M3105 : Conception et programmation objet avancées

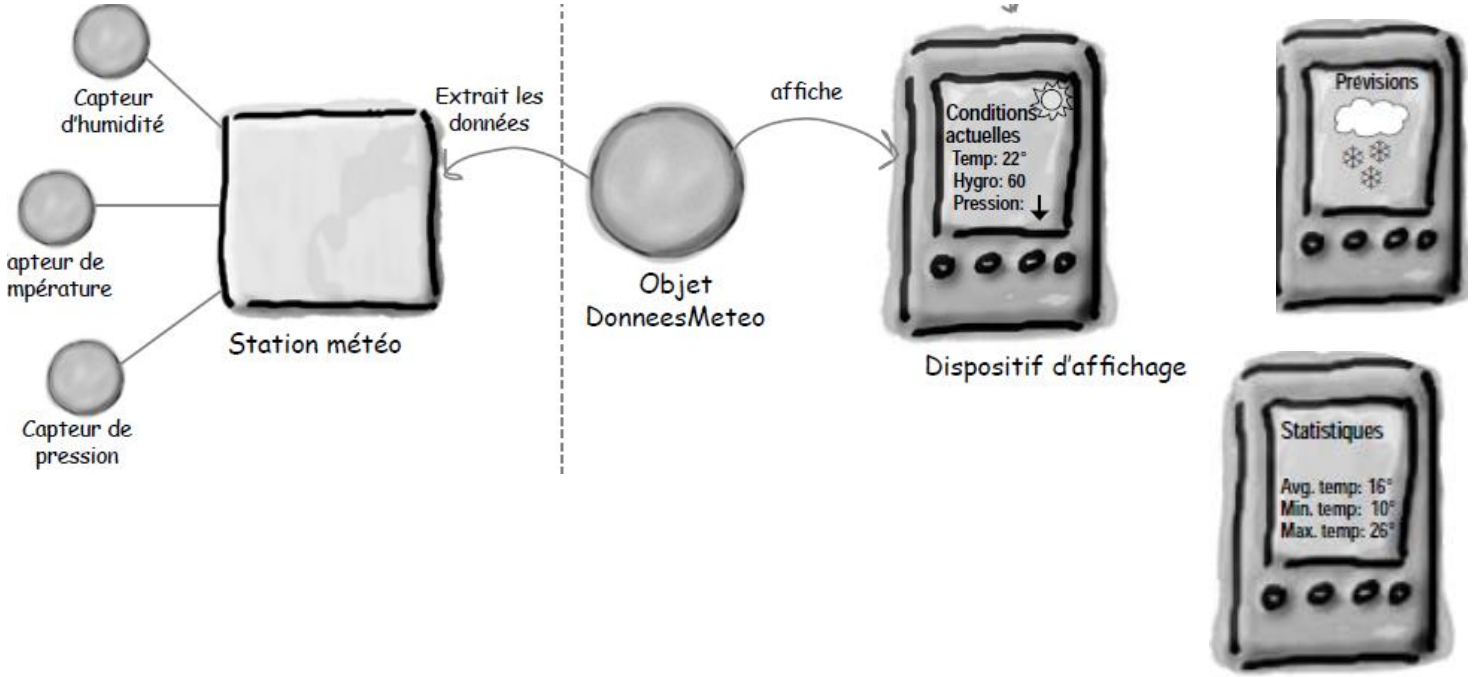
Contenus

I. Introduction – bilan de S2
II. Compléments en java - les collections
III. Patrons de conception
    Introduction
    1. Stratégie
    2. Adaptateur
    3. Itérateur
    4. Décorateur
    5. Observateur
IV. Eléments d'architecture logicielle
    1. MVC

III.5. Observateur (Observer)

Partons d'un exemple concret …

- Une station météo fournit des données qui sont actualisées régulièrement et 3 affichages différents doivent être générés



Source : référence 4– E. Freeman

Un premier essai d'implémentation …

```
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.

Source : référence 4– E. Freeman

Quelles critiques pouvons-nous faire ?

```
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.

Source : référence 4– E. Freeman

Quelles critiques pouvons-nous faire ?

```
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.

Source : référence 4– E. Freeman

Quelles critiques pouvons-nous faire ?

```
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...

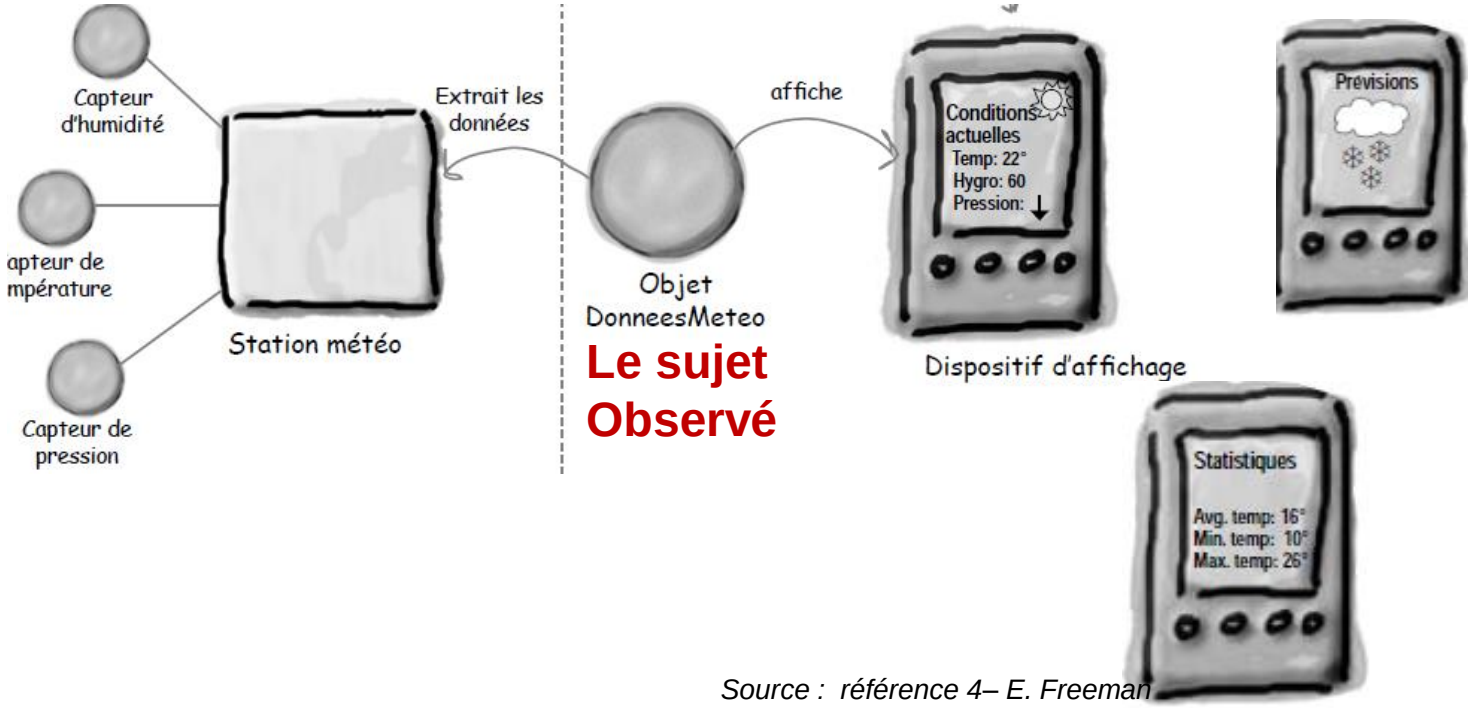
En codant des implémentation concrètes (affichageXXX), il n'y a pas moyen d'ajouter ou de supprimer des éléments sans modifier le programme.

La méthode actualiser est commune aux affichages → interface commune est à envisager ?

→ **une solution est proposée avec le patron Observateur**

Partons d'un exemple concret …

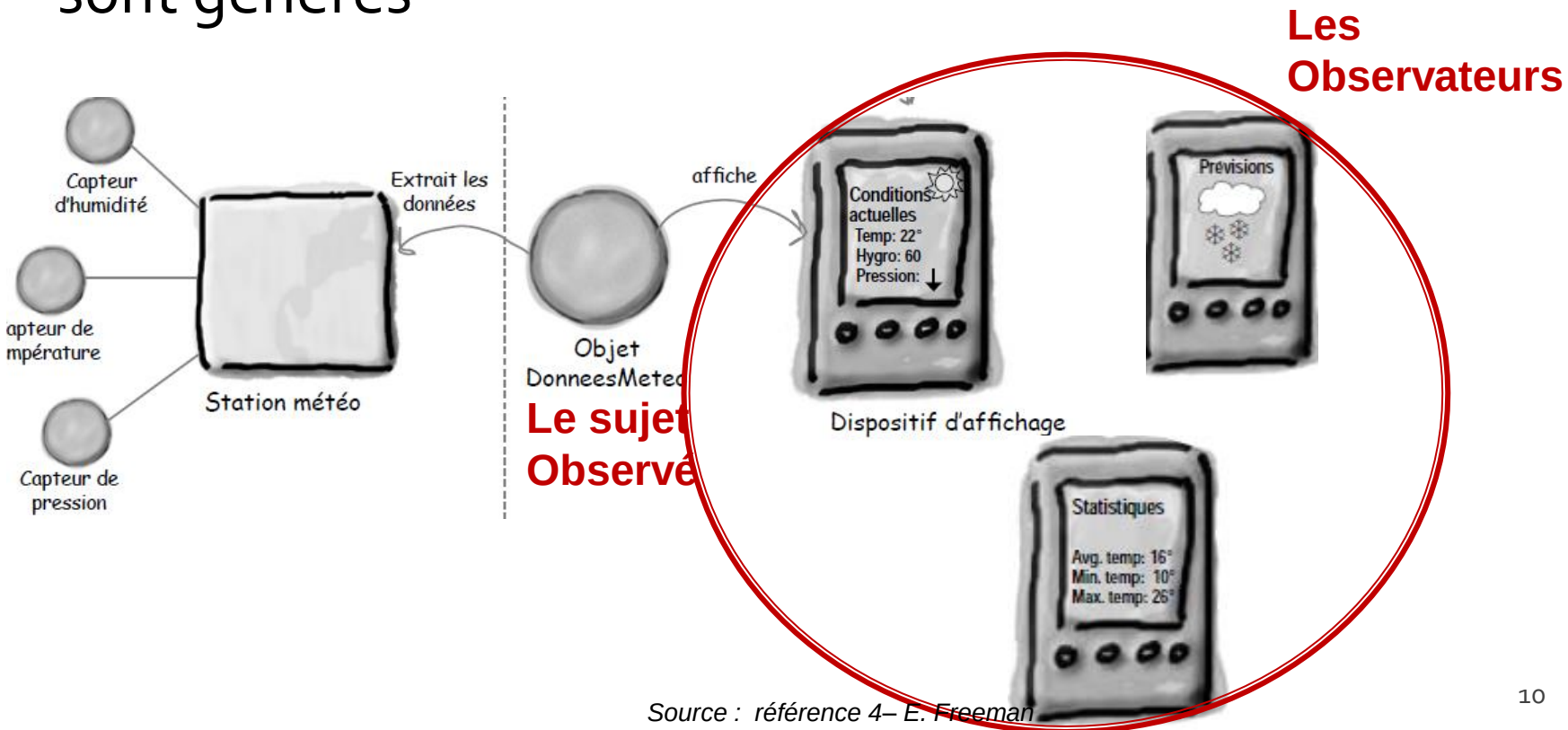
- Une station météo fournit des données qui sont actualisées régulièrement et 3 affichages différents sont générés



Source : référence 4– E. Freeman

Partons d'un exemple concret …

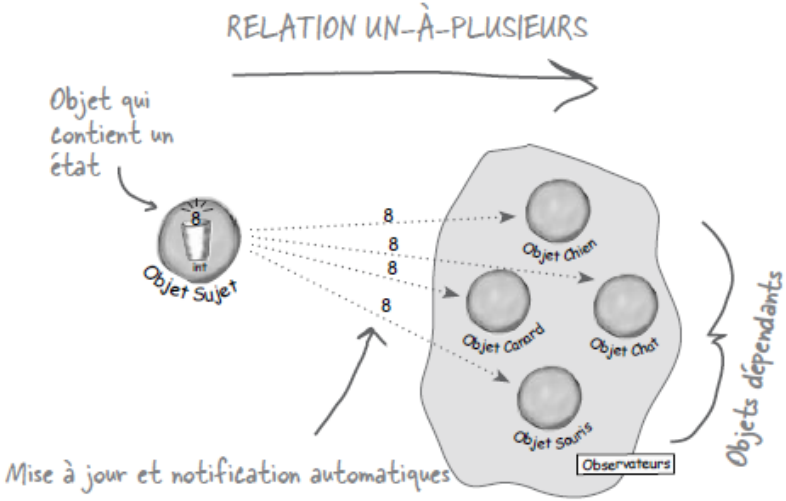
- Une station météo fournit des données qui sont actualisées régulièrement et 3 affichages différents sont générés



Source : référence 4– E. Freeman

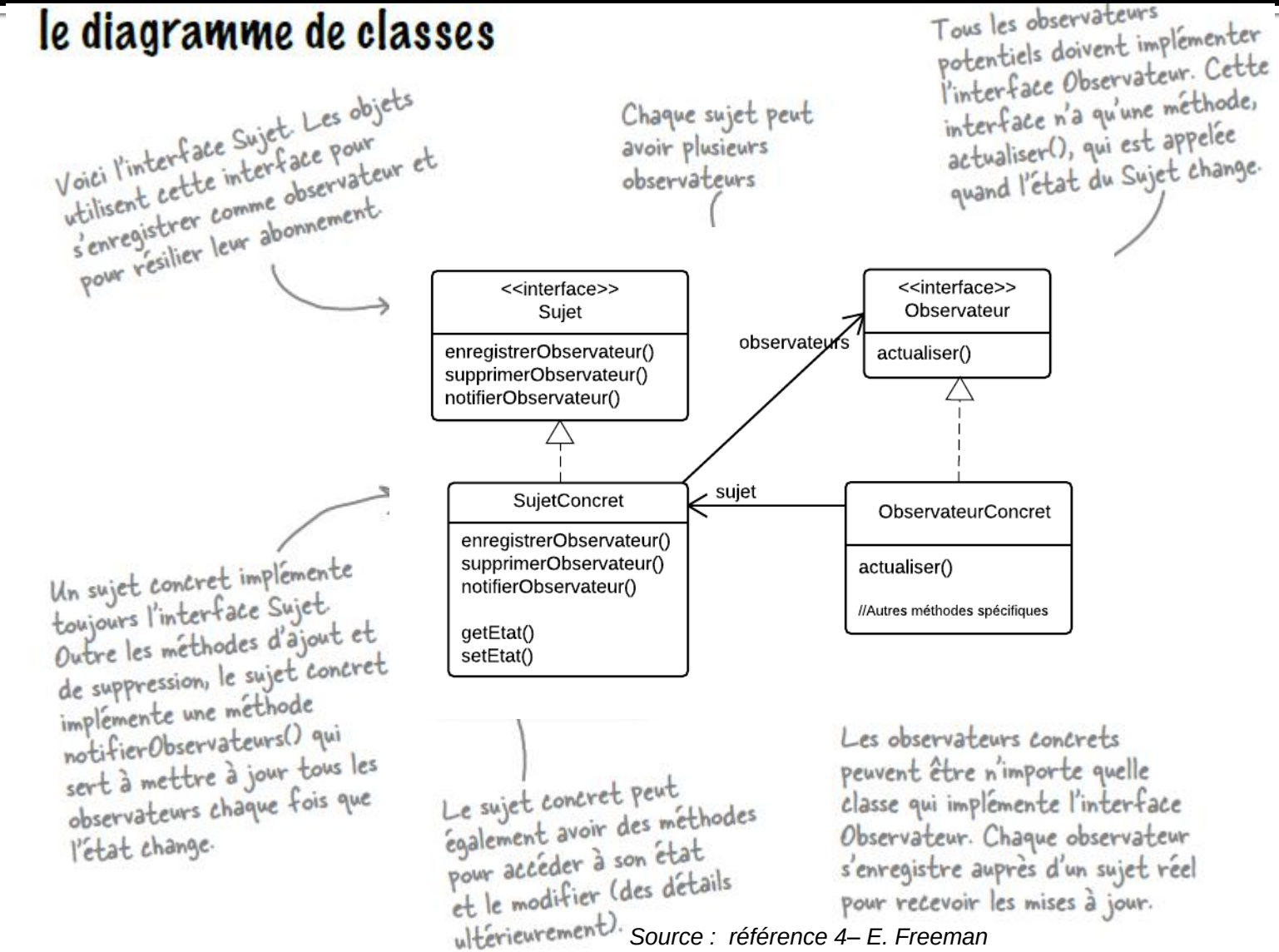
Le patron Observateur (Observer)

- Catégorie
  - Patron de comportement
- Raison d'être
  - Définir une dépendance de un (**le sujet observé**) à plusieurs (**les observateurs**).
    - Quand le sujet observé change d'état, les observateurs qui se sont enregistrés auprès de lui sont notifiés et mis à jour.



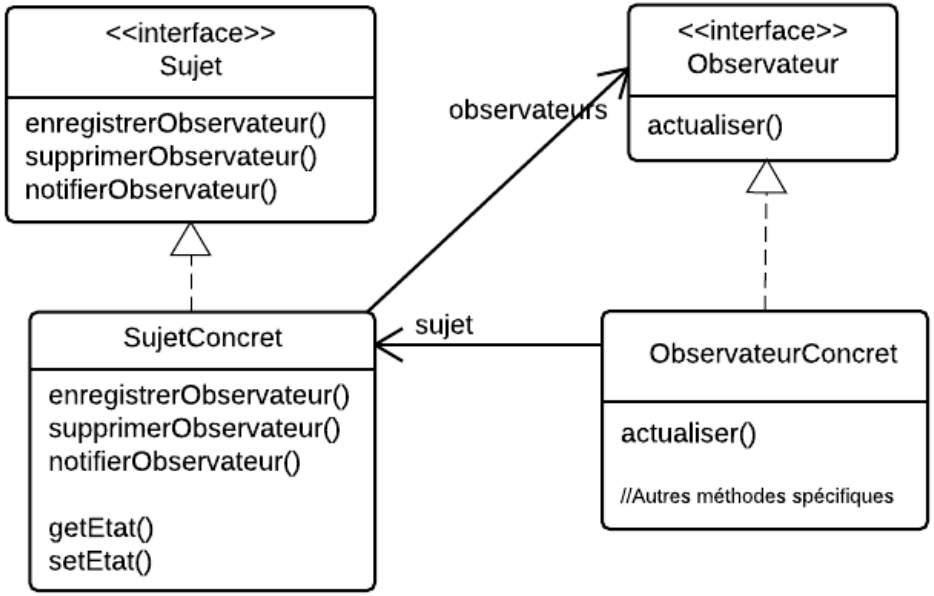
Source : référence 4– E. Freeman

Le patron Observateur (Observer)



Source : référence 4– E. Freeman

Le patron Observateur (Observer)



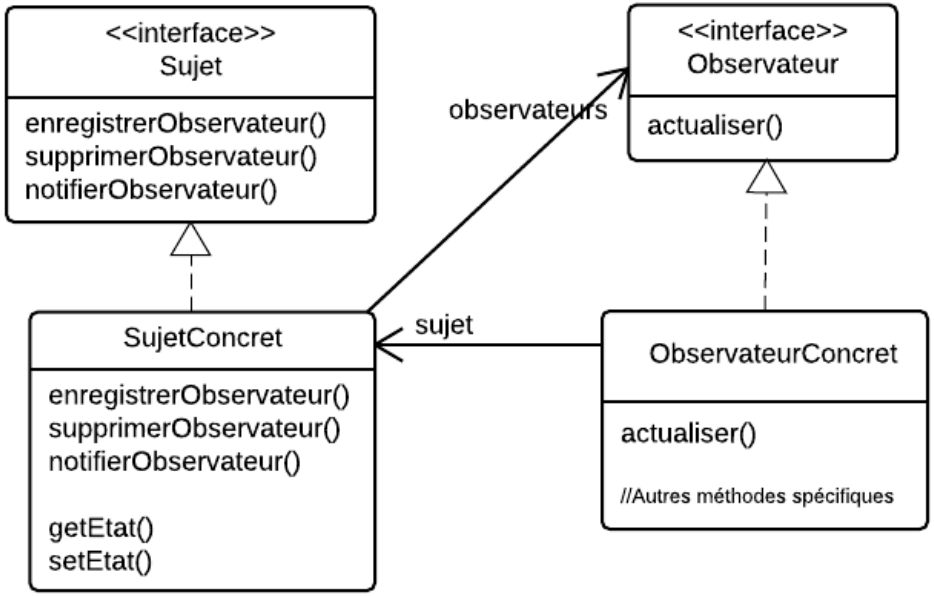
- **Faible couplage**
  - Lorsque deux objets sont faiblement couplés, ils peuvent interagir sans pratiquement se connaître

En effet :

- Le sujet ne sait qu'une chose à propos de l'observateur : il implémente l'interface *Observateur*
- Des observateurs peuvent être ajoutés à tout moment
- Les modifications de code de Sujet n'affectent pas les observateurs et réciproquement

Source : référence 4– E. Freeman

Le patron Observateur (Observer)



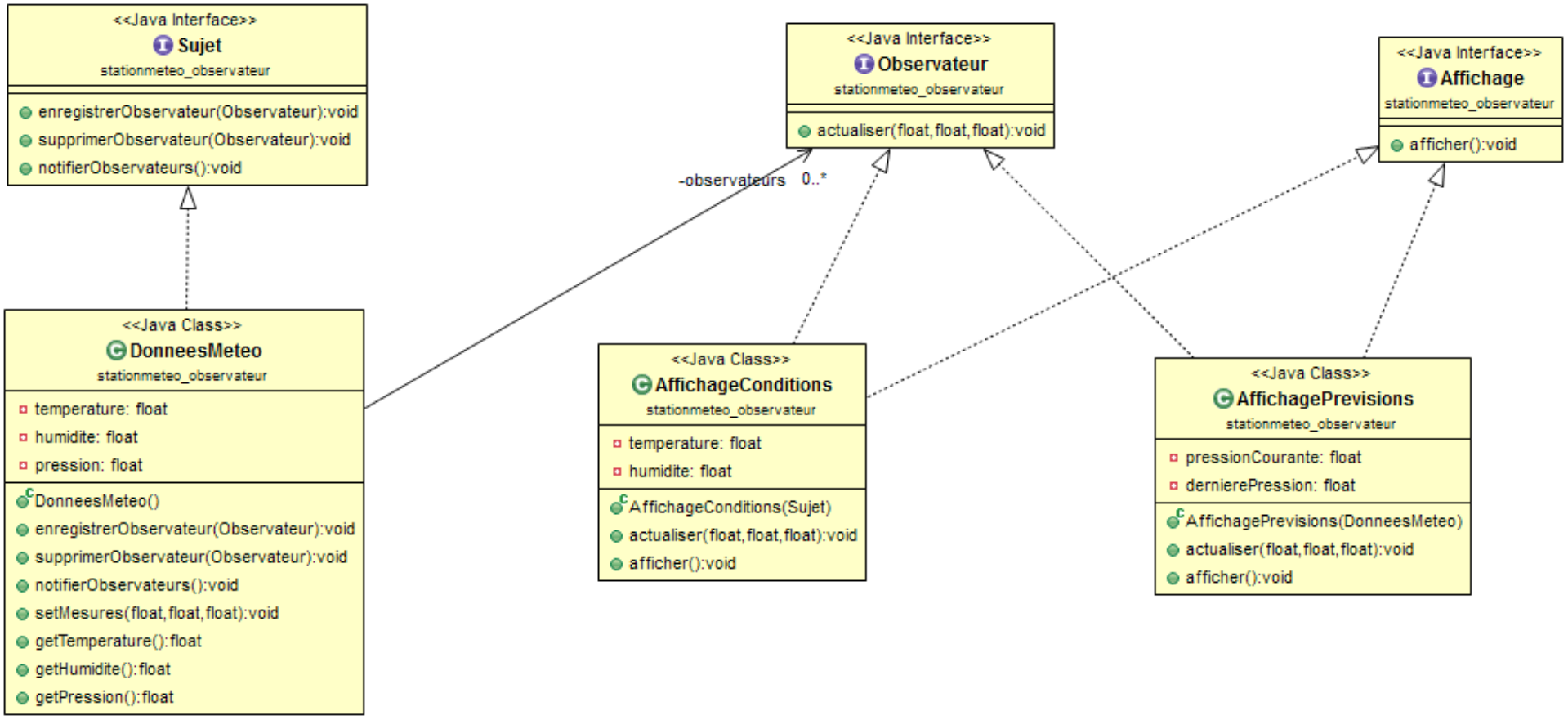
- **Faible couplage**
  - Lorsque deux objets sont faiblement couplés, ils peuvent interagir sans pratiquement se connaître

En effet :

- Le sujet ne sait qu'une chose à propos de l'observateur : il implémente l'interface *Observateur*
- Des observateurs peuvent être ajoutés à tout moment
- Les modifications de code de Sujet n'affectent pas les observateurs et réciproquement

Les conceptions faiblement couplées permettent de construire des systèmes souples, capables de faire face aux changements car ils minimisent l'interdépendance entre les objets

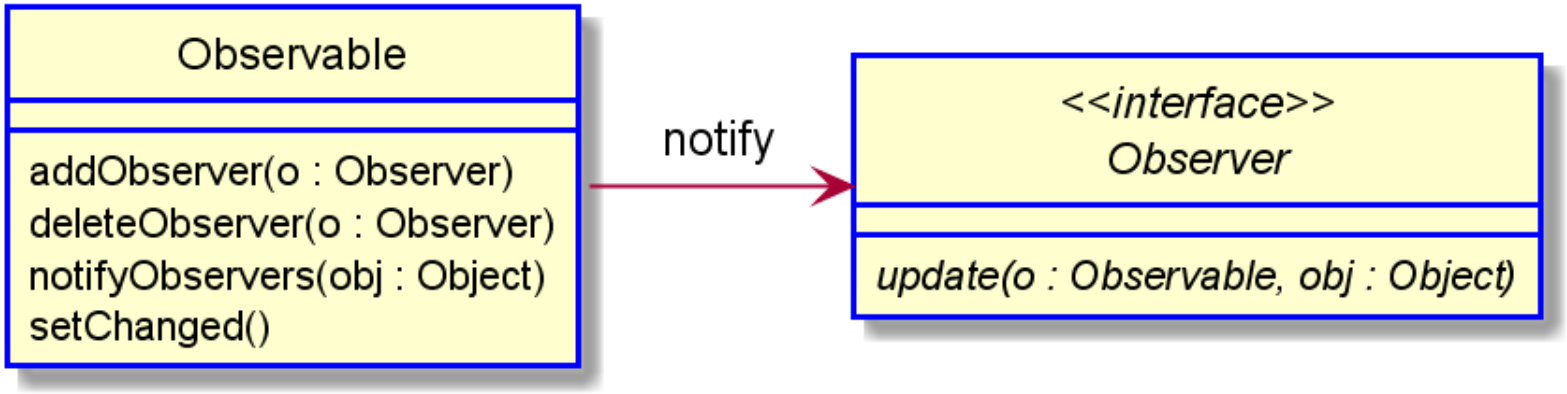
Diagramme de classes de la station météo



Cf. stationmeteo_observateur.zip

Java propose une variante du patron Observateur

- Une **classe Observable** (attention différence par rapport au patron Observateur) et une **interface Observer**



Classe Observable

- Méthodes de la classe Observable *(java.util.Observable)*
  - addObserver(Observer) ajoute un objet d'Observer à la liste des objets Observer de l'instance
  - countObservers() retourne le nombre d'objets Observer
  - deleteObservers(Observer) enlève un objet d'Observer de la liste des objets Observer de l'instance
  - setChanged() indique qu'un changement s'est produit sur l'instance de l'objet
  - notifyObservers() ou notifyObservers(Object) indique aux objets Observer qu'un changement a eu lieu après que la méthode setChanged() ait été utilisée

Classe Observable

- Précisions sur les appels de setChanged et notifyObservers
  - L'appel de setChanged fait passer un attribut boolean *changed* à true
  - A l'appel de notifyObservers, 2 possibilités :
    - Si *changed* est à true,
      - l'objet Observable appelle la méthode **update** sur chaque objet Observer
      - Puis *changed* est mis à false
    - Si *changed* est à false, la méthode update n'est pas appelée sur les objets Observer

Conclusion : nécessité de coupler les appels à setChanged et notifyObservers pour que les modifications soient répercutées au niveau des Observers

Interface Observer *(java.util.Observer)*

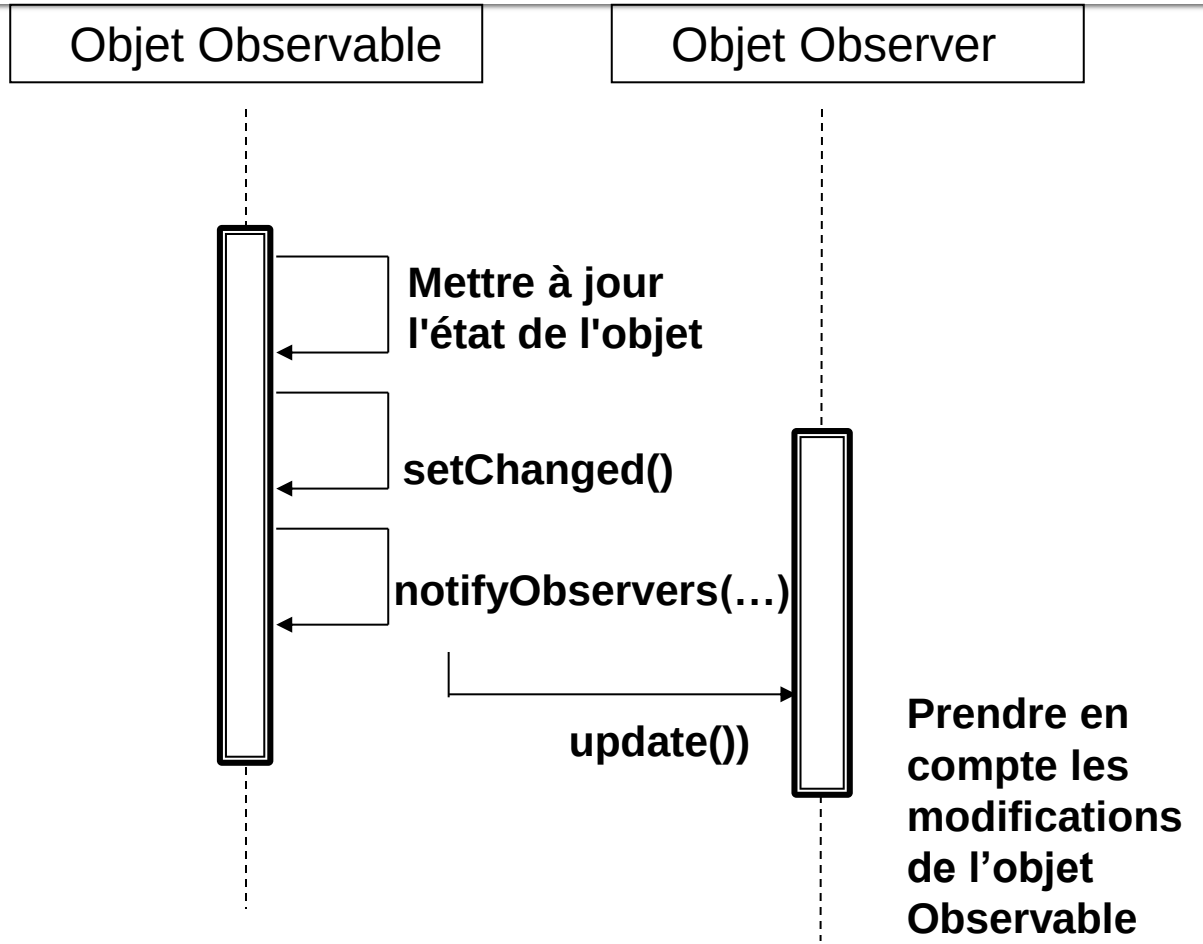
- Contient l'unique signature de la méthode **update** appelée quand l'instance appartient à la liste des objets Observer de **o** :

  public abstract void update (Observable o, Object arg)

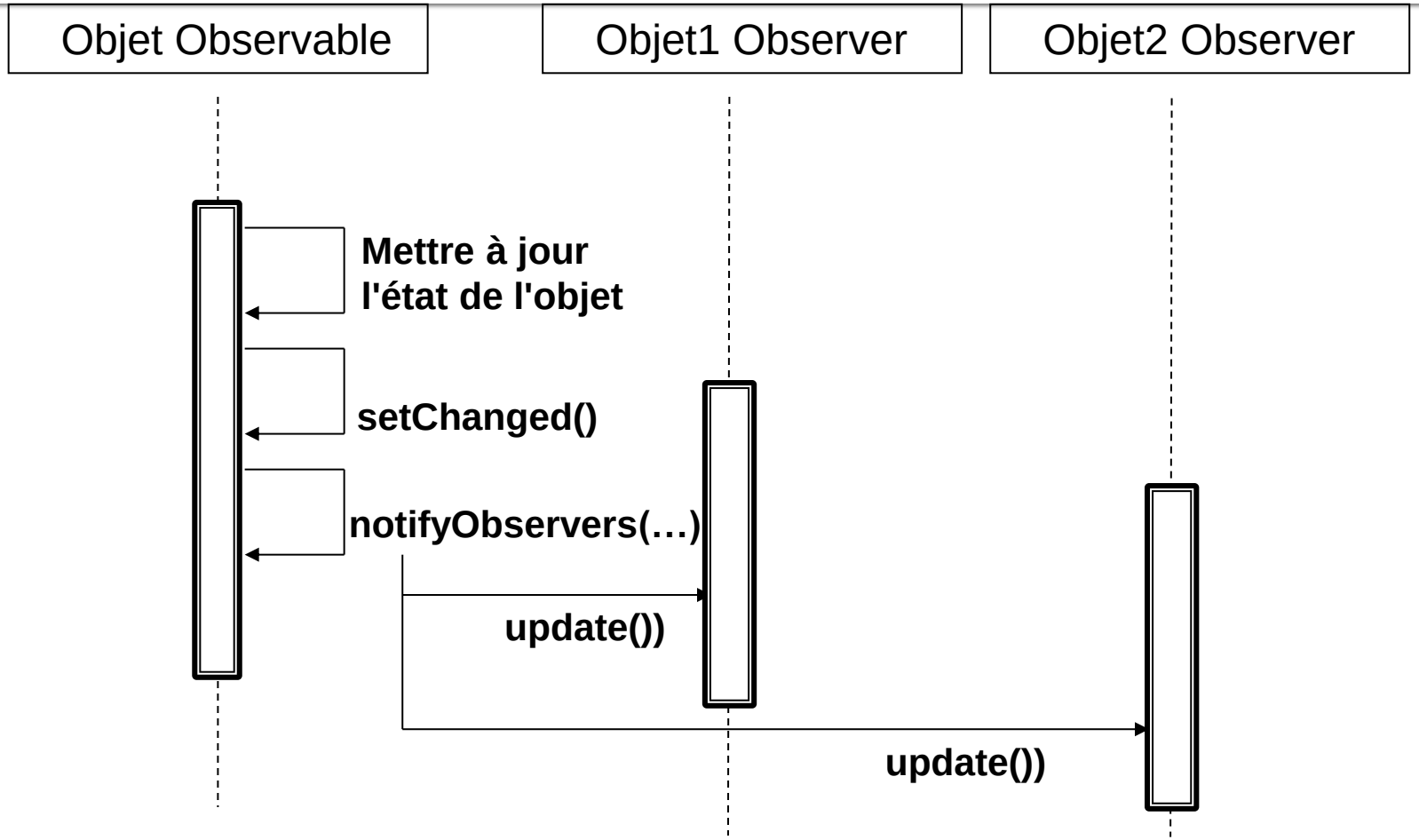
  - arg permet de passer une information de l'Observable à ses objets Observer associés : soit null, soit le paramètre d'appel de notifyObservers()

  Remarque : l'objet Observable est donc connu par l'objet Observer grâce au paramètre o de la méthode update

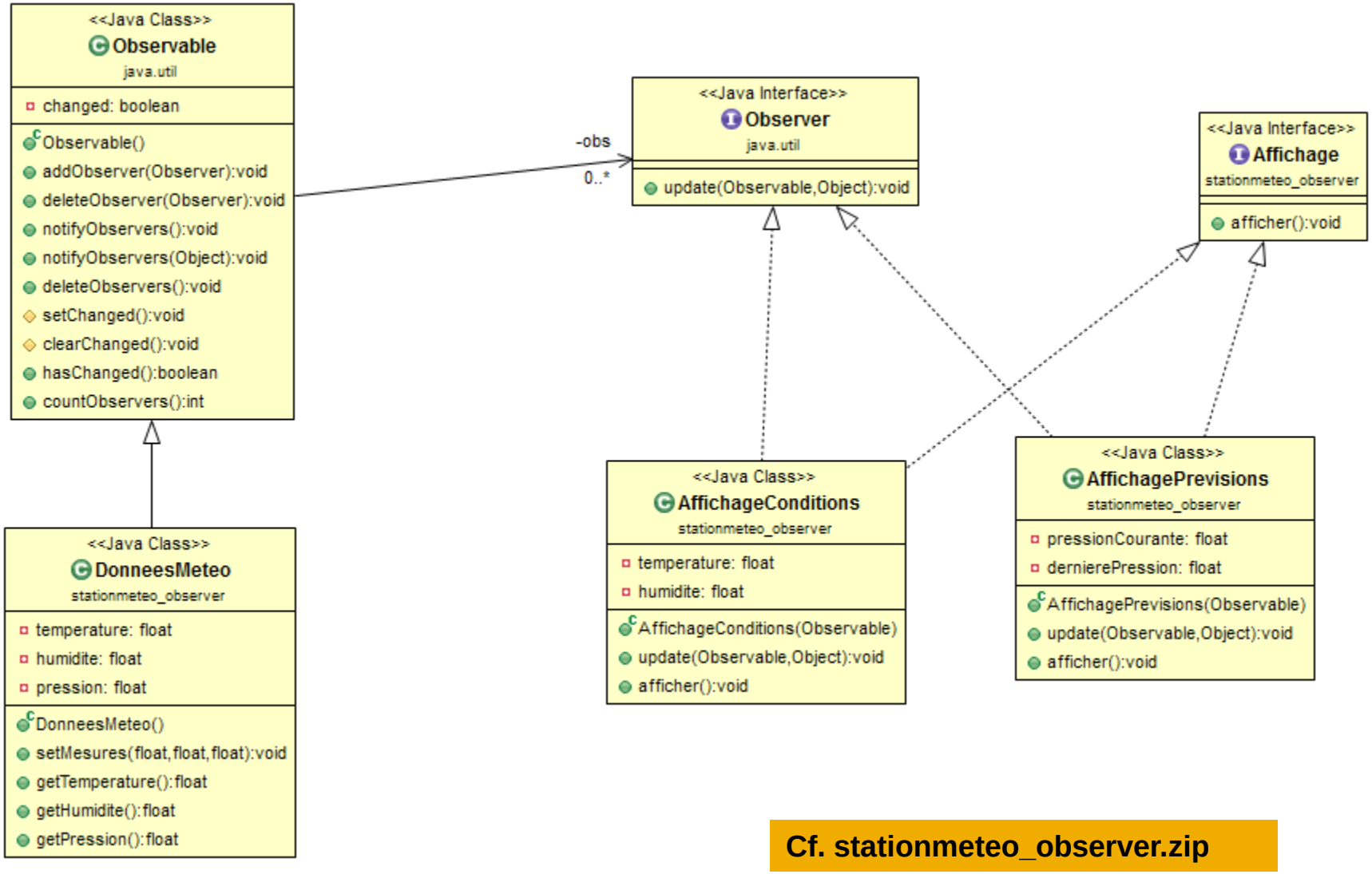
Observer/Observable



Observer/Observable



Station météo avec Observer/Observable



Cf. stationmeteo_observer.zip

Observer/Observable

- Intérêt
  - Plus besoin de mémoriser les observateurs (Observer) dans une structure de données adaptée, la super classe Observable s'en charge
  - L'appel à setChanged() puis notifyObservers() lance automatiquement les méthodes update sur chaque observateur

- Mais
  - Observable est une classe et elle doit être sous-classée → on ne peut pas ajouter le comportement d'Observable à une classe existante étendant déjà une classe

Utilisation du pattern Observateur dans Java

- **Gestion des événements graphiques**
  - Repose sur 3 types d'objets :
    - Les **événements graphiques** qui dérivent de *java.util.EventObject*
      - Ex : ActionEvent, KeyEvent, …
    - Les **objets sources** qui dérivent de *java.awt.Component*
      - Ex : JButton, JPanel, …
    - Les **objets écouteurs d'événements** qui implémentent une ou des interfaces dérivant de *java.util.EventListener*
      - Ex : ActionListener, KeyListener, …

L'objet source est observé. Les Listeners sont les observateurs chargés de réagir aux événements en lançant une méthode spécifique. Une fois enregistrés, les listeners sont notifiés dès qu'un événement se produit sur les composants.

IV. Eléments d'architecture logicielle

Définition

- **Architecture Logicielle**
  - décrit d'une manière symbolique et schématique les différents éléments d'un ou de plusieurs systèmes informatiques, leurs interrelations et leurs interactions.
  - le **modèle d'architecture**, produit lors de la phase de conception, ne décrit pas ce que doit réaliser un système informatique mais plutôt **comment il doit être conçu** de manière à répondre aux spécifications.
    - L'analyse décrit le « quoi faire » alors que l'architecture décrit le « comment le faire ».

Classification des architectures

- Les **modèles architecturaux** décrivent des formes d'organisation des systèmes logiciels que l'on retrouve fréquemment.
  Deux types de classification :
  - **Les styles architecturaux**
    - Descriptions très générales , reflétant une certaine philosophie d'organisation des systèmes – Ex : Architecture en couches
  - **Les patrons architecturaux**
    - Assemblages précis de composants destinés à résoudre un problème particulier dans un contexte donné – Ex : le patron architectural MVC
      - Limites entre patrons architecturaux et patrons de conception assez floues – certain patrons de conception sont centraux dans certains patrons architecturaux

IV.1. Le patron architectural MVC

Le patron architectural Modèle-Vue-Contrôleur (MVC)

- Concerne les applications interactives
  - Vise à séparer traitements, données et présentations
    - Essentiel pour faciliter l'évolution des interfaces utilisateur et l'adaptation à différents contexte d'utilisation – plusieurs interfaces possibles
  - Divise l'application en trois types de composants et définit leurs interactions
    - **Modèle**
    - **Vue**
    - **Contrôleur**

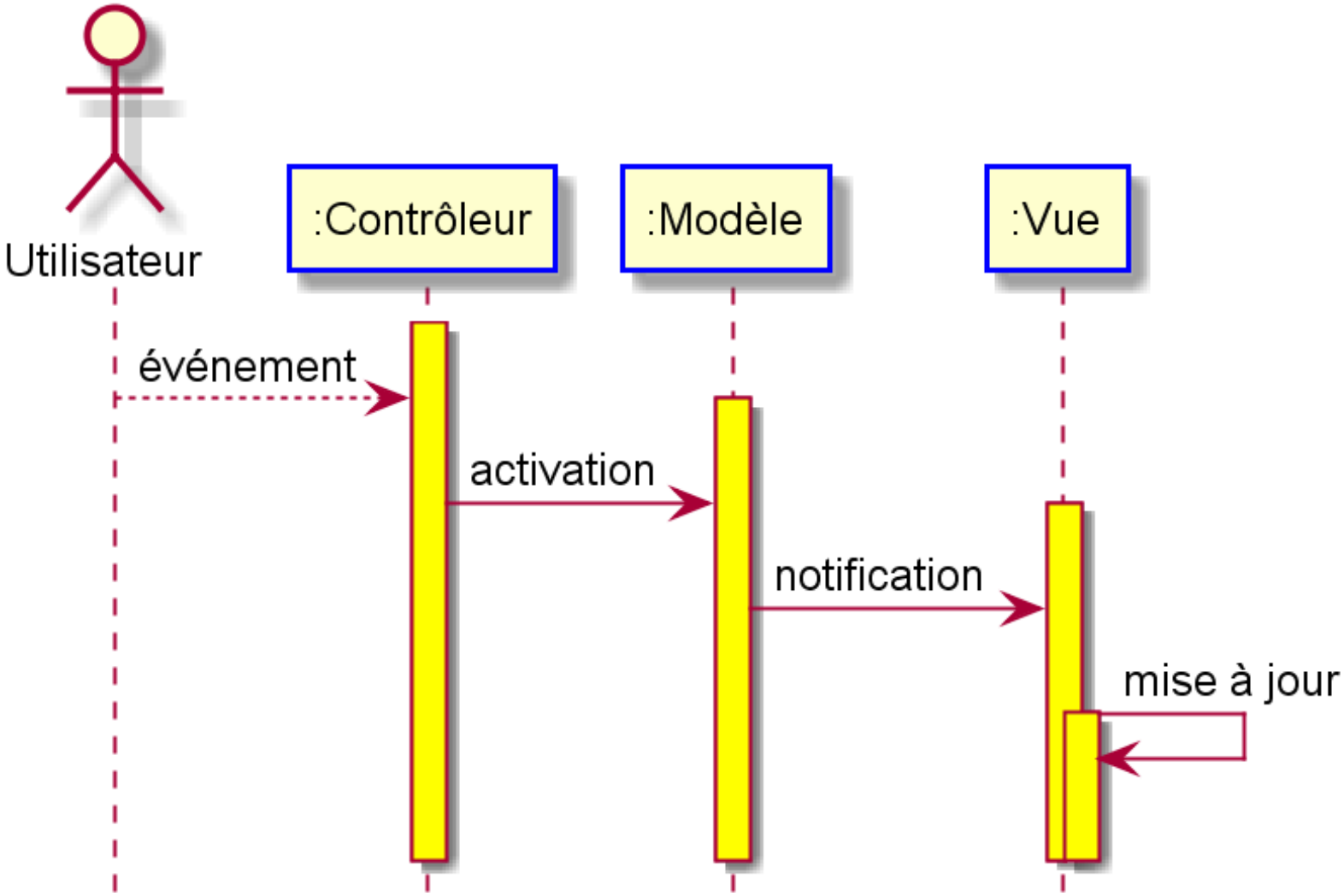
MVC – rôles

- Un Modèle
  - Correspond à une entité gérée par l'application avec ses **données** et les traitements associés
- Une Vue
  - Est une **représentation** externe du modèle
- Un contrôleur
  - Reçoit les **événements** de l'utilisateur

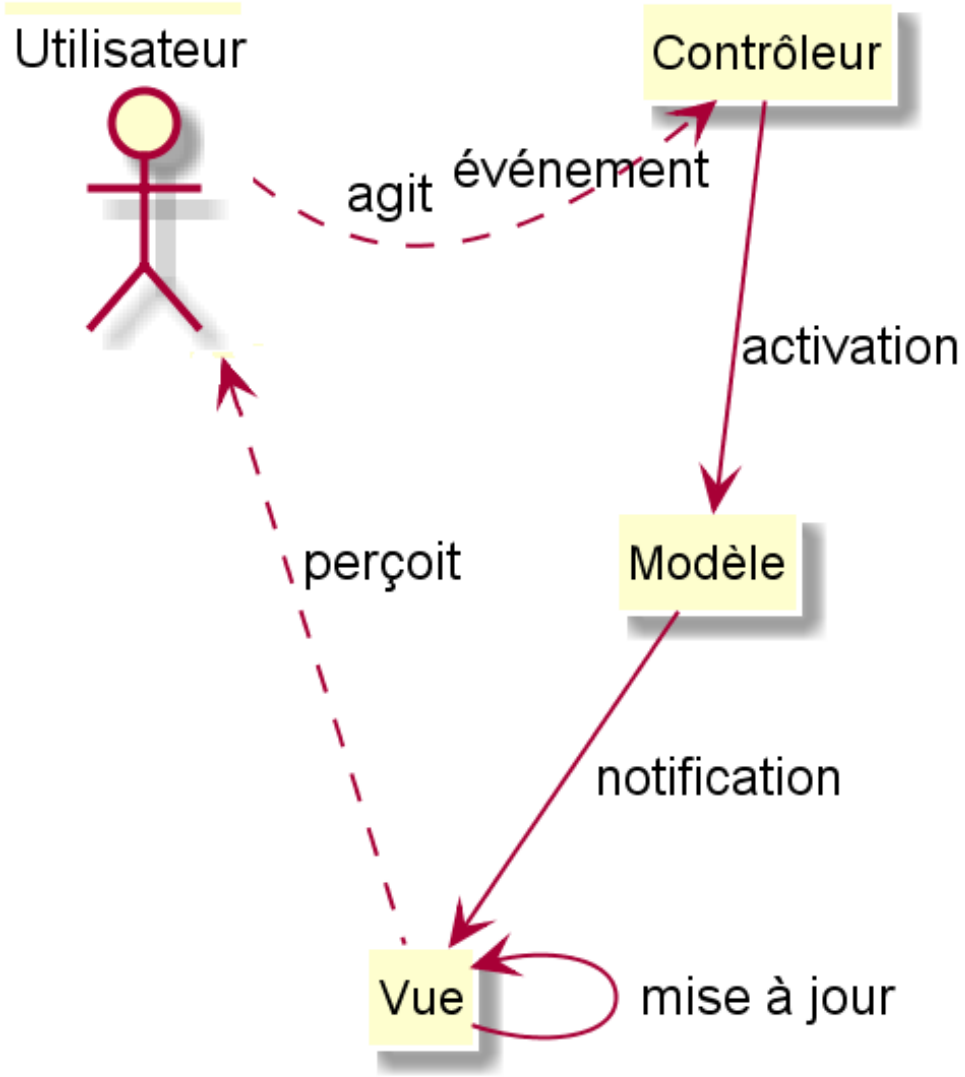
MVC – rôles et interactions

- Un Modèle
  - Correspond à une entité gérée par l'application avec ses **données** et les traitements associés
  - Est activé par le contrôleur et notifie les vues de ses changements
- Une Vue
  - Est une **représentation** externe du modèle
  - Se met à jour quand elle est notifiées par le(s) modèle(s)
- Un contrôleur
  - Reçoit les **événements** de l'utilisateur
  - Déclenche les traitements à effectuer par le(s) modèle(s)

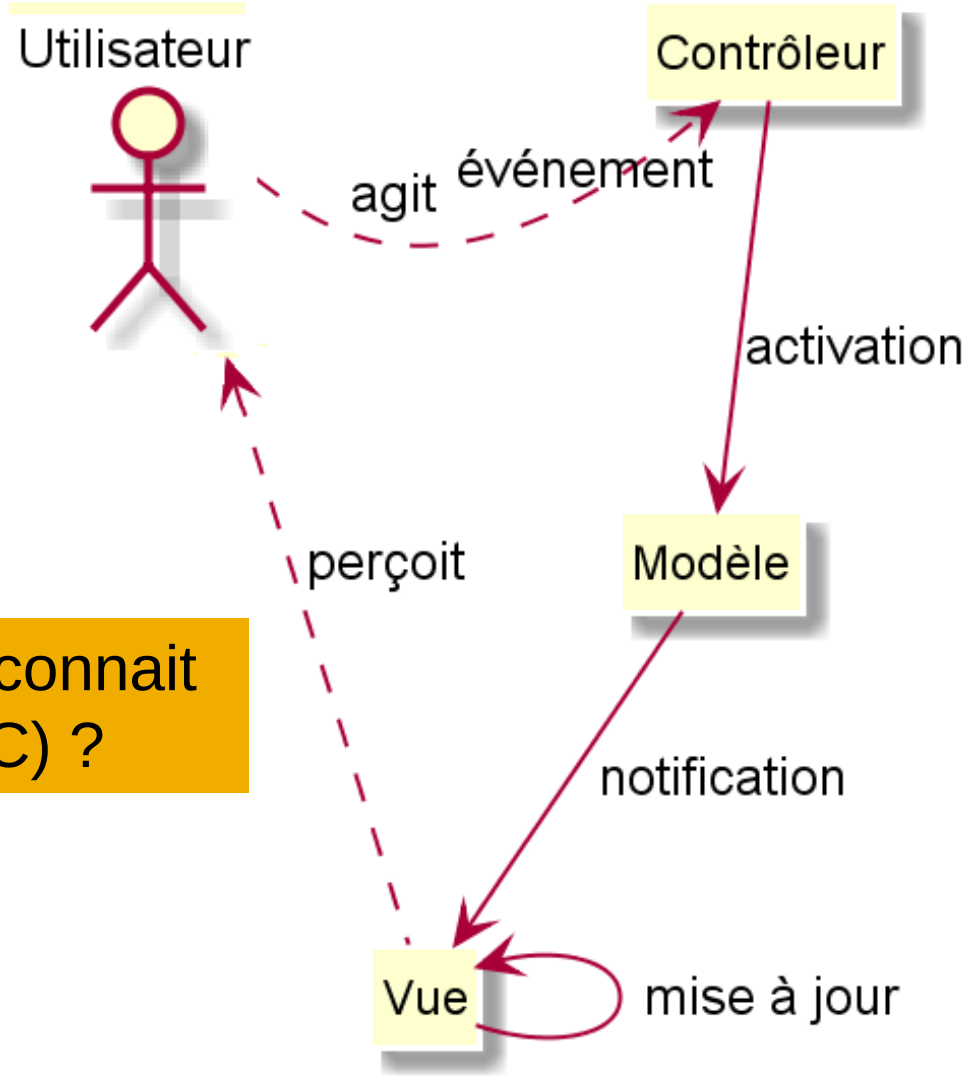
MVC - interactions



MVC - interactions

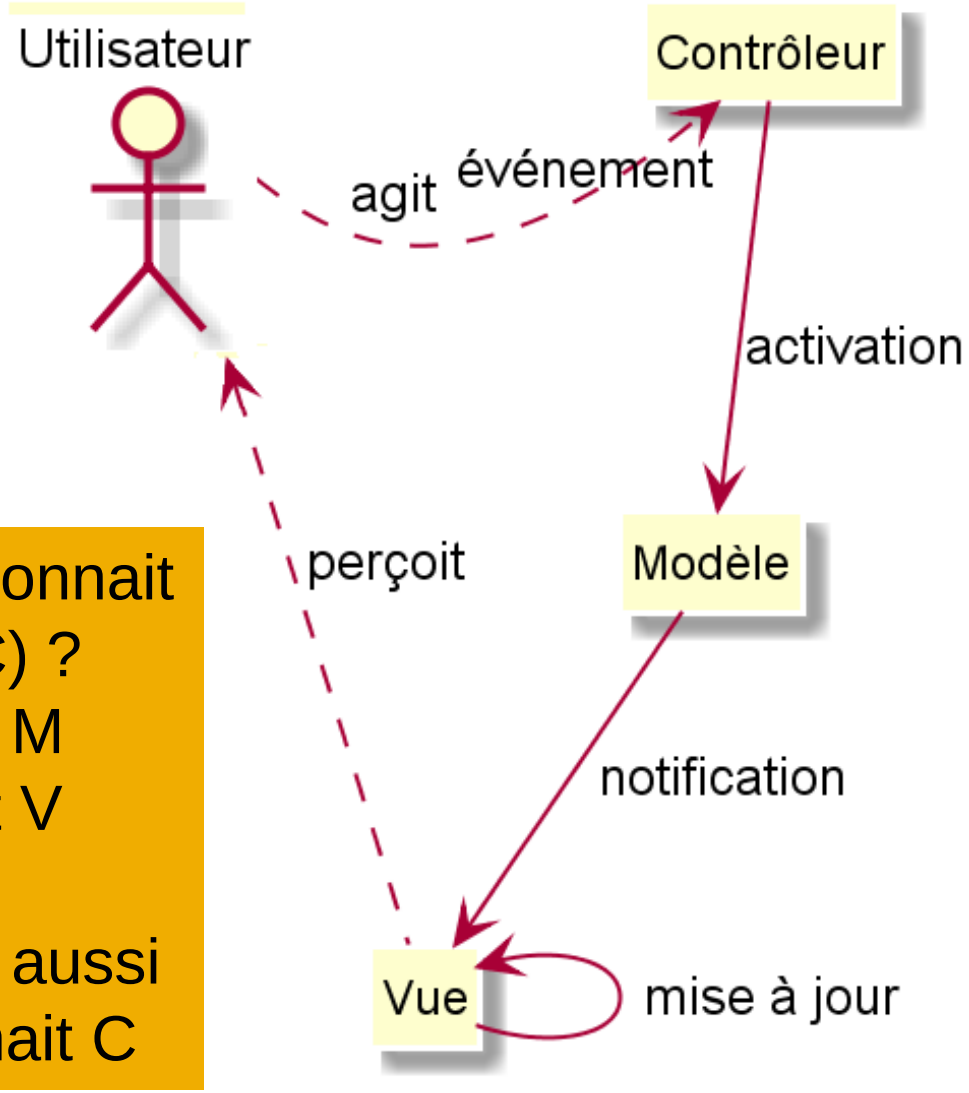


MVC - interactions



Qui (M/V/C) connait qui (M/V/C) ?

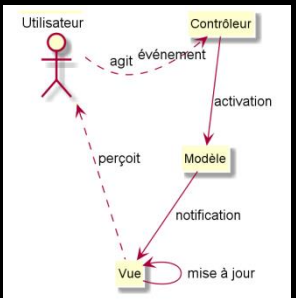
MVC - interactions



Qui (M/V/C) connait qui (M/V/C) ?
C connait M
M connait V

Mais on peut aussi avoir V connait C

MVC - implantation



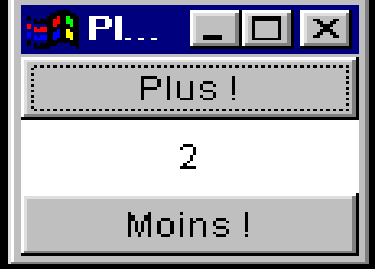
- **Modèle-Vue**
  - Relation fondées sur le **patron Observateur** :
    - Les vues sont des observateurs des modèles
    - Les vues s'enregistrent auprès des modèles et sont notifiées par eux quand ils évoluent
- **Vue-Contrôleur**
  - Relation fondées sur le **patron Stratégie** :
    - En java, les contrôleurs sont des écouteurs des événements (Listener) générés sur les vues – les vues délèguent aux contrôleurs la gestion des actions de l'utilisateur.

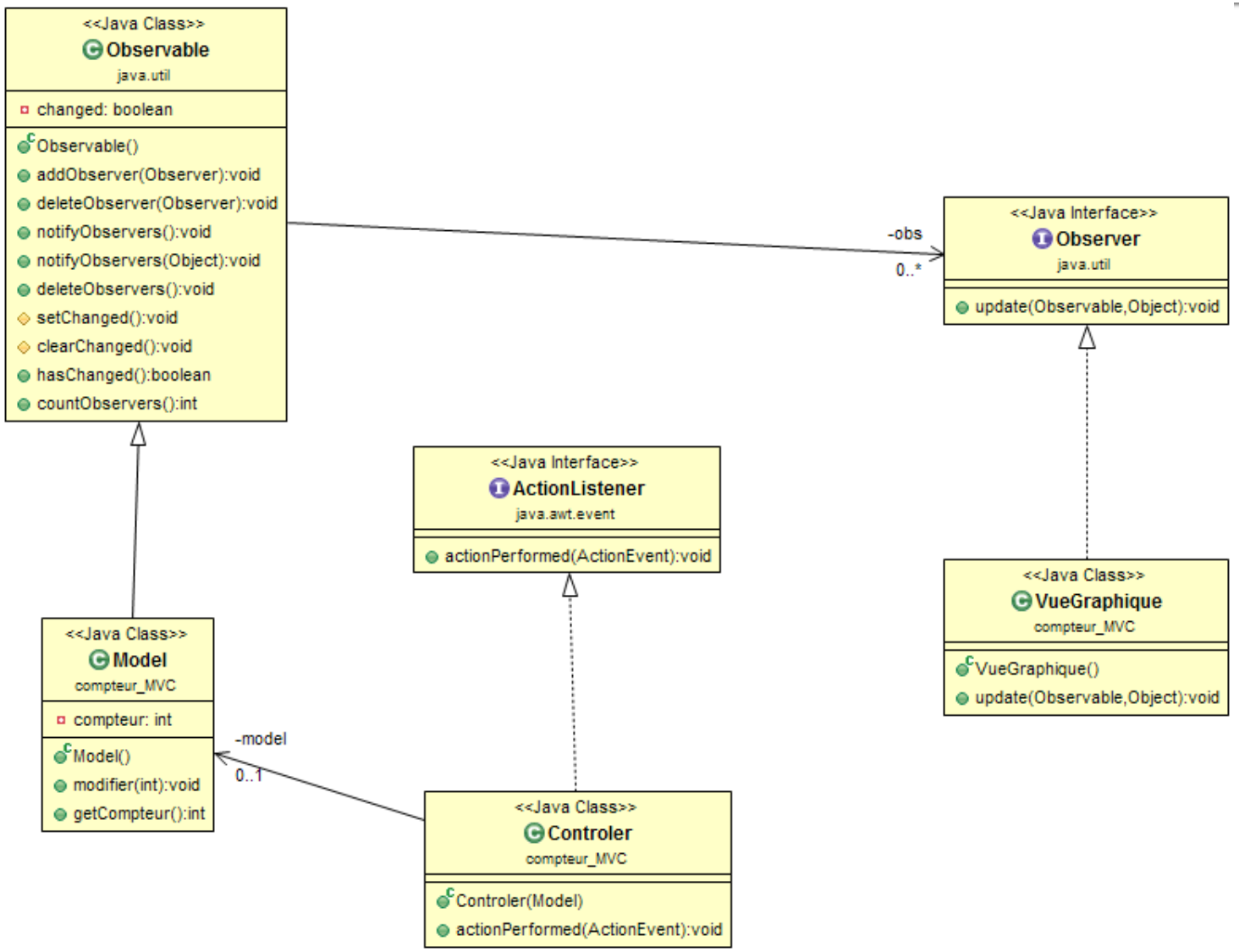
MVC - Exemple

- Modèle ?
- Vue ?
- Contrôleur ?

Cf. compteur_mvc.zip

MVC - Exemple


Pl...
Plus !
2
Moins !


<<Java Class>>
Observable
java.util
changed: boolean
Observable()
addObserver(Observer):void
deleteObserver(Observer):void
notifyObservers():void
notifyObservers(Object):void
deleteObservers():void
setChanged():void
clearChanged():void
hasChanged():boolean
countObservers():int
-obs
0..*
<<Java Interface>>
Observer
java.util
update(Observable,Object):void
<<Java Interface>>
ActionListener
java.awt.event
actionPerformed(ActionEvent):void
<<Java Class>>
VueGraphique
compteur_MVC
VueGraphique()
update(Observable,Object):void
<<Java Class>>
Model
compteur_MVC
compteur: int
Model()
modifier(int):void
getCompteur():int
-model
0..1
<<Java Class>>
Controler
compteur_MVC
Controler(Model)
actionPerformed(ActionEvent):void

Exercice

- Les fichiers des classes de l'exemple du cours figurent dans le fichier ***compteur_mvc_exercice.zip*** sur arche. Faire un projet sous Eclipse et y inclure les fichiers dans un package *compteur_mvc_exercice*.
  - Ajouter une vue (**classe VueConsole**) au MVC de l'exemple permettant **d'afficher le compteur dans la console**.
  - Faire le diagramme de classe - **le faire valider par votre enseignant-.**