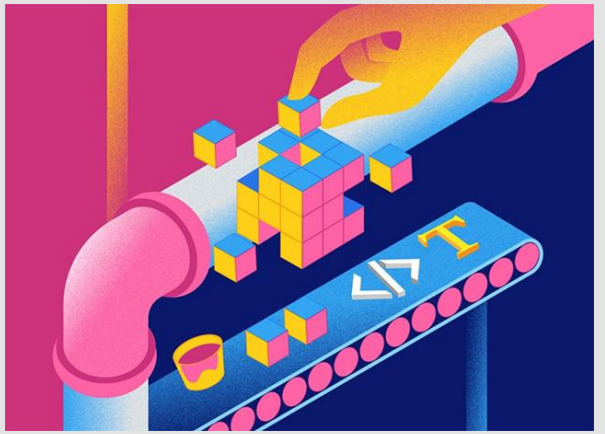
ENSET
ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA
المدرسة العليا لأساتذة التعليم التقني المحمدية

Design Patterns

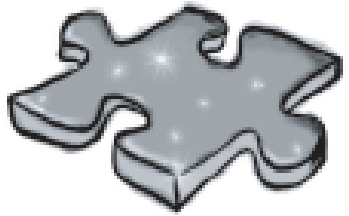
2 ème année Cycle Ingénieur

Pr. SARA RETAL

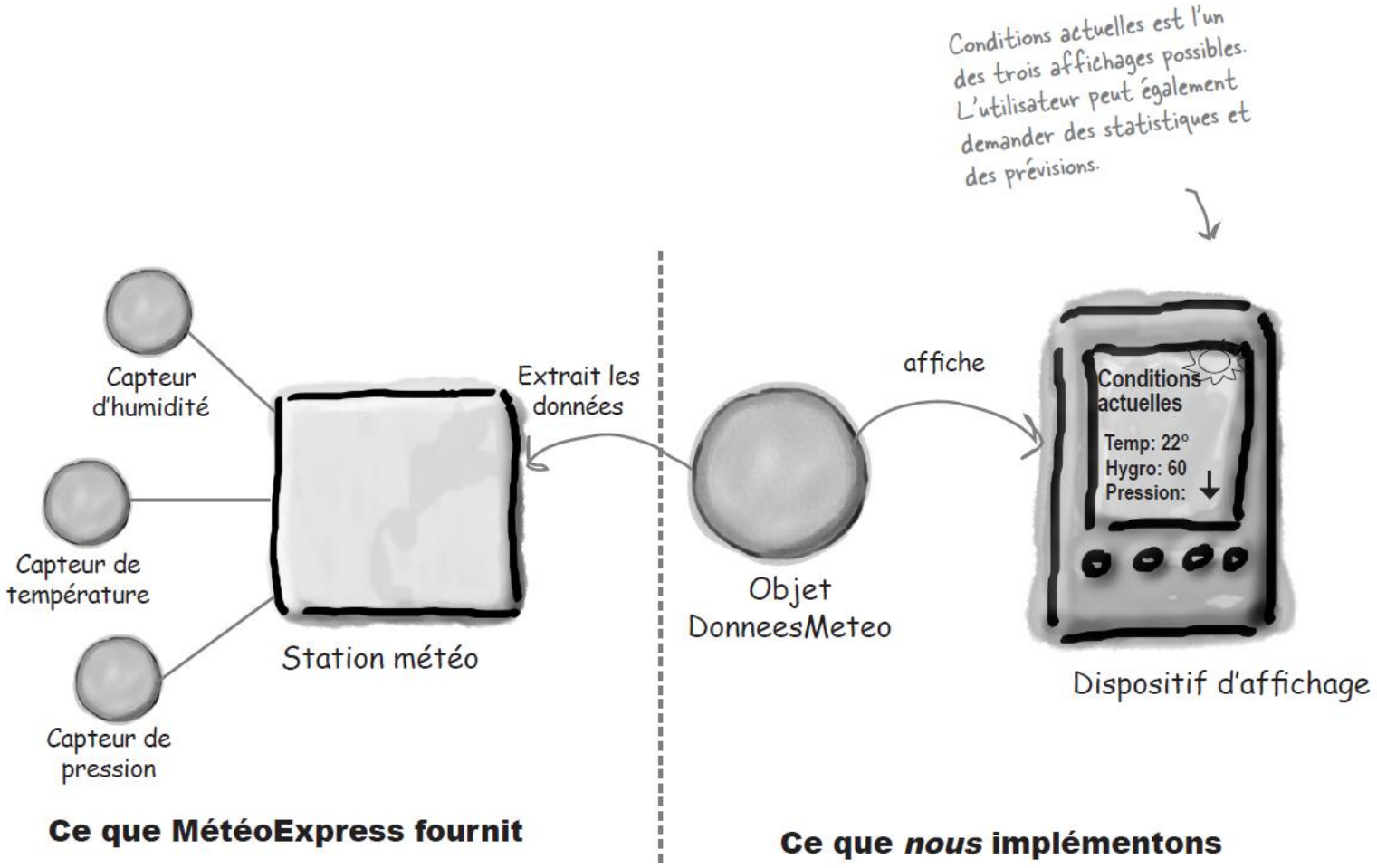
GLSID 2, ICCN 2 & IIBDCC 2



Problèmes de conception

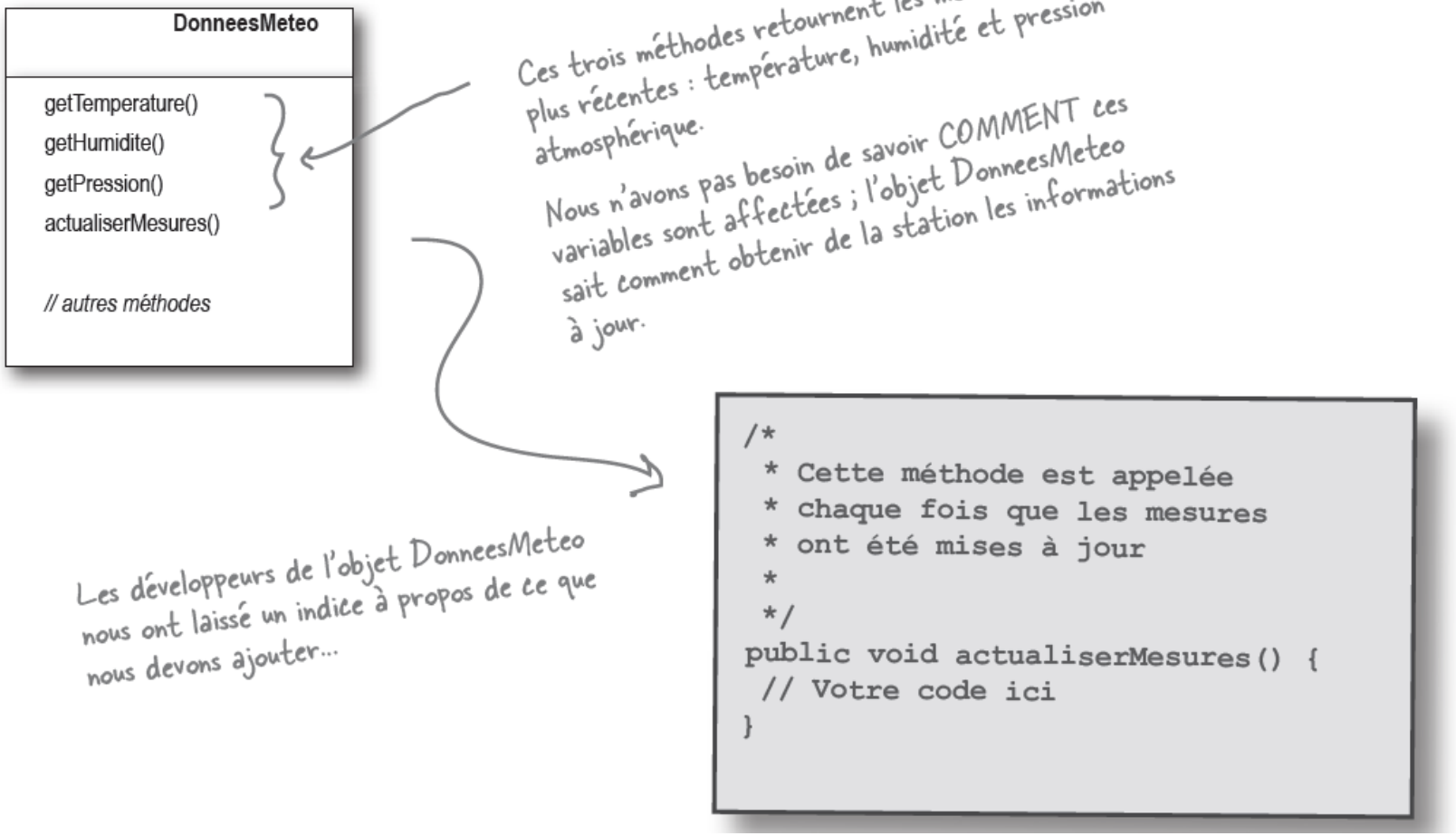


Exemple

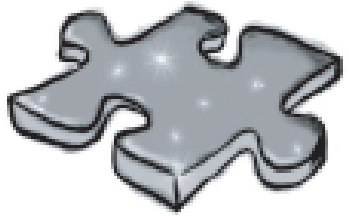


Problèmes de conception

Exemple



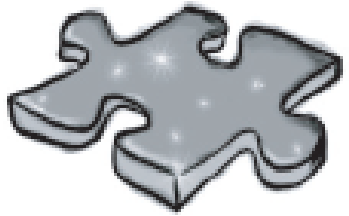
Problèmes de conception



Exemple

```
actualiserMesures(
```

Problèmes de conception



Exemple

```
public class DonneesMeteo {

    // déclaration des variables d'instance

    public void actualiserMesures() {

      float temp = getTemperature();
      float humidite = getHumidite();
      float pressure = getPression();

      affichageConditions.actualiser(temp, humidite, pression);
      affichageStats.actualiser(temp, humidite, pression);
      affichagePrevisions.actualiser(temp, humidite, pression);
    }
    // autres méthodes de DonneesMeteo
}
```

Obtenir les mesures les plus récentes en appelant les méthodes get de Donnees-Meteo (déjà implémentées).

Actualiser les affichages...

Appeler chaque élément pour mettre à jour son affichage en lui transmettant les mesures les plus récentes.

QUiZ

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

(Plusieurs réponses possibles.)

A. Nous codons des implémentations concrètes, non des interfaces.

B. Nous devons modifier le code pour chaque nouvel élément d'affichage.

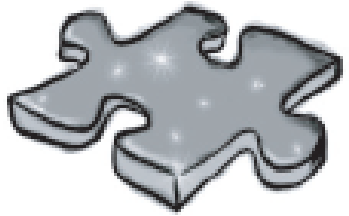
C. Nous n'avons aucun moyen d'ajouter (ou de supprimer) des éléments d'affichage au moment de l'exécution.

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.

Problèmes de conception



Exemple

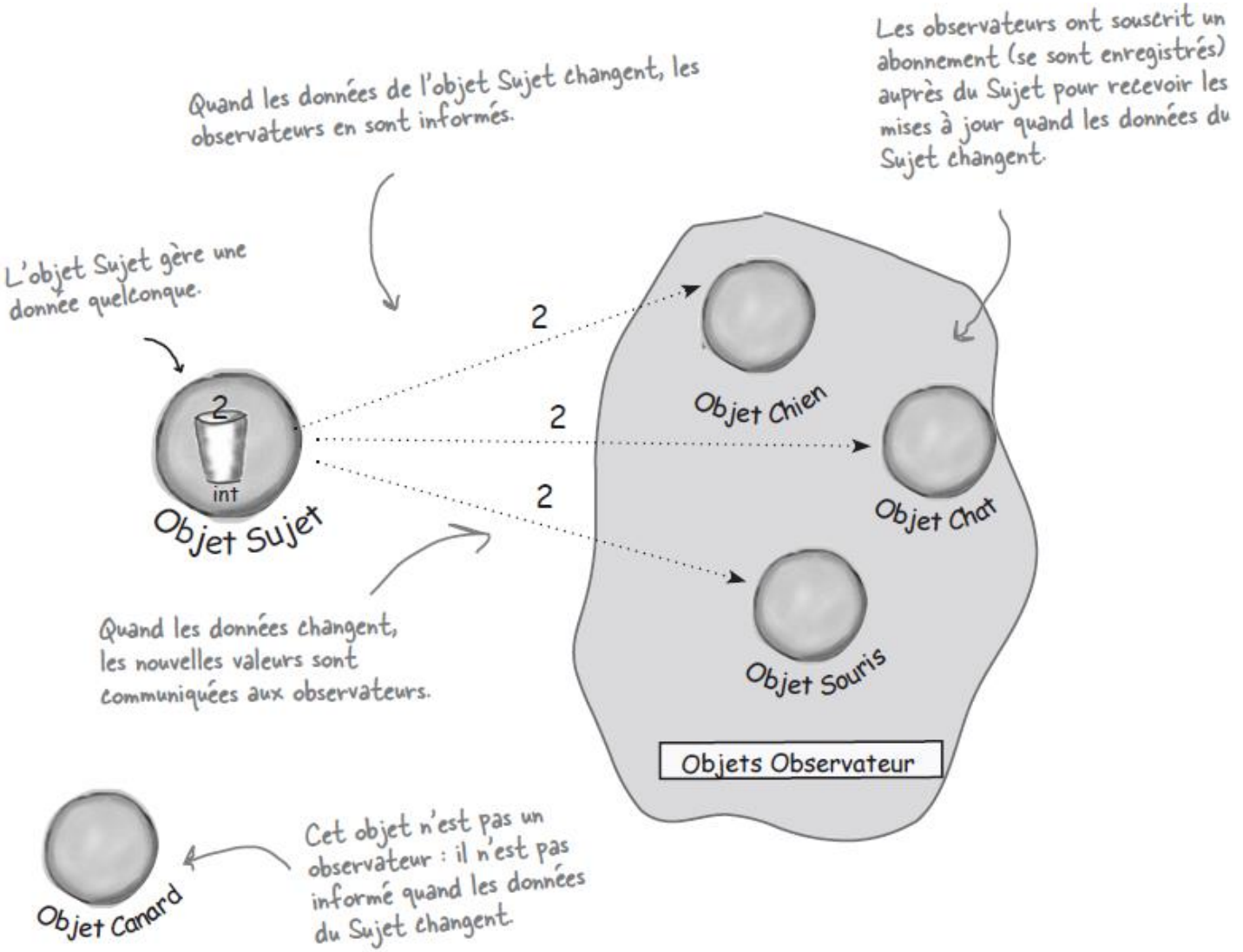
```
public void actualiserMesures() {
   float temp = getTemperature();
   float humidite = getHumidite();
   float pression = getPression();

   affichageConditions.actualiser(temp, humidite, pression);
   affichageStats.actualiser(temp, humidite, pression);
   affichagePrevisions.actualiser(temp, humidite, pression);

}
```

Point de variation : nous devons l'encapsuler.

En codant des implémentations 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 actualiser() qui lit les valeurs de temp, humidite et pression.

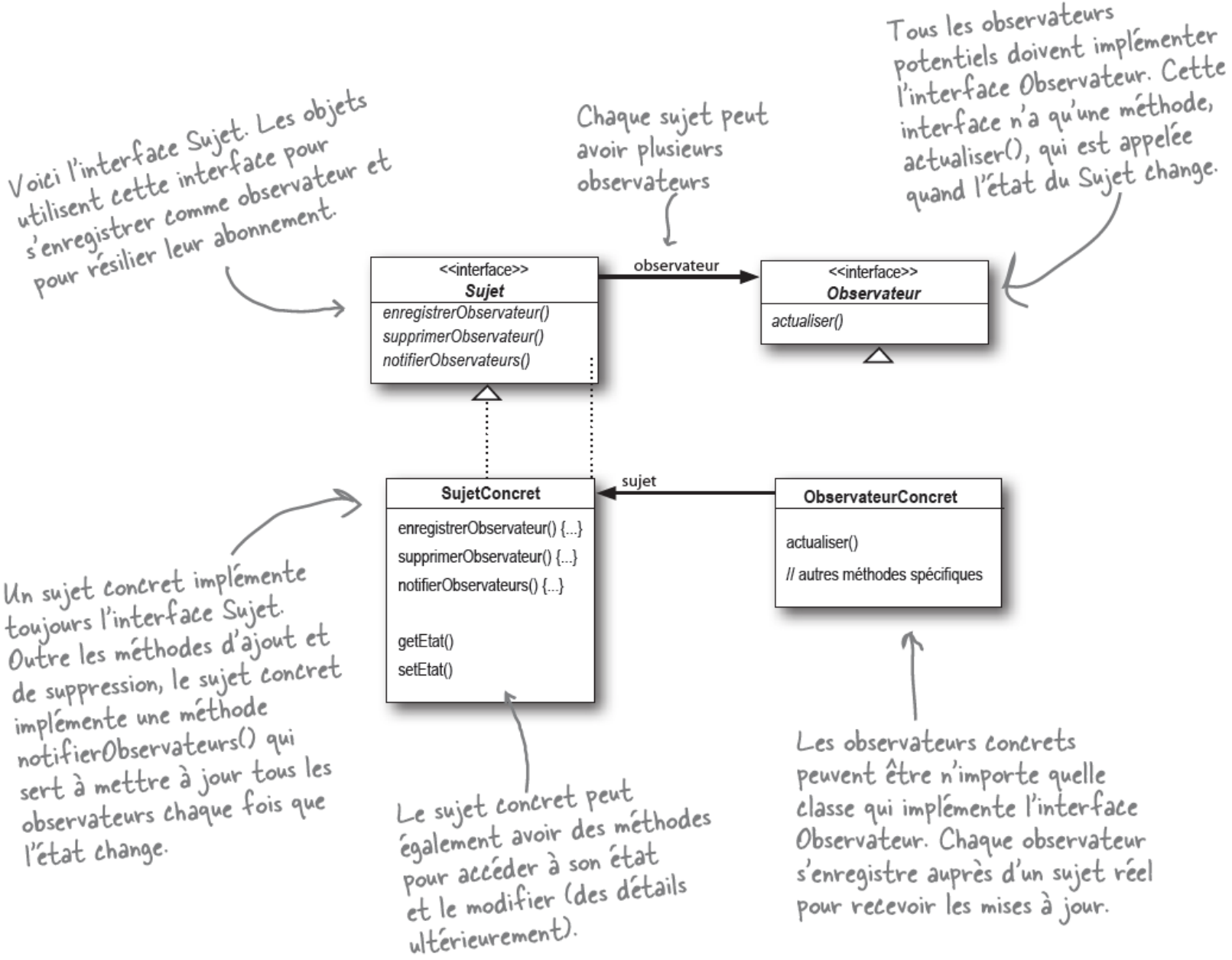
Diffusion + Souscription = pattern Observateur



Diffusion + Souscription = pattern Observateur

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

Diffusion + Souscription = pattern Observateur



Questions

- **Quel est le rapport avec la relation un-à-plusieurs ?**

Questions

- **Et que vient faire la dépendance là-dedans ?**

Couplage faible

- Lorsque deux objets sont faiblement couplés, ils peuvent interagir sans pratiquement se connaître.
- Le pattern Observateur permet une conception dans laquelle le couplage entre sujets et observateurs est faible.

Couplage faible

- Le sujet ne sait qu'une chose à propos de l'observateur : il implémente une certaine interface (l'interface Observateur)
- Nous pouvons ajouter des observateurs à tout moment
- Nous n'avons jamais besoin de modifier le sujet pour ajouter de nouveaux types d'observateurs
- Nous pouvons réutiliser les objets et les sujets indépendamment les uns

des autres

- Les modifications des sujets n'affectent pas les observateurs et inversement

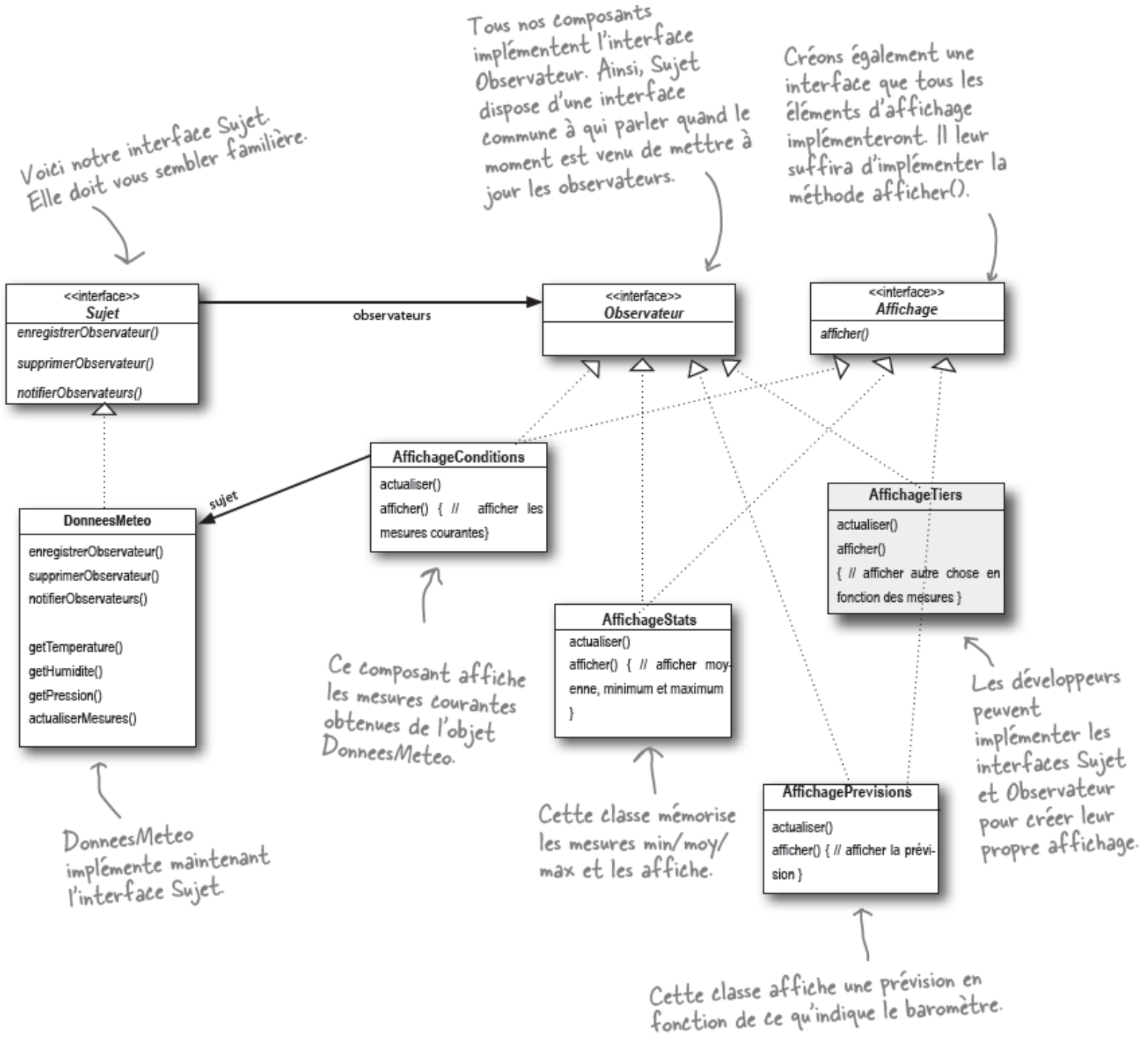
Principe de conception

- *Efforcez-vous de coupler faiblement les objets qui interagissent.*

Exercice

- Essayez d’esquisser les classes nécessaires pour implémenter la Station Météo, notamment la classe DonneesMeteo et ses différents affichages. Vérifiez que votre diagramme montre la façon dont tous les composants s’assemblent et dont un autre développeur pourrait implémenter son propre affichage.

Solutions


Voici notre interface Sujet. Elle doit vous sembler familière.
Tous nos composants implémentent l'interface Observateur. Ainsi, Sujet dispose d'une interface commune à qui parler quand le moment est venu de mettre à jour les observateurs.
Créons également une interface que tous les éléments d'affichage implémenteront. Il leur suffira d'implémenter la méthode afficher().
<<interface>> Sujet
enregistrerObservateur()
supprimerObservateur()
notifierObservateurs()
observateurs
<<interface>> Observateur
<<interface>> Affichage
afficher()
AffichageConditions
actualiser()
afficher() { // afficher les mesures courantes}
sujet
DonneesMeteo
enregistrerObservateur()
supprimerObservateur()
notifierObservateurs()
getTemperature()
getHumidite()
getPression()
actualiserMesures()
AffichageTiers
actualiser()
afficher()
{ // afficher autre chose en fonction des mesures }
AffichageStats
actualiser()
afficher() { // afficher moyenne, minimum et maximum }
AffichagePrevisions
actualiser()
afficher() { // afficher la prévision }
Ce composant affiche les mesures courantes obtenues de l'objet DonneesMeteo.
Les développeurs peuvent implémenter les interfaces Sujet et Observateur pour créer leur propre affichage.
DonneesMeteo implémente maintenant l'interface Sujet.
Cette classe mémorise les mesures min/moy/max et les affiche.
Cette classe affiche une prévision en fonction de ce qu'indique le baromètre.

Implémenter la Station Météo

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

Ces deux méthodes acceptent un Observateur en argument ; autrement dit, l'Observateur à enregistrer ou à supprimer.

Cette méthode est appelée pour notifier tous les observateurs que l'état de Sujet a changé.

```
public interface Observateur {
        public void actualiser(float temp, float humidite, float pression);
}
```

Les valeurs de l'état que les Observateurs obtiennent du Sujet quand une mesure change

L'interface Observateur étant implémentée par tous les observateurs, ils doivent tous implémenter la méthode actualiser(). Ici, nous appliquons l'idée de Marie et Anne et nous transmettons les mesures aux observateurs.

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

L'interface Affichage ne contient qu'une méthode, afficher(), que nous appellerons quand un élément devra être affiché.

Implémenter la Station Météo

```
public class DonneesMeteo implements Subject {
    private ArrayList observateurs;
    private float temperature;
    private float humidite;
    private float pression;

    public DonneesMeteo() {
        observers = new ArrayList();
    }
```

DonneesMeteo implémente maintenant l'interface Sujet.

Nous avons ajouté une ArrayList pour contenir les observateurs et nous la créons dans le constructeur.

Quand un observateur s'enregistre, nous l'ajoutons simplement à la fin de la liste.

Implémenter la Station Météo

Ici, nous implémentons l'interface Sujet.

```
public void enregisterObservateur(Observateur o) {
    observers.add(o)r;
}

public void supprimerObservateur (Observateur o) {
    int i = observateurs.indexOf(o);
    if (i >= 0) {
        observateurs.remove(i);
    }
}

public void notifierObservateurs() {
    for (int i = 0; i < observateurs.size(); i++) {
        Observer observer = (Observer)observers.get(i);
        observer.update(temperature, humidite, pression);
    }
}
```

De même, quand un observateur veut se « désenregistrer » nous le supprimons de la liste.

Voici le plus intéressant. C'est là que nous informons les observateurs de l'état du sujet. Comme ce sont tous des objets Observateur, nous savons qu'ils implémentent tous actualiser() et nous savons donc comment les en notifier.

Implémenter la Station Météo

```
    public void actualiserMesures() {
        notifierObservateurs();
    }

    public void setMesures(float temperature, float humidite, float pression) {
        this.temperature = temperature;
        this.humidite = humidite;
        this.pression = pression;
        actualiserMesures();
    }

    // autres méthodes de DonneesMeteo
}
```

Nous notifions les observateurs quand nous recevons de la Station Météo des mesures mises à jour.

Bon, nous voulions accompagner chaque exemplaire de ce livre d'une jolie petite station météo mais l'éditeur a refusé. Au lieu de lire les données sur une vraie station, nous allons donc utiliser cette méthode pour tester nos affichages. Ou bien, pour le plaisir, vous pouvez écrire un programme qui récupère des relevés sur le web.

Implémenter la Station Météo

```
    public void actualiserMesures() {
        notifierObservateurs();
    }

    public void setMesures(float temperature, float humidite, float pression) {
        this.temperature = temperature;
        this.humidite = humidite;
        this.pression = pression;
        actualiserMesures();
    }

    // autres méthodes de DonneesMeteo
}
```

Nous notifions les observateurs quand nous recevons de la Station Météo des mesures mises à jour.

Bon, nous voulions accompagner chaque exemplaire de ce livre d'une jolie petite station météo mais l'éditeur a refusé. Au lieu de lire les données sur une vraie station, nous allons donc utiliser cette méthode pour tester nos affichages. Ou bien, pour le plaisir, vous pouvez écrire un programme qui récupère des relevés sur le web.

Implémenter la Station Météo

Cet affichage implémente Observateur pour pouvoir obtenir les changements de l'objet DonneesMeteo.

Il implémente également Affichage car notre API va avoir besoin de tous les éléments pour implémenter cette interface.

```
public class AffichageConditions implements Observateur, Affichage {
    private float temperature;
    private float humidite;
    private Sujet donneesMeteo;

    public AffichageConditions(Sujet donneesMeteo) {
        this.donneesMeteo = donneesMeteo;
        donneesMeteo.enregistrerObservateur(this);
    }

    public void actualiser(float temperature, float humidite, float pression) {
        this.temperature = temperature;
        this.humidite = humidite;
        afficher();
    }

    public void afficher() {
        System.out.println("Conditions actuelles : " + temperature
        + " degrés C et " + humidite + " % d'humidité");
    }}
```

Nous transmettons au constructeur l'objet DonneesMeteo (le Sujet) et nous l'utilisons pour enregistrer l'affichage en tant qu'observateur.

Quand actualiser() est appelée, nous sauvegardons température et humidité et nous appelons afficher().

La méthode afficher() affiche simplement les mesures de température et d'humidité les plus récentes.

Implémenter la Station Météo

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

        AffichageConditions affichageCond = new AffichageConditions(donneesMeteo);

        AffichageStats affichageStat = new AffichageStats(donneesMeteo);
        AffichagePrevisions affichagePrev = new AffichagePrevisions(donneesMeteo);

        donneesMeteo.setMesures(26, 65, 1020);
        donneesMeteo.setMesures(28, 70, 1012);
        donneesMeteo.setMesures(22, 90, 1012);
    }
}
```

Créons d'abord l'objet DonneesMeteo.

Si vous ne voulez pas télécharger le code, vous pouvez mettre ces deux lignes en commentaire et l'exécuter.

Créer les trois affichages et leur transmettre l'objet DonneesMeteo.

Simuler les nouvelles mesures.

Exercice

Le PDG de MétéoExpress, Jean-Loup Ragan, vient de vous appeler : ils ne peuvent pas commercialiser leur station sans un affichage de l'humidex. Voici les détails :

L'humidex est un indice qui combine la température et l'humidité afin de déterminer la température apparente (la chaleur réellement ressentie). Pour le calculer, on prend la température, T, et l'humidité relative (HR) et on applique cette formule :

$$\begin{aligned}\text{humidex} = {} & 16.923 + 1.85212 * 10^{-1} * T + 5.37941 * HR - 1.00254 * 10^{-1} * T \\ & * HR + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * HR^2 + 3.45372 * 10^{-4} \\ & * T^2 * HR - 8.14971 * 10^{-4} * T * HR^2 + 1.02102 * 10^{-5} * T^2 * HR^2 - \\ & 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * HR^3 + 1.42721 * 10^{-6} * T^3 * HR \\ & + 1.97483 * 10^{-7} * T * HR^3 - 2.18429 * 10^{-8} * T^3 * HR^2 + 8.43296 * \\ & 10^{-10} * T^2 * HR^3 - 4.81975 * 10^{-11} * T^3 * HR^3\end{aligned}$$

Transmission de l’état mis à jour aux observateurs

- Ils ont pensé que transmettre directement les mesures aux observateurs était la méthode la plus simple pour mettre à jour l’état. Pensez-vous que cela soit judicieux ?
- Indication : est-ce un point de l’application susceptible de changer dans le futur ? Si c’est le cas, le changement sera-t-il bien encapsulé ou entraînera-t-il des modifications dans de nombreuses parties du code ?
- Voyez-vous d’autres façons de résoudre le problème de la transmission de l’état mis à jour aux observateurs ?

Transmission de l'état mis à jour aux observateurs

Sujet

Oui, je pourrais vous laisser **tirer** mon état. Mais est-ce que ce ne serait pas moins pratique pour vous ? Si vous devez me contacter chaque fois que vous voulez quelque chose, il vous faudra une quantité d'appels de méthodes pour obtenir tout l'état dont vous avez besoin. C'est pourquoi je préfère **pousser**... En procédant ainsi, vous avez tout ce qu'il vous faut dans une seule notification.

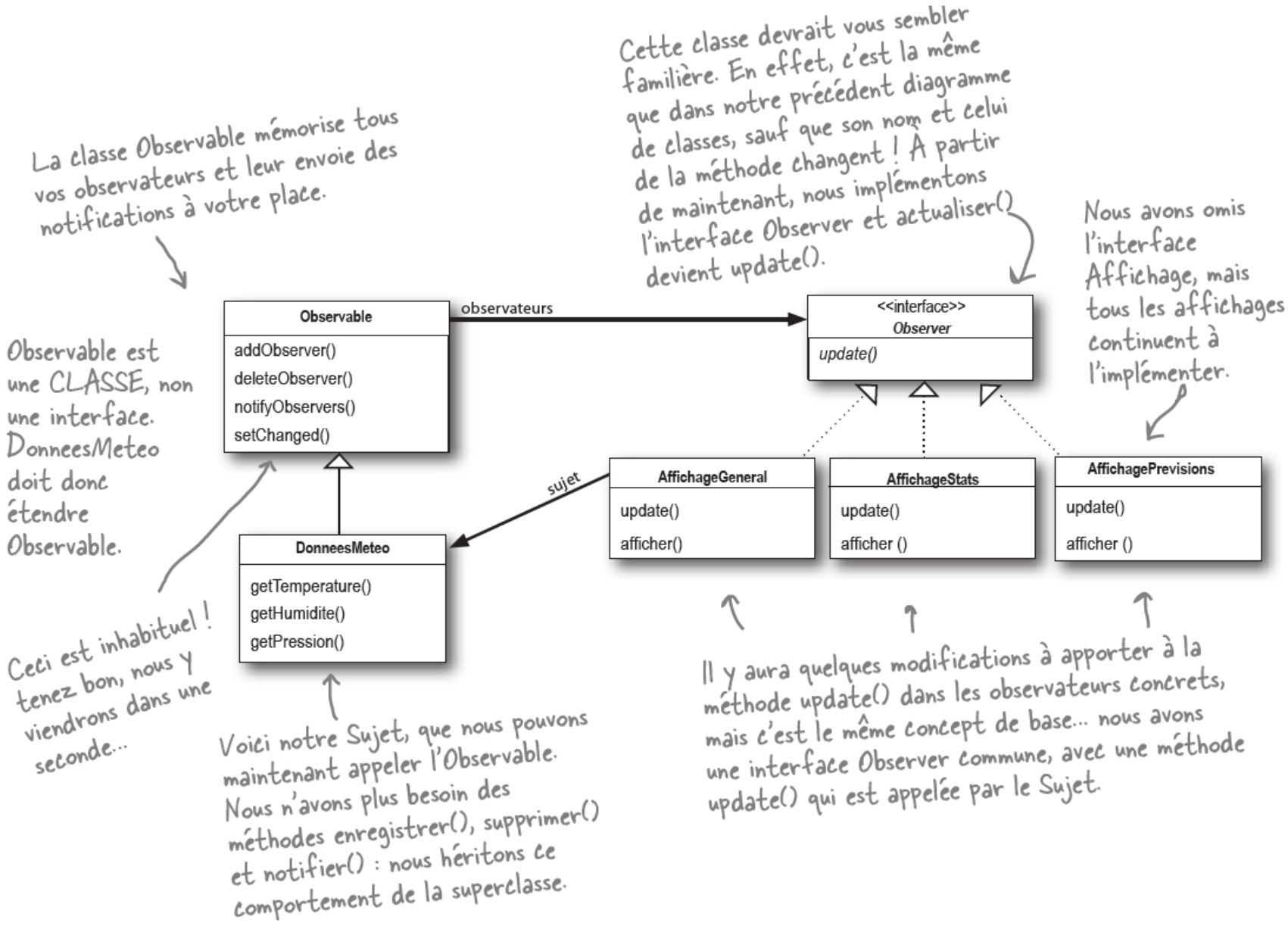
Observateur

Pourquoi ne pas simplement écrire des méthodes get publiques qui nous permettraient de retirer l'état dont nous avons besoin ?

Ne vous poussez pas du col comme ça ! Il y a tellement de types d'observateurs différents que vous n'avez aucun moyen d'anticiper ce dont nous avons besoin. Laissez-nous venir vers vous pour obtenir votre état. Comme ça, si certains d'entre nous n'ont besoin que d'une petite partie de l'état, nous ne sommes pas forcés de l'avoir en entier. Cela faciliterait aussi les modifications. Imaginons que vous évoluez et que vous ajoutez quelque chose à l'état. Si nous tirons, il devient inutile de modifier les appels des méthodes de mise à jour sur chaque observateur. Il suffit de vous modifier pour permettre à d'autres méthodes get d'accéder à l'état supplémentaire.

Solutions

Le pattern Observateur de Java



Solutions

Le pattern Observateur de Java

Pour qu'un objet devienne Observateur

Comme d'habitude, implémenter l'Interface observateur (cette fois java.util.Observer) et appeler la méthode addObserver() de l'objet Observable. De même, pour supprimer un observateur, il suffit d'appeler deleteObserver().

Pour que l'Observable envoie des notifications

Vous devez d'abord appeler la méthode setChanged() pour signifier que l'état de votre objet a changé

Puis vous appelez l'une des deux méthodes notifyObservers() :

Soit `notifyObservers()` soit `notifyObservers(Object arg)`

Pour qu'un observateur reçoive des notifications

```
update(Observable o, Object arg)
```

Solutions

Le pattern Observateur de Java

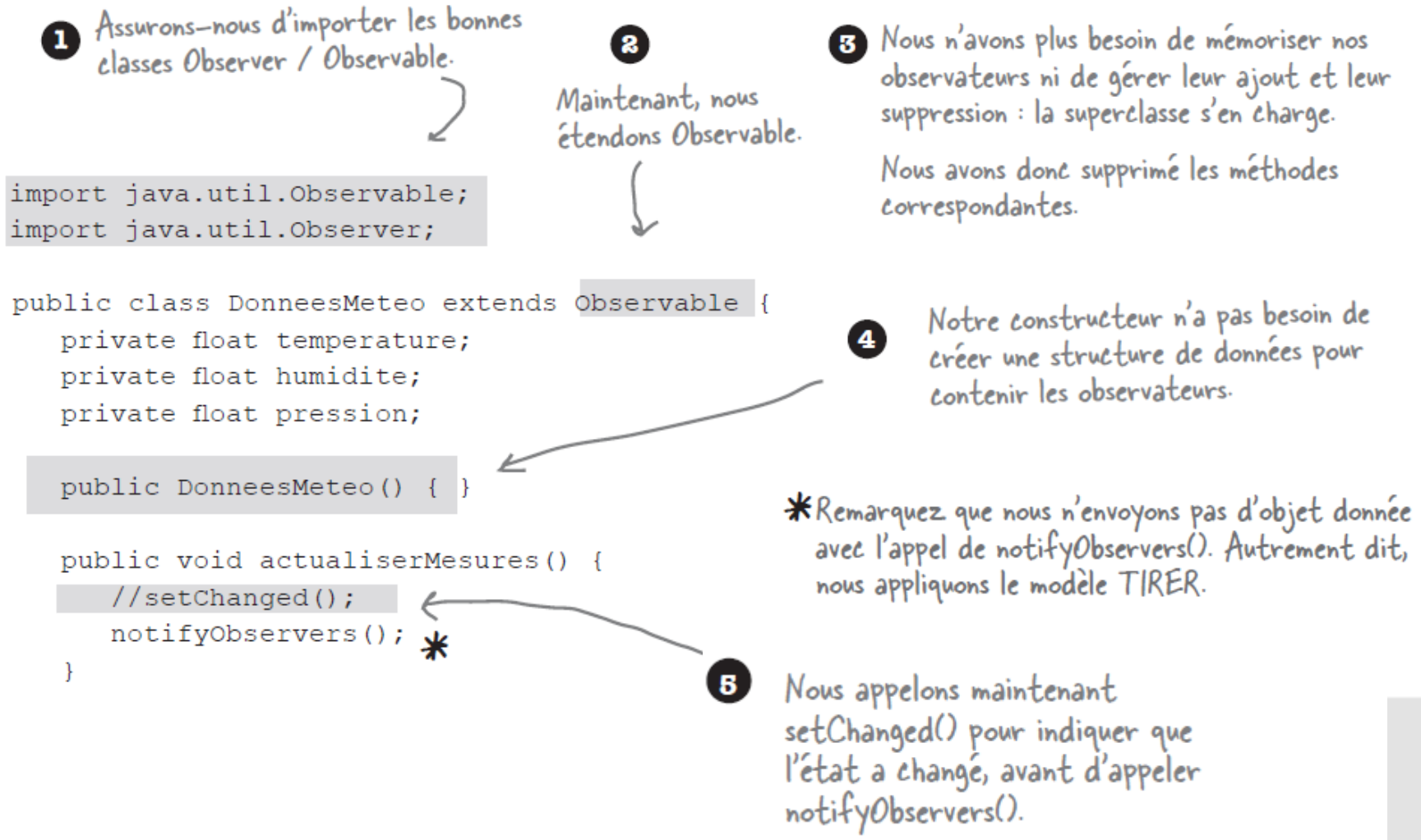
```
setChanged() {
    changed = true
}
notifyObservers(Object arg) {
    if (changed) {
        pour chaque observateur de la liste{
          appeler update(this, arg)
        }
        changed = false
}
}
notifyObservers() {
   notifyObservers(null)
}
```

La méthode setChanged() positionne un drapeau "changed" à true.

notifyObservers() ne notifie les observateurs que si la valeur du drapeau est TRUE.

Puis elle notifie les observateurs et remet le drapeau à false.

Récrivons d'abord DonneesMeteo pour qu'elle étende java.util.Observable



Récrivons d'abord DonneesMeteo pour qu'elle étende java.util.Observable

```
    public void setMesures(float temperature, float humidite, float pression) {
        this.temperature = temperature;
        this.humidite = humidite;
        this.pression = pression;
        actualiserMesures();
    }

    public float getTemperature() {
        return temperature;
    }

    public float getHumidite() {
        return humidite;
    }

    public float getPression() {
        return pression;
     }
}
```

6 Ces méthodes ne sont pas nouvelles, mais comme nous allons "tirer" nous avons pensé à vous rappeler leur existence. Les observateurs les utiliseront pour accéder à l'état de l'objet DonneesMeteo.

Récrivons d'abord DonneesMeteo pour qu'elle étende java.util.Observable

1 Nous importons de nouveau les bonnes classes Observer / Observable.

2 Nous implémentons maintenant l'interface java.util.Observer

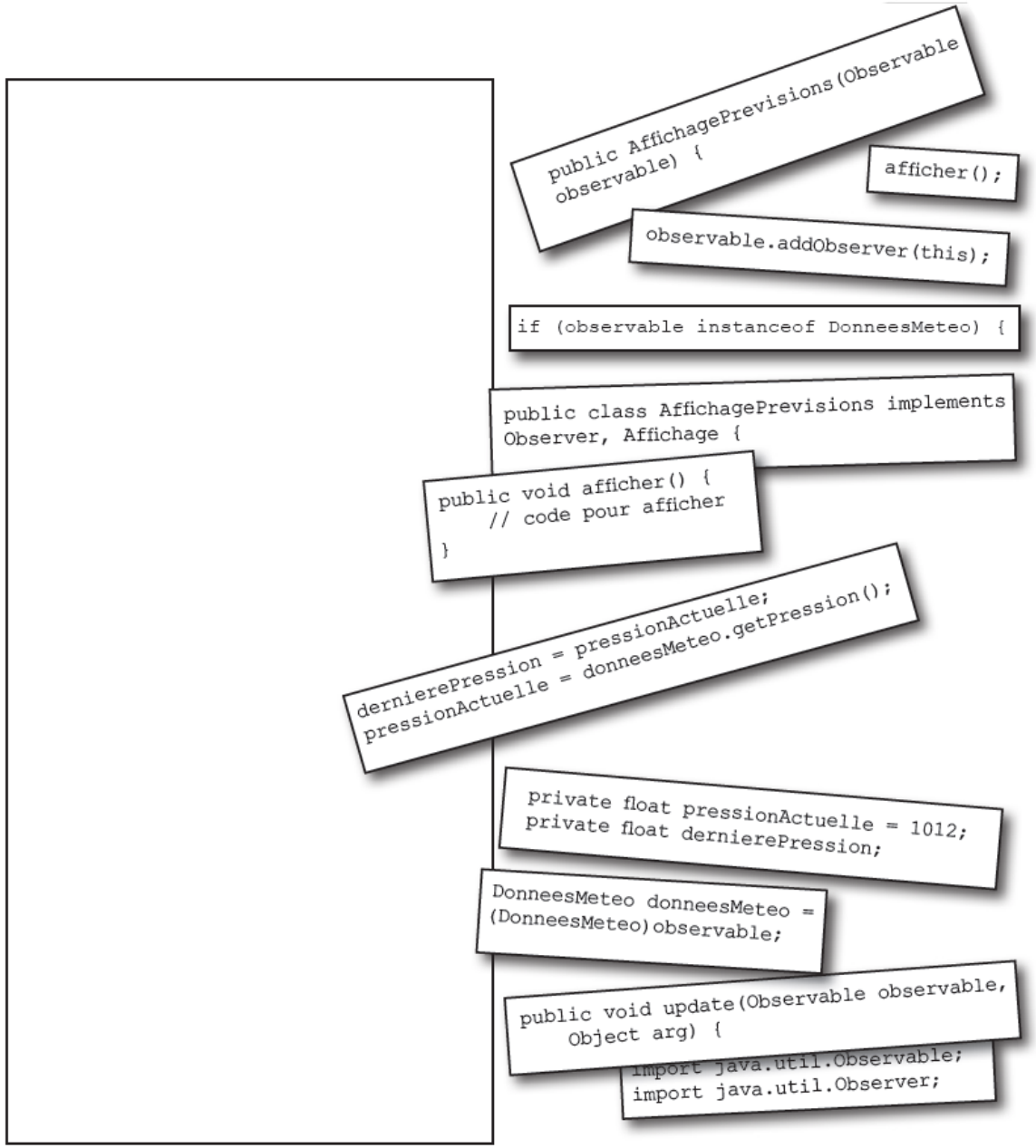
```
import java.util.Observable;
import java.util.Observer;

public class AffichageConditions implements Observer, Affichage {
   Observable observable;
   private float temperature;
   private float humidite;

   public AffichageConditions(Observable observable) {
      this.observable = observable;
      observable.addObserver(this);
   }

   public void update(Observable obs, Object arg) {
      if (obs instanceof DonneesMeteo) {
         DonneesMeteo donneesMeteo = (DonneesMeteo)obs;
         this.temperature = donneesMeteo.getTemperature();
      this.humidite = donneesMeteo.getHumidite();
         afficher();
      }
   }

   public void afficher() {
      System.out.println("Conditions actuelles : " + temperature
         + " degrés C et " + humidite + " % d'humidité");
   }
}
```

3 Notre constructeur accepte maintenant un Observable et nous utilisons « this » pour enregistrer l'objet en tant qu'observateur.

4 Nous avons modifié la méthode update() pour qu'elle accepte en argument à la fois un Observable et des données optionnelles.

5 Dans update(), nous nous assurons d'abord que l'observable est de type DonneesMeteo, puis nous appelons ses méthodes get pour obtenir la température et l'humidité. Puis nous appelons afficher().

Exercice


public AffichagePrevisions(Observable observable) {
afficher();
observable.addObserver(this);
if (observable instanceof DonneesMeteo) {
public class AffichagePrevisions implements Observer, Affichage {
public void afficher() {
// code pour afficher
}
dernierePression = pressionActuelle;
pressionActuelle = donneesMeteo.getPression();
Private float pressionActuelle = 1012;
Private float dernierePression;
DonneesMeteo donneesMeteo = (DonneesMeteo)observable;
public void update(Observable observable, Object arg) {
import java.util.Observable;
import java.util.Observer;

Principe de conception

- Encapsulez ce qui varie.
- Préférez la composition à l'héritage.
- Programmez des interfaces, non des implémentations.
- **Efforcez-vous de coupler faiblement les objets qui interagissent.**

QUiZ

Pour chaque principe de conception, dites comment le pattern Observateur l'applique.

Principe de conception

Identifiez les aspects de votre application qui varient et séparez-les de ce qui demeure inchangé.

Principe de conception

Programmez des interfaces, non des implémentations.

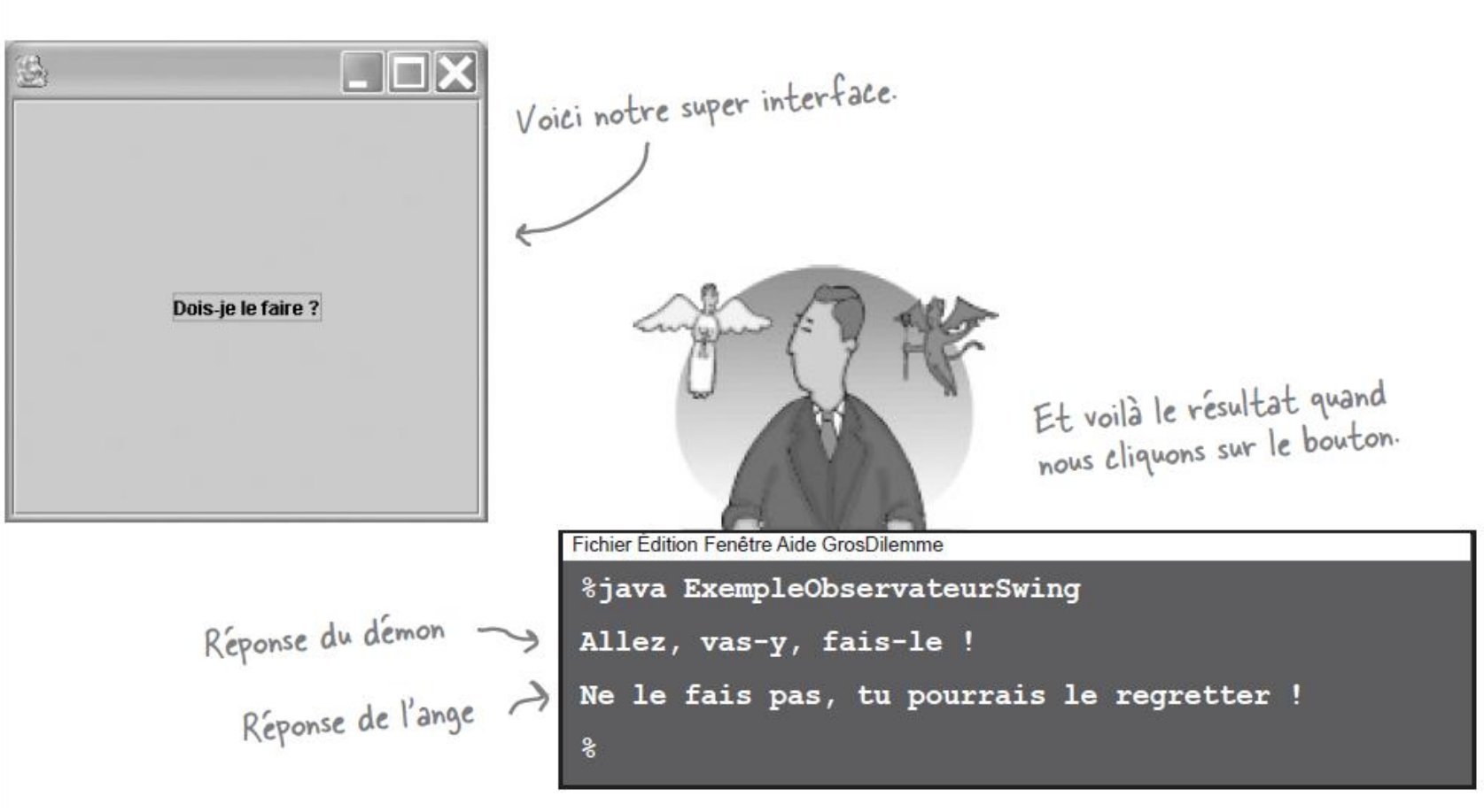
Principe de conception

Préférez la composition à l'héritage.

l'API Swing possède également son propre implémentation du pattern

- Si vous regardez rapidement la superclasse de JButton, AbstractButton, vous verrez qu'elle a un tas de méthodes « add » et « remove ». Ces méthodes vous permettent d'ajouter et de supprimer des observateurs,ou, comme on les appelle dans la terminologie Swing, des auditeurs, afin d'être à l'écoute des différents types d'événements qui se produisent dans le composant Swing. Par exemple, un ActionListener vous permet d' «écouter» tous les types d'actions qui peuvent être appliquées à un bouton, comme les clics. Vous trouverez différents types d'auditeurs dans toute l'API Swing.

l'API Swing possède également son propre implémentation du pattern



l'API Swing possède également son propre implémentation du pattern

Simple application Swing qui se borne à créer un cadre et à y insérer un bouton.

```
public class ExempleObservateurSwing {
  JFrame cadre;

  public static void main(String[] args) {
    ExempleObservateurSwing exemple = new ExempleObservateurSwing ();
    exemple.go();
  }

  public void go() {
    cadre = new JFrame();
    JButton bouton = new JButton("Dois-je le faire ?");
    bouton.addActionListener(new AuditeurAnge());
    bouton.addActionListener(new AuditeurDemon());
    cadre.getContentPane().add(BorderLayout.CENTER, bouton);
    // Défnir les propriétés du cadre ici
  }

class AuditeurAnge implements ActionListener {
   public void actionPerformed(ActionEvent event) {
      System.out.println("Ne le fais pas, tu pourrais le regretter!");
   }
 }

 class AuditeurDemon implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      System.out.println("Allez, vas-y, fais-le ");

}
}
}
```

Créer les objets ange et démon (les observateurs) qui écoutent les événements du bouton.

Voici les définitions de classes des observateurs. Ce sont des classes internes (mais ce n'est pas obligatoire).

Au lieu d'update(), la méthode actionPerformed() est appelée quand l'état du sujet (en l'occurrence le bouton) change.

EXERCICE

- Vous devez concevoir une application qui suit les variations des prix d'actions boursières.
- Les traders peuvent s'abonner à des actions spécifiques.
- Chaque fois qu'une action change de prix, tous les traders abonnés à cette action doivent être notifiés de la mise à jour.

Gestion des données dynamiques avec Angular

- Vous devez concevoir un système de gestion de tâches dans une application Angular. Cette application dispose de deux composants principaux :

1. **Liste des tâches** : qui affiche toutes les tâches en temps réel.
2. **Compteur de tâches** : qui montre le nombre total de tâches en temps réel.

- Chaque fois qu'une tâche est ajoutée ou supprimée, les deux composants doivent être mis à jour automatiquement sans rafraîchir la page. Vous devez modéliser le fonctionnement sous-jacent de cette application