Chap.4 – Design patterns (part. 1)

V. Deslandres ©

Conception d'Architectures Logicielles

LP DevOps

IUT de Lyon - Université Lyon 1

Sommaire de ce cours

Les Design Patterns : c'est quoi ?

- **Design patterns** = Modèles de conception (*patrons de conception*) pour la POO
- Répondent à des problèmes **récurrents** de la conception OO
  - **Diminution du couplage**, **Séparation des rôles,** Indépendance vis-à-vis des plateformes, **Réutilisation** du code existant, Facilité **d'extension**
- Proposer un catalogue de **meilleures pratiques** issue de **l'expérience** de concepteurs chevronné
- **Analogie avec l'algorithmique** :
  - L'algorithmique concerne le corps des méthodes (intra classe), alors que les *patrons* concernent l'organisation des classes entre elles (inter classe)

*E. GAMMA, R. HELM, R. JOHNSON et J. VLISSIDES (the Gang of Four), Addison-Wesley, « **Design Patterns – Catalogue de Modèles de Conception Réutilisables** », International Thomson Publishing France, 1996*

Bénéfices des Design Patterns

1. **Capitalisation** de l'expérience et **réutilisation** de solutions
   - Plus puissant que la réutilisation de codes
   - Amène souvent la réutilisation de *composants*
2. **Vocabulaire** commun pour la conception
   - « On fait un Singleton ? »
3. Niveau d'abstraction **plus élevé**
   - Constructions logicielles de meilleure qualité
4. Souvent : meilleure **robustesse**
   - Impression de simplicité (ex. les IHM avec OBSERVER)
   - L'API Java en utilise beaucoup dans ses bibliothèques (ex. les flux Java sont des DECORATOR, les menus reposent sur COMMAND)

Les inconvénients

- Effort de synthèse
  - Difficile à **reconnaître**
  - Haut niveau d'abstraction
- Les patrons **se « dissolvent »** dans le code
- Ils sont **nombreux**
  - Lesquels sont identiques ?
  - Pas tous du même niveau :
    - Certains patterns s'appuient sur d'autres
- Ils nécessitent un **temps d'apprentissage**
  - Pas toujours facile sur du code en production
  - Passer par des exercices : *seule la pratique* permet d'en voir les avantages

Typologie des Design Pattern / Description

- **Classification des 23 patrons de Gamma :**
  - Selon la **fonction**
    - modèle de **création**,
    - modèle de **structure** (assemblage d'objets),
    - modèle **comportemental**
  - Selon la **portée**
    - **classes** : héritage
    - **objets** : encapsulation, délégation
- Exemples :
  - Pattern Composite = structurel / objets
  - Pattern Abstract Factory = création / classes

Description standard :

- **Nom** du design pattern
- **Objectif** = but
- **Problème** = qu'il s'efforce de résoudre
- **Solution** = proposée (contexte donné)
- **Participants** = entités impliquées
- **Conséquences** = ce qui se passera en implémentant le pattern
- **Implémentation** = mise en œuvre concrète (DCL)
- Référence GoF

Patrons de Création

- Objectif **: proposer différentes « formes » de création**

- Abstraire le processus d'instanciation
  - Cacher ce qui est créé, qui crée, où, comment et quand.
- Rendre indépendant la façon dont les objets sont créés, composés ou initialisés

Patrons de Structure

- **Expliciter les formes de structure**
  - Comment les objets sont assemblés
  - Comment les patrons sont complémentaires les uns des autres

- En Conception Objet, la structure porte sur :
  - Les **classes**
  - Les **packages**
  - Les **composants** physiques

Patrons de Comportement

- Décrire les formes de comportement :
  - Les algorithmes
  - Les comportements entre objets
  - Les formes de communication entre objet

- Objectif : concevoir des modules **à forte cohésion** et **faible couplage**

Image : titline.fr

Présentation de quelques Patrons de Conception

State, Façade, Singleton, Strategy, Composite

	Patrons créateurs	Patrons structuraux	Patrons comportementaux
Classes	*Factory Method*	*Adapter** (class)	Interpreter *Template Method*
Objets	Abstract Factory Builder Prototype *Singleton**	Adapter (object) Bridge *Composite* *Decorator* *Façade** *Proxy*	Command Iterator Mediator Memento *Observer** *State** *Strategy** Visitor

* *vus dans ce cours*

Le pattern Singleton

Pattern de Création à portée Objets

http://yavkata.co.uk

Design pattern « Singleton »

- Une des techniques les plus utilisées en conception objet
- « Comment s'assurer de n'instancier qu'**une seule fois** une classe (utilisée plusieurs fois) ? »
- Permet de référencer l'instance d'une classe lorsqu'elle est, ***par construction***, le seul et unique représentant de la classe
  - Ex. : une connexion à une BD, un fichier de log, un spooler d'imprimante, un gestionnaire de cache, le moteur d'un jeu, etc.
  - Permet aussi de limiter l'usage des ressources.
- Objet unique : accessible par les autres instances de classes.

Calendar : un Singleton

- ex. classe ***java.util.Calendar*** utilise un singleton pour renvoyer la date courante
- Un singleton est donc une classe appelée *Singleton* composé d'un attribut :
  - *instance* qui recevra la référence de l'objet unique
- et d'une opération :
  - *getInstance()* qui va chercher cette référence et la stocke dans *instance* si l'objet existe

Singleton : caractéristiques

- `getInstance(): Singleton` se charge d'automatiquement construire l'objet **unique** au 1er appel :

```
public synchronized static Singleton getInstance() {
        if (_instance == null)
          _instance = new Singleton();
        return _instance;
}
```

- Le constructeur est **privé**
- `Synchronized` empêche toute instanciation multiple, même par différents threads

Pattern Singleton

```
                Singleton

private static Singleton uniqueInstance
....
singletonData

private Singleton()
public static synchronized Singleton
getInstance()
public static synchronized void
releaseInstance()
....
singletonOperation()
```

```
return
uniqueInstance
```

Variante : on peut aussi créer l'instance lors de la définition de la variable :

```
private static final Singleton _instance = new Singleton();
```

Du coup, plus besoin de synchronized : getInstance() retourne simplement l'instance.

Java

```
final class MonSingleton  {

    // variable de classe privée :
    private static MonSingleton uniqueInstance = null;
    // constructeur privé :
    private MonSingleton() {}
    // méthode qui crée une instance unique du singleton :
    public static synchronized MonSingleton getInstance() {
       if (uniqueInstance == null)
            uniqueInstance = new MonSingleton();
         return uniqueInstance;
    }
     // Méthode qui libère l'instance :
    public static synchronized void releaseInstance() {
     uniqueInstance = null;
    }
    // Autre méthode du singleton :
    public static void affiche() {
       System.out.println( »*** On est dans le Singleton");
    }
 } // de MonSingleton
```

Ex. Journalisation

- Pour la journalisation, le concepteur désire créer en local un **seul** et **unique** fichier de traces **par jour.**

➔ On va utiliser un *singleton*

➔ Et enrichir la méthode `getInstance()` pour contrôler la date de création

Un fichier de traces **par jour de la semaine**

```
public class FichierTraceJour {

   private Date _dateCreation;      // date de création (jour) du dernier ficherTrace créé
   private static FichierTraceJour _instance;  // l'objet fichierTrace créé pour le jour
    private FileOutputStream leFichier; // le fichier de trace du jour de semaine

   public static FichierTraceJour getInstance() {

   // la classe Calendar utilise aussi un Singleton pour la date courante :
      int day = Calendar.getInstance().get(Calendar.DAY_OF_WEEK);

         if (_instance == NULL || _dateCreation.getDay() != day) {
               _instance = new FichierTraceJour(day);
         }
         return _instance; // retourne l'instance du FichierTraceJour du jour
   }
```

```
   //constructeur (privé):
       private FichierTraceJour(int day) {
             if (NULL != leFichier) leFichier.close();
             setDateCreation( Calendar.getInstance() );
             leFichier = new FileOutpuStream(); // etc.
    }
   …
} //de FichierTraceJour
```

On enregistre la date de création de la dernière instance créée

Le pattern State

Pattern comportemental à portée Objets

Etat d'un objet ?

- Si l'état est défini par un seul attribut : ex.: une commande (validée, en cours, etc.)
  - Imaginons que **plusieurs méthodes** peuvent modifier cet état, par ex.: définir(), ajouterProduit(), annuler(), définirDateLivraison(), archiver()
  - De fait on aura peut-etre besoin de distinguer ces différents états : en cours / validée / livrée / archivée … pour savoir ce qu'il est possible de faire.

→ La logique dépend de l'état de l'objet

→ Elle est répartie dans différentes méthodes de la classe

→ Code répété : `si (état == e1) faire ceci sinon si (état == e2) faire cela,` etc…

L'idée de **STATE** :

→ Constituer des classes pour chaque Etat et répartir cette logique dans ces classes.

DP State

Notion de workflow

- **Objectif**
- Il permet à un objet de **modifier son comportement** quand son état interne change.
- Permet d'exécuter des actions **en fonction d'un contexte** et de définir **l'état suivant**

Exemple : commande de produits

- Une commande possède une liste de produits
- Elle passe de l'état en cours, à validée, puis livrée et archivée.
- Seule une commande en cours peut voir sa liste de produits évoluer.
- Une commande validée dont la livraison est effectuée passe à l'état 'Livrée'
- Après une période définie (12 mois), la commande est archivée.

Première implémentation

- On pourrait traiter la commande **de façon unitaire**, de bout en bout, en contrôlant les états et les traitements sur la commande.
- Exemple :

  À la **création** d'un nouvelle commande : `setEtat("enCours");`

  Dans la méthode `ajouterProduit()` :

  ```
  if (this.état == "enCours")        // ajouter un produit
  ```

  Idem pour **modifier / supprimer** un produit d'une commande

  Dans la méthode `setDateLivraison()` :

  ```
  if (this.état == "validée")        // définir la date de livraison, l'état passe à : « à livrer »
  ```

  Dans la méthode `setLivraisonEffectuée(boolean b) :`

  ```
  if (this.état == "à livrer")       // on vérifie que la commande devait être livrée, et on l'archive
  ```

  Etc.

- Rend interdépendants les différents traitements
- Code difficile à faire évoluer et à maintenir : par exemple, si on introduit la possibilité d'annuler une commande (au moins 24h avant sa livraison) → nouveaux tests, enchevêtrement des logiques, etc.

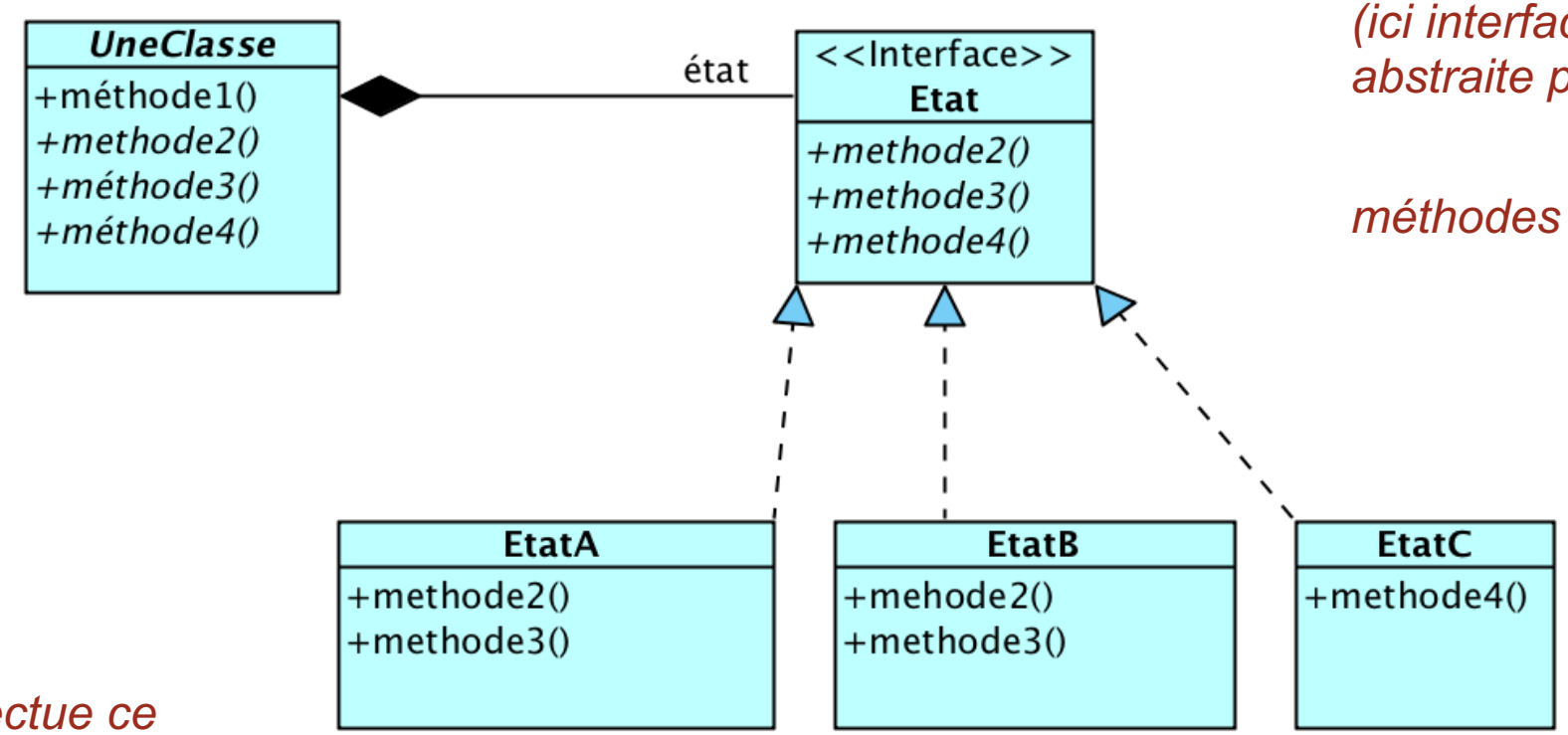
Coder avec le DP State

- L'idée est de gérer chaque étape du traitement de la commande, **indépendamment**.
- On crée des objets pour chaque étape, chacun ayant les comportements dédiés à chaque étape :
  - Ex. état 'Validée' : on peut définir la date de livraison, mais pas modifier les articles ;
  - Cela permet d'ajouter par la suite de nouveaux états, sans modifier ce qui existe (OCP).
  - Sépare clairement les rôles des étapes (SRP).

- Chaque classe Etat mentionne aussi **l'état suivant**, une fois le traitement effectué : cela permet de **modéliser le processus.**

DP State

*La classe de contexte, qui décrit **tous les comportements possibles**, et possède **différents états** EtatA, EtatB…, où seul l'état courant est gardé*



(ici interface, mais classe abstraite possible)

méthodes abstraites

*Chaque opération effectue ce qui **correspond à l'état actuel** de la classe et précise **l'état suivant***

methode2() et methode3() :
//… traitement quand dans l'EtatA,
puis :
uneClasse.**setEtat(new EtatB())**;

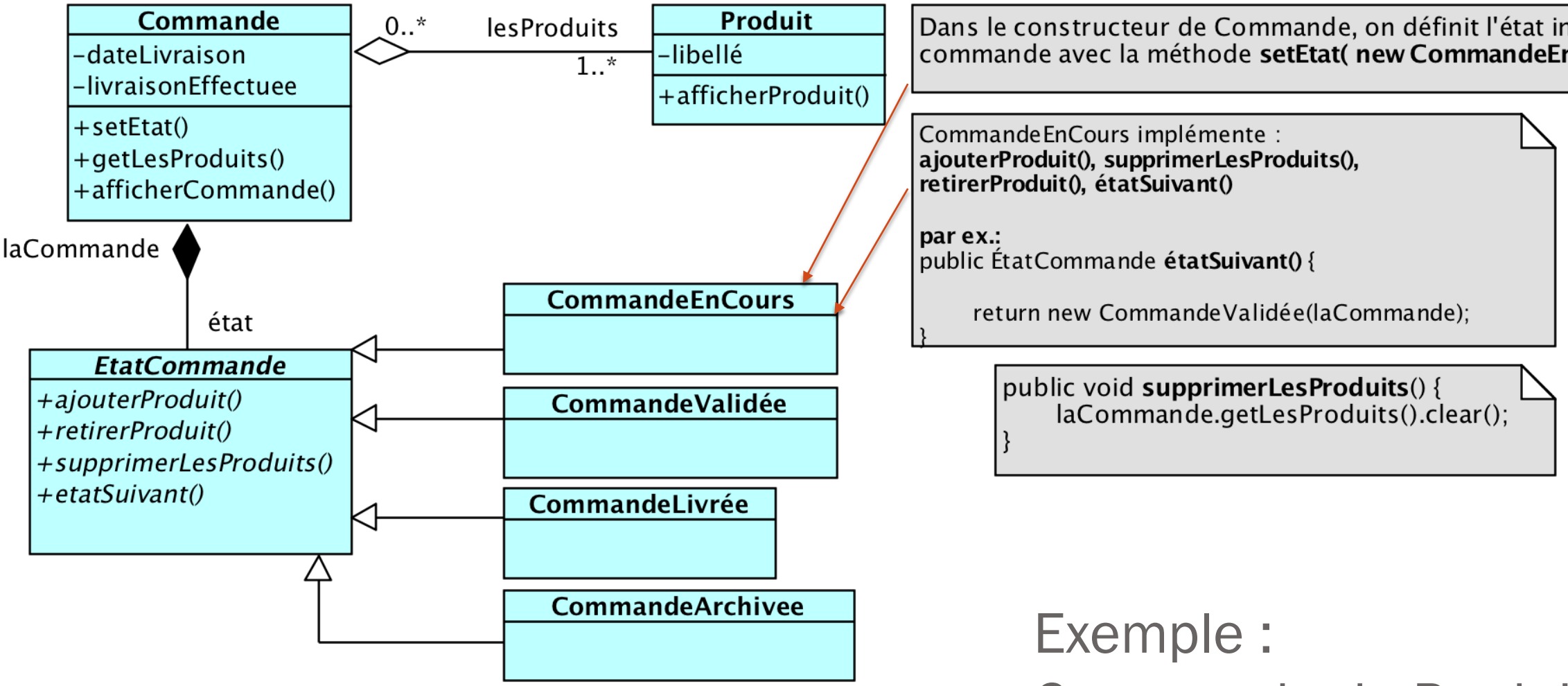
methode2() et methode3() :
//… traitement qd en EtatB, puis
uneClasse.**setEtat(new EtatC());**

Participants

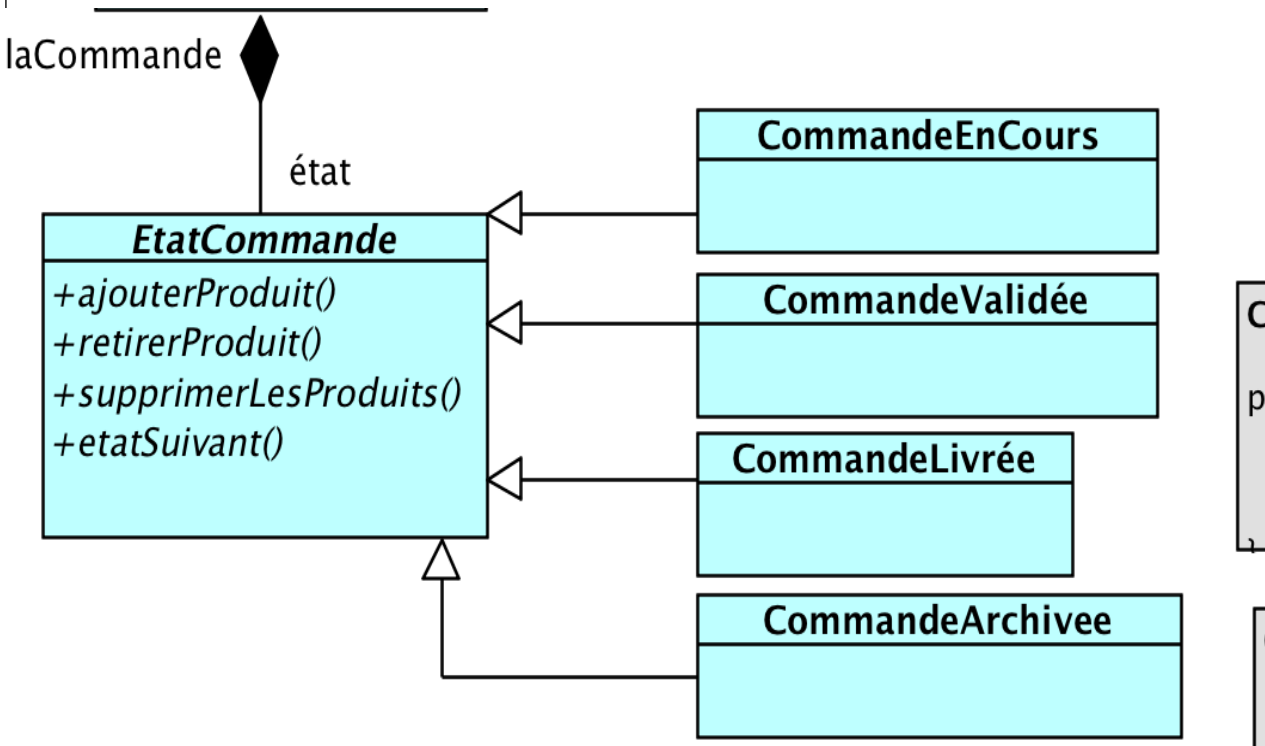
- *UneClasse* : définit l'objet dont ont veut gérer l'état. L'attribut *_état* définit l'état courant de l'objet. Cet attribut est lui-même un objet implémentant « *Etat* ».
- *Etat* : Définit une interface pour *encapsuler le comportement* correspondant à un état de l'objet
- *Etat1*, *Etat2*…: sous-classes définissant chacune un état concret et surtout le comportement possible de cet état (faire passer l'objet d'un état à un autre). Les états n'ont pas conscience les uns des autres. On peut en ajouter, en supprimer, sans modifier *UneClasse*

Fonctionnement

- *UneClasse* délègue les invocations des opérations à l'objet *« Etat »* représentant l'état courant.
- Le changement d'état d'un objet est défini dans les opérations des sous-classes *Etat1*, *Etat2*…



Exemple : Commande de Produit



CommandeValidee n'implémente que :

```
public ÉtatCommande étatSuivant() {
      if (laCommande.livraisonEffectuee)
            return new CommandeLivree(laCommande);
      return this;
}
```

CommandeArchivee :

```
public ÉtatCommande étatSuivant() {
     return this;
}
```

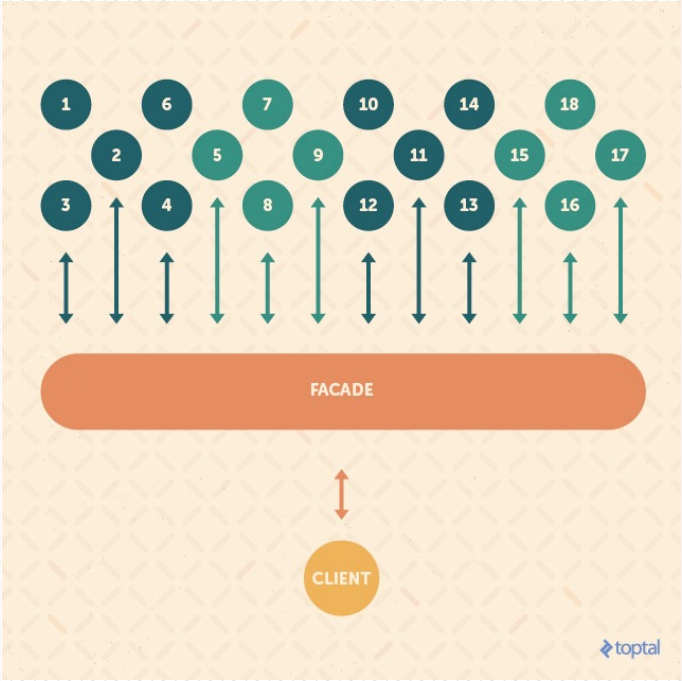
CommandeLivree :

```
public ÉtatCommande étatSuivant() {
     if ( laCommande.dateLivraison.getYear()+1 == Calendar.getDate().getYear()
)
            return new CommandeArchivée(laCommande);
     return this;
}
```

Cf TD

Le pattern Façade

Un modèle de conception de type Structure à portée Objets



Le pattern « Façade »

- **Problème :** on a besoin de n'utiliser qu'un sous-ensemble d'un système existant
- **But** : offrir une interface **s**implifiée à un ensemble de composants
- **Conséquence** : fournit une interface de plus haut niveau
  - Cela rend le sous-système plus facile à utiliser
  - Mais certains fonctionnalités pourront rester inaccessibles au client.

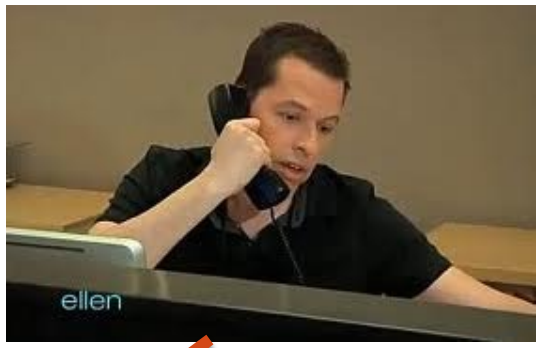
Pattern Façade (2)

- **Implémentation :** définir une nouvelle classe possédant l'interface requise; implémenter cette classe à l'aide des fonctionnalités du système existant

- **Cas d'utilisation** :
  - Soit interface actuelle pas assez conviviale
  - Soit on cherche à utiliser le système d'une façon particulière
    - ex. utiliser un logiciel 3D pour faire de la 2D
    - On va isoler les fonctionnalités utiles pour la partie Cliente
  - Soit on veut limiter l'accès à une partie du sous système

Ex. un Standard

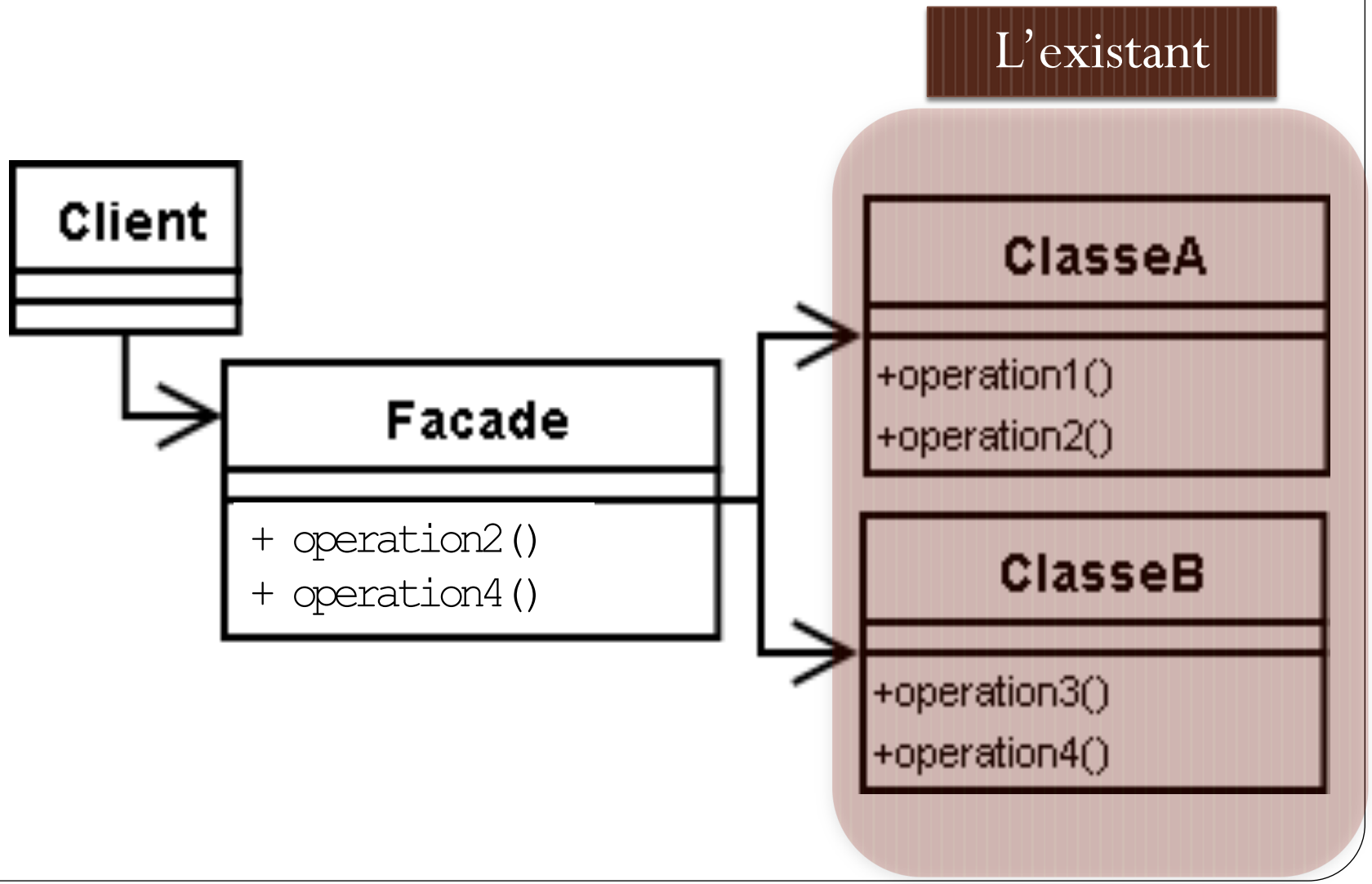
Le standard masque la complexité de l'organisation, le client qui appelle n'en connaît pas la structure



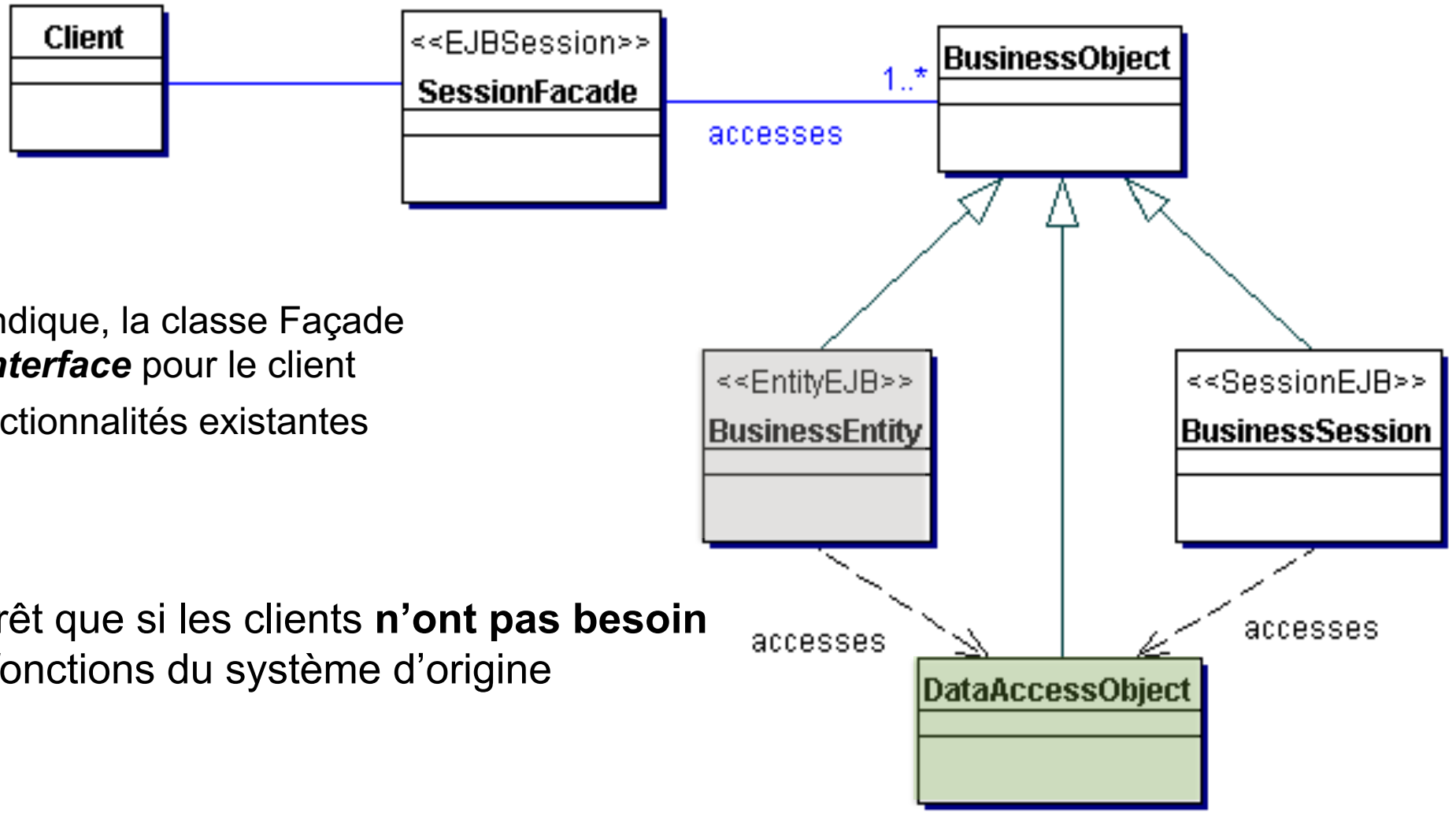
Service Commercial

Service Approvisionnement

Comptabilité



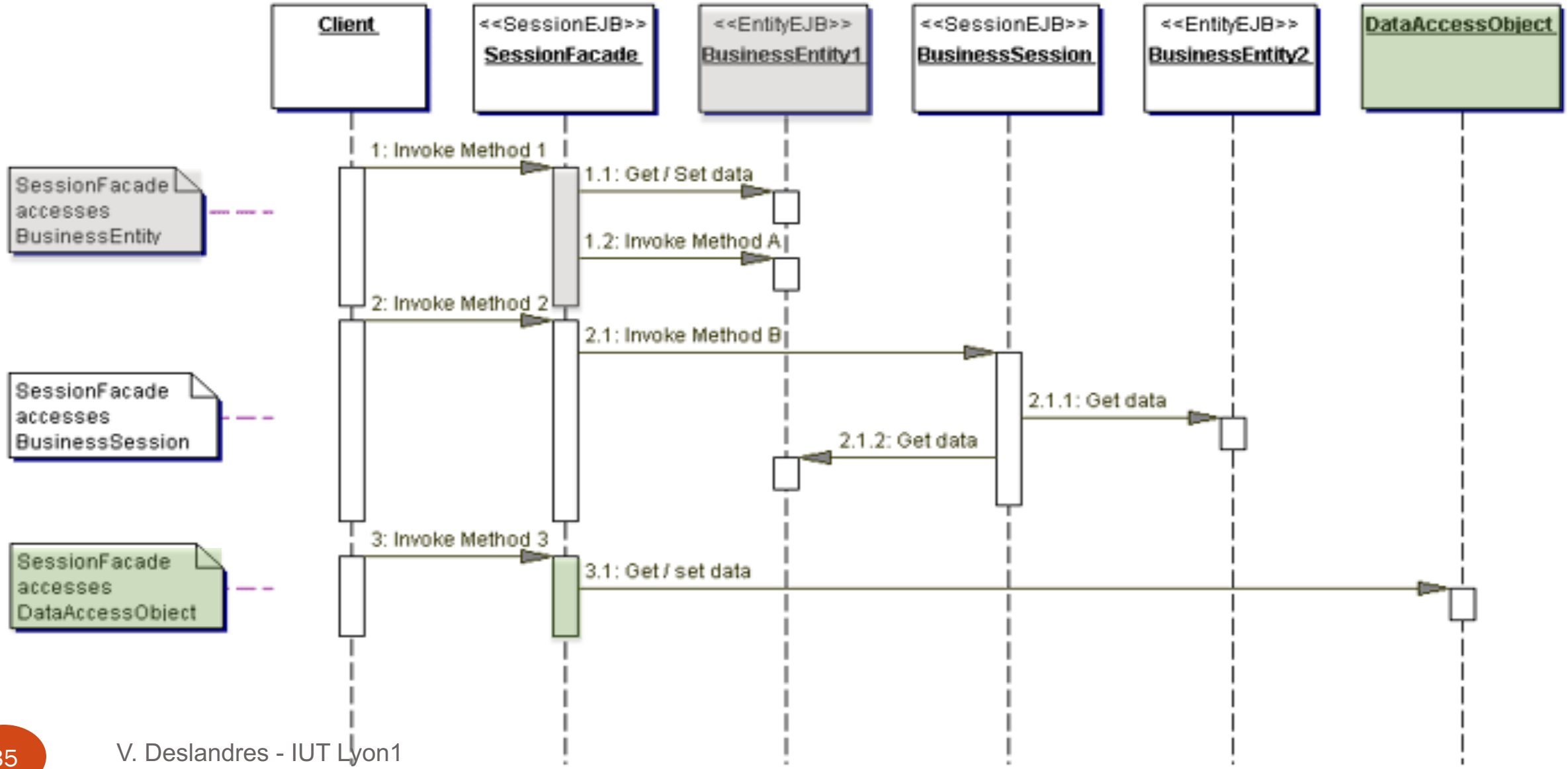
Ex. Pattern Façade : accès aux services et données avec JEE



- Comme son nom l'indique, la classe Façade offre une nouvelle ***interface*** pour le client
- Élaborée sur les fonctionnalités existantes

Ce pattern n'a d'intérêt que si les clients **n'ont pas besoin** d'utiliser **toutes** les fonctions du système d'origine

Pattern Façade appliqué à JEE (suite)



```
/* pattern Façade */
 class UserfriendlyDate {
     GregorianCalendar gcal;
     public UserfriendlyDate(String isodate_ymd) {
         String[] a = isodate_ymd.split("-");
         gcal = new GregorianCalendar(Integer.parseInt(a[0]),
               Integer.parseInt(a[1])-1 /* careful ! */, Integer.parseInt(a[2]));
     }
     public void addDays(int days) {
       gcal.add(Calendar.DAY_OF_MONTH, days);
     }
     public String toString() {
       return String.format("%1$tY-%1$tm-%1$td", gcal);
     }
}
 /* Client */
 class TestFacadePattern {
     public static void main(String[] args) {
         UserfriendlyDate d = new UserfriendlyDate("2018-08-20");
         System.out.println("Date : "+d);
         d.addDays(20);
         System.out.println("20 jours après : "+d);
     }
 }
```

Ex. façade pour une utilisation simplifiée du calendrier de l'API Java

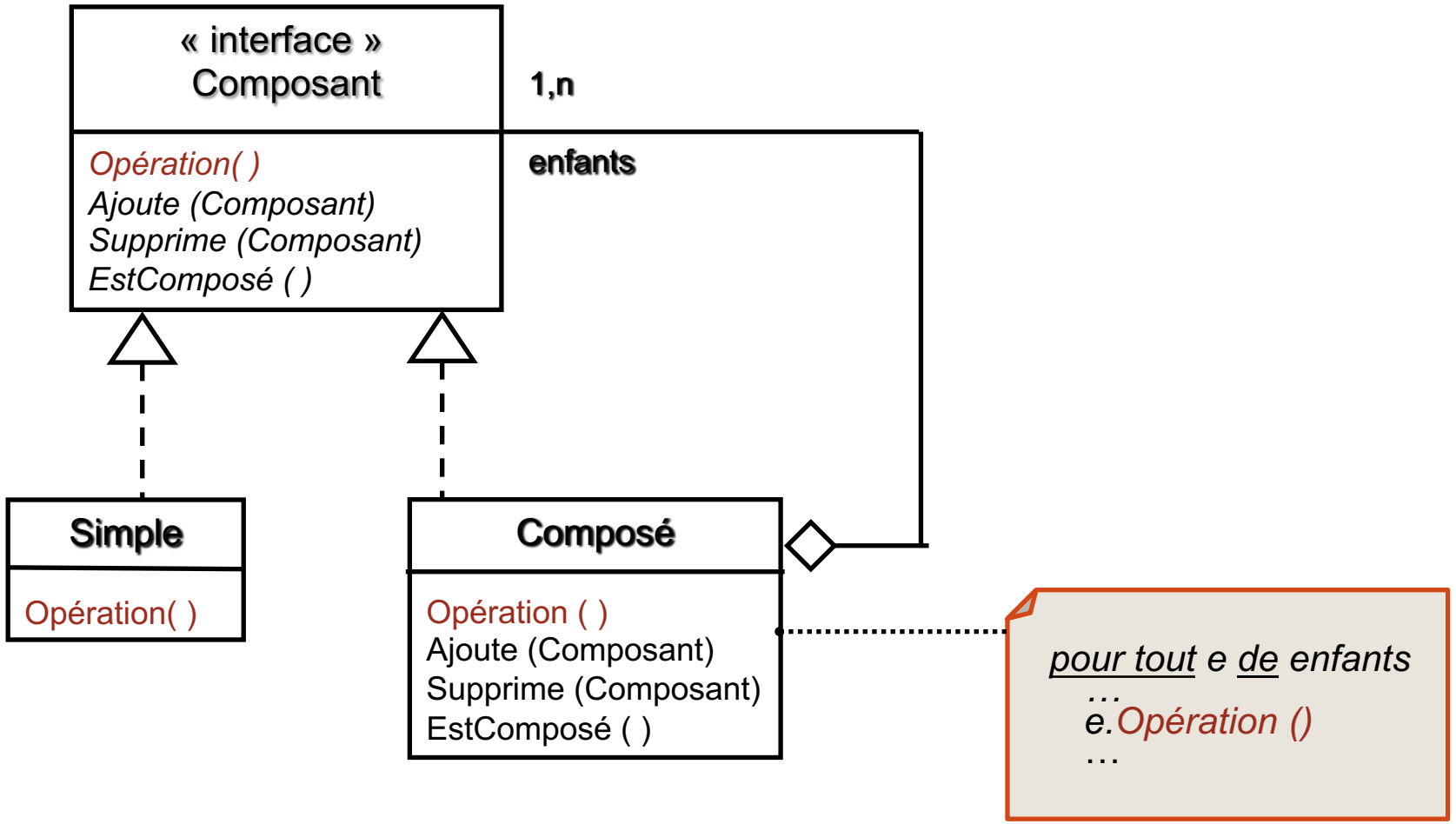
Le Pattern Composite

Patron de Structure

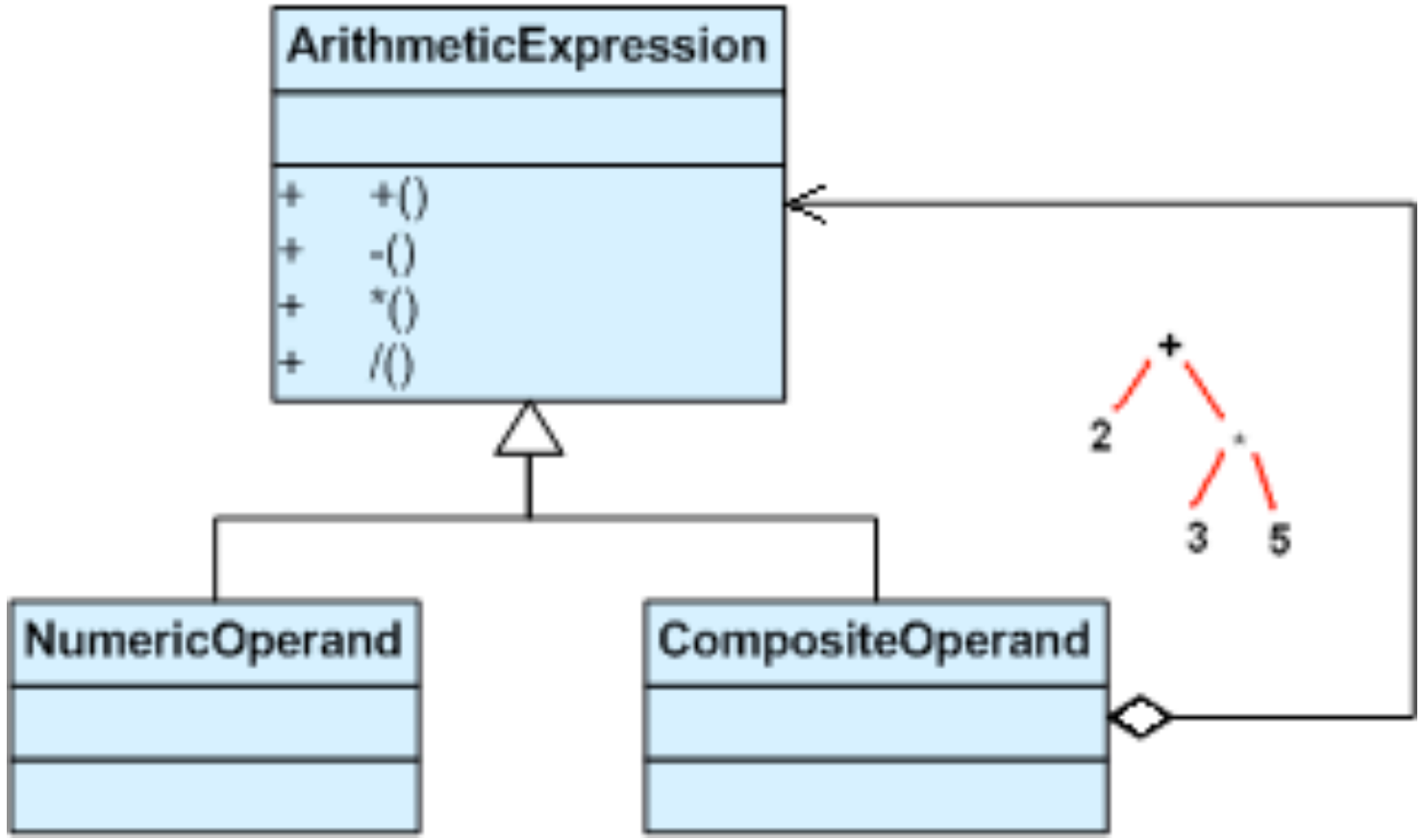
Patron Composite

- Un objet *Composite* est composé d'autres objets
  - Qui peuvent eux-mêmes être composé, ou des objets atomiques
  - Permet de créer une arborescence d'éléments
  - Les traitements s'effectuent sur les objets, indépendamment du fait qu'ils soient Composé ou Atomique
- *Opération()* est appelée « de façon récursive » sur chaque feuille composant les agrégats
  - Utilise un itérateur, par ex. pour calculer le prix de l'objet global à partir du prix de chaque composant
- On pourrait ajouter une méthode *getEnfant()* dans un Composite, qui retourne :
  - Un container des enfants
  - Un itérateur sur la racine, etc.

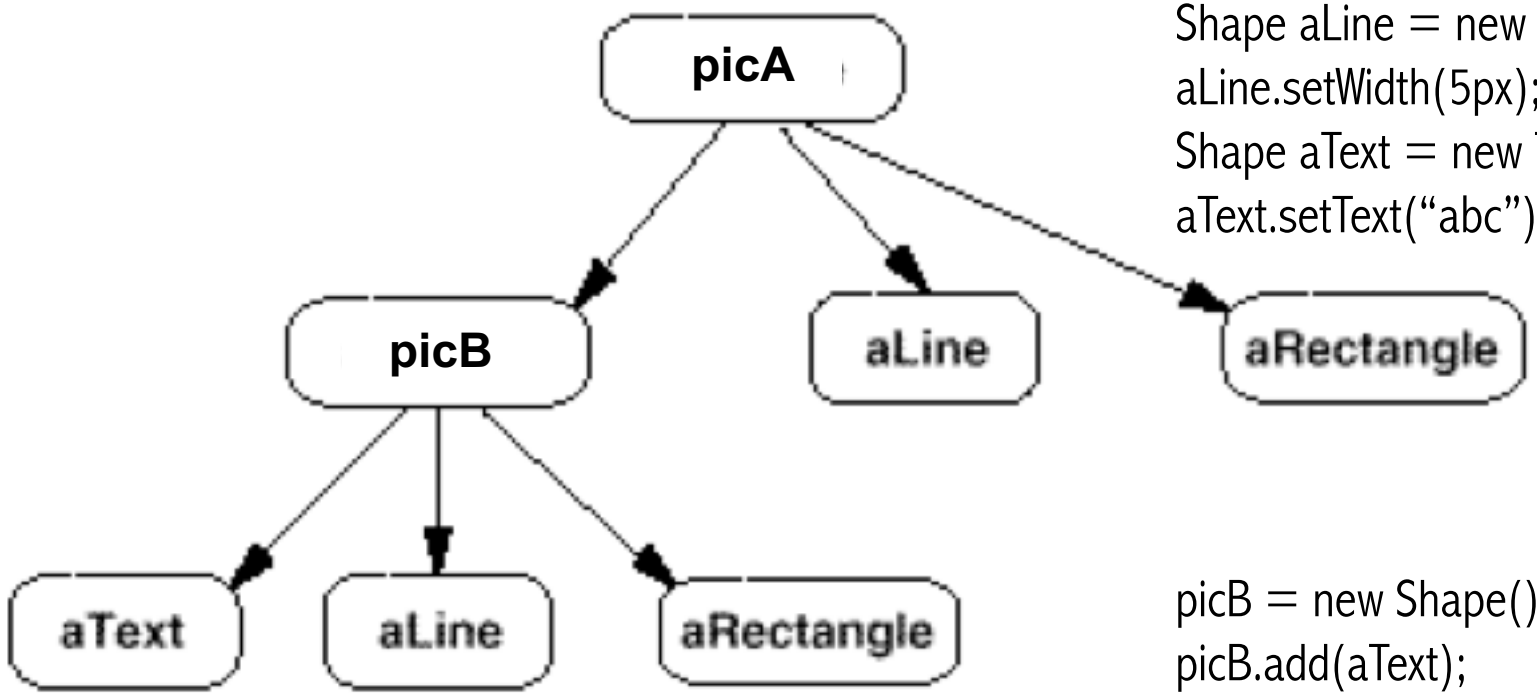
Pattern Composite



Exemple : expression arithmétique



Exemple : graphisme




```
Shape aRect = new Rect();
Shape aLine = new Line();
aLine.setWidth(5px);
Shape aText = new Text();
aText.setText("abc");
```

```
picB = new Shape();
picB.add(aText);
picB.add(aLine);
picB.add(aRect);

picA = new Shape();
picA.add(picB);
picA.add(aLine);
picA.add(aRect);
```

Polymorphisme :

```
for (Shape g : listShapes) {
    g.draw();
}
```

Le pattern Strategy

Pattern comportemental à portée Objets

Le pattern STRATEGY (1/2)

- **Problème :** réorganiser une solution particulière (ex.: algorithme) pour en faire une solution générique

- **But** : définir un **ensemble d'algorithmes répondant à un même problème**, encapsuler chacun et les rendre interchangeables

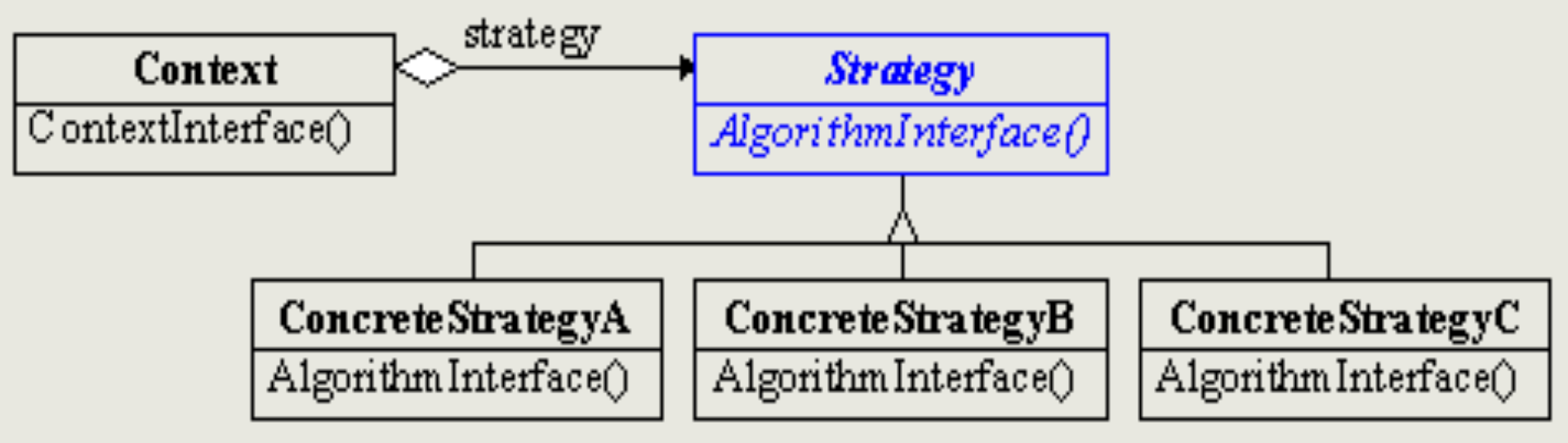
- **Conséquence** : le pattern définit une **famille d'algorithmes**, définit des classes de réalisation indépendantes, les rend dynamiquement interchangeables.
  - On peut ajouter / supprimer des algorithmes (OCP)
  - Il est possible d'échanger dynamiquement d'algorithme sans modifier les classes clients qui les utilisent.

Le pattern STRATEGY (2/2)

Cas d'utilisation

- On a une hiérarchie de classes nombreuses qui se distinguent uniquement par leurs **comportements**
  - Ex.: pour des animaux, le comportement alimentaire ou de reproduction ; pour un véhicule, l'énergie et le *ground* de déplacement d'un véhicule (air, sol, mer…), etc.
- Différentes **versions** d'algorithmes sont nécessaires
- Une classe définit plusieurs comportements qui sont autant de **branches conditionnelles** dans ses méthodes
  - Swith/case
  - If imbriqués

Architecture STRATEGY (cf TD)



Délégation d'opération via un attribut *uneStratégie* de type *Strategy*, dans Context :

```
public void contexteInterface() {
    uneStratégie.algorithmeInterface();
}
```

(le choix de la stratégie est affectée lors de l'instanciation du contexte, et peut être modifiée ensuite de manière transparente)

Le pattern Observer

Pattern comportemental

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

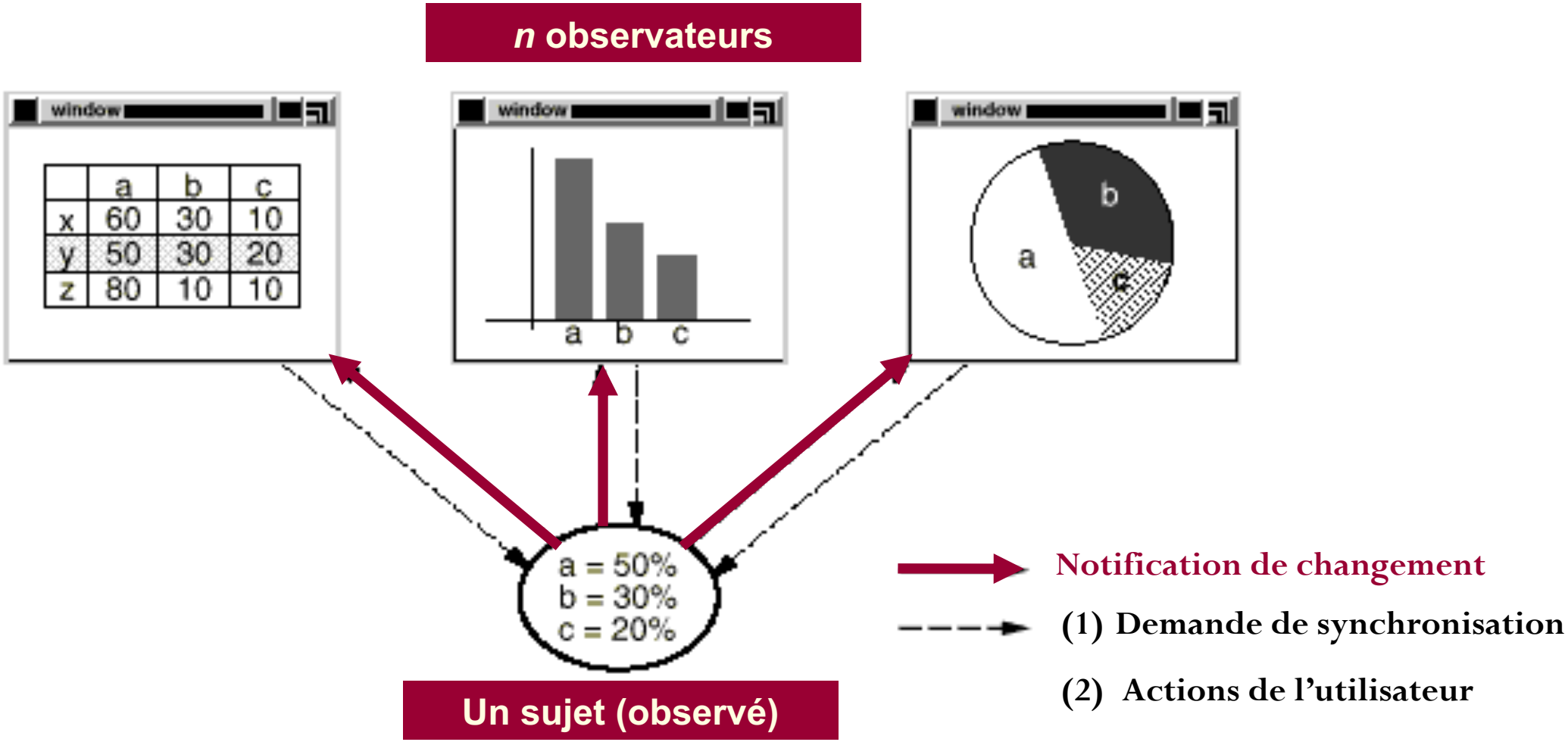
  - sans toutefois coupler trop fortement les classes pour ne pas réduire leur réutilisabilité

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.

Observer : le contexte

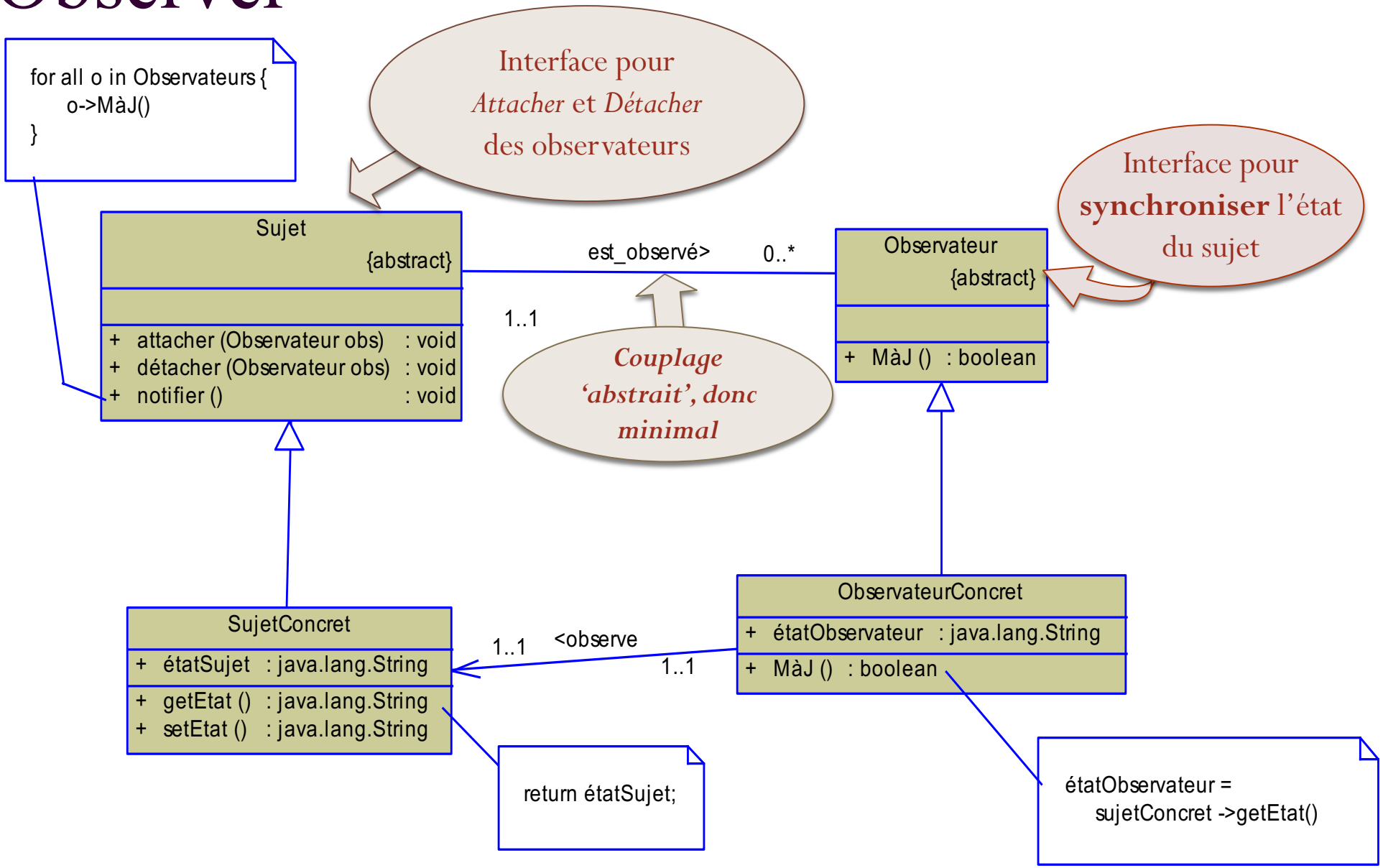


Mécanisme du *publish-subscribe*

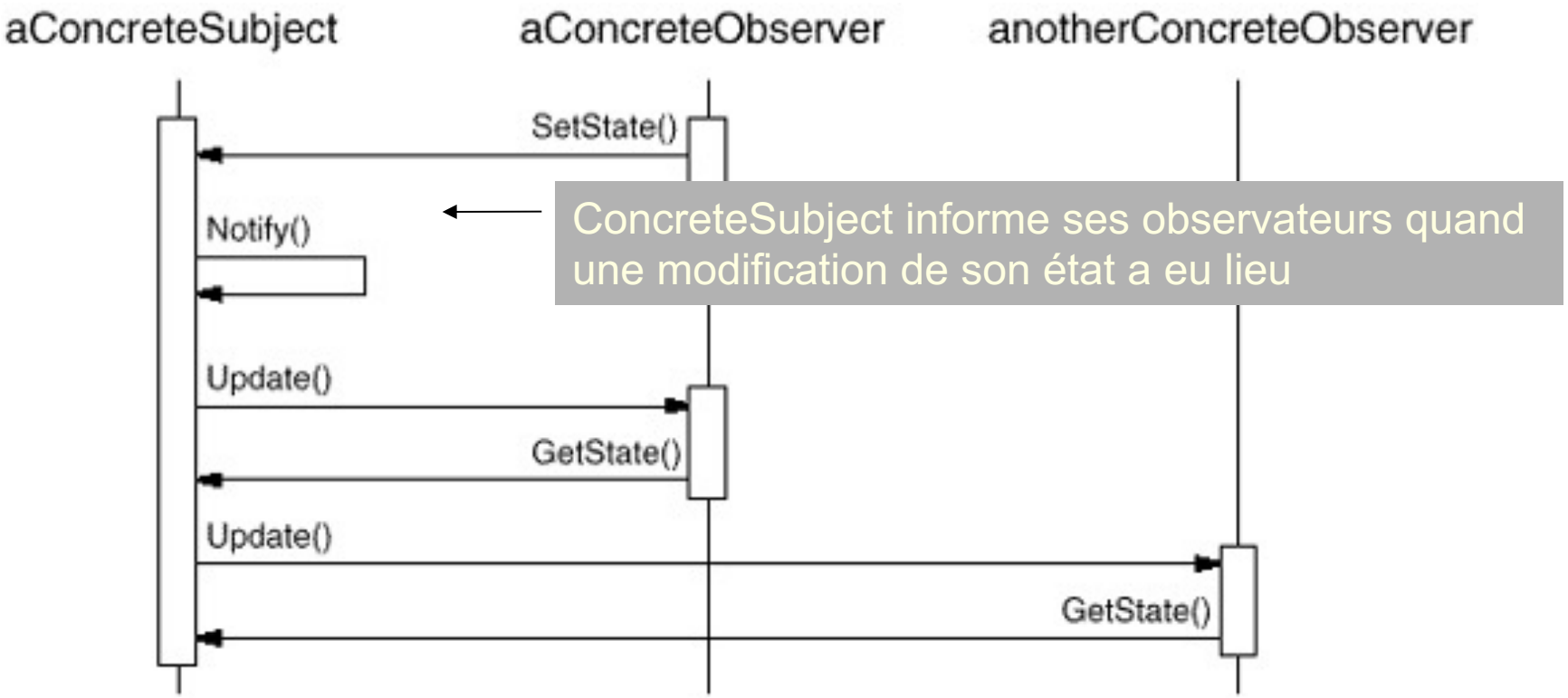
- **Ces interactions** entre un sujet et ses observateurs sont connues sous le nom de **publication / abonnement**

- 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.

Pattern Observer



Un fonctionnement d'Observer



Les notifications ne sont pas tjrs demandées **par le sujet**.
Il existe différentes formes de notification.

Conséquences de l'utilisation d'Observer

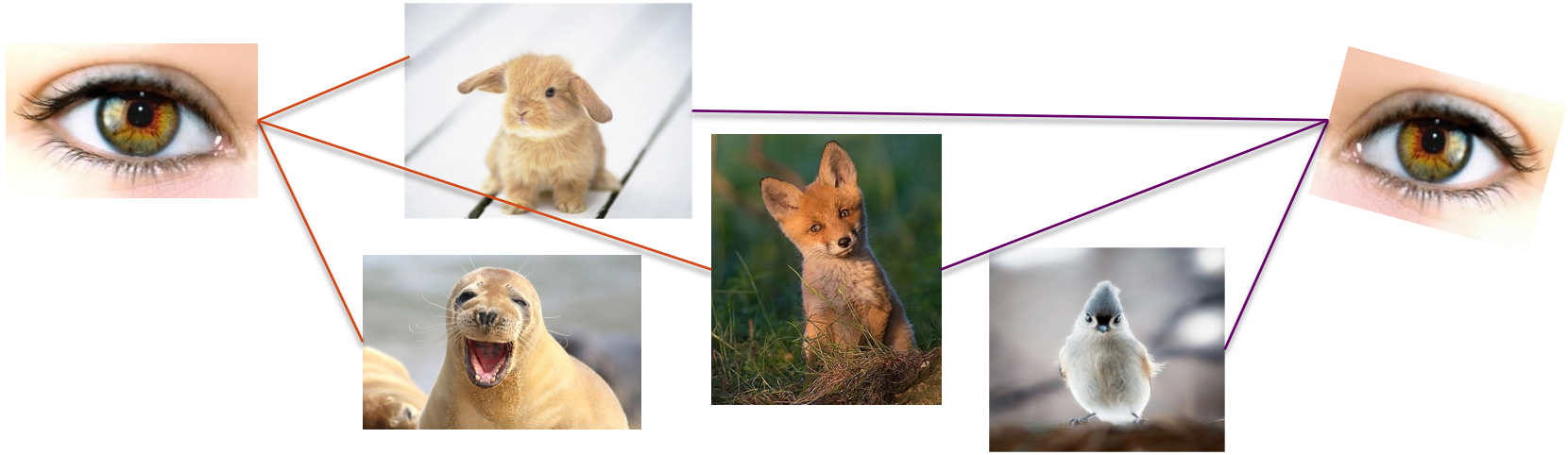
- **Couplage sur les classes abstraites**, donc minimal
  - Tout ce qu'un Sujet sait, c'est qu'il a une liste d'Observateurs, et que chacun se conforme à l'interface commune (méthode `Update()`) leur permettant de synchroniser leur état avec le sien.
- Le patron Observer permet de manipuler les objets 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 opération a priori inoffensive sur le sujet peut causer des mises à jour en cascade de la part des Observateurs et des objets liés.
  - 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()

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**

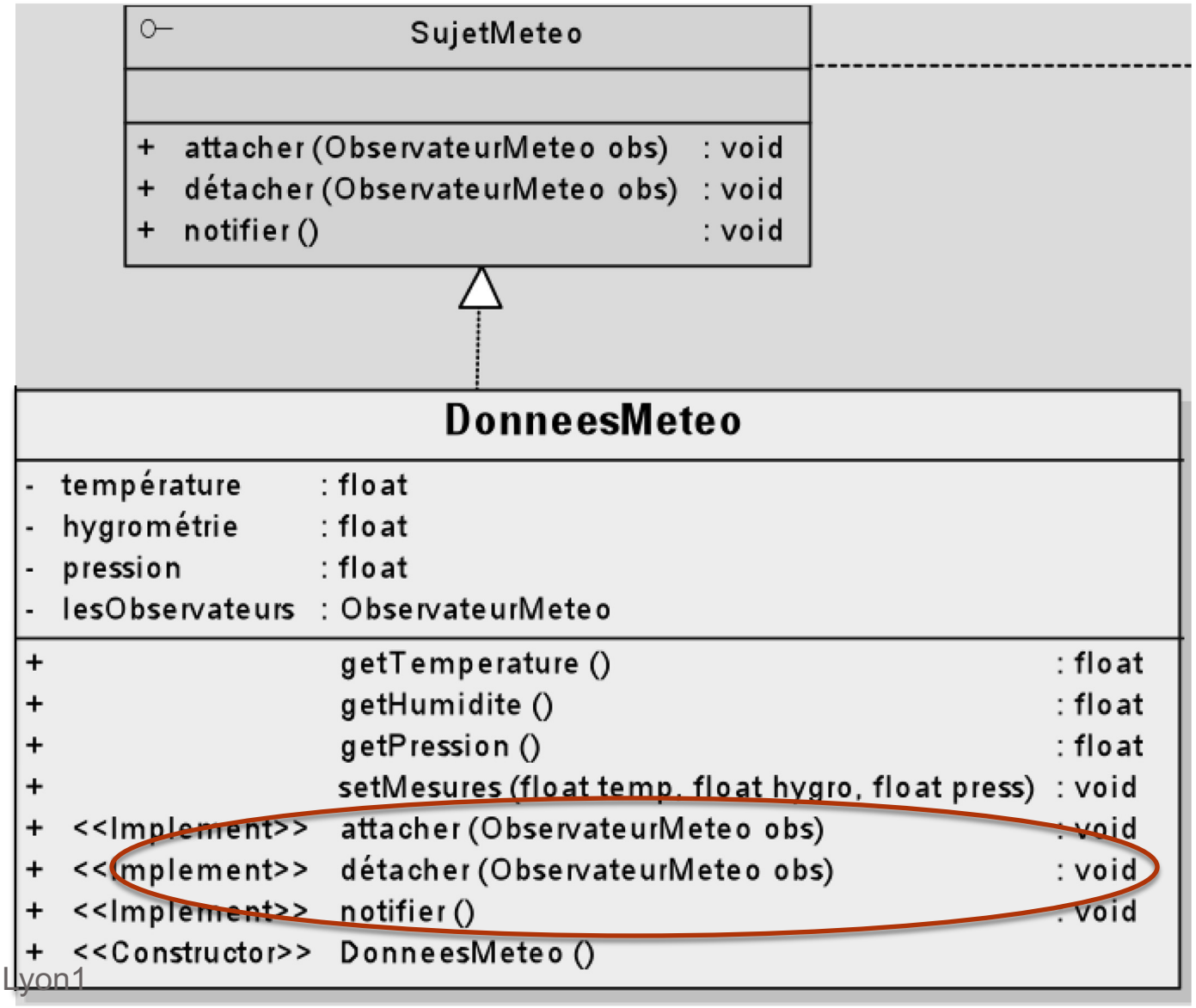
Pattern Observer sur l'exemple Météo

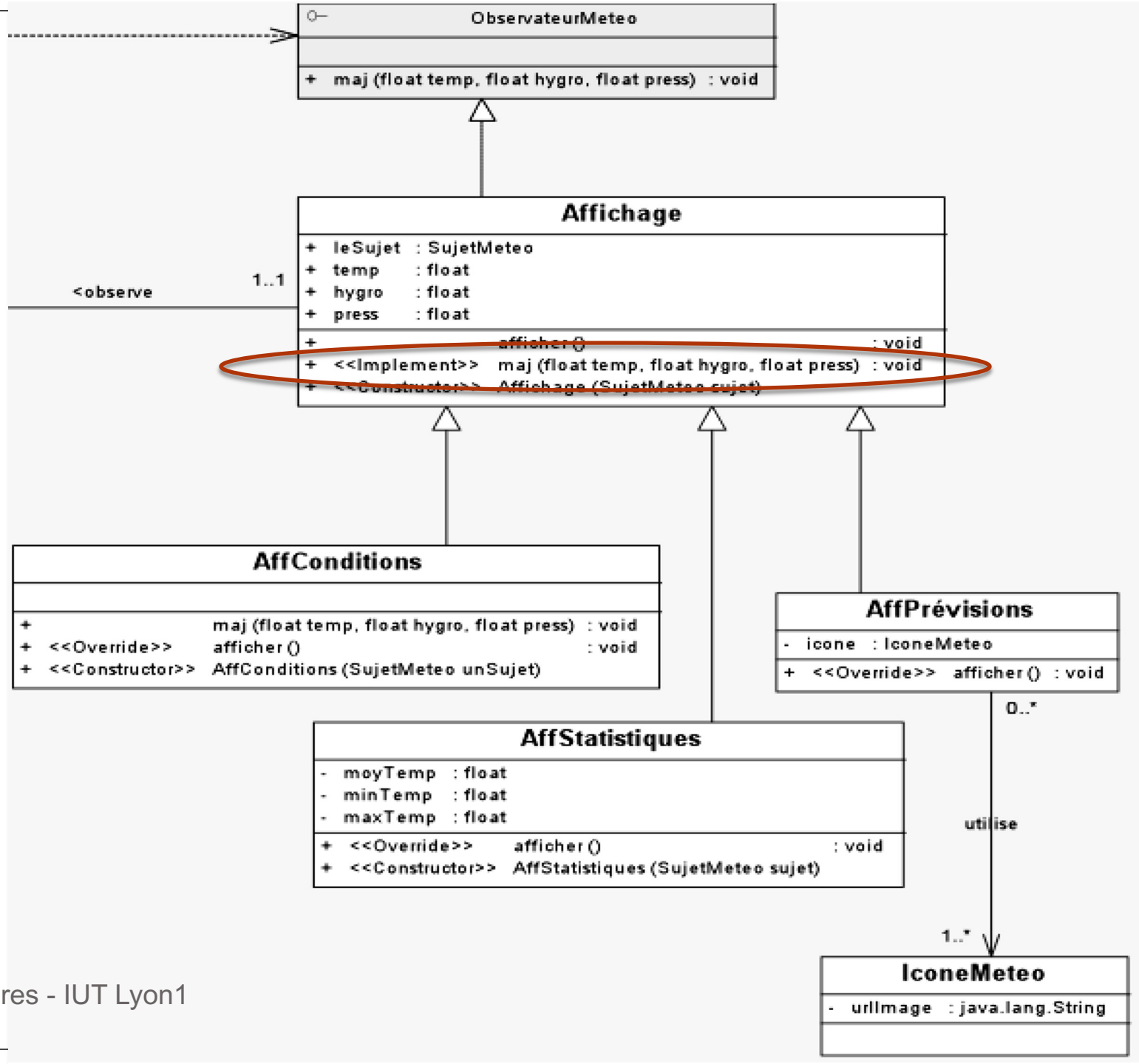
- Qu'est-ce qui **varie** dans cette application ? (les isoler dans des classes concrètes)
  - Les données de météo
  - L'affichage des 3 composants envisagés
  - Le nb et le type des affichages envisagés

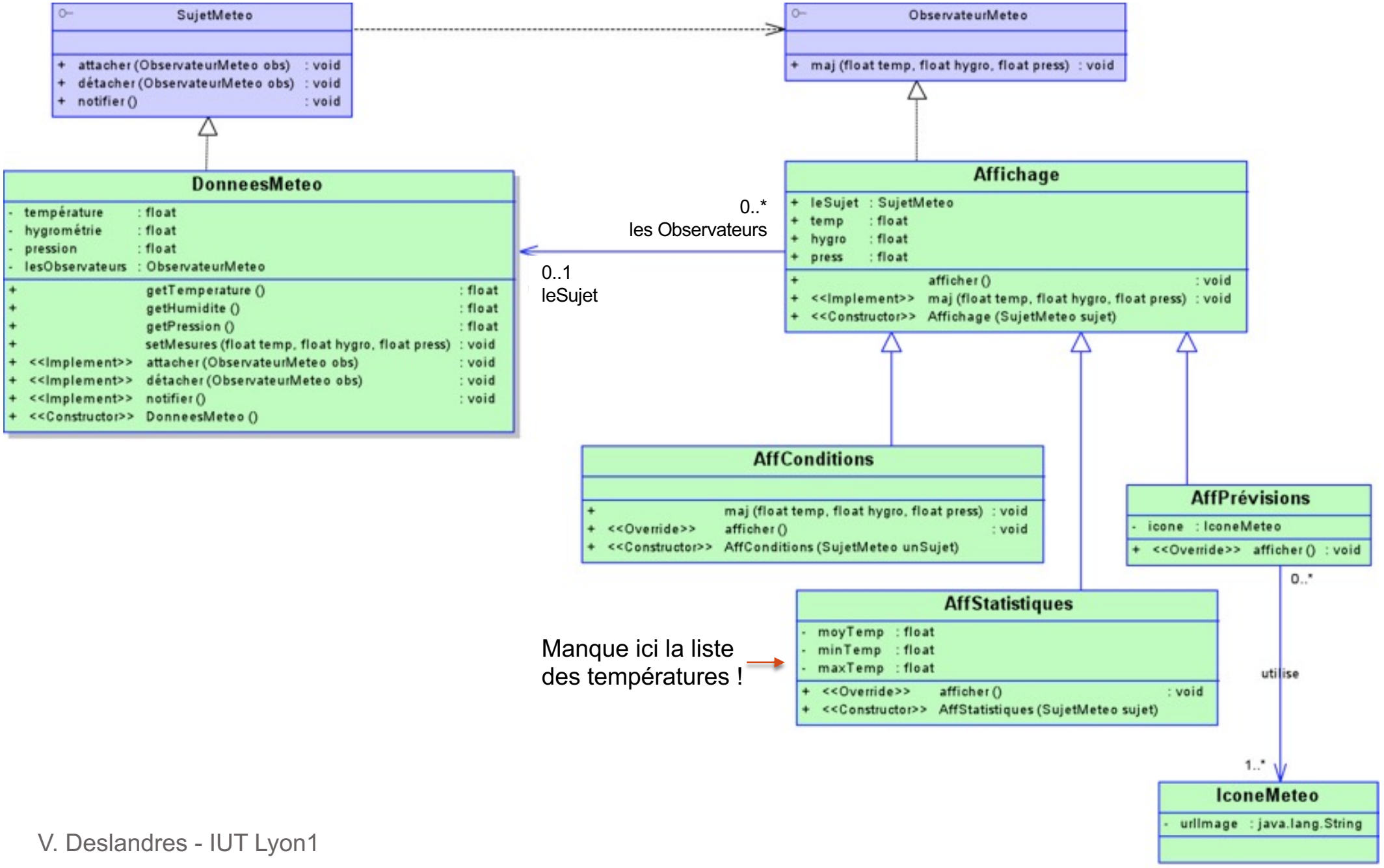
- Qu'est-ce qui est **stable** ? (l'encapsuler aussi)
  - La récupération des données météo : c'est la même qui va être effectuée par les 3 affichages envisagés
  - On crée une classe concrète Affichage avec la méth *update()*

  (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
AffStatistiques
- moyTemp : float
- minTemp : float
- maxTemp : float
+ <<Override>> afficher () : void
+ <<Constructor>> AffStatistiques (SujetMeteo sujet)
0..*
utilise
1..*
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)
Manque ici la liste
des températures !
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...");

  }
```

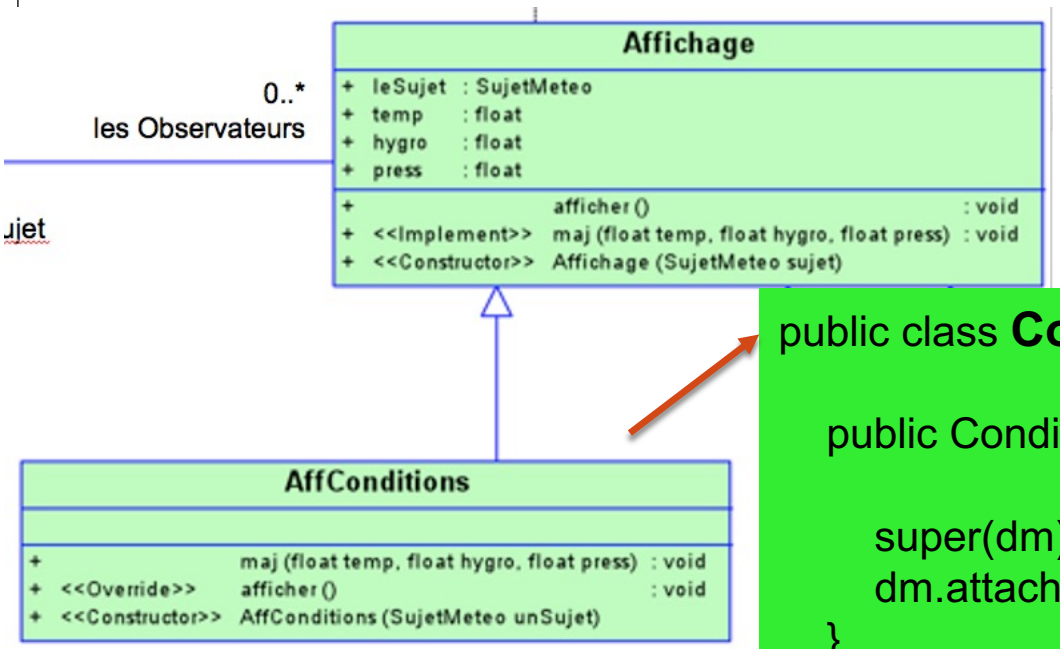
Météo : Extraits de code (2)

```
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