

Chap.8 – Design patterns (partie 2)

V. Deslandres ©

Licence Professionnelle SIL option DEVOPS

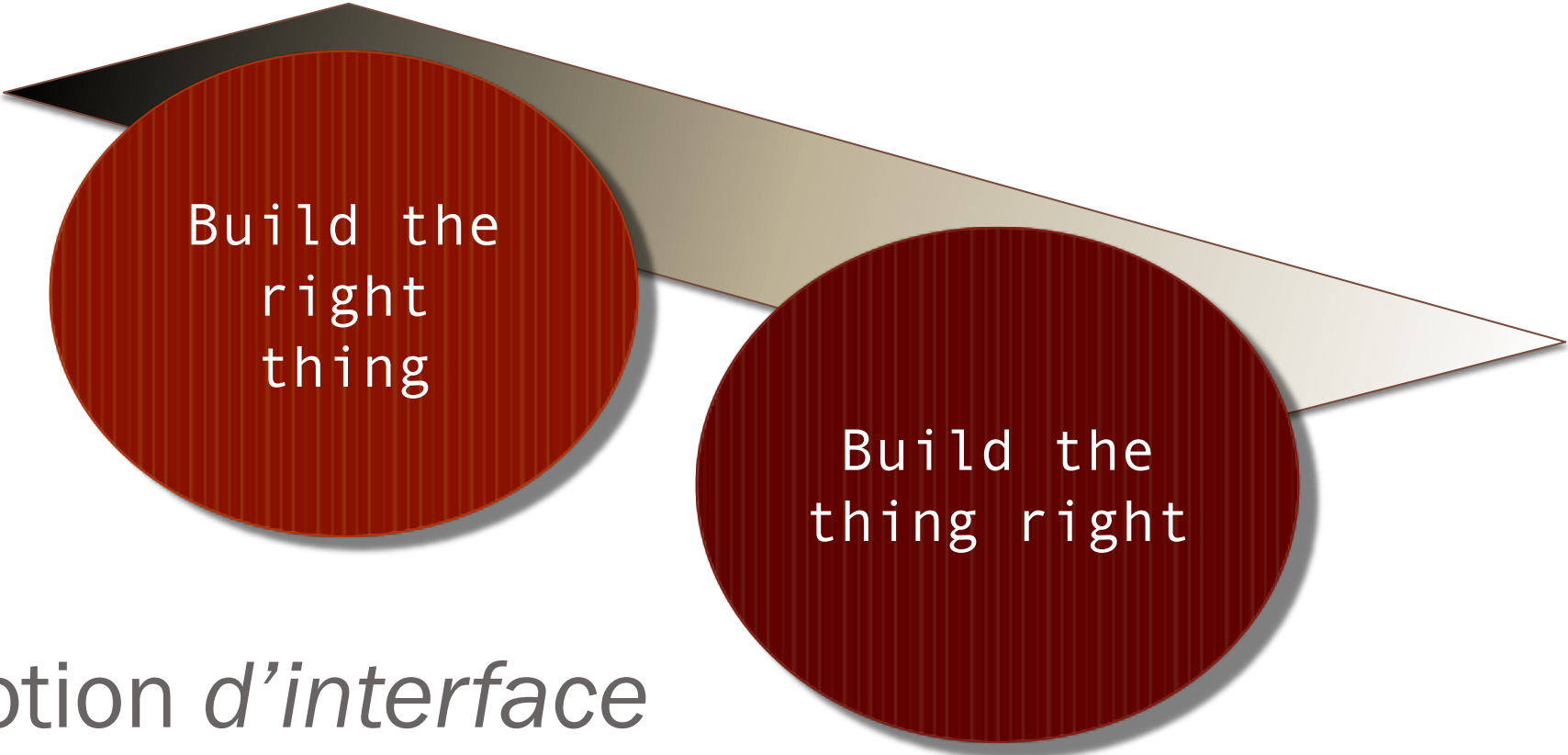
Développeur et Administrateur de Systèmes d'information, courant DEVOPS

IUT de Lyon - Université Lyon 1



Sommaire du cours DP - part.2

- Notion d'interface #3
- Le pattern Observer #7
 - Illustration Affichage Temps [#14](#)
 - Exercice Météo [#19](#)
- Les patterns FactoryMethod [#33](#)
- Le pattern Composite [#41](#)
- Le pattern TemplateMethod [#46](#)
- Conclusion sur les DP [#51](#)
 - 5 lois à connaître



Build the
right
thing

Build the
thing right

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.

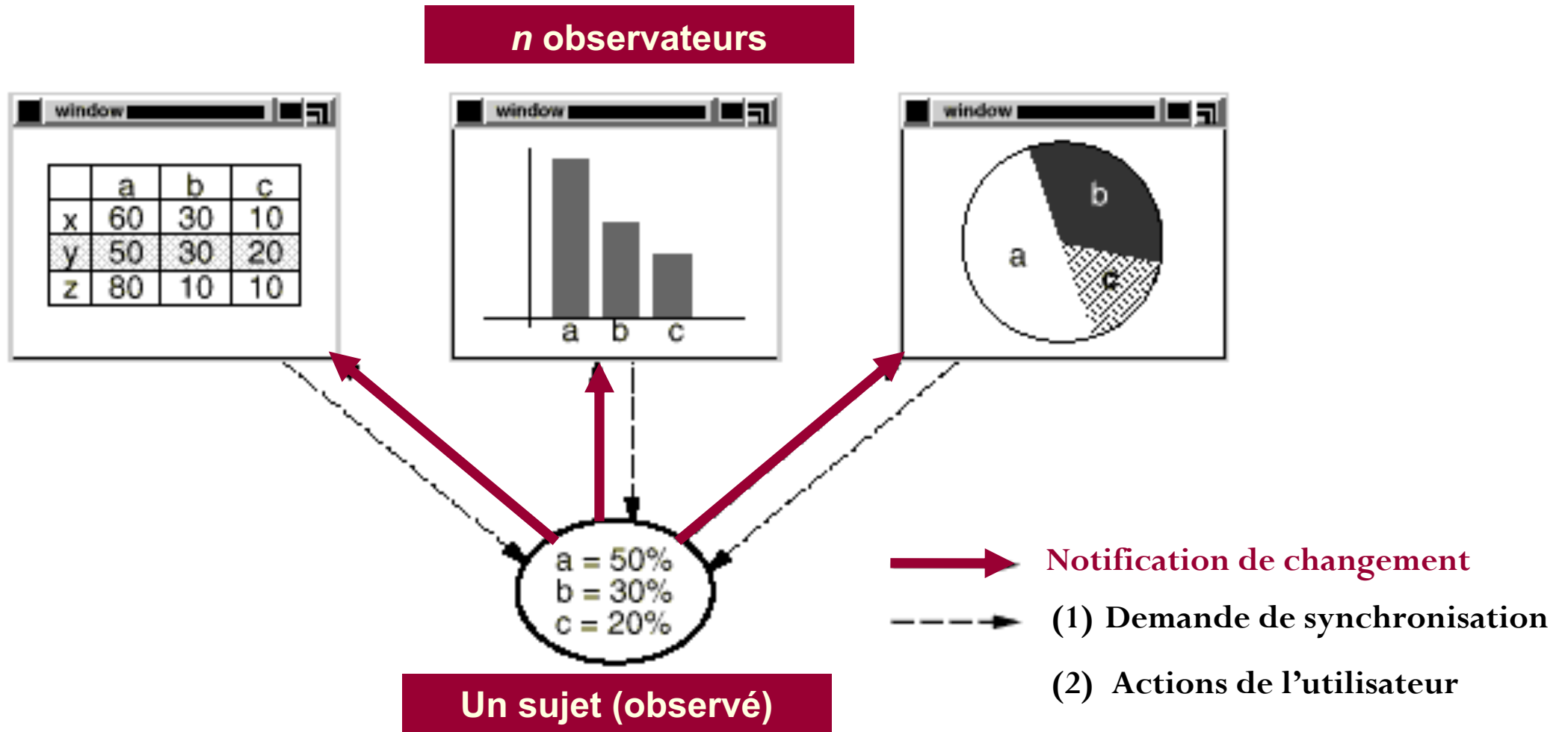
Motivation

- 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.

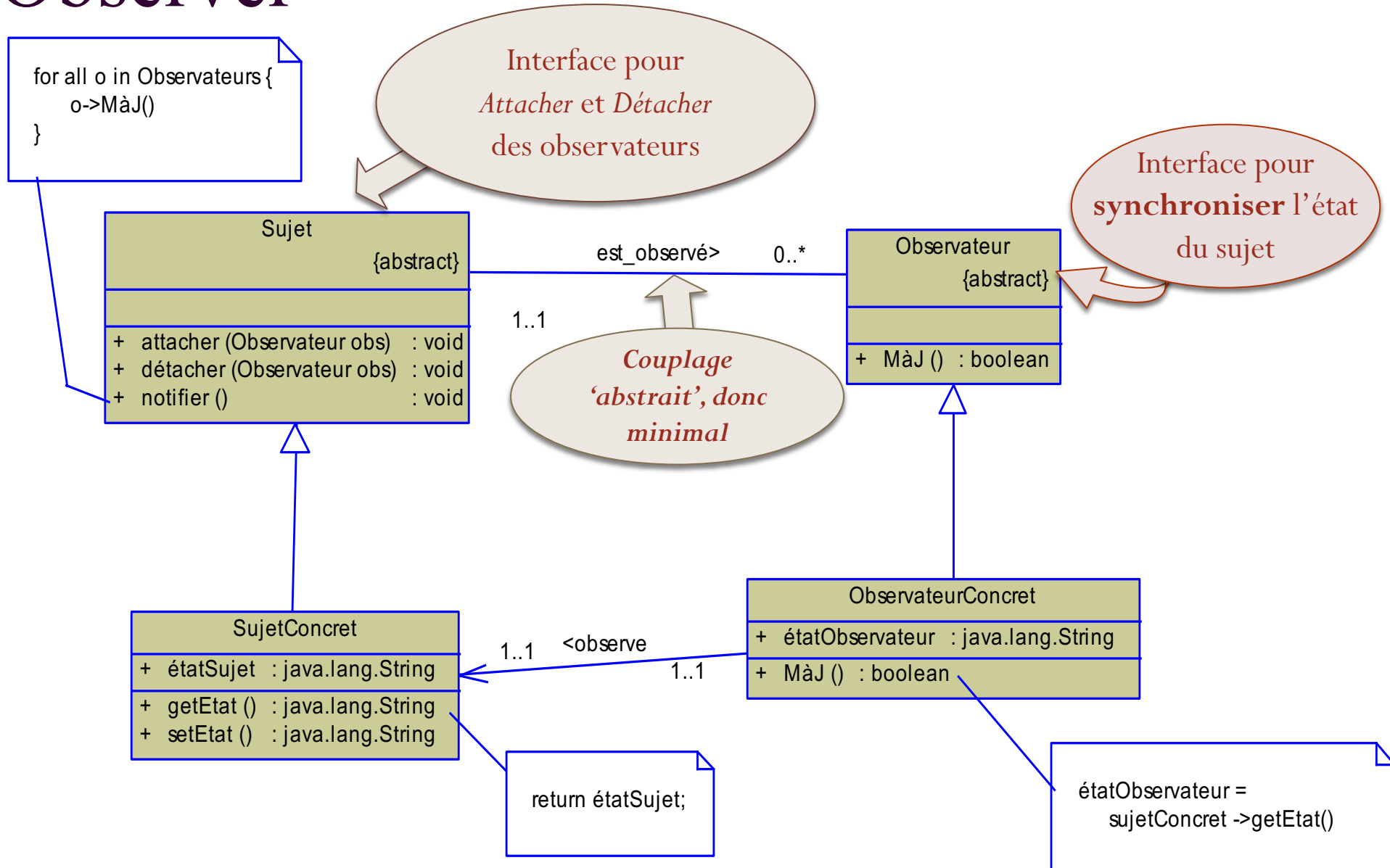
Observer : le contexte



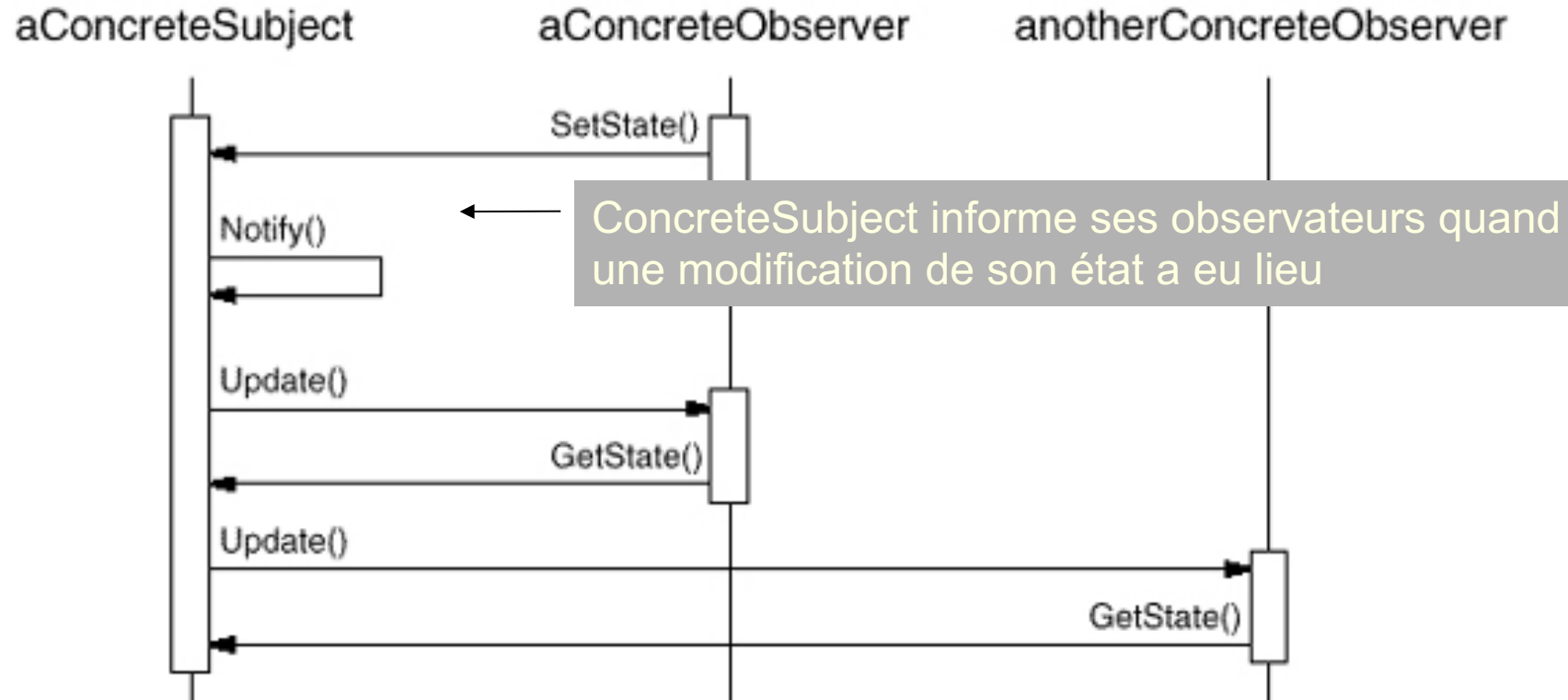
Mécanisme du *publish-subscribe*

- 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**.
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

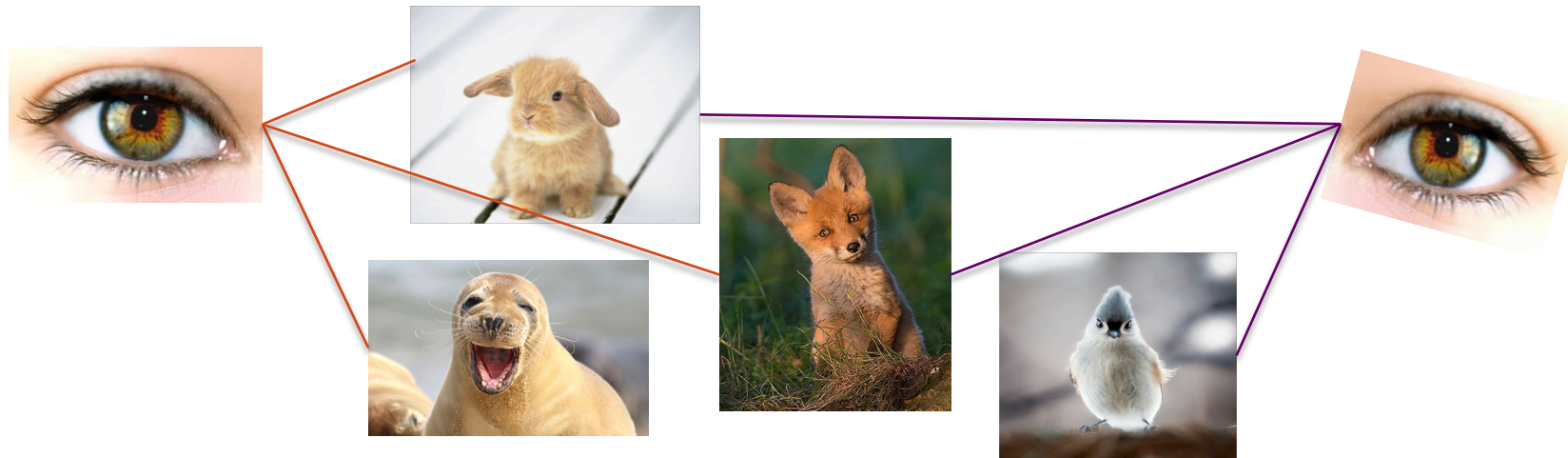
■ Mises à jour en cascade

- Une 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()

Illustration : affichage du temps



Classe abstraite Observer

```
/* RESPONSABILITES :
```

- * - Connait le sujet observé
- * - Sait comment mettre à jour le dernier état du sujet

```
*/
```

```
abstract class ObserverTimer {  
    protected AbstractTimer leSujetTimer;  
    abstract void update(AbstractTimer t);  
}
```

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

Classe abstraite Sujet (Observable)

```
abstract class AbstractTimer {  
    protected List<ObserverTimer> lesObservateurs;  
  
    abstract void attach(ObserverTimer ot);  
    abstract void detach(ObserverTimer ot);  
  
    void notifier() {  
        for (ObserverTimer o: lesObservateurs) {  
            o.update(this);  
        }  
    }  
}
```

Classe concrète pour le Sujet : **ClockTimer**

```
public class ClockTimer extends AbstractTimer {
```

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

```
// constructeur
```

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

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

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



Classe concrète pour le Sujet : **ClockTimer** (2)



```
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()
```

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

Opération **notifier()** : pour tous les observateurs, appel à update(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(ObserverTimer ot) {  
    lesObservateurs.add(ot);  
}  
  
@Override  
void detach(ObserverTimer ot) {  
    if (lesObservateurs.contains(ot)) {  
        lesObservateurs.remove(ot);  
    }  
}
```


Classe concrète : Observer



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

A sa création, l'instance du DigitalClock s'abonne au sujet ClockTimer

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 défaut
    }

    @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)

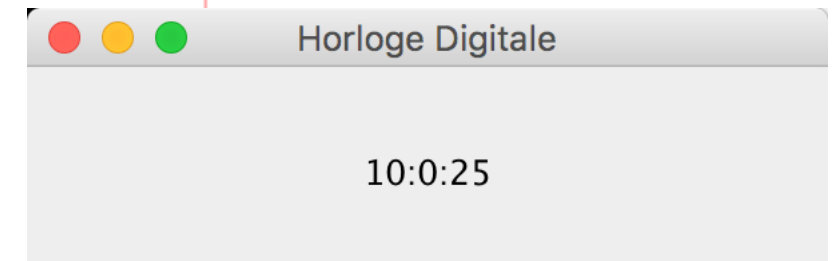


■ Affichage

```
// 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 **AnalogClock()** 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 **AnalogClock()** 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 :

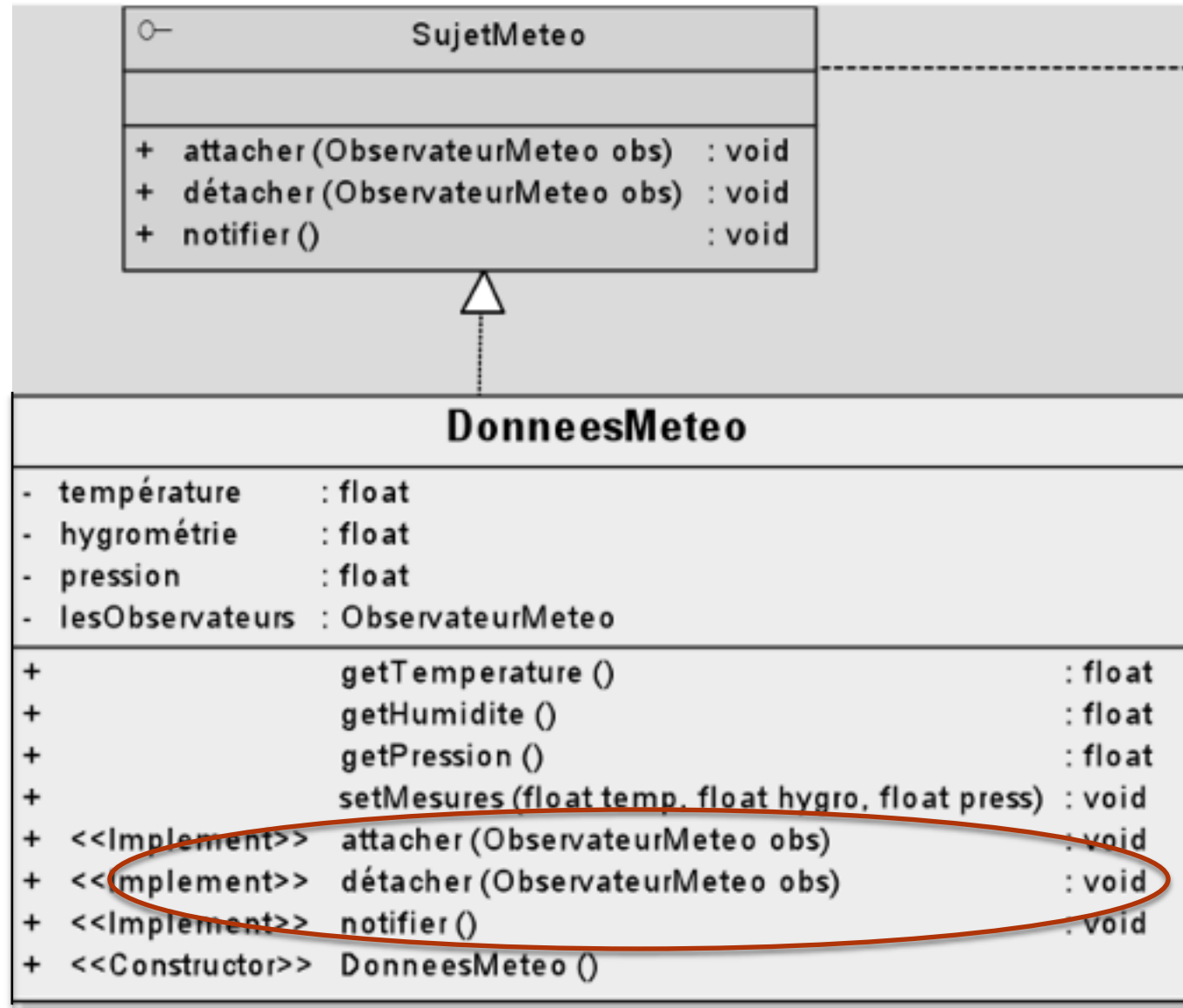
- Laisser la possibilité d'ajouter de **nouveaux types d'affichage**

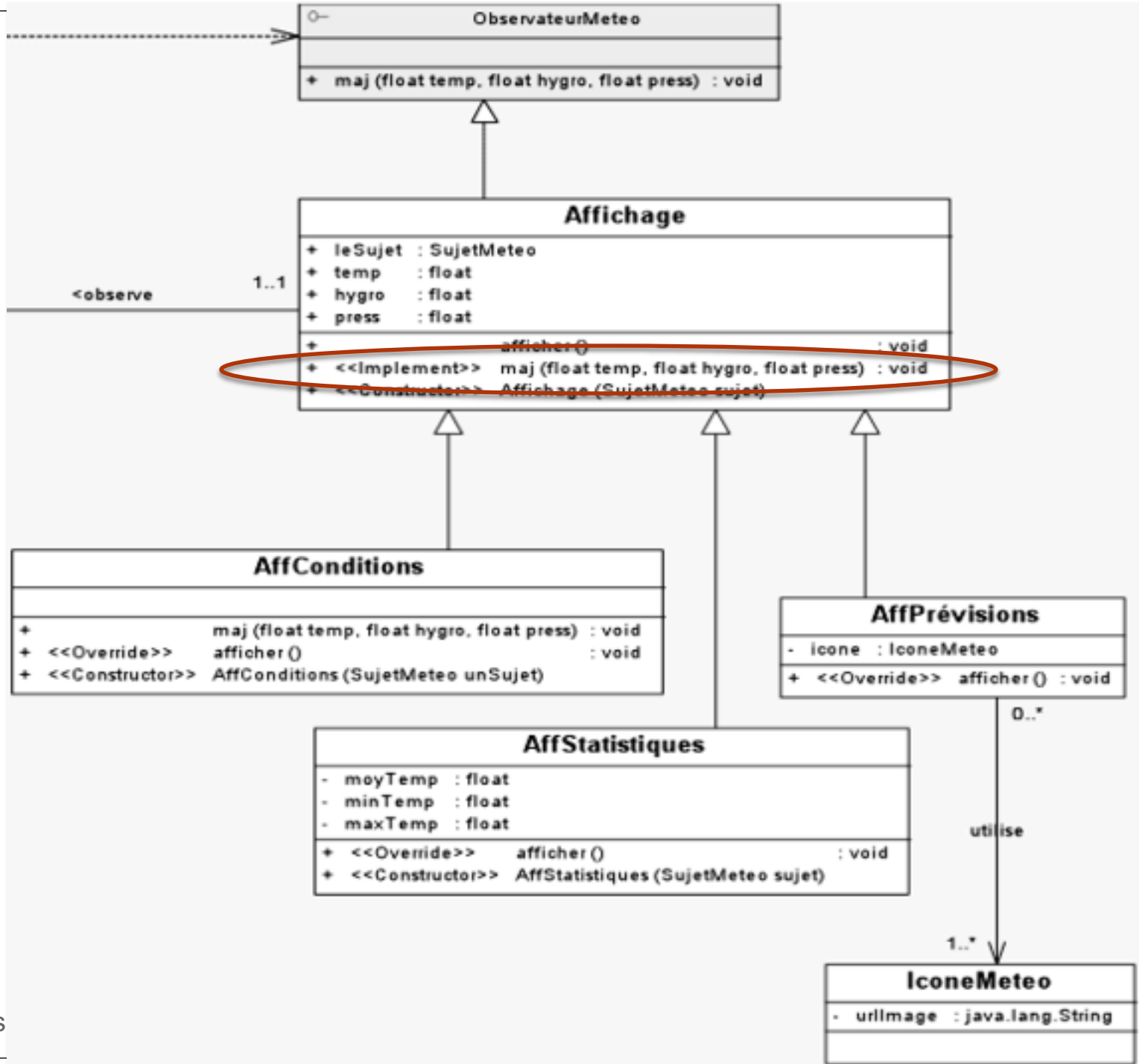
Pattern Observer sur l'exemple Météo

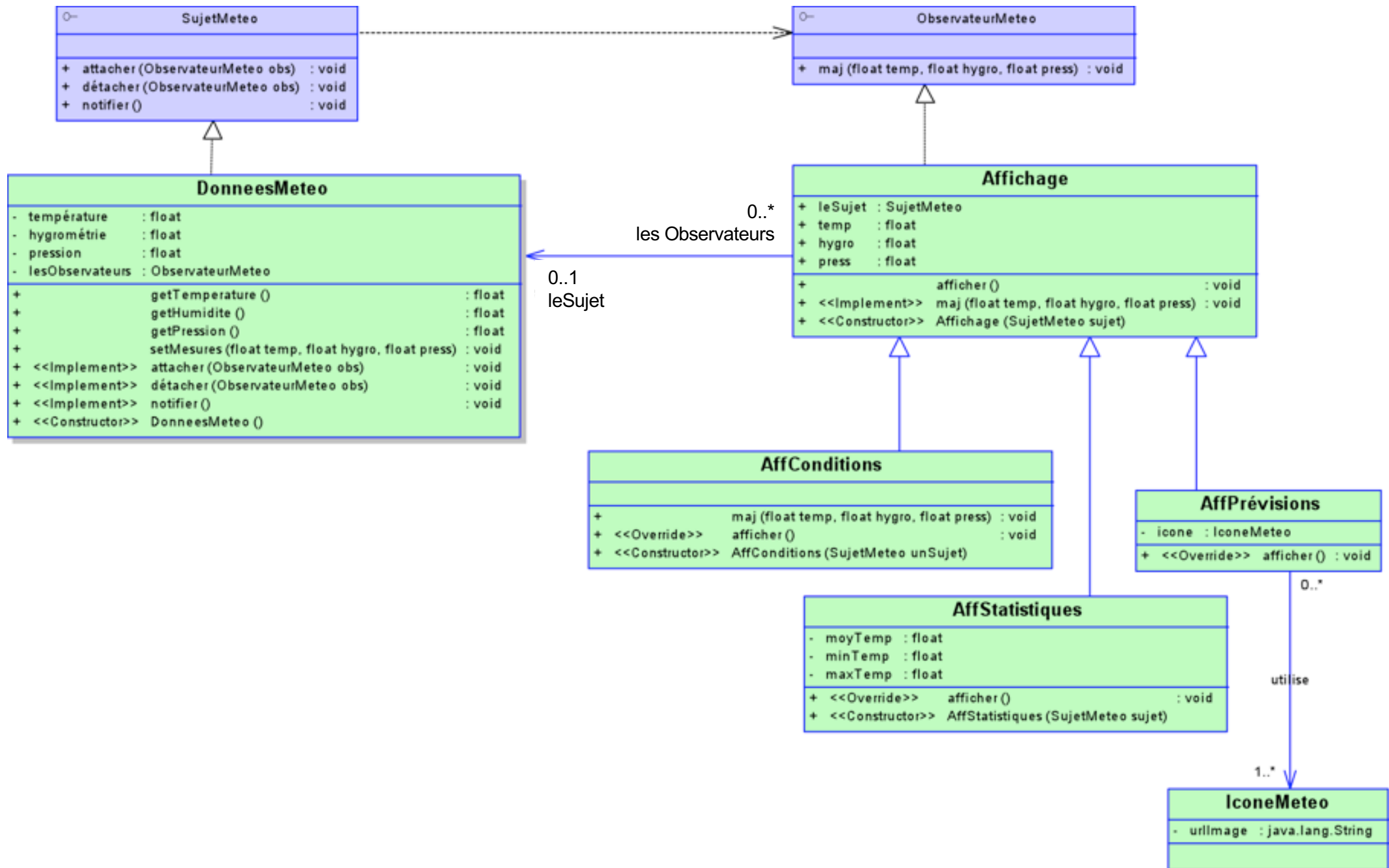
- Qu'est-ce qui **varie** dans cette application ? (les isoler dans des classes concrètes)
 - Les données de météo
 - L'affichage des 3 composants envisagés
 - Le nb et le type des affichages envisagés
- Qu'est-ce qui est **stable** ? (l'encapsuler aussi)
 - La récupération des données météo : c'est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







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

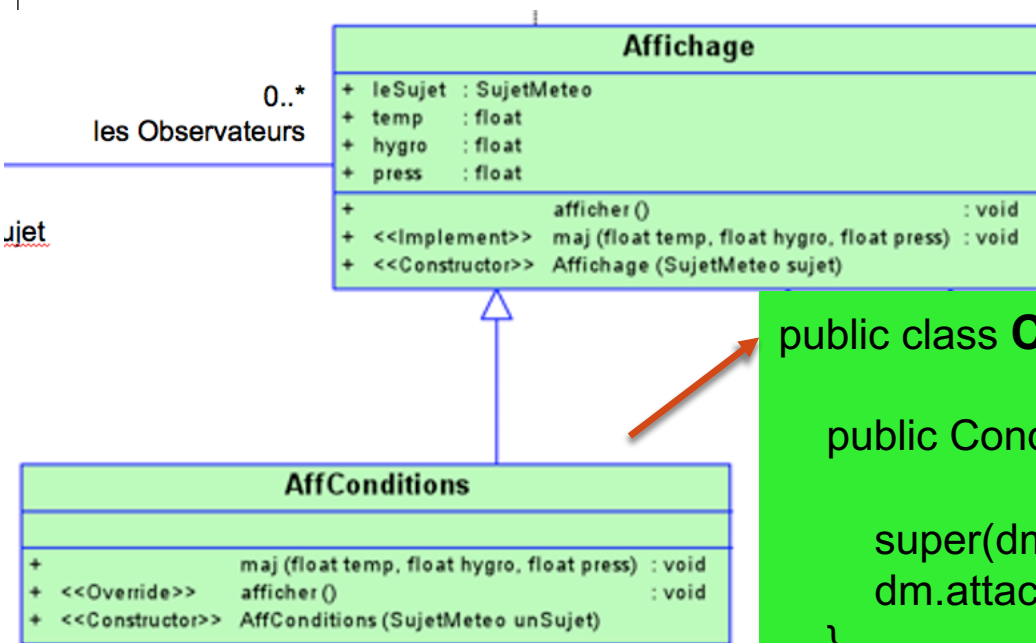
(2)

```
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 + " degrees 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

FactoryMethod

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

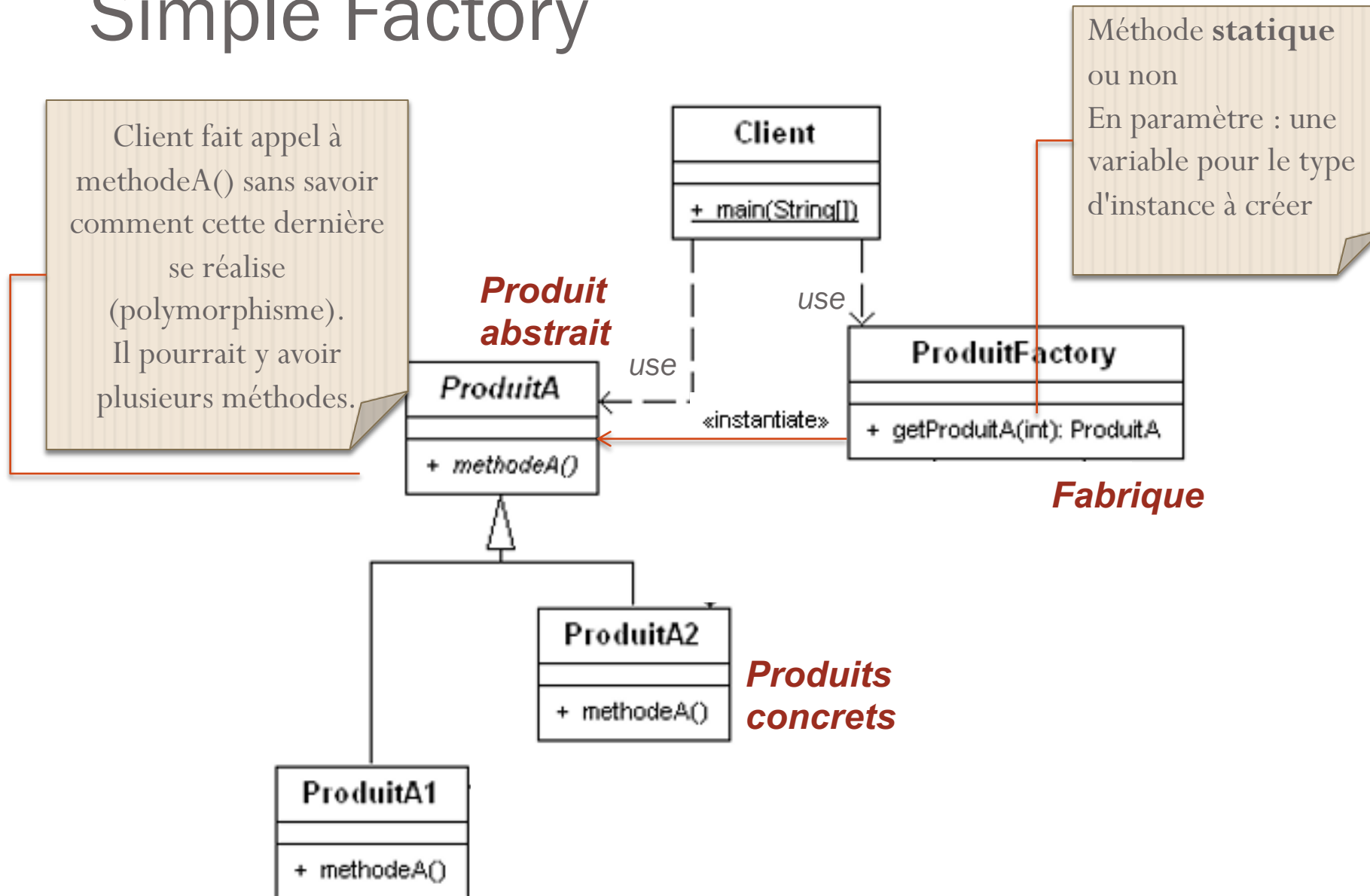


RAPPEL : simple Factory (pas un vrai pattern)

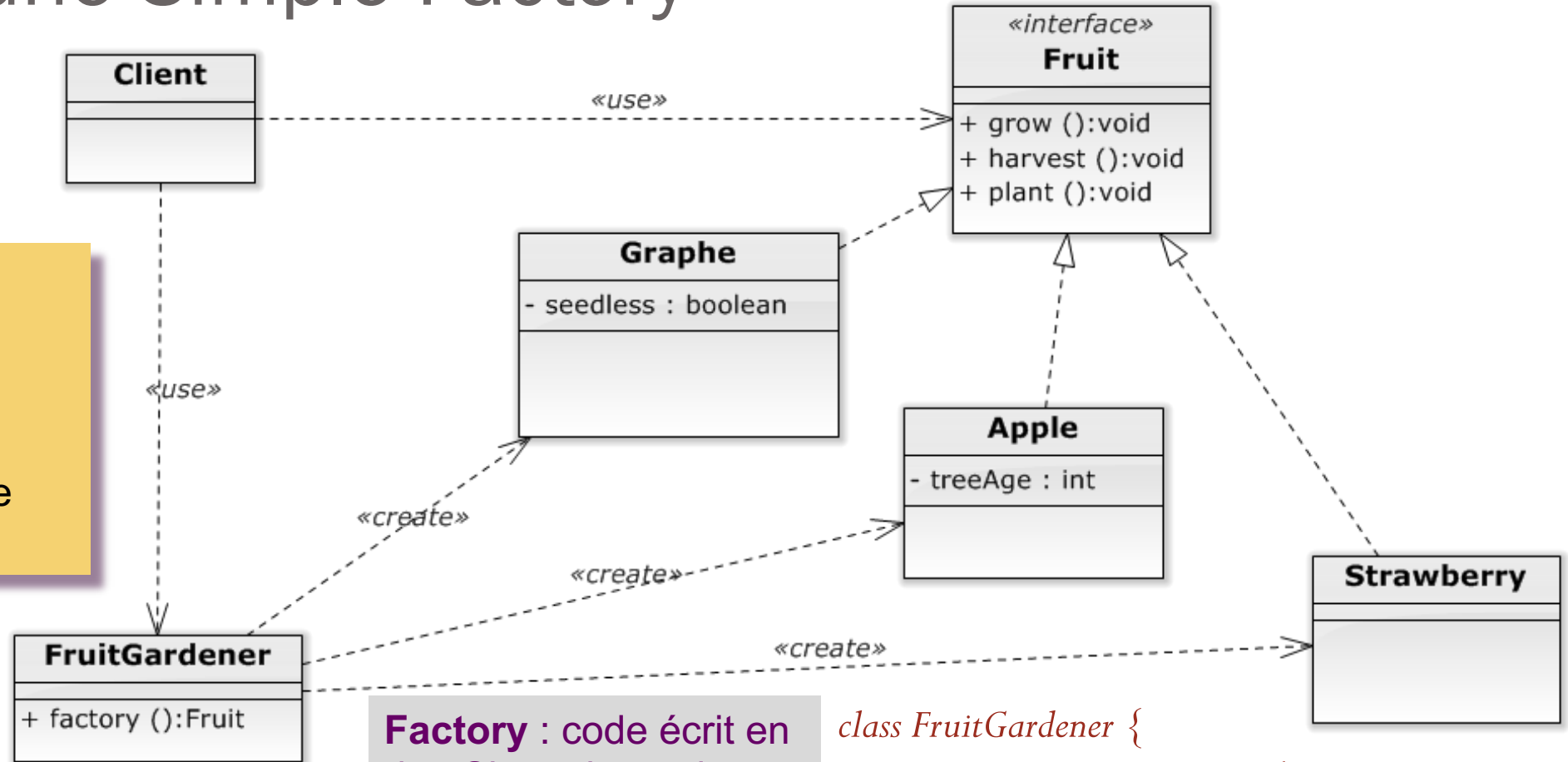
Un idiome de programmation

- 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 est inconnu
- Principe : passer par une classe spéciale, chargée de **CRÉER** les objets spécifiés 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

Simple Factory



Exemple d'une Simple Factory



Un client demande des fruits au jardinier... il ne connaît pas les classes, juste leur nom.

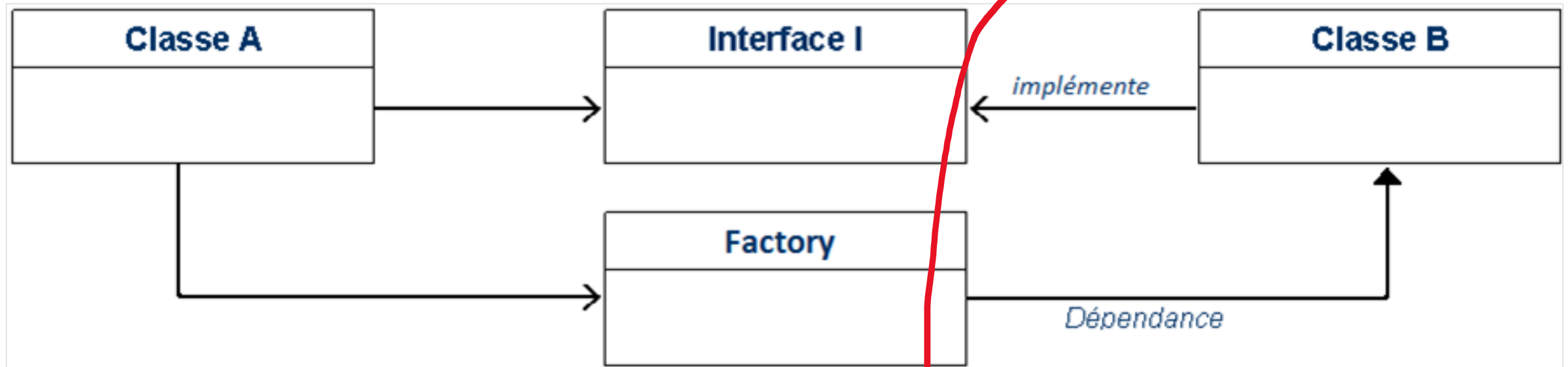
factory(a) pour avoir une *Apple*

Factory : code 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...
    }
}
```

Illustration2 : implémentation du DIP

- C'est une classe **Factory** qui va gérer les dépendances.
- Cette *factory* possède des méthodes qui vont instancier la dépendance (ici **Classe B**) 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*.



Code associé

```
public class Factory {  
    public I getDependency(int x) {  
        switch (x) {  
            case 1: return new B1();  
            case 2: return new B2();  
            ...  
            case 12: return new B12();  
            default: return new B1();  
        } }  
}
```

```
public class A {  
    int leChamp;  
  
    // constructeur qui caractérise la classe  
    A(int a) {  
        leChamp = a;  
    }  
  
    public static void main(String[] args) {  
        // Transmet à la factory le type d'objet B dont on aura besoin  
        I b = new factory().getDependency(leChamp);  
        b.someMethod();  
    }  
}
```

Vérifier avec la solution proposée par constructeur ! <https://www-igm.univ-mlv.fr/~dr/XPOSE2010/guice>

Design pattern « Factory Method »

- On ajoute une **abstraction** supplémentaire : cette fois une classe abstraite (*la fabrique*) 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'emballage, 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.: 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

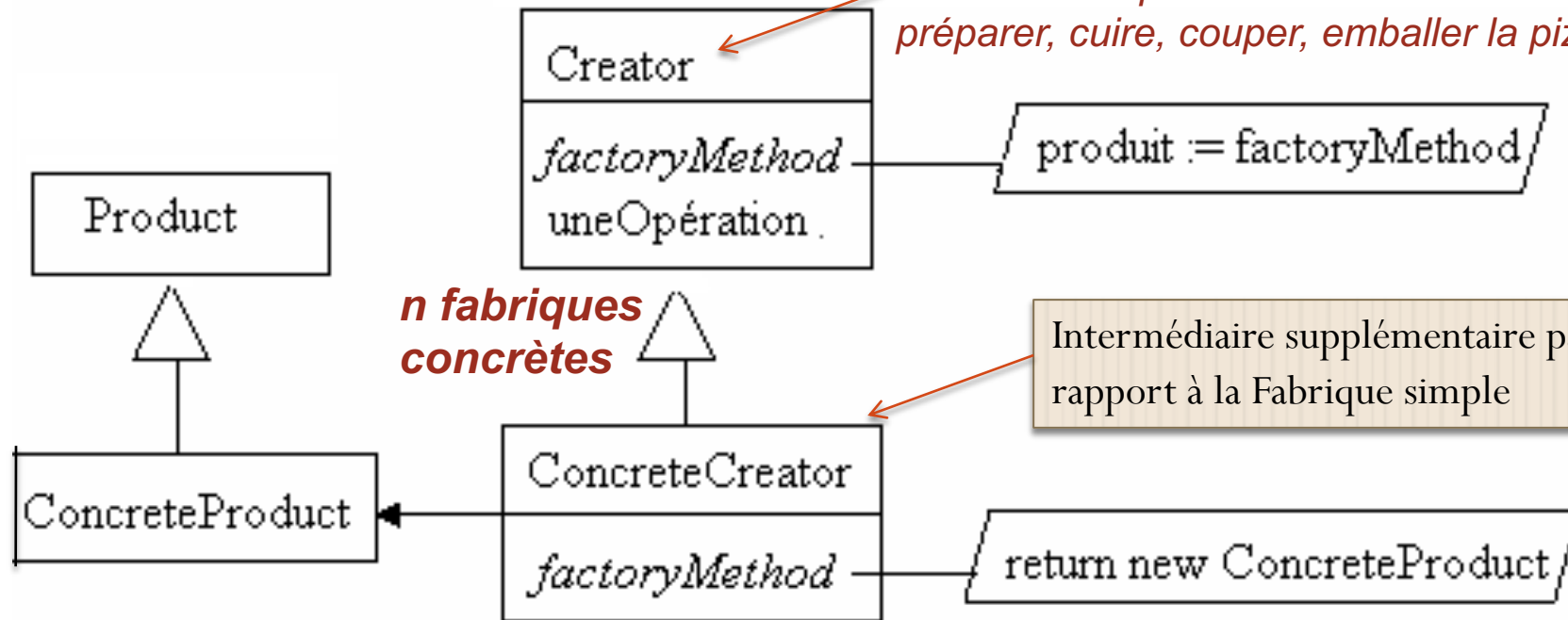
Principe de Factory Method



Fabrique abstraite

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

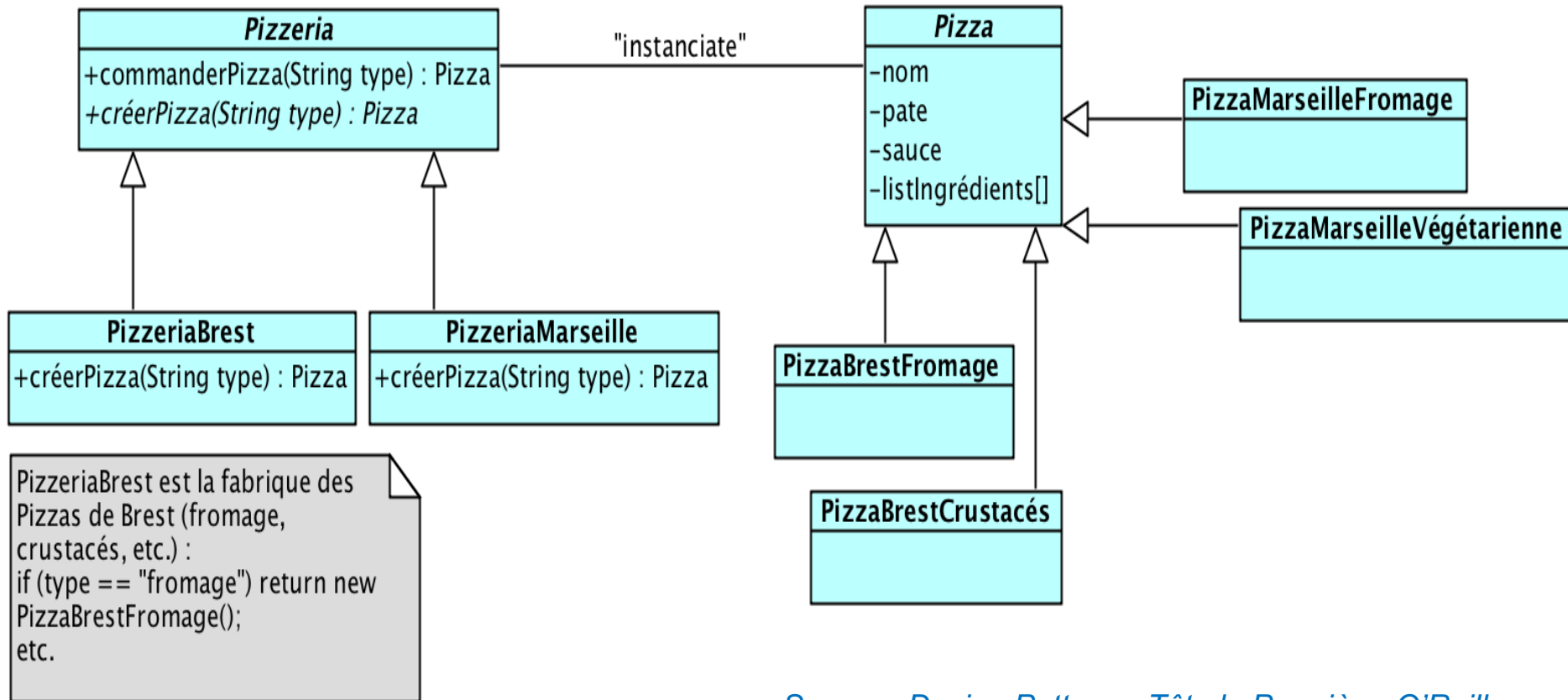
Produit abstrait



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

Illustration Factory Method

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



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

Abstract Factory

- Il existe une 3^{ème} forme de design pattern FactoryMethod
 - 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

Le Pattern Composite

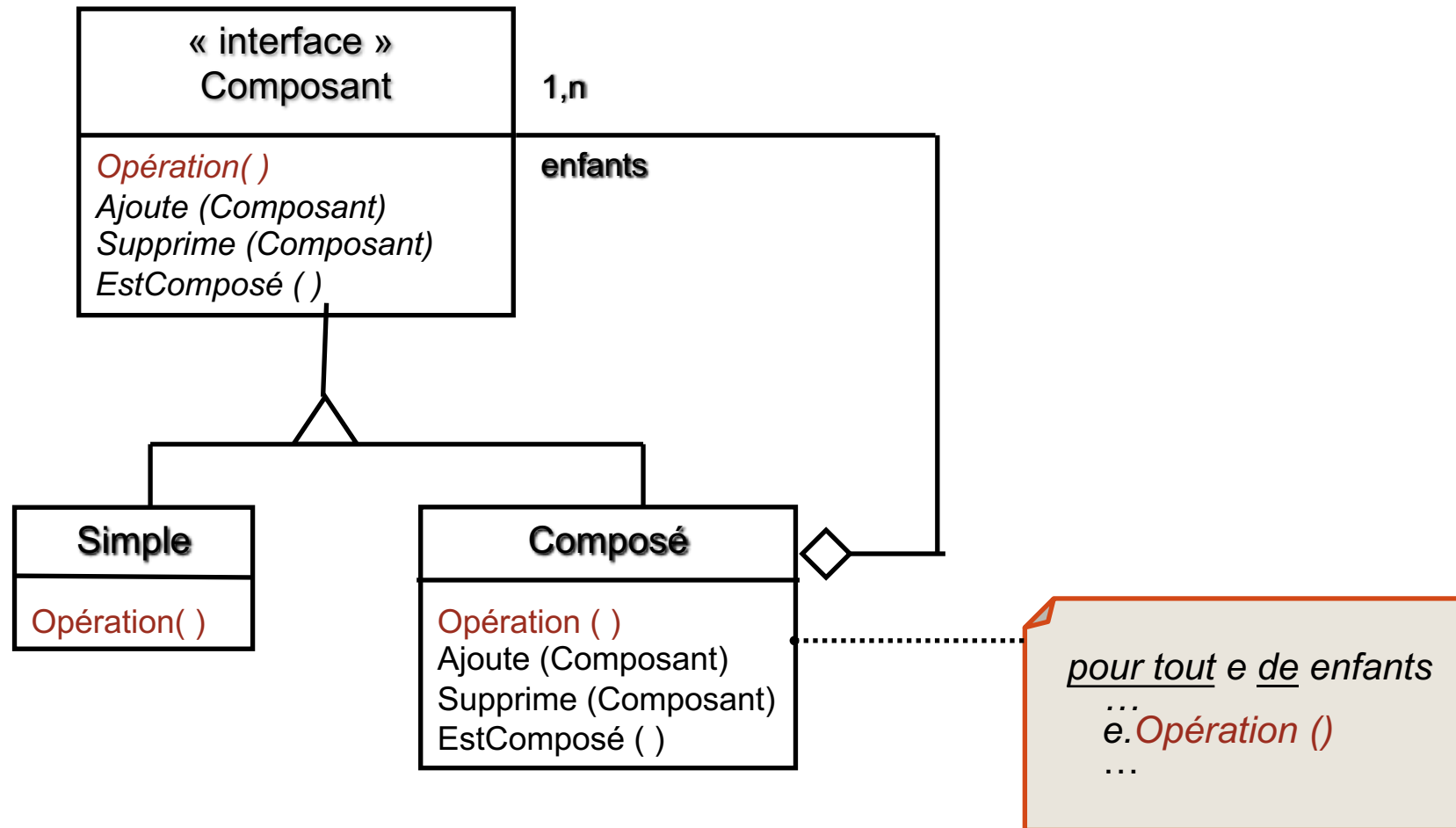


Patron de Structure

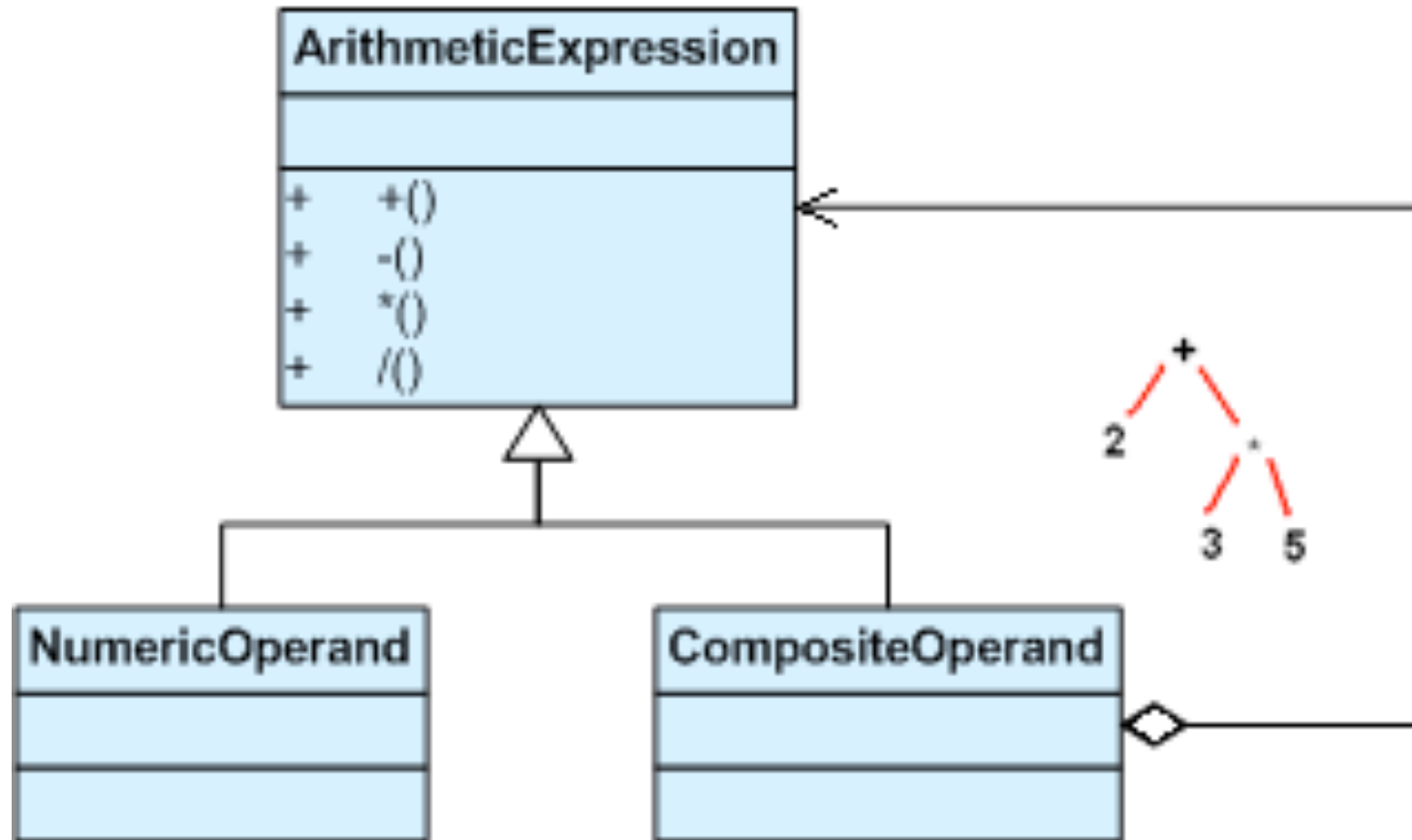
Patron Composite

- 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.

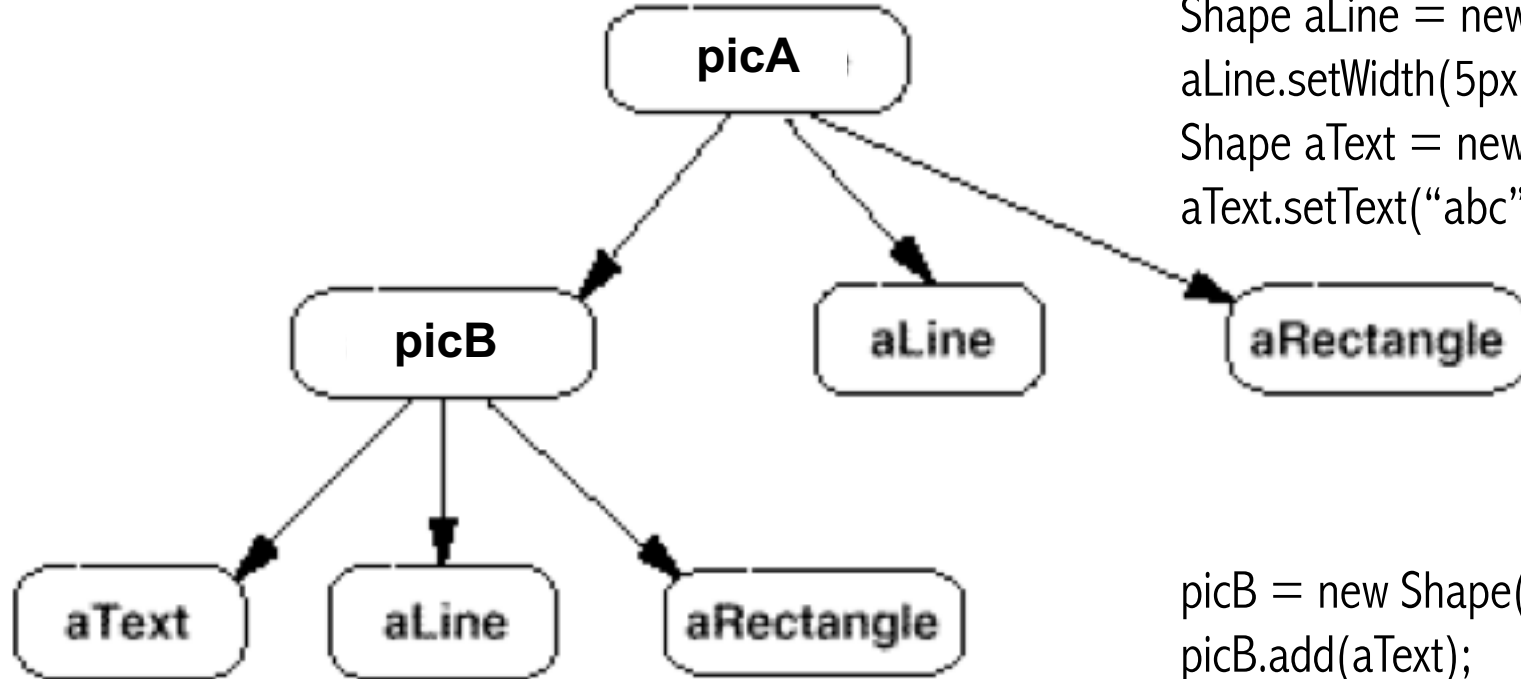
Pattern Composite



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();  
}
```

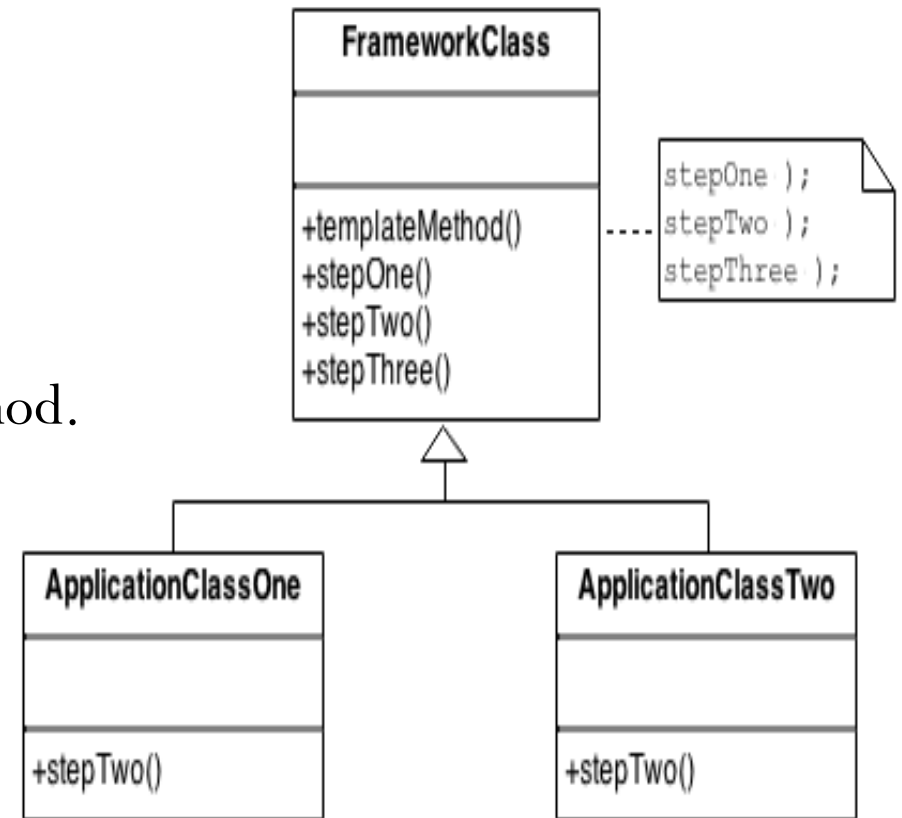

Le pattern TemplateMethod

Pattern comportemental à portée de Classe

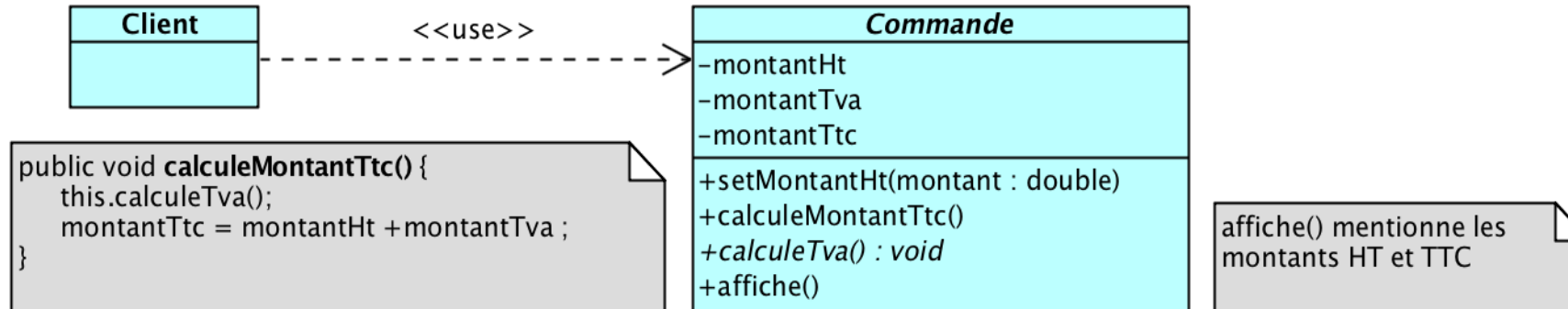


Fonctionnement et structure

- Une classe de base définit un algorithme (appelé *template method*) qui est composé de différentes étapes (= une succession de méthodes), certaines communes, d'autres spécifiques.
 - Des sous-classes définissent les méthodes spécifiques.
- Le client appelle la bonne version de la méthode ou peut recréer une sous-classe avec une nouvelle version (OCP)
→ Les Frameworks utilisent beaucoup le DP Template Method.



Exemple : calcul de montants TTC



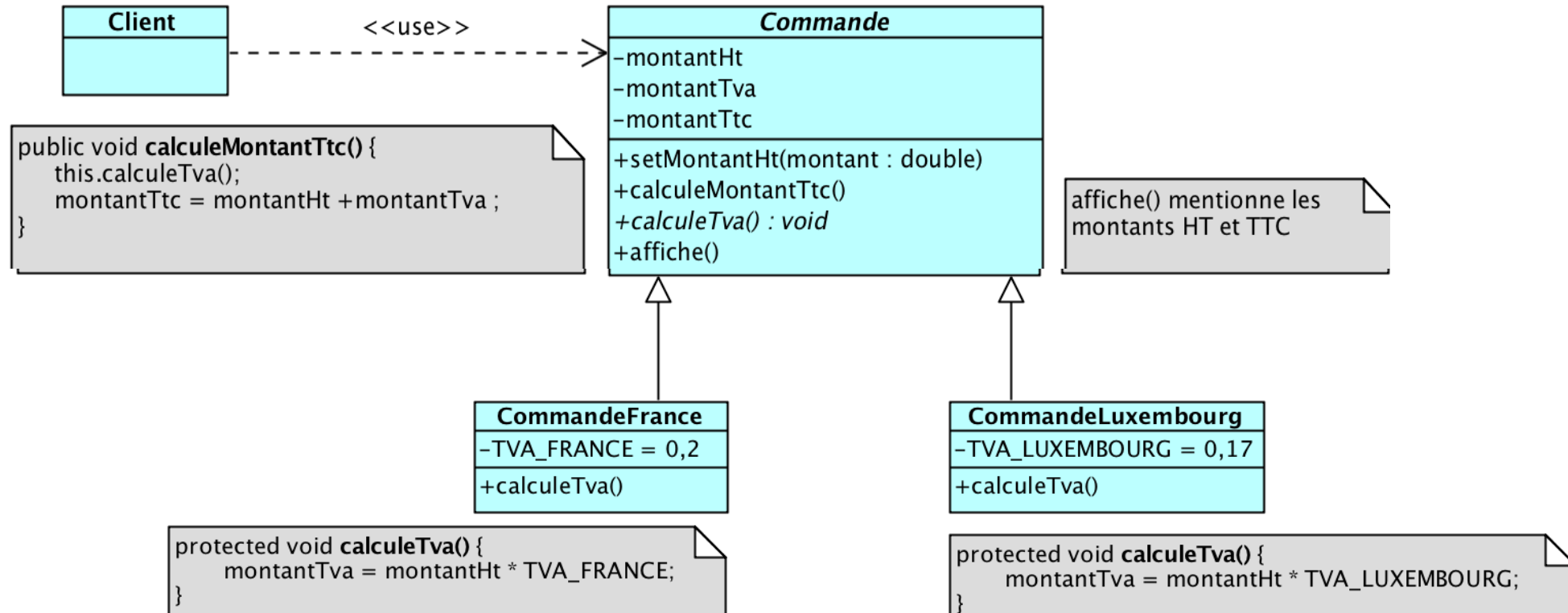
Ici **this** fera appeler à la « bonne » implémentation (celle de la sous-classe instanciée)

La **templateMethod** ici est :

- `setMontantHt()`
- `calculMontantTtc()`
- `affiche()`

L'implémentation des méthodes **communes** à toutes les sous-classes sont écrites dans la **classe de base**

Exemple : calcul de montants TTC



*Les méthodes **spécifiques aux sous-classes** sont abstraites ici et implémentées dans les sous-classes*

TemplateMethod vs. d'autres patterns

- **Strategy** ressemble à **Template Method**, mais :
 - **Template Method** utilise *l'héritage* pour la part variable de l'algorithme alors que **Strategy** utilise la *délégation* pour des versions entières d'algorithmes ;
 - **Strategy** modifie la logique des objets individuels. **Template Method** modifie la logique de **toute une classe**.
- **Factory Method** est une spécialisation de **Template Method**.

Conclusion



Sur les Design Patterns

Conclusion sur les DP

- Les Design Patterns fournissent un **outil puissant** :
 - **d'abstraction** des problèmes rendant le code facile à faire évoluer (ce qui augmente significativement la durée de vie du projet) ;
 - de documentation de **savoir-faire**;
 - de **nommage de concepts** universellement utilisés;
 - de **réutilisation** dans les projets.
- **Points négatifs**
 - *Complexification* du code car ils augmentent le nbre de classes ;
 - L'utilisation du polymorphisme propre aux design patterns *pénalisent les performances* en terme d'exécution, de mémoire et de compilation.