, vous avez compris l'essentiel du patterns Observateur, sauf que nous appelons l'éditeur le **SUJET** et les abonnés les **OBSERVATEURS**

Faites connaissance avec le pattern Observateur

- Regardons de plus près



Pattern Observateur : définition

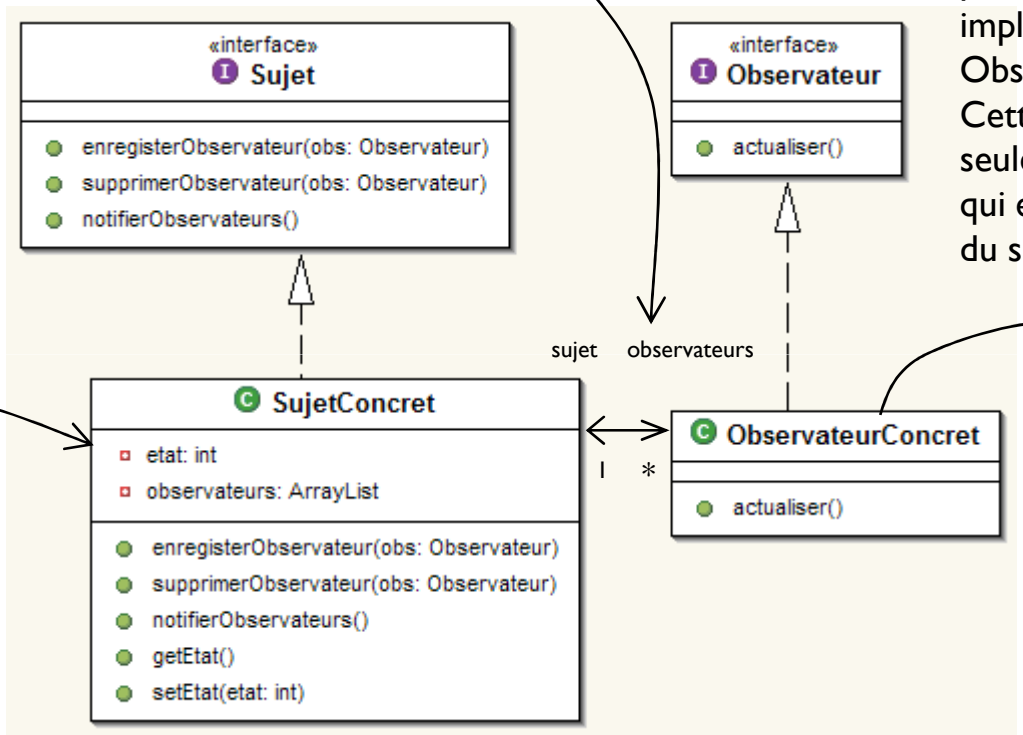
- *Le pattern observateur définit une relation entre les objets de type un à plusieurs, de façon que, lorsqu'un objet change d'état, tous ce qui en dépendent en soient informés et soient mis à jour automatiquement*

Pattern Observateur : Diagramme de classes

Chaque sujet peut avoir plusieurs observateurs

Voici l'interface Sujet les objets utilise cette interface pour s'enregistrer comme observateur et pour résilier leur abonnement

Tous les observateurs potentiels doivent implémenter l'interface Observateur.
Cette interface n'a qu'une seule méthode actualiser() qui est appelée quand l'état du sujet change



Un sujet concret implémente toujours l'interface Sujet.
Outre les méthode d'ajout et de suppression d'observateurs, le sujet concret implémente la méthode notifierObservateurs() qui sert à mettre à jour tous les observateurs chaque fois que l'état change.

Le sujet concret peut également avoir des méthodes pour accéder à son état (getEtat()) et le modifier (setEtat())

Les observateurs concrets peuvent être n'importe quelle classe qui implémente l'interface Observateur.
Chaque observateur s'enregistre au près du sujet réel pour recevoir les mises à jour;

Le pouvoir du faible couplage

- Lorsque deux objets sont faiblement couplés, ils peuvent interagir sans pratiquement se connaître.
- Le pattern observateur permet une conception dans laquelle le couplage entre sujet et observateurs est faible:
  - **Le sujet ne sait qu'une seule chose :**
    - L'observateur implémente une certaine interface (**Observateur**).
    - Il n'a pas donc besoin de connaître ni la classe concrète de l'observateur, ni ce qu'il fait ni quoi que ce soit d'autre.
  - **Nous pouvons ajouter des observateurs à tout moment :**
    - Comme le sujet dépend uniquement d'une liste d'objets qui implémente l'interface Observateur (observateurs:ArrayList), nous pouvons ajouter, supprimer, modifier des observateurs à volonté pendant l'exécution.
    - Le sujet continue de fonctionner comme si rien n'était
  - **Nous n'avons jamais besoin de modifier le sujet pour ajouter de nouveaux types d'observateurs :**
    - Disons qu'une nouvelle classe concrète se présente et a besoin d'être un observateur, nous n'avons besoin d'ajouter quoi que ce soit au sujet pour gérer ce nouveau type.
    - Il suffit qu'elle implémente l'interface Observateur, et de l'enregistrer entant que observateur.
    - Le sujet continue de diffuser des notifications à tous les observateurs.
  - **Nous pouvons réutiliser les observateurs et les sujets indépendamment les uns des autres.**
    - Si nous avons un autre emploi d'un sujet ou d'un observateur, nous pouvons les réutiliser sans problème par ce qu'ils sont faiblement couplés.
  - **Les modifications des sujets n'affectent pas les observateurs et inversement**

Le pouvoir du faible couplage

Principe de conception:

Efforcez-vous à coupler faiblement les objets qui interagissent

4er Principe de conception

Les conceptions faiblement couplées nous permettent de construire les systèmes OO souples, capables de faire face aux changements par ce qu'ils minimisent l'interdépendance entre les objets.

A vos crayons

- Avant de continuer, essayer d'esquisser les classes et les interfaces nécessaires pour implémenter la station météo, notamment la classe DonneesMeteo et les classes des différents affichages.
  - S'il vous faut un peu d'aide, lisez la page suivante : vos collègues sont déjà entrain de penser à la conception de la station météo.

Conversation dans un box

- *Marie: Et bien, ça aide de savoir que nous utilisons le pattern Observateur.*
- *Anne: Oui… mais comment allons nous l'appliquer.*
- *Marie : regardons à nouveau la définition :*
  - Le pattern observateur définit une relation entre les objets de type un à plusieurs, de façon que, lorsqu'un objet change d'état, tous ce qui en dépendent en soient informés et soient mis à jour automatiquement
- *Marie : ça a l'air claire quand on y pense : Notre classe DonneesMeteo est le « un », et le « plusieurs », ce sont les éléments qui affichent les différentes mesures;*
- *Anne : c'est vrai, la classe DonneesMeteo a avidement un état… la température, l'humidité, la pression atmosphérique, et bien sûr, ces données changent.*
- *Marie : Oui, et quand ces données changent, il faut notifier les éléments d'affichage pour qu'ils puissent mettre à jour l'affichage en utilisant les nouvelles mesures.*
- *Anne : super, maintenant je crois que je vois comment appliquer le pattern Observateur à notre problème.*
- *Marie : mais il y'a encore deux ou trois choses que je ne suis pas sûre d'avoir comprises.*
- *Anne : Pae exemple ?*
- *Marie : D'abord, comment faire pour que les éléments d'affichage puissent obtenir les mesures.*

Conversation dans un box

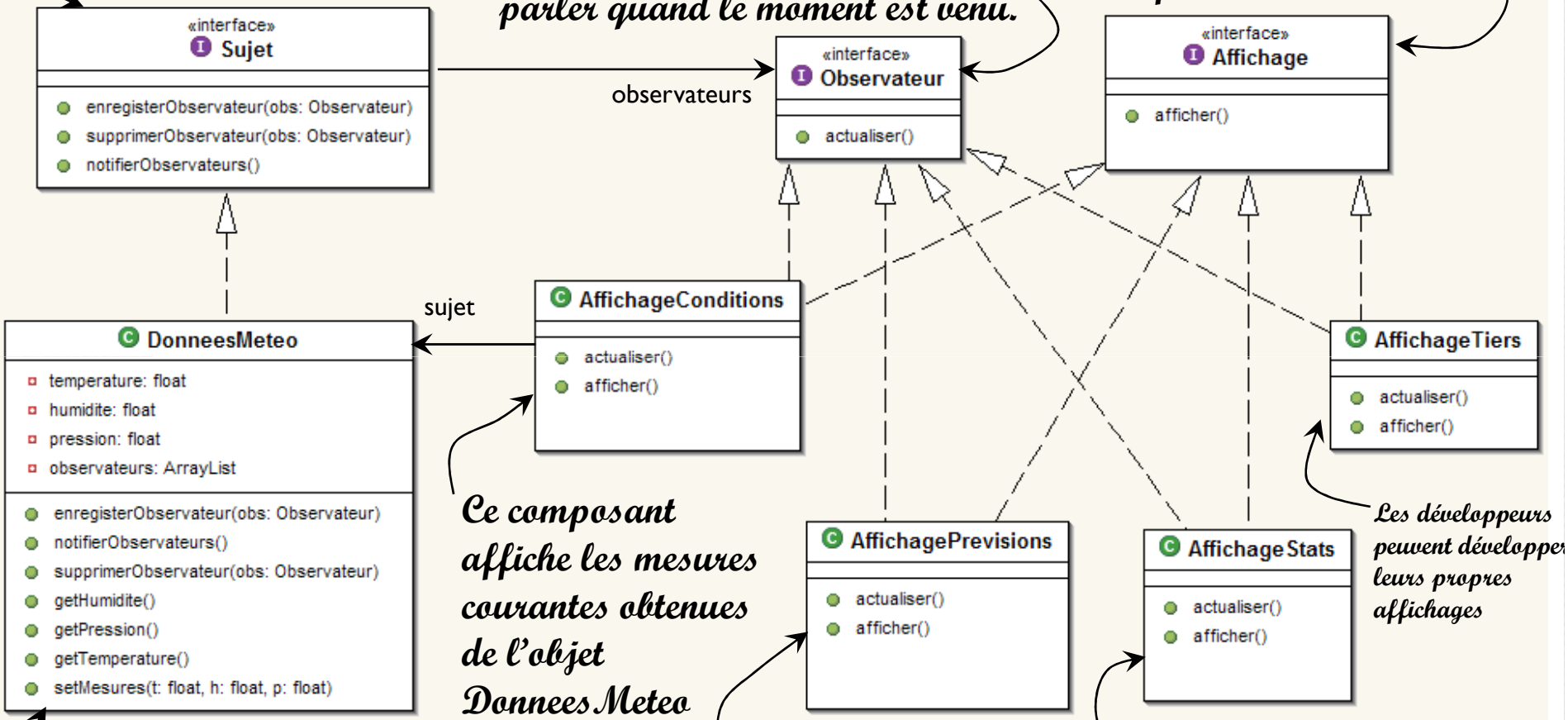
- *Anne: Eh bien, en regardant à nouveau le diagramme de classes du pattern Observateur, si nous faisons de l'objet DonneesMeteo le sujet et des éléments d'affichage les observateurs, alors les affichages vont s'enregistrer eux-mêmes auprès de l'objet DonneesMeteo pour obtenir les informations dont ils auront besoin, non ?*
- *Marie: Oui… et une fois que la station météo est au courant de l'existence d'un élément d'affichage, elle peut simplement appeler une méthode pour lui transmettre les mesures.*
- *Anne: Il faut se souvenir que chaque affichage peut être différent… et c'est là que je crois que qu'intervient une interface commune. Même si, chaque composant est d'un type différent, ils doivent implémenter tous une même interface pour que l'objet DonneesMeteo sache comment leur transmettre les mesures.*
- *Marie: je vois ce que tu veux dire. Chaque affichage aura par exemple une méthode actualier( ) que DonneesMeteo va appeler.*
- *Anne: Et actualiser( ) est définie dans une inteface commune que tous les éléments d'affichage implémenteront.*

Concevoir la station météo

Voici l'interface Sujet

Tous nos composants implémentent l'interface Observateur. Ainsi le sujet dispose d'une interface à qui parler quand le moment est venu.

Tous les affichages possèdent la méthode afficher( ). Il implémentent donc cette interface



Ce composant affiche les mesures courantes obtenues de l'objet DonneesMeteo

Les développeurs peuvent développer leurs propres affichages

DonneesMeteo implémente maintenant l'interface Sujet

Cette classe Mémorise les mesures MIN, MOY et MAX et les affiche.

Cette classe affiche une prévision en fonction de l'indicateur du baromètre

Implémenter la station meteo

- Commençons par les interfaces

```
public interface Sujet {
        void enregisterObservateur(Observateur obs);
        void supprimerObservateur(Observateur obs);
        void notifierObservateurs();
}
```

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

}
```

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

Implémenter l'interface sujet dans DonneesMeteo

```
import java.util.ArrayList;
public class DonneesMeteo implements Sujet {
   private float temperature;
   private float humidite;
   private float pression;
    private ArrayList observateurs;
public DonneesMeteo(){
   observateurs=new ArrayList();
}
public void enregisterObservateur(Observateur obs) {
      observateurs.add(obs);
 }
public void supprimerObservateur(Observateur obs) {
   int i=observateurs.indexOf(obs);
   if(i>=0){
      observateurs.remove(i);
   }
 }
```

Implémenter l'interface sujet dans DonneesMeteo (suite)

```
public void notifierObservateurs() {
  for(int i=0;i<observateurs.size();i++){
  Observateur o=(Observateur)observateurs.get(i);
  o.actualiser(temperature,humidite,pression);
      }
  }
  public void setMesures(float t, float h, float p)
  {
      this.temperature=t;
      this.humidite=h;
      this.pression=p;
      actualiserMesures();
  }
      public void actualiserMesures() {
      notifierObservateurs();
  }
}
```

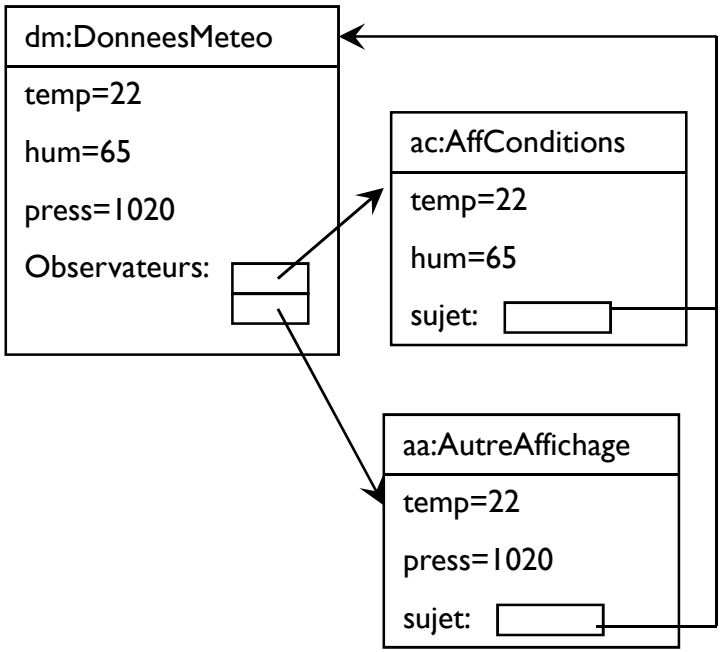
Implémenter les affichages

```
public class AffichageConditions implements Observateur,
   Affichage {
   private float temperature;
   private float humidite;
   private Sujet donneesMeteo;
   public AffichageConditions(Sujet dm){
       this.donneesMeteo=dm;
       donneesMeteo.enregisterObservateur(this);
   }
   public void afficher() {
       System.out.println("Conditions actuelles:"+
              temperature+" degès et "+humidite+" %
   d'hmidité");
   }
   public void actualiser(float t, float h, float p) {
       this.temperature=t;
       this.humidite=h;
       afficher();
   }
   }
```

Mettre en route la station météo

TestDonneesMeteo.java

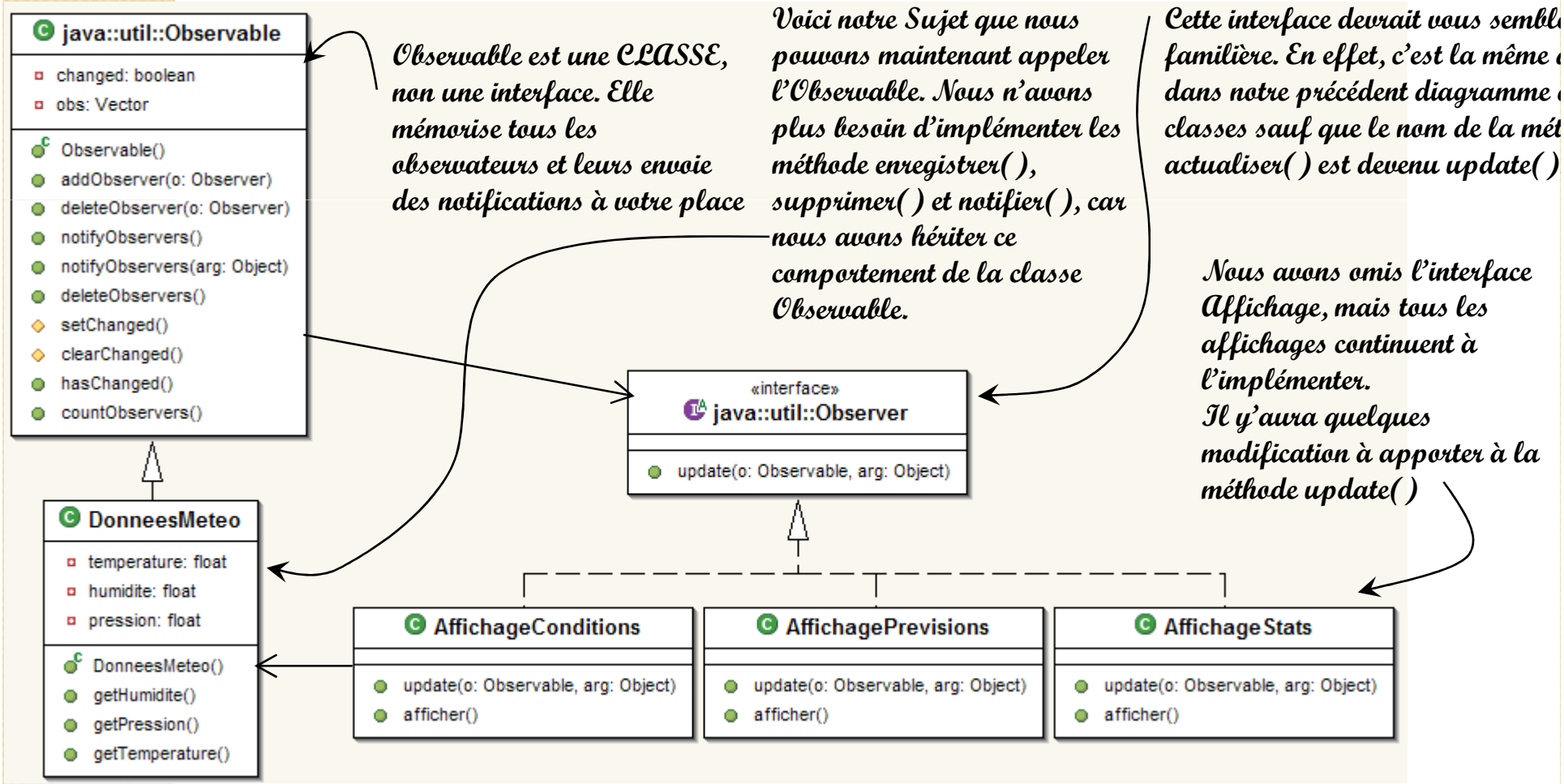
```
public class TestDonneesMeteo {
    public static void main(String[] args) {
DonneesMeteo dm=new DonneesMeteo();
AffichageConditions ac=new AffichageConditions(dm);
dm.setMesures(22,65,1020);
dm.setMesures(25,75,1000);
dm.setMesures(23,30,1800);
    }

}
```




```
Conditions actuelles:22.0 degès et 65.0 % d'hmidité
Conditions actuelles:25.0 degès et 75.0 % d'hmidité
Conditions actuelles:23.0 degès et 30.0 % d'hmidité
```

Utiliser le pattern Observateur de java

- Jusqu'ici, nous avons écrit notre propre code pour appliquer le pattern Observateur,mais java le prend en charge nativement dans plusieurs de ses API.
- Le cas le plus général est constitué d'une interface Observer et d'une classe Observable du package java.util.
- Elles sont très similaires à nos interfaces, mais elles apportent d'autres fonctionnalités « toutes prêtes ».



Comment fonctionne le pattern Observateur en java

- Le pattern Observateur intégré à java fonctionne un peu différemment de l'implémentation que nous avons choisie pour notre première station météo.
- La différence la plus évidente réside dans le faite que DonneesMeteo (notre sujet) étend maintenant la classe Observable et nous héritons (entre autres) des méthodes addObserver(), deleteObserver() et notifierObservers(). Voici comment nous utilisons la version java de cet Observateur:
  - Pour qu'un objet devient observateur…
    - Comme d'habitude, implémenter l'interface Observer et appeler la méthode addObserver() de l'objet Observable.
    - De même pour supprimer un observateur, il suffit d'appeler la méthode deleteObserver().

Comment fonctionne le pattern Observateur en java

- Pour que l'observable envoie des notifications…
  - Tout d'abord, il faut qu'il devienne Observable en étendant la super classe java.util.Observable. Puis l'on possède à deux étapes:
    - Vous devez d'abord appeler la méthode setChanged(), pour signifier que l 'état de votre objet à changé
    - Puis vous appelez l'une des deux méthodes notifyObservers():
      - public void notifyObservers() : méthode dans argument
      - notifyObservers(Object arg) : méthode avec argument arg de type Object
- Pour qu'un observateur reçoive des notifications…
  - Il implémente la méthode update comme auparavant, mais sa signature est diffère légèrement:
    - Public void update(Observer o, Object arg):
      - Le premier paramètre représente le sujet qui envoyé la notification
      - Le deuxième paramètre représente l'objet qui contient les données transmises à notifyObservers ou null si aucun objet n'a été spécifié.
    - Si vous voulez pousser des données vers des observateurs, vous pouvez les transmettre sous forme d'objet donnée à la méthode notifyObservers(Object arg).
    - Si non, l'observateur doit tirer les données de l'objet Observable qui lui a été transmis.
- Voyons comment faire tout cela dans notre station météo.

Méthode setChanged()

- Avant de commencer, on peut remarquer l'existence inhabituelle de la méthode setChanged() dans la classe Observable.
- A quoi sert donc cette méthode?
- Réponse:
  - Cette méthode sert à signifier que l'état a changé et que notifyObservers(), quand elle est appelée, doit mettre à jour les observateurs.
  - Si notifyObservers() est appelée sans qu'on ait d'abord appelé setChanged(), les observateurs ne seront pas notifiés.
  - Jetons un coup d'œil dans les coulisses de la classe Observable pour voir comment cala fonctionne.

Coulisses de Obsarvale

```
protected void setChanged() {
   changed = true; }

protected void clearChanged() {
   changed = false;    }

public void notifyObservers() {
   notifyObservers(null); }

public void notifyObservers(Object arg) {
   Object[] obsArr;
   if (!changed)  return;
   obsArr = obs.toArray();
   clearChanged();
   for (int i = obsArr.length-1; i>=0; i--){
     Observer o=(Observer) obsArr[i];
     o.update(this, arg);
   }
}
```

La méthode setChanged positionne un drapeau changed à true.

La méthode clearChanged positionne le drapeau changed à false.

notifyObserwers( ) ne notifie les observateurs que si la valeur du drapeau est true;

Si le drapeau changed est true, elle le remet à false et puis notifie les observateurs

Intérêt du drapeau changed

- La méthode setChanged() est conçue pour vous autoriser plus de souplesse dans la façon dont vous mettez à jour les observateurs en vous permettant d'optimiser les notifications.
- Par exemple,
  - Dans le cas de notre station météo, imaginez que les capteurs de mesures soient si sensibles que les températures affichées varient constamment de quelques dixième de degrés.
  - L'objet DonneesMeteo pourraient émettre des notification en permanence.
  - Mais si nous voulions émettre des notifications toutes moins fréquentes, nous n'appelons setChanged() que lorsque la température aura augmentée ou diminuée d'un demi degré.
- Vous n'utiliserez, peut-être pas très souvent cette fonctionnalité, mais sachez qu'elle existe en cas de besoin.
- Dans tous les cas, vous devez appeler la méthode setChanged() pour que les notifications soient effectuées.
- Comme vous pouvez également appeler clearChanged(), réinitialise le drapeau « changed » à false et la méthode hasChanged() qui retourne la valeur du drapeau.

Retravailler la station météo avec le pattern Observer du package java.util

DonneesMeteo.java

```
import java.util.Observable;

public class DonneesMeteo extends Observable {
  private float temperature;
  private float humidite;
  private float pression;

  public DonneesMeteo(){}

  public float getHumidite() {
      return humidite;
  }
  public void actualiserMesures(){
      setChanged();
      notifyObservers();
  }
```

Retravailler la station météo avec le pattern Observer du package java.util

DonneesMeteo.java (suite)

```
public void setMesures(float t, float h, float p) {
        this.temperature=t;
        this.humidite=h;
        this.pression=p;
        actualiserMesures();
    }
    public float getPression() {
        return pression;
    }
    public float getTemperature() {
        return temperature;
    }
}
```

AffichageConditions.java (suite)

```
import java.util.Observable;
import java.util.Observer;
public class AffichageConditions implements Observer,Affichage {
   Observable sujet;
   private float temperature;
   private float humidite;
   public AffichageConditions(Observable obs){
        this.sujet=obs;
        sujet.addObserver(this);
   }
   public void update(Observable obs, Object arg) {
        if(obs instanceof DonneesMeteo){
                DonneesMeteo dm=(DonneesMeteo)obs;
                this.temperature=dm.getTemperature();
                this.humidite=dm.getHumidite();
                afficher();
        }
   }
   public void afficher() {
        System.out.println("Conditions actuelles:"+
        temperature+" degès et "+humidite+" % d'hmidité");
   }
}
```

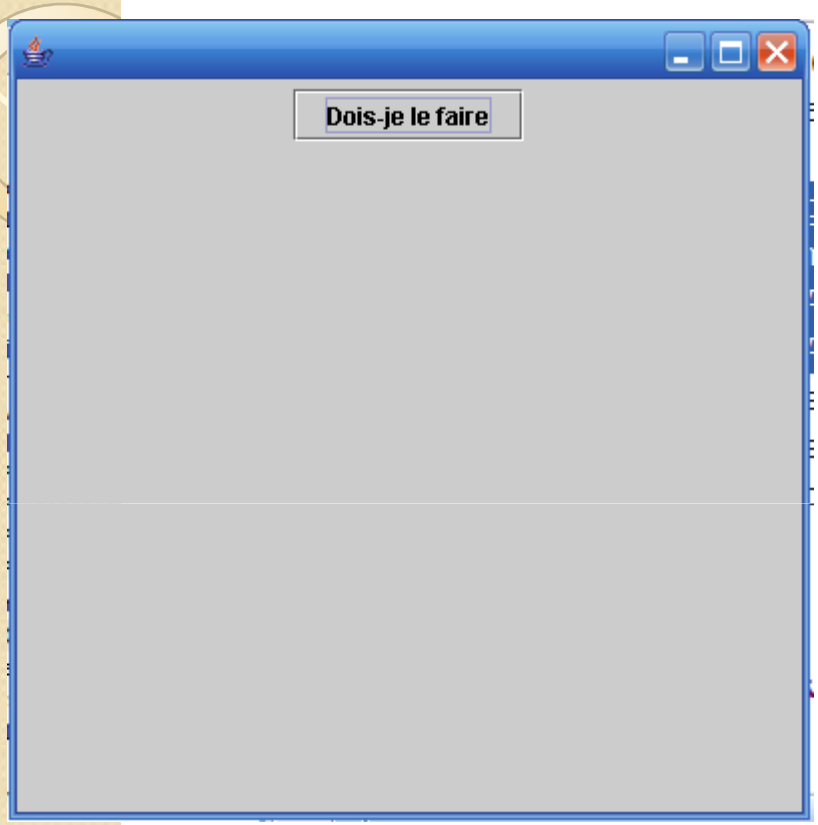
Travail à faire

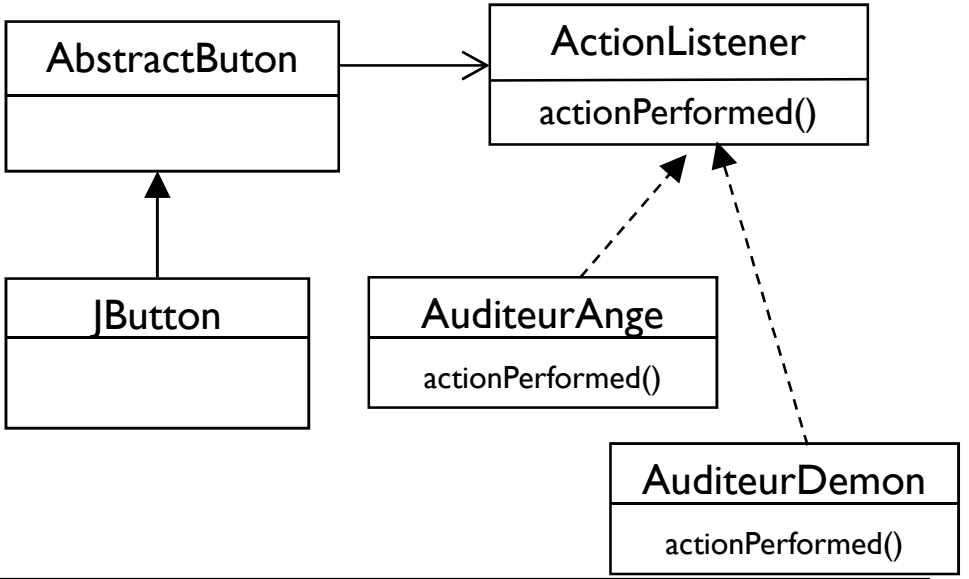
- Ecrire une implémentation de AffichagePrevisions de façon à afficher les messages suivants:
  - « Amélioration en cours » si la différence de pression est positive
  - « le temps se rafraîchit », si la différence de pression est négative.
  - « Dépression bien installée », si la pression ne change pas.

Autres endroits ou se trouve le pattern observateur dans le JDK.

- Le composant Swing JButton est un sujet qui hérite de la classe AbstractButton. Si vous regarder la structure de cette classe vous regardez des méthodes comme add et remove qui permettent d'ajouter et de supprimer des observateurs qu'on appelle dans la terminologie de Swing des listenners (Auditeurs).
- Ces listeners sont à l'écoute des différents types d'événements qui peuvent se produire dans le composant JButton.
- Par exemple un ActionListener vous permet d'acouter tous les types d'actions qui peuvent être appliquées à un bouton, comme les clics.
- Si l'un des événements se produit sur le bouton, ce dernier notifie tous les observateurs qui sont de type ActionListener en faisant appel à la méthode actionPerformed() de chaque ActionListener.

Exemple pour le plaisir





- Créer un JFrame
- Créer un JButton : Sujet
- Créer deux listeners ange et demon : Observateurs
- enregistrer les deux listeners chez le bouton en faisant appel à la méthode addActionListener()
- Si un utilisateur clique sur le bouton, son état change et tous les observateurs sont notifiés en exécutant leurs méthodes actionPerformed()

Console:

```
Fais le tu as tout à gagner!
Ne le fais pas, tu pourrais le regréter
```

Exemple

ExObsSwing.java:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.JButton;
import javax.swing.JFrame;
public class ExObsSwing{
   private JFrame jf;
   private JButton bouton;
// Créer un premier observateur qui implémente ActionListener
   class AutiteurAnge implements ActionListener{
    public void actionPerformed(ActionEvent e) {
System.out.println("Ne le fais pas, tu pourrais le regréter");
        }
   }
// Créer un deuxième observateur qui implémente ActionListener
   class AutiteurDemon implements ActionListener{
        public void actionPerformed(ActionEvent e) {
System.out.println("Fais le tu as tout à gagner!");
        }
   }
```

Exemple

```
public void go(){
        jf=new JFrame();
// Créer le sujet
        bouton=new JButton("Dois-je le faire");
// Créer deux observateurs ange et demon
        AuditeurAnge ange=new AuditeurAnge();
        AuditeurDemon demon=new AuditeurDemon();
// Enregistrer les deux observateurs chez le bouton
        bouton.addActionListener(ange);
        bouton.addActionListener(demon);
        jf.getContentPane().setLayout(new FlowLayout());
        jf.getContentPane().add(bouton);
        jf.setBounds(100,100,400,400);
        jf.show();
// Le bouton s'affiche dans la frame.
// En cliquant sur celui-ci, tous les listeners enregistrés
// chez le boutons sont notifiés et exécutent la méthode
// actionPerformed(); équivalent à update()
   }
   public static void main(String[] args) {
        ExObsSwing ex=new ExObsSwing();
        ex.go();
   }
}
```