Chap.3 – Design patterns (partie 2)

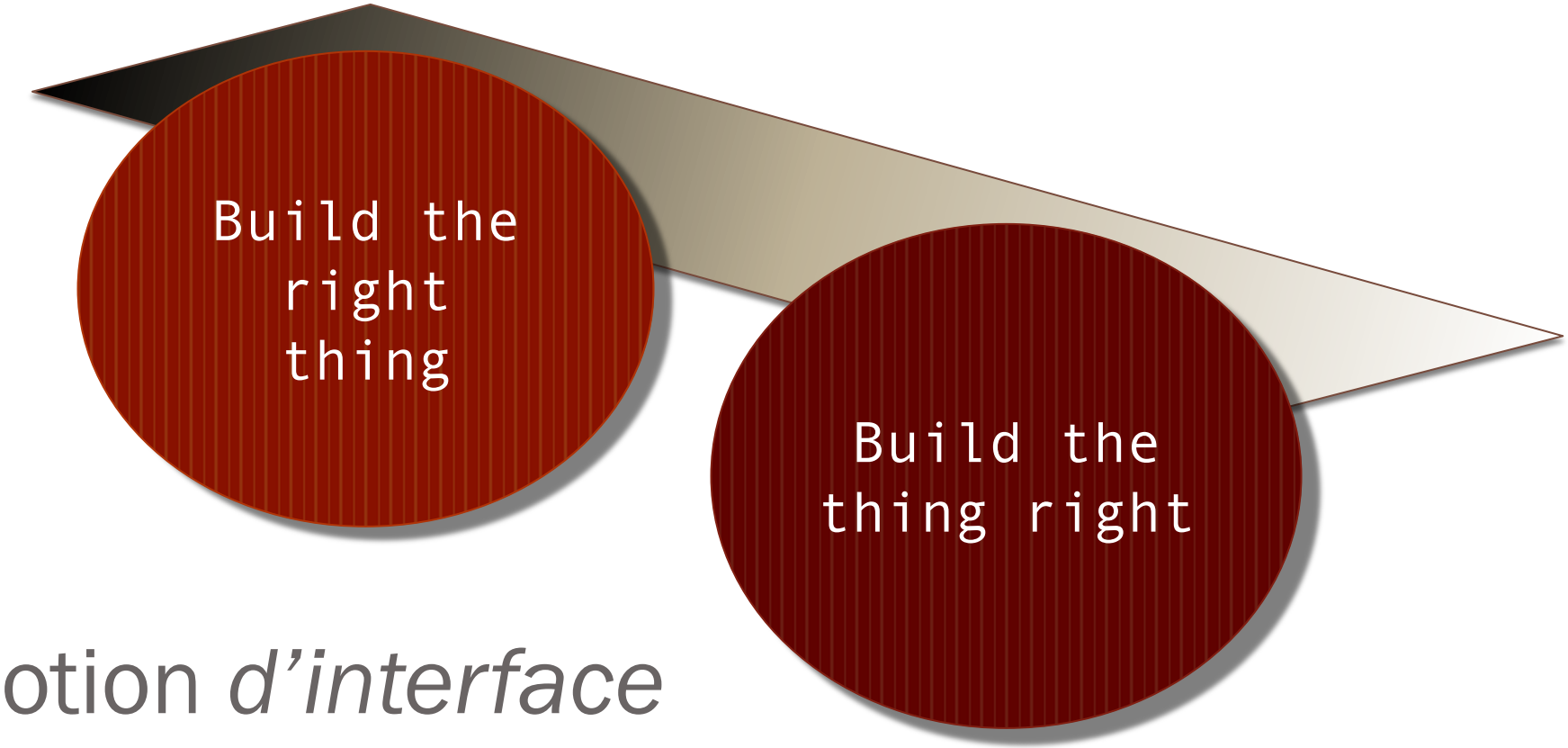
V. Deslandres ©

BUT Informatique, s3

Réalisation d'Applications

IUT de Lyon - Université Lyon 1

Sommaire du cours DP - part.2



Préambule : notion *d'interface*

- « **Interface** » : sens général
  - Pas seulement Java
  - Ensemble des méthodes associées à un composant (classe, package, module), par ex.: interface d'une API

Le pattern OBERVER

Un modèle de comportement

Design pattern 'Observer'

- Définit une dépendance (1,n) entre des objets de telle sorte que quand un objet change d'état, tous les objets dépendants de lui sont notifiés et mis à jour automatiquement.

Motivation

- Un effet de bord du découpage en composants et packages des systèmes
- Un grand nombre de classes collaborent

  ➔ Besoin de maintenir la cohérence

  - sans toutefois coupler trop fortement les classes pour ne pas réduire leur réutilisabilité

Observer

- Exemple **: toutes les Interfaces Graphiques (GUI, IHM)**
  - Les classes relatives à l'IHM et à l'application peuvent être réutilisées indépendamment les unes des autres
  - Mais elles travaillent 'ensemble' aussi

- Un Tableur et un Histogramme représentent la même information sous des formes différentes.
  - Ces deux objets ne savent rien l'un de l'autre, c'est l'utilisateur qui choisit le mode de représentation sur lequel il agit.
  - Ils se comportent néanmoins comme s'ils échangeaient : lorsque les données sont modifiées dans le tableur, l'histogramme reflète le changement immédiatement, et vice versa.

Illustration

- Ex.: un graphique montrant le compte de résultat d'un **bilan comptable**
  - Il peut y avoir **plusieurs vues** de ce bilan, dans l'application
  - Les données **peuvent évoluer** au cours du temps (→ évolution du modèle)
  - **La façon de la visualiser** peut changer selon les technologies employées (→ vue)

Mois	Recettes	Dépenses
Janvier	10 000 €	8 000 €
Février	12 000 €	10 000 €
Mars	9 000 €	10 000 €
Avril	15 000 €	12 000 €
Mai	14 000 €	11 000 €
Juin	18 000 €	15 000 €

Sans Observer...

- Le **comptable** modifie des valeurs dans son **tableau de résultats**
- L'objet graphique *tableau* (ex. JTable) signifie à son modèle que ses valeurs ont été modifiées
- Le *modèle de la table* signifie à toutes les **autres vues** (courbes sur le poste du Chef de Service, camemberts sur le poste du PDG…) qu'elles **doivent se mettre à jour**

- Si jamais la **DSI** ajoute une vue du Bilan (nouvelle application, nouveau poste), il faut modifier le code précédent du modèle pour ajouter la notification de mise à jour associée à cette nouvelle vue…
- Le **DP Observer** va inverser ce mécanisme : ce sont les vues (observateurs ») qui vont s'abonner aux données (le sujet « observé »). Ce dernier les avertira en cas de MàJ.

*Qui est « **observateur** » et « **observé** » dans notre exemple ?*

Mécanisme du *publish-subscribe*

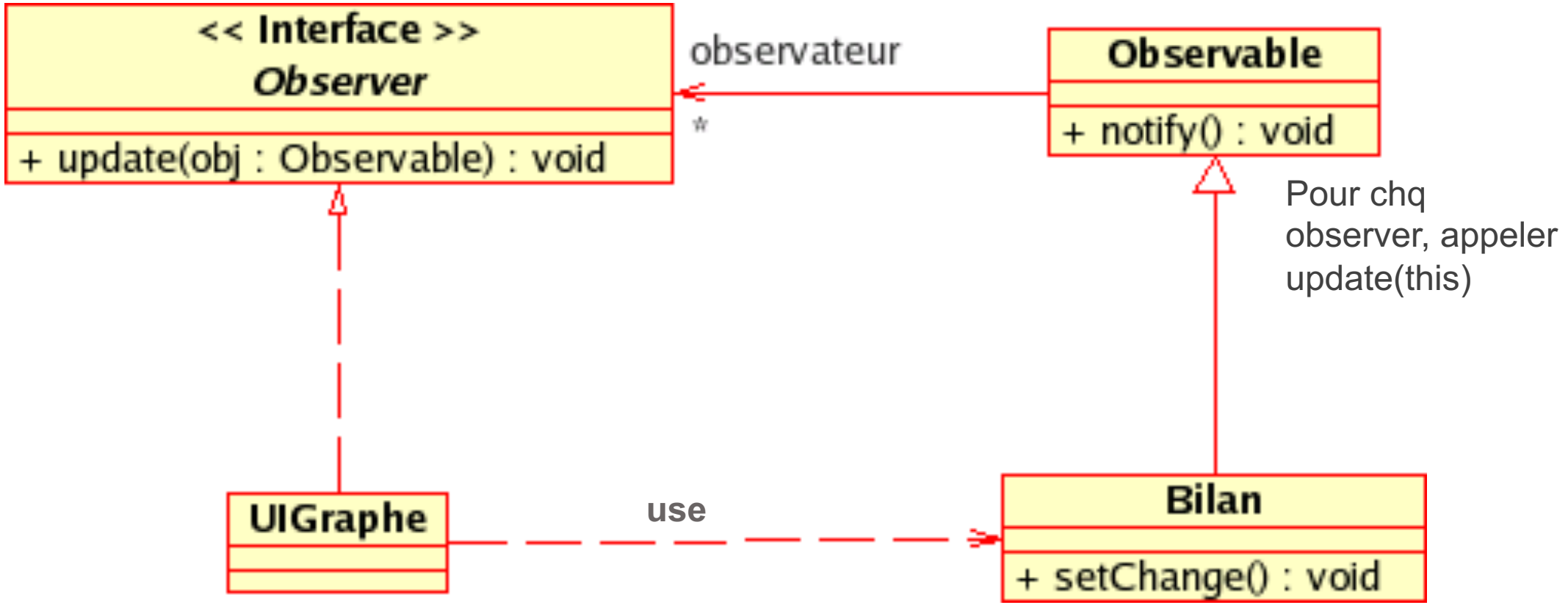
- **Ces interactions** entre un sujet et ses observateurs sont connues sous le nom de **publication / abonnement**

- Le *sujet* est celui qui publie des notifications de changement d'état.
  - Il envoie ces notifications sans avoir besoin de connaître **qui** sont ses observateurs.
- Les objets *observateurs* s'abonnent pour recevoir les notifications de changements, et se mettre à jour.

Principes du DP Observer

- Définir un comportement générique de ces 2 éléments :
  - La classe Observer signale à la classe observée qu'elle le « suit » (elle s'abonne)
  - Quand l'observé est modifié (actions utilisateurs sur la vue, ou modification du modèle par des traitements)
    - Méthode **`notify()`** dans Observable (observé)
  - Le sujet « observé » signale à *tous ses observateurs* (abonnés) qu'ils doivent se mettre à jour :
    - Méthode **`update()`** dans Observateur
- Utilisation **d'interfaces ou classes abstraites** pour tout ce qui est générique :
  - Le Bilan et le Graphique seront des **classes d'implémentation**

DP Observer pour l'exemple



Code source de l'exemple

```
public interface Observer {
   public void update(Observable o);
}
```

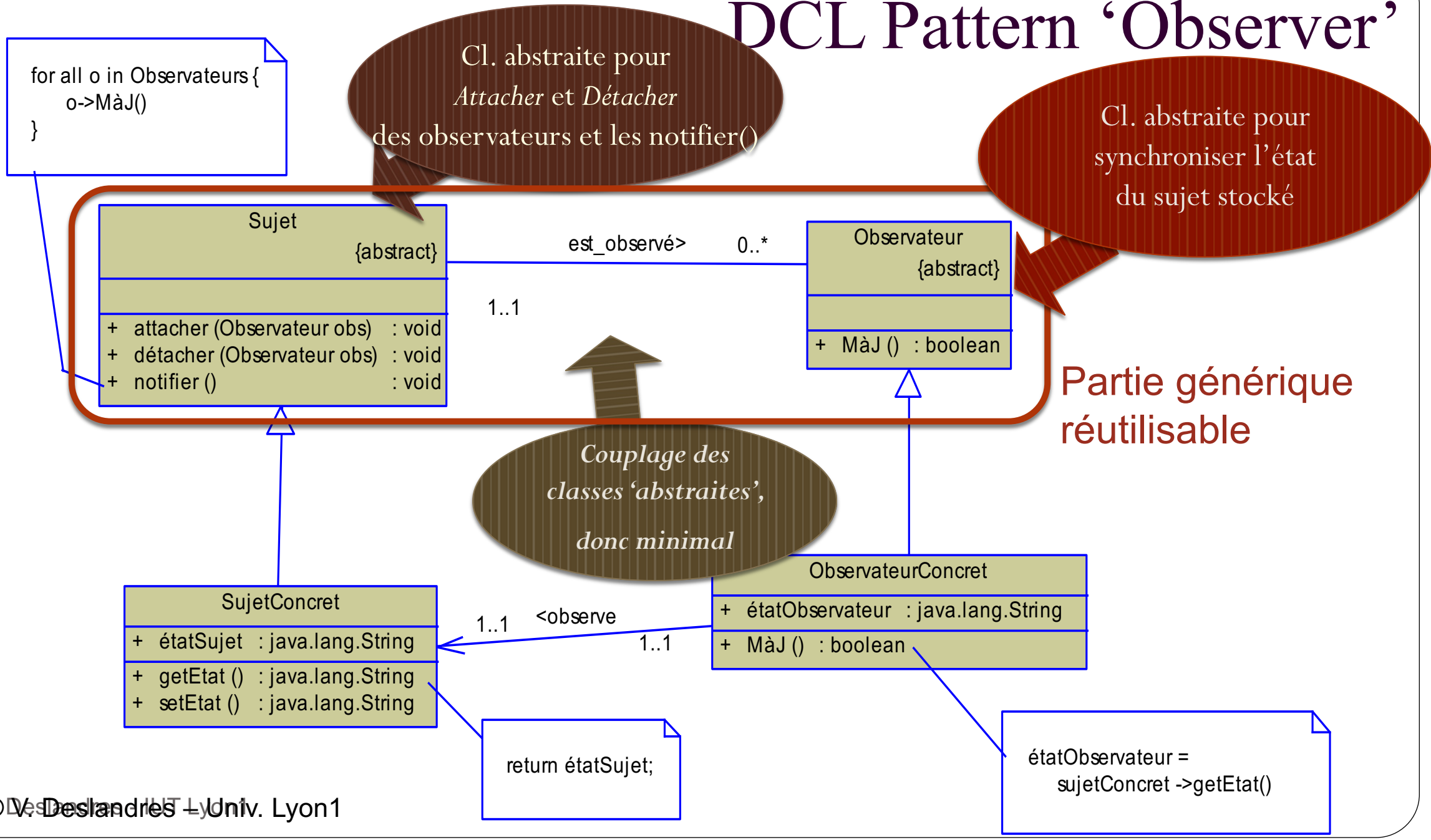
```
public class Observable {
      Collection observateurs;

      public void notify() {
            Iterator it = this.iterator();
            while (it.hasNext()) {
                  ((Observer) it.next()).update(this);
            }
      }
      public void addObserver(Observer o) {
            observateurs.add(o);
      } ...
}
```

```
public class Bilan extends Observable {
      void setChange() {
            notify();
      } ...
}
```

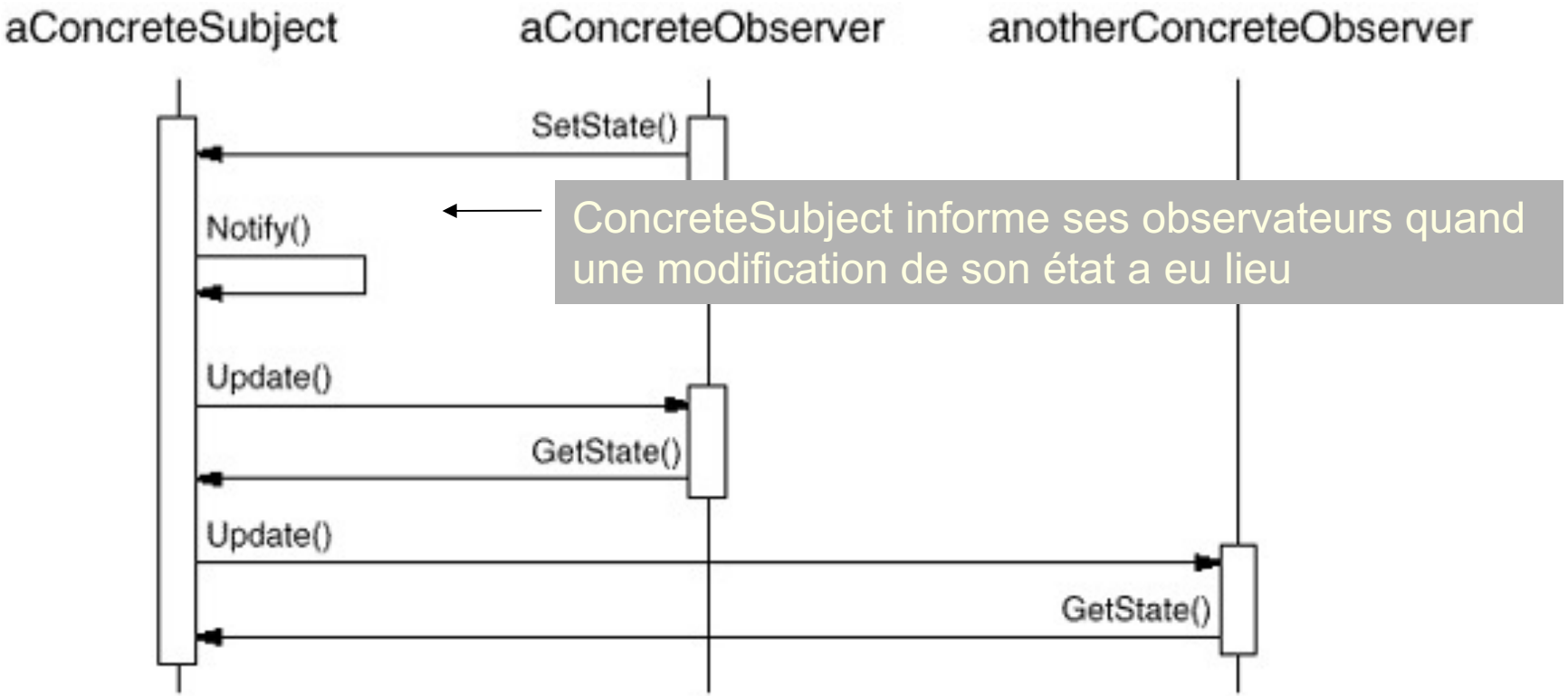
```
public class UIGraphe implements Observer {

      public void update(Observable o) {
            Bilan unbilan = (Bilan) o;
            // on récupère la valeur à jour du Bilan :
            double compteResultat = unbilan.getCompteResultat();
            ...
      }
      ...
}
```

Nota : le patron Observer existe dans l'API Java, il est un peu différent (méthode setChange())

DCL Pattern 'Observer'



Un fonctionnement d'Observer



ConcreteSubject informe ses observateurs quand une modification de son état a eu lieu

Les notifications ne sont pas tjrs demandées **par le sujet**.
On verra différentes formes de notification lors de l'implémentation.

Conséquences de l'utilisation d'Observer

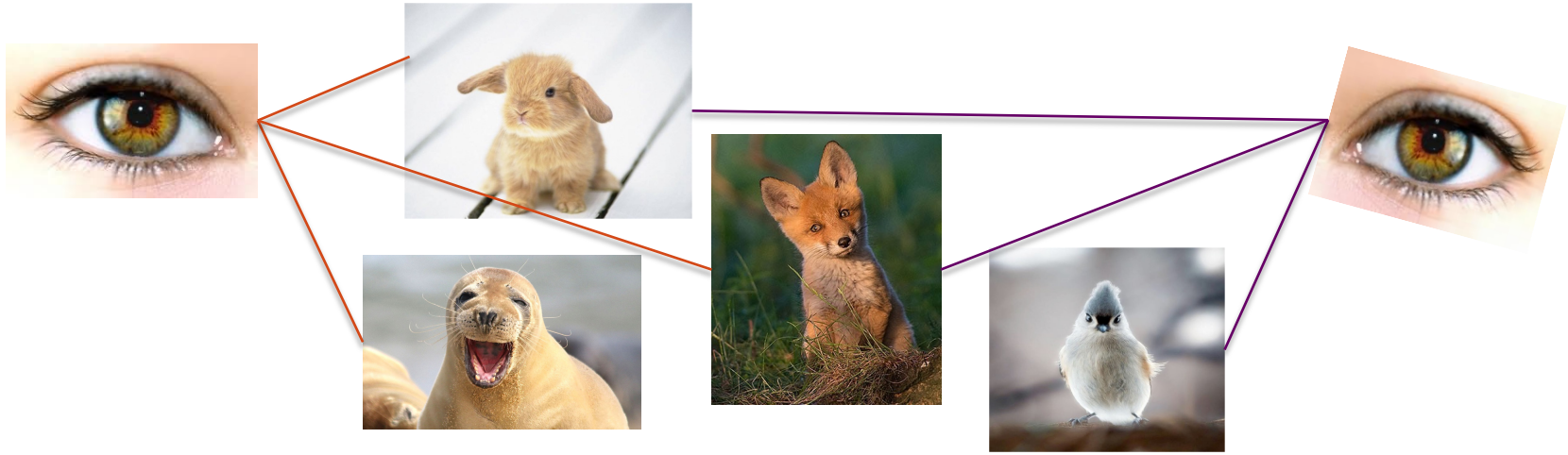
- **Couplage sur les classes abstraites**, donc minimal
  - Tout ce qu'un Sujet sait, c'est qu'il a une liste d'Observateurs, et que chacun se conforme à l'interface commune (méthode `Update()`) leur permettant de synchroniser leur état avec le sien.
- Le patron Observer permet de manipuler les objets Sujet et Observateurs de **façon indépendante et variée.**
  - On peut *réutiliser* les sujets sans les observateurs, et réciproquement;
  - On peut aussi *ajouter* des observateurs sans modifier le sujet et les autres observateurs (respect du principe d'OCP)

Risques associés à Observer

- **Mises à jour en cascade**
  - Une succession de modifications sur le sujet peut causer des mises à jour en cascade de la part des Observateurs.
  - Comme les observateurs n'ont pas connaissance de la présence des autres, ils peuvent ne pas savoir le coût imposé par certaines modifications du sujet.

- **Un protocole de MàJ un peu simpliste**
  - Comme les Observateurs n'ont pas de moyen de savoir quels changements ont eu lieu, cela peut coûter cher parfois d'aller 'voir' ;
  - La méthode de *mise à jour* de l'interface actuelle ne le permet pas : elle est très souvent paramétrée pour contrôler les mises à jour.

Extensions Observer : *n* sujets observés



- Par ex. : un tableur portant sur *n* sources de données
- Il est alors nécessaire d'étendre Update() afin que l'Observateur sache **quel sujet** a envoyé la notification.

- Implémentation possible :
  - Le sujet peut envoyer son nom en paramètre de la méthode Update()

Illustration : affichage du temps

Classe abstraite Observer

```
/* RESPONSABILITES :
 *      - Connait le sujet observé
 *      - Sait comment mettre à jour le dernier état du sujet
 */

abstract class AbstractObserverTimer {

    protected AbstractSujetTimer leSujetTimer;

    abstract void update(AbstractSujetTimer t);

}
```

Cette implémentation permet d'observer plusieurs sujets : on mentionne **quel** sujet a changé dans update().

Classe abstraite Sujet (Observable)

```
abstract class AbstractSujetTimer {
    protected List<AbstractObserverTimer>
lesObservateurs;

    abstract void attach(AbstractObserverTimer ot);
    abstract void detach(AbstractObserverTimer ot);

    void notifier() {
        for (AbstractObserverTimer o: lesObservateurs)
{
            o.update(this);
        }
    }
}
```

Classe concrète pour le Sujet : **SujetClockTimer**

```
public class SujetClockTimer extends AbstractSujetTimer {

    private int h;
    private int min;
    private int sec;
    private UniteTemps uniteTemps;

    // constructeur
    public SujetClockTimer(int h, int m, int s) {
        this.h = h;
        min = m;
        sec = s;
        uniteTemps = UniteTemps.seconde;
        lesObservateurs = new ArrayList<AbstractObserverTimer>();
    }
```

Ici on utilise une classe qui stocke et maintient un temps interne selon l'unité de temps choisie (énumération)

RESPONSABILITES :
- Connait son état (temps) et la façon de changer son état
- A une liste d'observateurs abonnés
- Sait donner son état (le temps) ou le détail de son état (h, min,...)
- Sait changer son état (une temps donné ou avancer un pas de temps) et notifier ses observateurs
- Décrit comment il attache / détache des abonnés

Classe concrète pour le Sujet : **SujetClockTimer**

L'opération **tick()** décompte le temps et appelle **notifier()** pour informer les observateurs du changement

```
public void tick() {

    // incrémente du temps
    switch (uniteTemps) {
        case heure:
            this.h++;
            if (h > 12) {
                h = 1;
            }
            break;
        case minute:
            this.min++;
            if (min >= 60) {
              // … break;
        case seconde:
              //…
            break;
    }
    this.notifier();

} // fin de tick()
```

Opération **notifier() :** pour tous les observateurs, appel à updtate(this)

```
// définir un temps donné et notifier ses observateurs :
public void setClockTimer(int h, int m, int s) {

    this.h = h;
    min = m;
    sec = s;
    this.notifier();
}
```

```
@Override
    void attach(AbstractObserverTimer ot)
{
        lesObservateurs.add(ot);
    }

    @Override
    void detach(AbstractObserverTimer
ot) {
        if (lesObservateurs.contains(ot)) {
            lesObservateurs.remove(ot);
        }
```

Classe concrète Observer : **ObsDigitalClock**

- Elle va hériter de la classe Observer précédente, et gérer l'affichage

A sa création, l'instance du ObsDigitalClock s'abonne au sujet SujetClockTimer

Avant que update() affiche le temps, elle vérifie que le **sujet** qui a notifié est le bon

Mise à jour de l'affichage

```
public class DigitalClock extends ObserverTimer {

    private int heure;
    private int minute;
    private int seconde;
    private UniteTemps uniteTemps;
    private ClockDisplay view;                  // affichage fenêtre

    // constructeur : nécessairement lié à un sujet
    public DigitalClock(AbstractTimer unTimer, String titre) {
        this.leSujetTimer = unTimer;
        this.leSujetTimer.attach(this);     // on s'abonne au sujet
        view = new ClockDisplay(titre);
        // DigitalClock récupère le temps de son sujet en appelant update()
        update(leSujetTimer);
        this.uniteTemps = UniteTemps.seconde; // par defaut
    }

    @Override
    void update(AbstractTimer t) {
        // on vérifie que c'est bien notre sujet qui a notifié un changement
        if (t == leSujetTimer) {
            ClockTimer ct = (ClockTimer) t; // on est obligé de caster car Abstr
            int[] temps = ct.getClockTimer();
            heure = temps[0];
            minute = temps[1];
            seconde = temps[2];
            view.labelHeure.setText(heure +":"+ minute +":"+ seconde);
        }
    }
```

Classe concrète : **Observer (2)**

- Affichage

```
// classe interne d'affichage
    class ClockDisplay extends JFrame {
        private JLabel labelHeure;

        public ClockDisplay() {
            super("Horloge Digitale");
            setSize(300, 100);
            setDefaultCloseOperation(EXIT_ON_CLOSE);

            labelHeure = new JLabel(heure +" : "+ minute +" : "+ seconde, SwingConstants.CENTER);
            getContentPane().add(labelHeure, BorderLayout.CENTER);
            this.setVisible(true);
        }

    }
}
```

On pourrait développer une classe **ObsAnalogClock()** à l'identique

Lancement du pattern

```
ObserverTimer ecran1, ecran2 ;
ClockTimer unTimerSec = new ClockTimer(10, 0, 0);          // en sec.
ClockTimer unAutreTimer = new ClockTimer(17, 20, 20);      // en sec.

ecran1 = new DigitalClock(unTimerSec, "Horloge 1");
ecran2 = new DigitalClock(unTimerSec, "Horloge 2");

unTimerSec.detach(ecran2);
unTimerSec.setClockTimer(20, 15, 30);

System.out.println("On a cree 2 instances de DigitalClock, seul le 1er est attaché au sujet, dont l'heure a été modifiée");
System.out.println(ecran1);
System.out.println(ecran2);
```

A chaque décompte du timer (opération `tick()`), les observateurs sont informés et les deux horloges s'affichent correctement.

Si on ajoute une horloge analogique **ObsAnalogClock()** abonnée au même sujet, les horloges s'afficheraient correctement à chaque modification du temps.

Exercice : Météo

Exercice : Météo

- On souhaite exploiter les données Météo (température, hygrométrie et pression atmosphérique)
- Développer une **API Météo** où pour l'instant 3 affichages sont envisagés :
  - Affichage des *conditions actuelles* (valeurs des 3 données)
  - Des *statistiques* (températures, moyenne, min et max)
  - De *prévisions* simples (icône pour le temps qu'il fera demain : nuage, soleil, pluie, neige)
- Ces affichages étant mis à jour en TR au fur et à mesure que les dernières données parviennent au système

METEO : quels sont les sujets / les observateurs ?

- Le(s) sujet(s) ?

  → Une classe **Météo** avec les attributs *température*, *hygrométrie*, *pression*

  Acquisition des mesures de la station météo ? De nouvelles valeurs arrivent régulièrement de capteurs, on va simplement les simuler avec des **setMesures()**…

- Le(s) Observateurs ?

  → Les 3 affichages (conditions météo, stat, prévisions)

  1 condition :

  - Laisser la possibilité d'ajouter de **nouveaux types d'affichage**

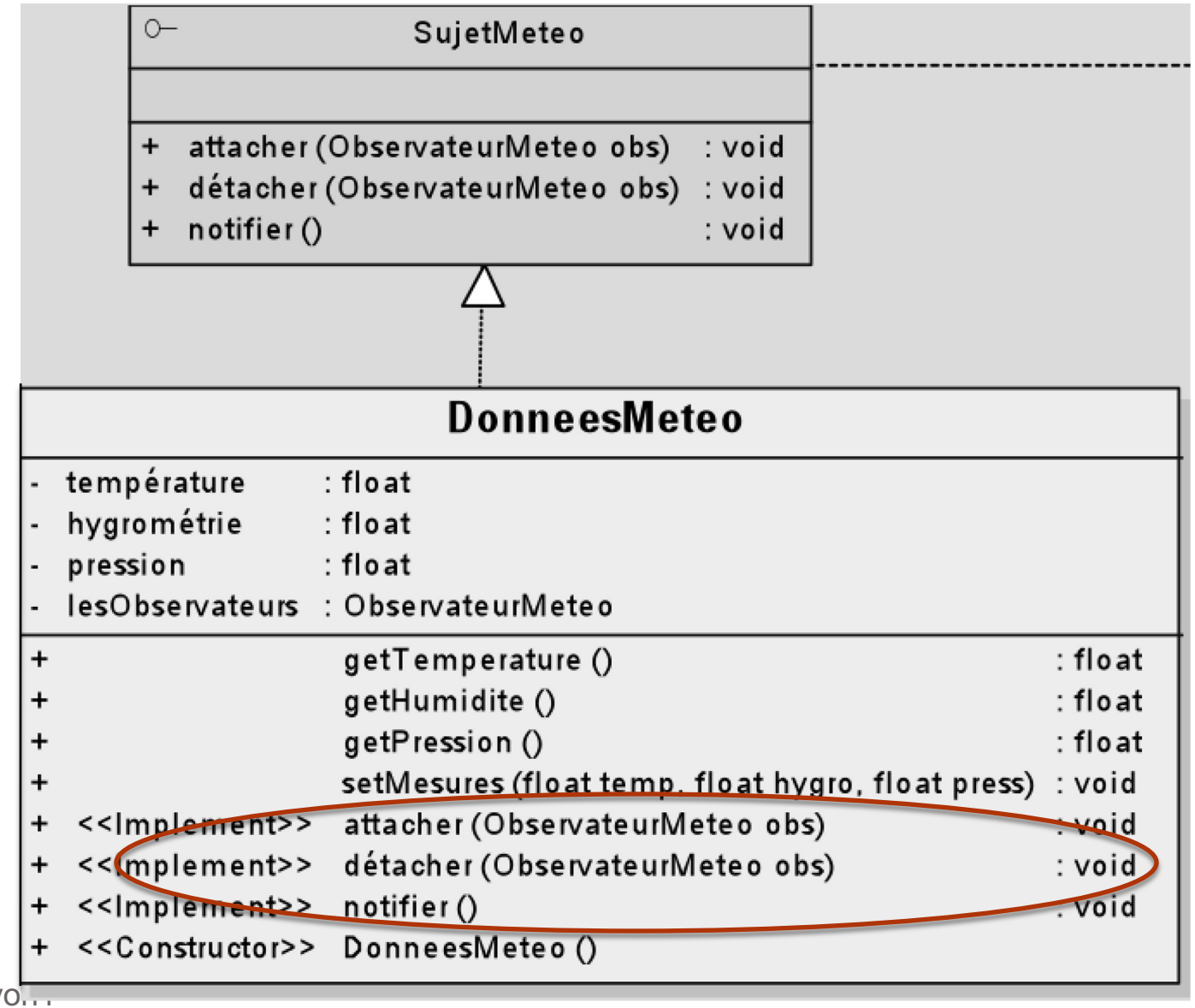
Pattern Observer sur l'exemple Météo

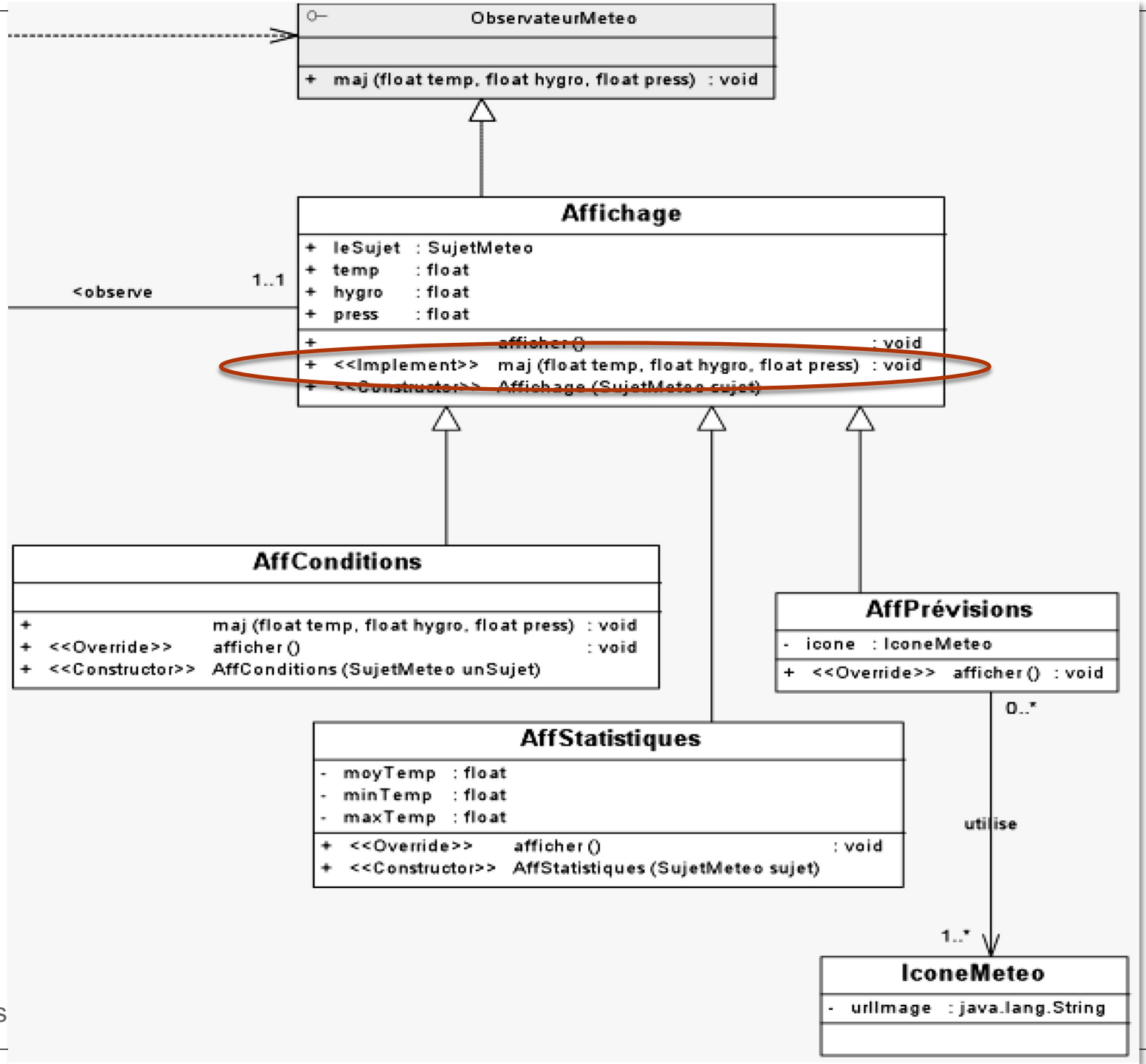
- Qu'est-ce qui **varie** dans cette application ? (les isoler dans des classes concrètes)
  - Les données de météo
  - L'affichage des 3 composants envisagés
  - Le nb et le type des affichages envisagés

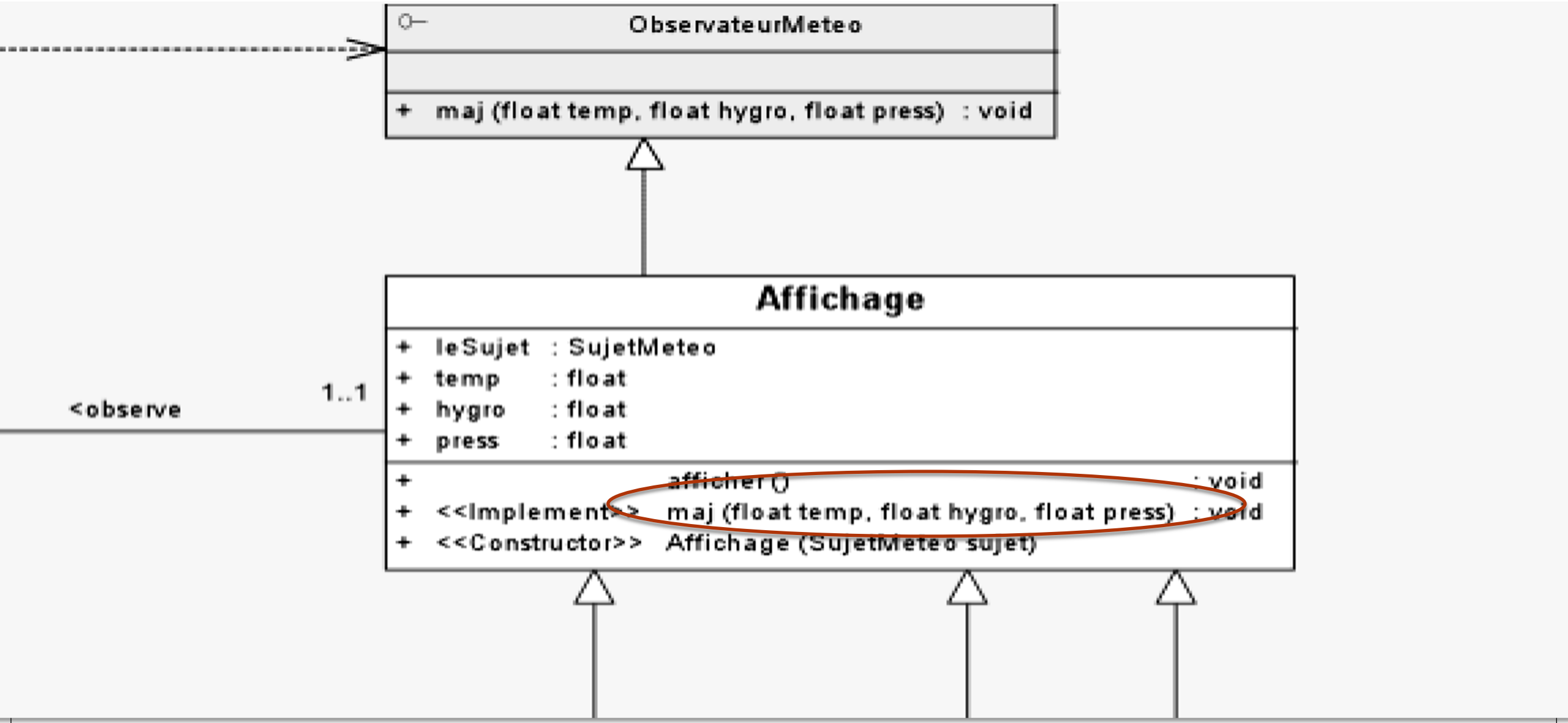
- Qu'est-ce qui est **stable** ? (l'encapsuler aussi)
  - La récupération des données météo : c'est la même qui va être effectuée par les 3 affichages envisagés
  - On crée une classe concrète Affichage avec la méth *update()*

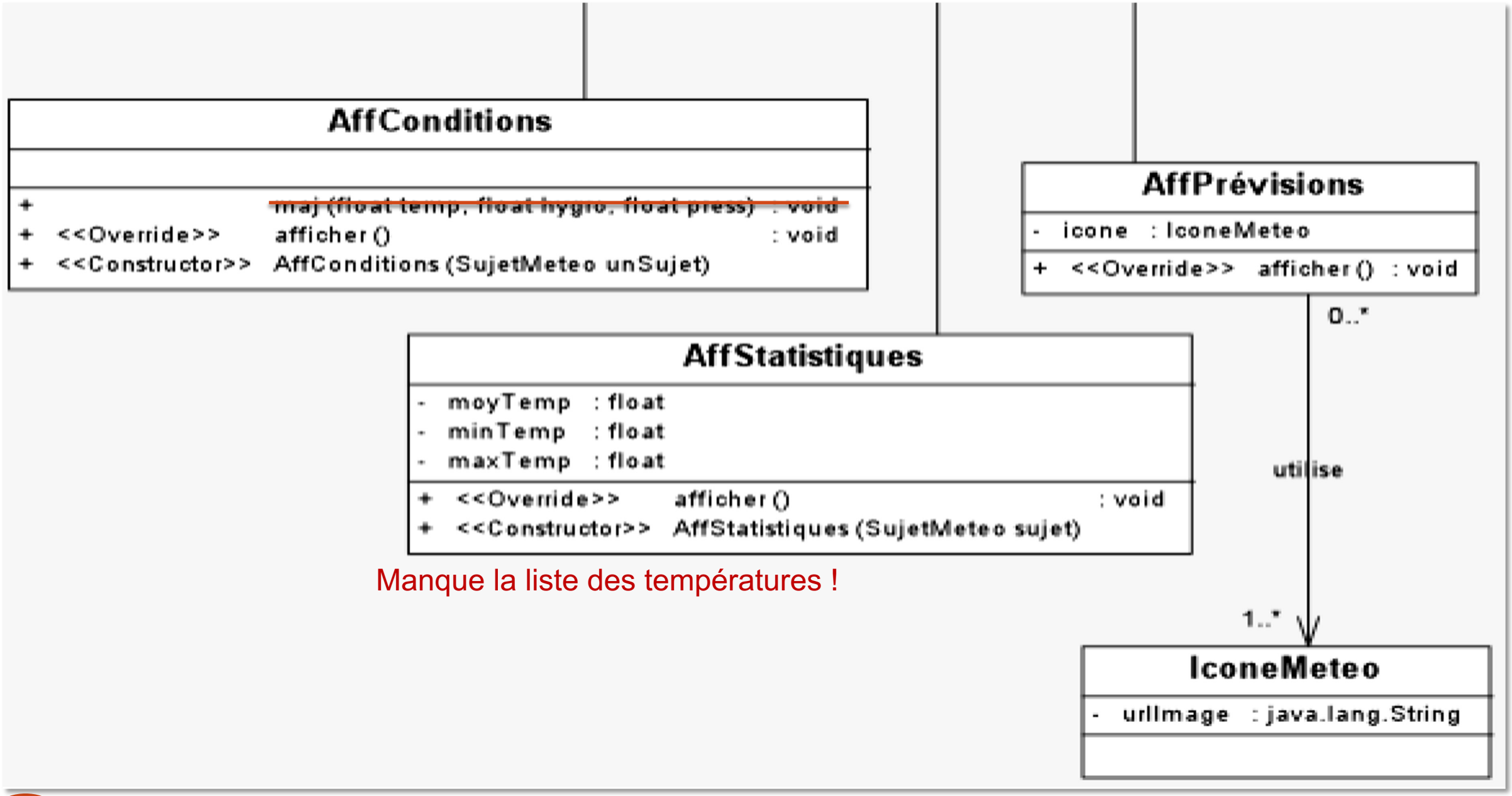
  (Les affichages doivent avoir une **interface commune** pour que le sujet Météo sache comment transmettre les modifications)

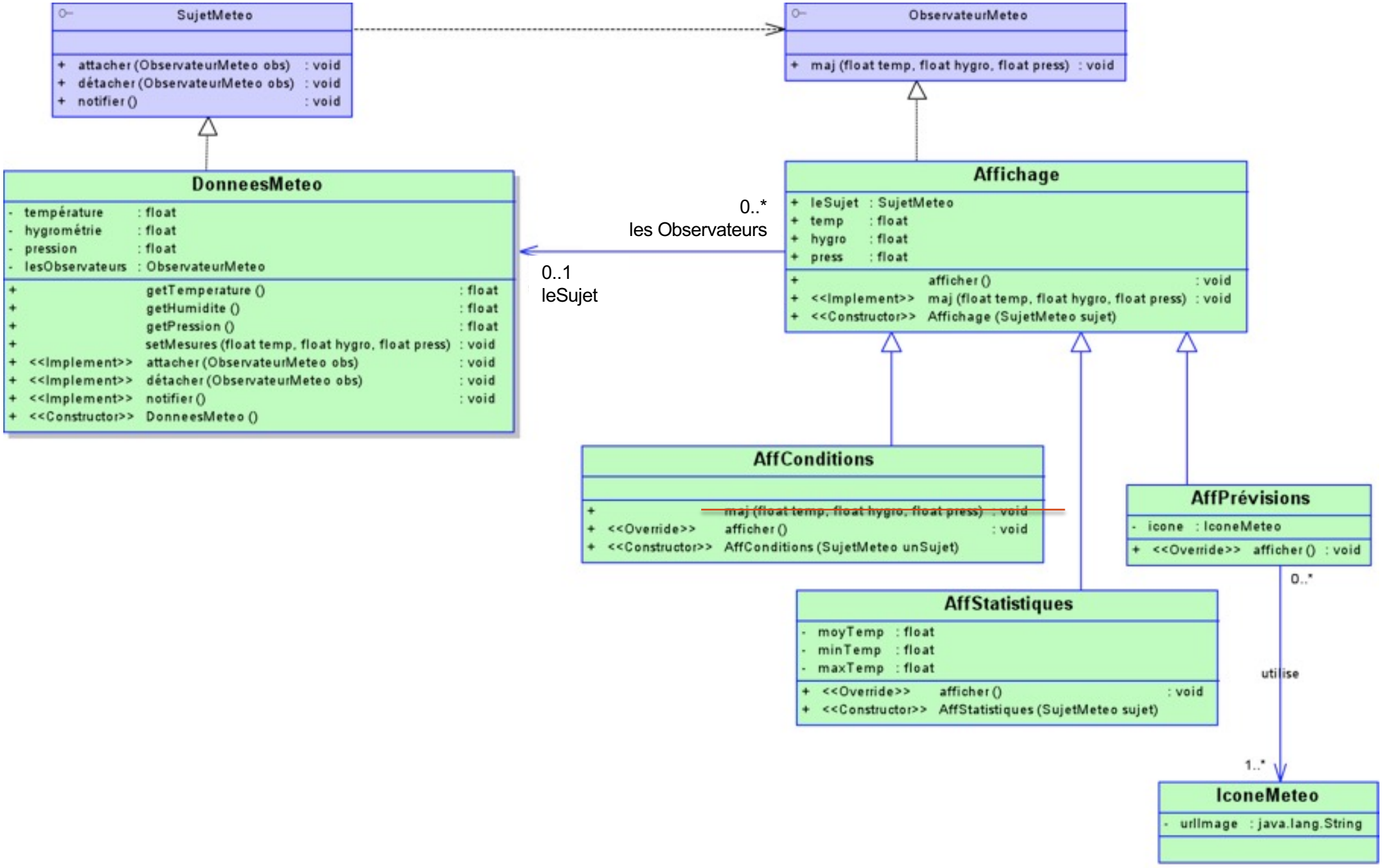
Pattern Observer sur l'exemple Météo




ObservateurMeteo
+ maj (float temp, float hygro, float press) : void
Affichage
+ leSujet : SujetMeteo
+ temp : float
+ hygro : float
+ press : float
+ afficher () : void
+ <<Implement>> maj (float temp, float hygro, float press) : void
+ <<Constructor>> Affichage (SujetMeteo sujet)
<observe
1..1
AffConditions
+ maj (float temp, float hygro, float press) : void
+ <<Override>> afficher () : void
+ <<Constructor>> AffConditions (SujetMeteo unSujet)
AffPrévisions
- icone : IconeMeteo
+ <<Override>> afficher () : void
0..*
AffStatistiques
- moyTemp : float
- minTemp : float
- maxTemp : float
+ <<Override>> afficher () : void
+ <<Constructor>> AffStatistiques (SujetMeteo sujet)
utilise
1..*
IconeMeteo
- urlImage : java.lang.String


ObservateurMeteo
+ maj (float temp, float hygro, float press) : void
Affichage
+ leSujet : SujetMeteo
+ temp : float
+ hygro : float
+ press : float
+ afficher() : void
+ <<Implement>> maj (float temp, float hygro, float press) : void
+ <<Constructor>> Affichage (SujetMeteo sujet)
<observe
1..1


AffConditions
+ maj (float temp, float hygro, float press) : void
+ <<Override>> afficher () : void
+ <<Constructor>> AffConditions (SujetMeteo unSujet)
AffPrévisions
- icone : IconeMeteo
+ <<Override>> afficher () : void
0..*
utilise
1..*
AffStatistiques
- moyTemp : float
- minTemp : float
- maxTemp : float
+ <<Override>> afficher () : void
+ <<Constructor>> AffStatistiques (SujetMeteo sujet)
Manque la liste des températures !
IconeMeteo
- urlImage : java.lang.String


SujetMeteo
+ attacher (ObservateurMeteo obs) : void
+ détacher (ObservateurMeteo obs) : void
+ notifier () : void
ObservateurMeteo
+ maj (float temp, float hygro, float press) : void
DonneesMeteo
- température : float
- hygrométrie : float
- pression : float
- lesObservateurs : ObservateurMeteo
+ getTemperature () : float
+ getHumidite () : float
+ getPression () : float
+ setMesures (float temp, float hygro, float press) : void
+ <<Implement>> attacher (ObservateurMeteo obs) : void
+ <<Implement>> détacher (ObservateurMeteo obs) : void
+ <<Implement>> notifier () : void
+ <<Constructor>> DonneesMeteo ()
0..*
les Observateurs
0..1
leSujet
Affichage
+ leSujet : SujetMeteo
+ temp : float
+ hygro : float
+ press : float
+ afficher () : void
+ <<Implement>> maj (float temp, float hygro, float press) : void
+ <<Constructor>> Affichage (SujetMeteo sujet)
AffConditions
+ maj (float temp, float hygro, float press) : void
+ <<Override>> afficher () : void
+ <<Constructor>> AffConditions (SujetMeteo unSujet)
AffPrévisions
- icone : IconeMeteo
+ <<Override>> afficher () : void
0..*
utilise
1..*
AffStatistiques
- moyTemp : float
- minTemp : float
- maxTemp : float
+ <<Override>> afficher () : void
+ <<Constructor>> AffStatistiques (SujetMeteo sujet)
IconeMeteo
- urlImage : java.lang.String

Météo : Extraits de code Java

```
public interface SujetMeteo {

  void attacher(ObservateurMeteo obs);

   void detacher(ObservateurMeteo obs);

   void notifier();
}
```

```
public class DonneesMeteo_sujetConcret
                                        implements SujetMeteo {

   private float temperature;
  private float hygrometrie;
  private float pression;

  // liste des observateurs du sujet :
  private ArrayList<ObservateurMeteo> lesObservateurs ;


  // constructeur
  public DonneesMeteo_sujetConcret ( float t, float h, float p ) {

        lesObservateurs = new ArrayList<ObservateurMeteo>();

        temperature = t; // affectation des valeurs
        hygrometrie = h;
        pression = p;

  }
```

Quelques méthodes de la classe DonneesMeteo :

```
public void setMesures(float t, float h, float p) {

    temperature = t; // affectation des valeurs

    hygrometrie = h;

    pression = p;

    this.notifier();  // notifie les observateurs

  }
```

```
public void notifier() {

    for (ObservateurMeteo obs : lesObservateurs)

            obs.maj(temperature, hygrometrie, pression);

  }
```

```
public void attacher(ObservateurMeteo obs) {

    lesObservateurs.add(obs);

    System.out.println("\n--> l'observateur  "+ obs.getClass().getName()+ "
a ete attache aux donnees Meteo...");

  }
```

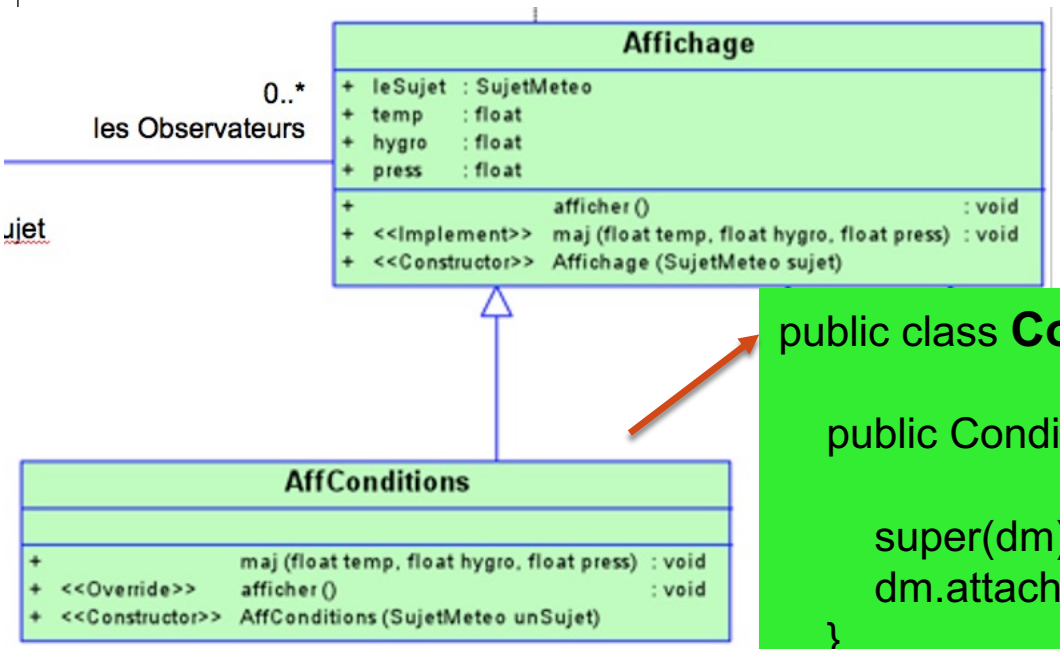
Météo : Extraits de code (2)

```
public interface ObservateurMeteo {

   void maj(float t, float h, float p);

}
```

Méthode update() d'Observer

```
public class Affichage_ObsConcret implements ObservateurMeteo {

   protected float temperature;
   protected float hygrometrie;
   protected float pression;

   protected DonneesMeteo_sujetConcret sujet;
/* on garde une reference sur le sujet pour s'enregistrer dans la liste de ses
observateurs */

   // Constructeur
public Affichage_ObsConcret(DonneesMeteo_sujetConcret dm) {
         this.sujet = dm;
   }

  // Actualise les dernieres valeurs et les affiche
   public void maj(float t, float h, float p) {

      this.temperature = t;
      this.hygrometrie = h;
      this.pression = p;

      this.afficher();

   }

   public void afficher() {
      // sera surchargee dans les sous-classes
   }
```

Météo : Extraits de code (3)




```
public class ConditionsMeteo extends Affichage_ObsConcret {

   public ConditionsMeteo(DonneesMeteo_sujetConcret dm) {

      super(dm);
      dm.attacher(this);
   }

   @Override
   public void afficher()  {

     System.out.println("\n*** Conditions actuelles :");
     System.out.println("- temperature :"+ temperature + " degres C");
     System.out.println("- hygrometrie :"+ hygrometrie + " %");
     System.out.println("- pression :"+ pression);

    }
```

```
public class Main {

    public static void main(String arg[] ) {

        DonneesMeteo_sujetConcret dm = new DonneesMeteo_sujetConcret(6f, 40.0f,
20.0f);

        // création de 2 observateurs affectés à cette source :

        ConditionsMeteo conditionsMeteo = new ConditionsMeteo(dm);
        StatMeteo statMeteo = new StatMeteo(dm);

        System.out.println("\nNb d'obs : "+ dm.getLesObservateurs().size());

       // simulation des arrivées de nouvelles valeurs :

        System.out.println("\n##### MIDI Collecte de nouvelles donnees #####");
        dm.setMesures(10f, 35.6f, 22.7f);
```

```
System.out.println("\n####### 15h Collecte de nouvelles donnees ######");
dm.setMesures(12.5f, 3f, 27.3f);

System.out.println("\nOn detache l'affichage des previsions...");
dm.detacher( previsionsMeteo );

System.out.println("\n###### 19h Collecte de nouvelles donnees ######");
dm.setMesures(10.5f, 35.6f, 22.7f);

// ajout d'un nouvel observateur :
previsionsMeteo = new PrevisionsMeteo(dm);
System.out.println("\nNb d'obs : "+ dm.getLesObservateurs().size());

System.out.println("\n###### 21h Collecte de nouvelles donnees ######");
dm.setMesures(8f, 12f, 2f);

} // du main
```

Remarque sur l'exemple

- Ici on a choisi d'implémenter nous-mêmes les classes du DP Observer
- Dans l'API Java, le pattern OBSERVER existe avec les classes Observer/Subject comprenant les méthodes *update()*, *attach() notify()*, etc.

- Avec ce DP de l'API, on peut choisir si on **pousse** ou on **tire** les données modifiées
  - En général, le mécanisme du 'pull' est jugé meilleur

Quand utiliser Observer ?

- Lorsqu'une modification d'un objet requiert celle d'autres objets, et qu'on ne sait pas combien d'objets il y a à modifier.

- Lorsqu'un objet devrait être capable de notifier d'autres objets, sans avoir besoin pour autant de connaître ces objets.
  - Càd. quand on ne souhaite pas **coupler** de façon trop forte des objets dépendants

FactoryMethod

Un pattern de **création** ciblé sur les **objets**

Problématique

- Ce pattern est utilisé lorsqu'on ne parvient pas à assurer les principes de *forte cohésion* et de *faible couplage* au sein d'un module.
- Le concepteur doit inventer de nouveaux concepts abstraits sans relation directe avec les objets du domaine. Ces concepts sont génériques.
  - Par exemple pour résoudre sa problématique de stockage, un concepteur va imaginer une classe chargée de la persistance des données. Ce n'est pas un objet métier, le concepteur choisit si on passe par un système de fichiers, une BD relationnelle ou une base noSQL.
  - Ici pour être générique, on délègue à une autre classe la responsabilité du dialogue avec la persistance. Pour éviter le couplage trop serré, on crée une Classe intermédiaire (abstraite, haut niveau) afin d'éviter de coupler directement l'objet Métier à son enregistrement en BD (implémentation, bas niveaux).

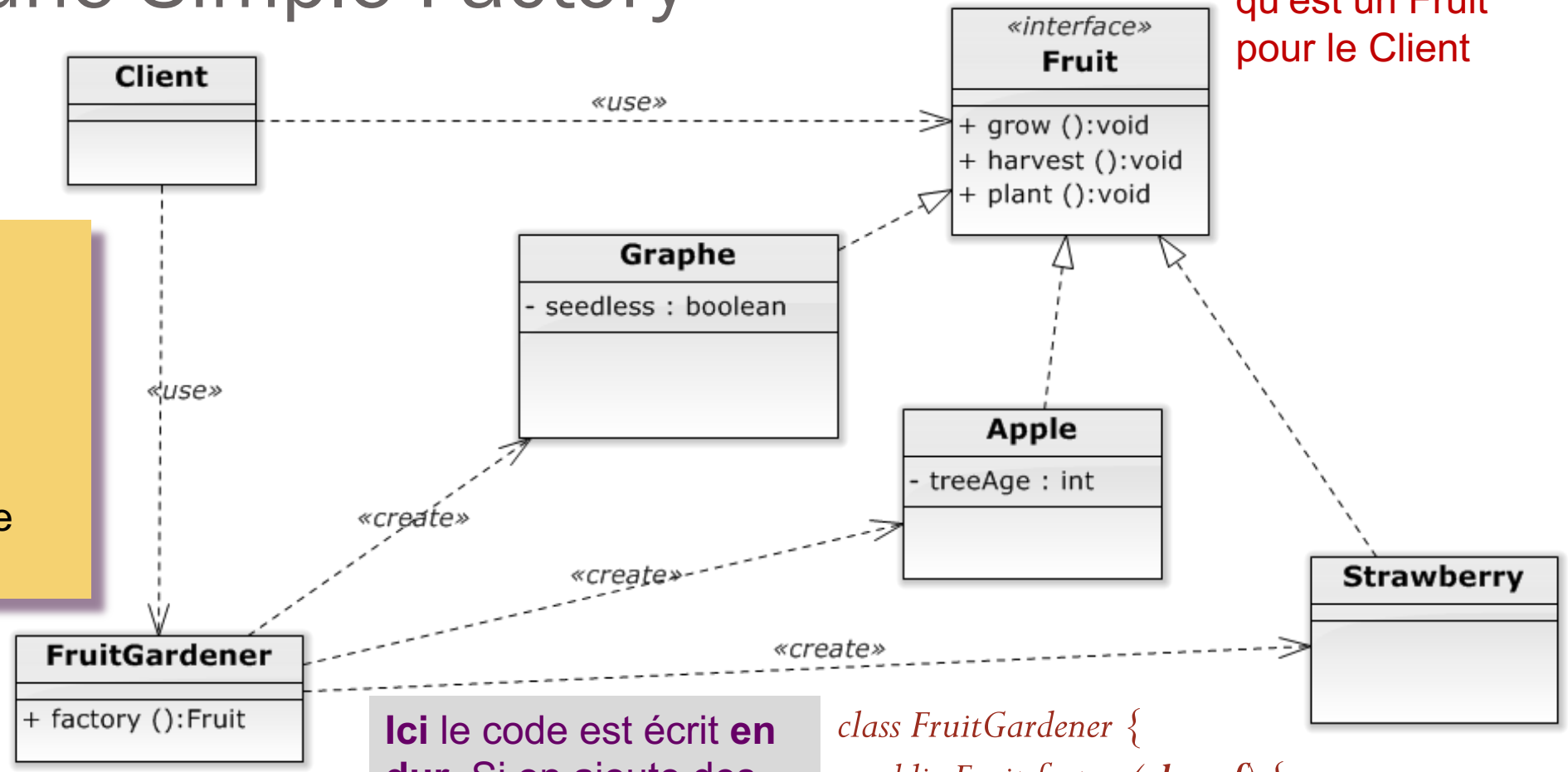
Simple Factory (pas un vrai pattern)

Un idiome de programmation

- Objectif : **créer un objet dont le type dépend du contexte**
- Contexte :
  - Une classe Client a besoin de créer des objets d'une famille de classes, dont le type peut varier en cours d'exécution.
    - Le Client n'a pas besoin de connaître dans quel cas créer tel ou tel type.
- Principe : passer par une classe spéciale, chargée de CRÉER les objets spécifié par le Client (paramètre)
  - L'objet retourné est donc toujours **du type de la classe mère**
  - Grâce au **polymorphisme** les traitements exécutés sont ceux de l'instance créée

Exemple d'une Simple Factory

Abstraction de ce qu'est un Fruit pour le Client



Un client demande des fruits au jardinier… il ne connait pas les classes, juste leur nom et leur contrat, par ex.:

factory(a) pour avoir une *Apple*

La simple Factory

Ici le code est écrit **en dur**. Si on ajoute des fruits, on doit modifier son code et faire l'appel au nouveau constructeur

```
class FruitGardener {
  public Fruit factory(char f) {
    if (f == 'a')
      return new Apple()
    else if (f == 'g')
      return new Graphe()
    etc...
```

Exemple informatique

Je veux fournir un **composant réutilisable implémentant un dictionnaire (ou une table associative)**.

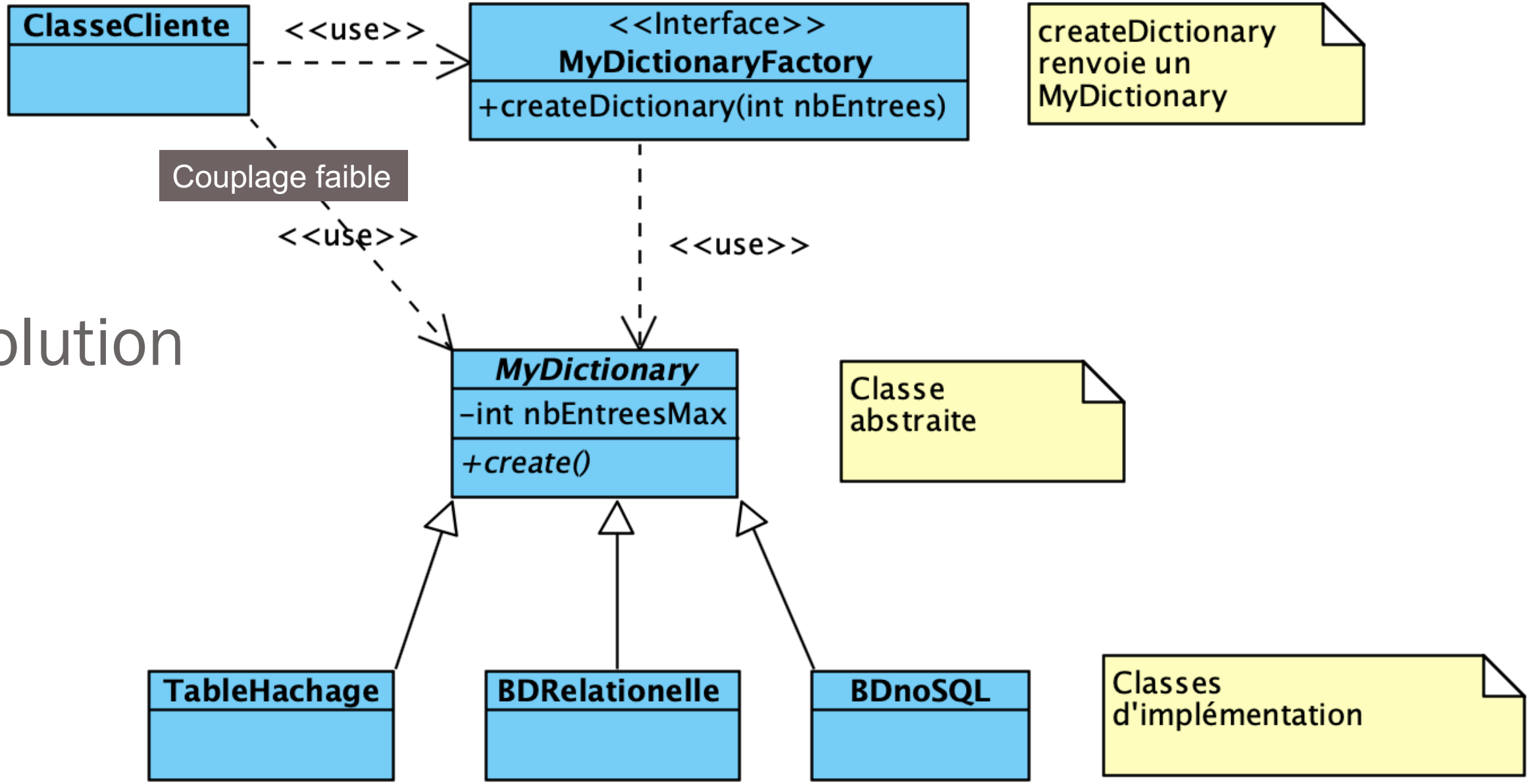
- Contraintes :
  - Ce dictionnaire doit être implémenté à l'aide de la structure de données la plus adaptée possible compte-tenu du nombre d'entrées qu'il va contenir

    Par exemple, une table de hachage tant que sa taille n'excède pas quelques mégas, une base de données sinon.
  - Les prochaines versions pourront intégrer de nouvelles structures de données.

- Solution envisagée : j'utilise une **classe abstraite `MyDictionnary`** et je définis une sous-classe **par structure de données à considérer**.

- Problème : Comment bien choisir la structure de données sans rendre publique l'existence des sous-classes ?

Solution


ClasseCliente
<<use>>
<<Interface>>
MyDictionaryFactory
+createDictionary(int nbEntrees)
createDictionary renvoie un MyDictionary
Couplage faible
<<use>>
<<use>>
MyDictionary
-int nbEntreesMax
+create()
Classe abstraite
TableHachage
BDRelationelle
BDnoSQL
Classes d'implémentation

Explications

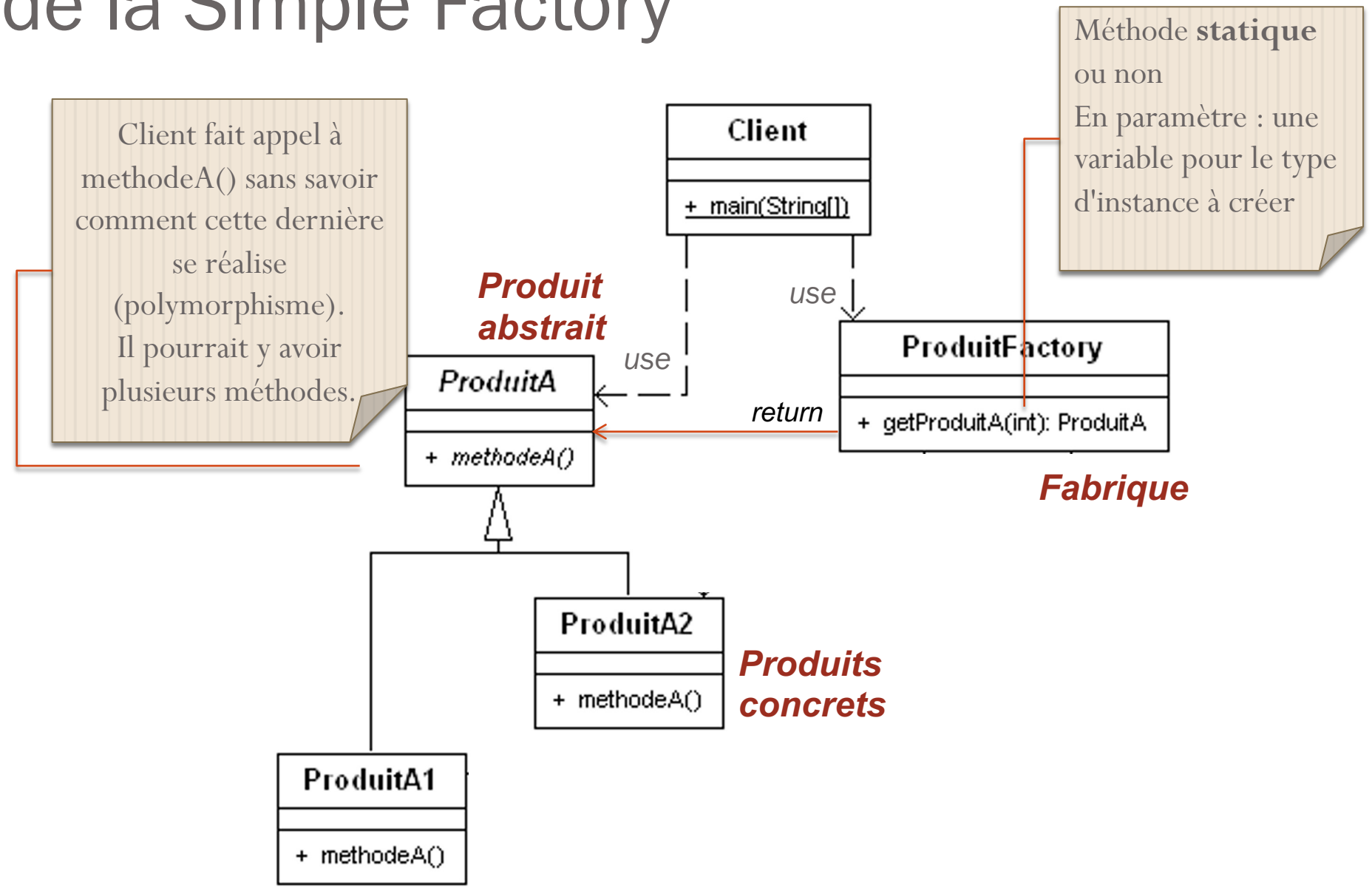
- On fournit une classe `MyDictionaryFactory`
  - qui répond au message `CreateDictionary (n)`

    où `n` est le nombre potentiel d'entrées dans le dictionnaire

- L'implémentation de cette classe choisit la **bonne sous-classe** et en retourne une instance, qui est vue comme un `MyDictionary`.

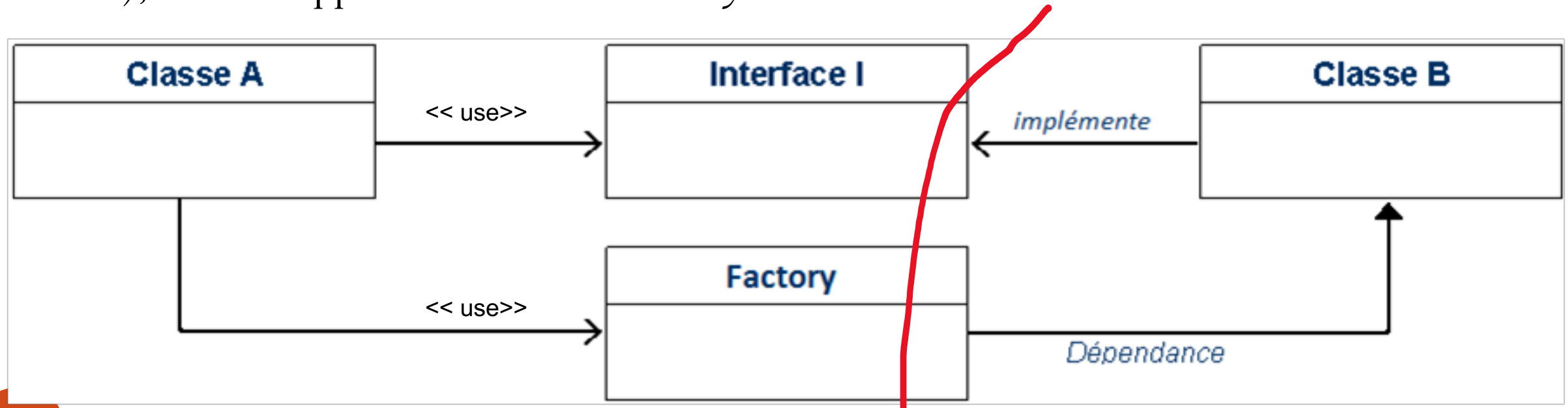
- On ne rend donc public que `MyDictionaryFactory` et `MyDictionary`
  - Pas les sous-classes

DCL de la Simple Factory



Implémentation du DIP

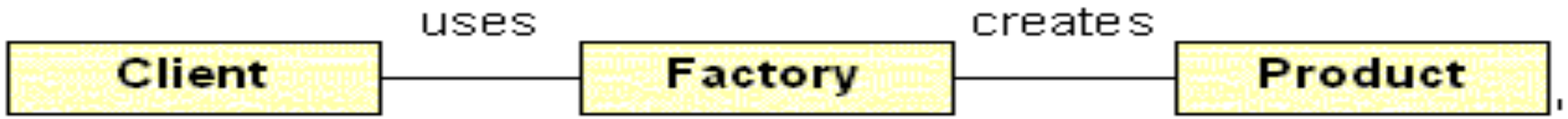
- *Inversion des Dépendances*
- C'est une classe **Factory** qui va gérer les dépendances vers les classes B (de bas niveaux), pas le Client.
  - *Factory* possède les méthodes qui vont instancier la dépendance et la retourner.
- Chaque fois qu'une dépendance devra être résolue (besoin d'un objet de type **Interface I**), la classe appelante utilisera la *Factory*.



Design pattern « Factory Method »

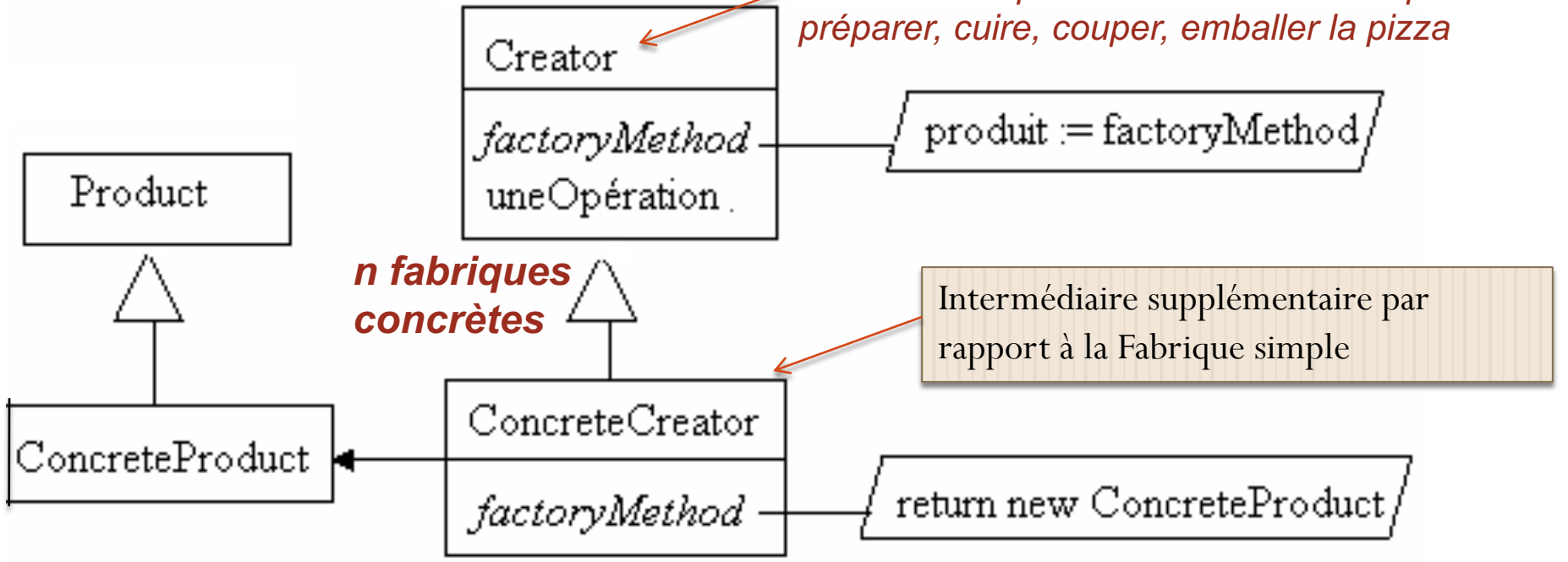
- On ajoute une **abstraction** supplémentaire : c'est une classe abstraite (*la fabrique*) qui délègue l'instanciation des objets (*les produits*) à une *fabrique concrète*, et il peut y en avoir *n*.
  - *Ex. créer des pizzas, pizzas de Brest ou de Marseille, de Brest végétarienne ou aux lardons, de Marseille végétarienne ou aux fruits de mer*
  - On **factorise le mécanisme de création, qui est commun à tous les produits** : une pizza se prépare, se cuit, se coupe et s'emballe, pour toutes les pizzas ; la pizza créée sera spécifique au besoin de création.
- Permet à un client de créer un « type » d'objet, sans qu'il ait à connaître son type précis
    - *Ex.: un dictionnaire (hashMap ou BD), une sauvegarde (quel type ? Peu importe pour le client)*
- Retourne une instance du produit spécifique créé (produit adapté)
  - Faciliter la création en respectant le principe d'O/F de code

Principe de Factory Method



Fabrique abstraite

Définit le comportement commun des produits : préparer, cuire, couper, emballer la pizza



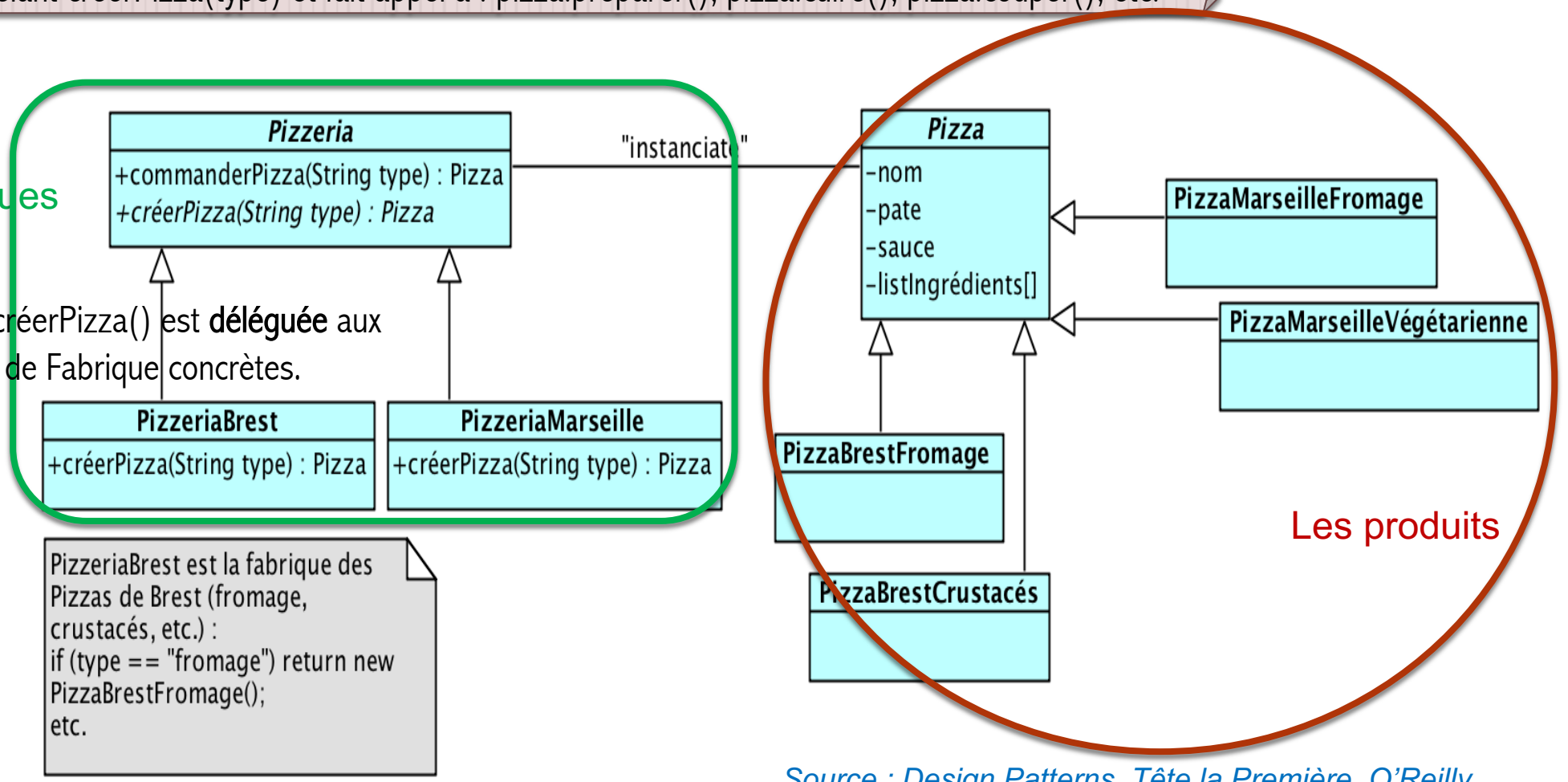
Définir une interface de création, et laisser les sous-classes décider du type

Illustration Factory Method

commanderPizza() est la méthode définissant le comportement générique des pizzas : elle crée une pizza en appelant créerPizza(type) et fait appel à : pizza.préparer(), pizza.cuire(), pizza.couper(), etc.

Les fabriques

La méthode créerPizza() est **déléguée** aux sous-classes de Fabrique concrètes.



Source : Design Patterns, Tête la Première, O'Reilly

Autre avantage : nommer les choses…

- Une fabrique peut aussi servir à différencier les constructeurs de classe à l'aide d'un nom.
  - En effet, il n'est pas toujours évident de dissocier les constructeurs seulement à l'aide de leur signature (du type des arguments).
  - Dans l'exemple du Dictionnaire, on peut vouloir construire un dictionnaire à l'aide d'un entier correspondant :
    - au nombre moyen d'entrées ;
    - au nombre minimal d'entrées ;
    - au nombre maximal d'entrées.
  - Pb : les constructeurs auraient tous la même signature (un entier) !
- SOLUTION : donner un nom **explicite** aux services de la fabrique pour dénoter le type de création (*createFromMean*, *createFromMin*, *createFromMax*).

Abstract Factory

- Il existe une 3ème forme de design pattern Factory
  - Avec un **niveau d'abstraction supplémentaire**
- *Abstract Factory* définit une fabrique abstraite pour *chaque étape du processus commun* de création d'une instance :
  - Ex.: *FabriqueIngrédientPate, FabriqueIngrédientSauce*, cuisson etc., avec *n* implémentations pour chacune : *PateFine, PateFeuilletée, SauceTomate, SauceViande,* etc.
  - C'est la fabrique concrète qui choisit l'implémentation qui convient à chaque étape :
    - PizzeriaBrest, pour sa pizza Fromage, va utiliser la Fabrique concrète PateFine, SauceTomate etc. (dans son constructeur)

http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm