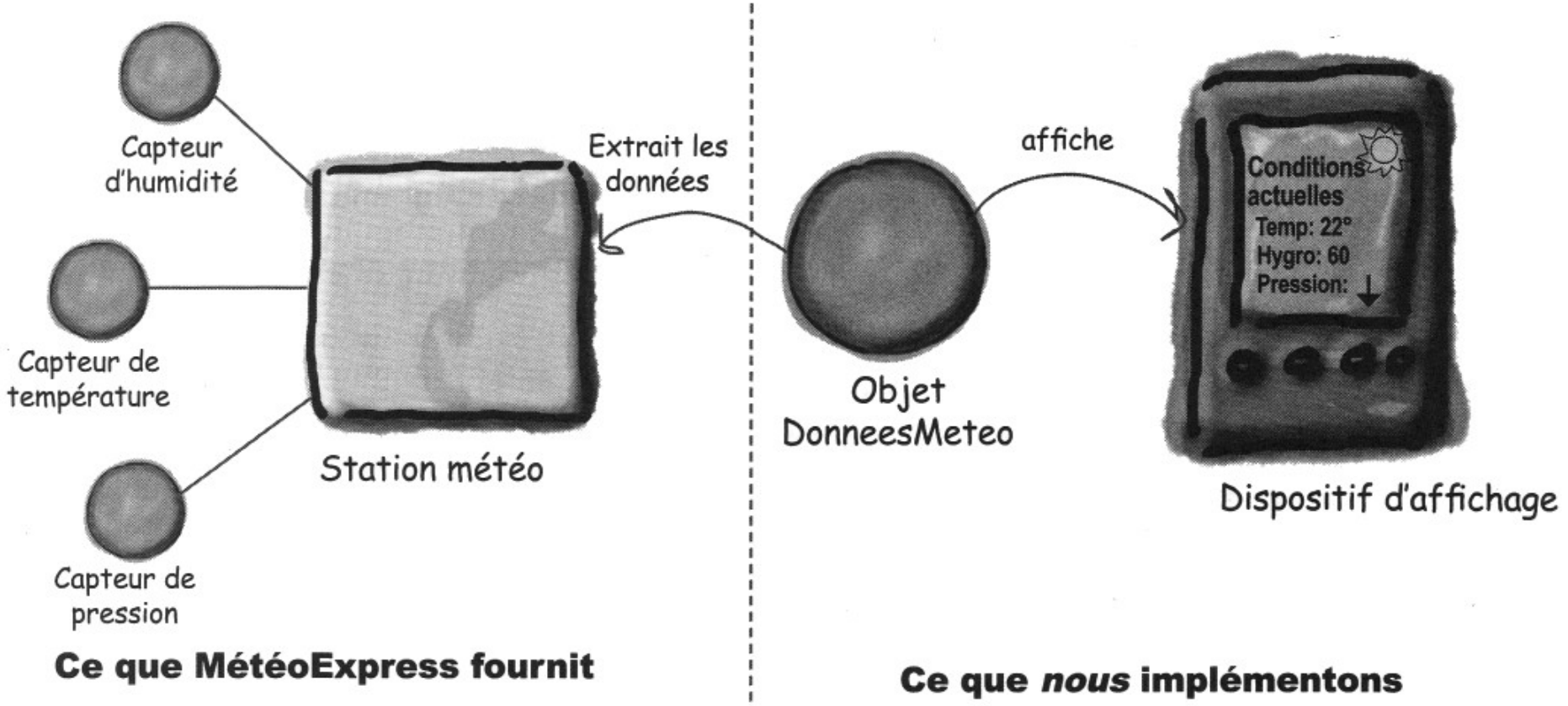
DESIGN PATTERNS

OBSERVATEUR/DECO/VISITEUR

Supports Frédéric Moal

Yohan BOICHUT
Université d'Orléans

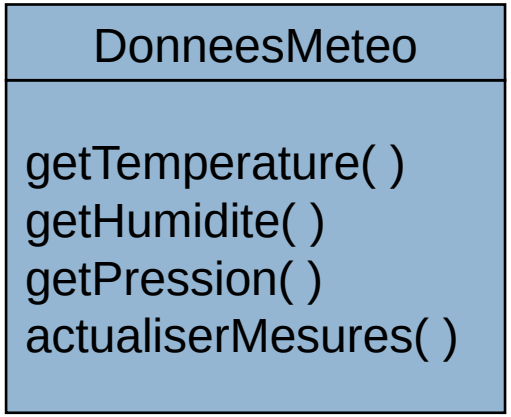
Premier contrat...



Définition du problème

- Méthode actualiserMesures à implémenter

  Elle est appelée automatiquement chaque fois qu'une nouvelle mesure est dispo



Pour l'instant, 3 affichages possibles : conditions actuelles, statistiques et prévisions

Solution 1 au problème

- Méthode actualiserMesures à implémenter

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

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

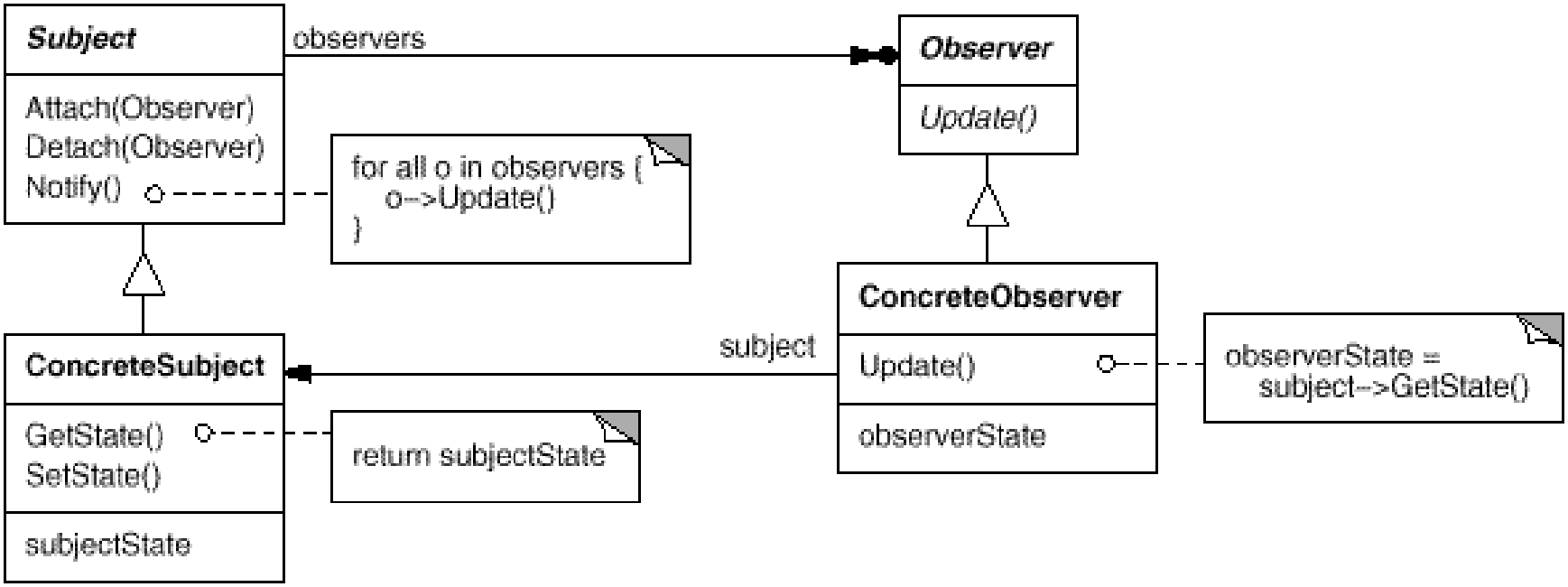
Solution 1 au problème

- Critiques de cette solution ?

Pattern Observateur / Observer

Objectif : permettre à un objet d'informer d'autres objets qu'il ne connaît pas de l'évolution de son état interne

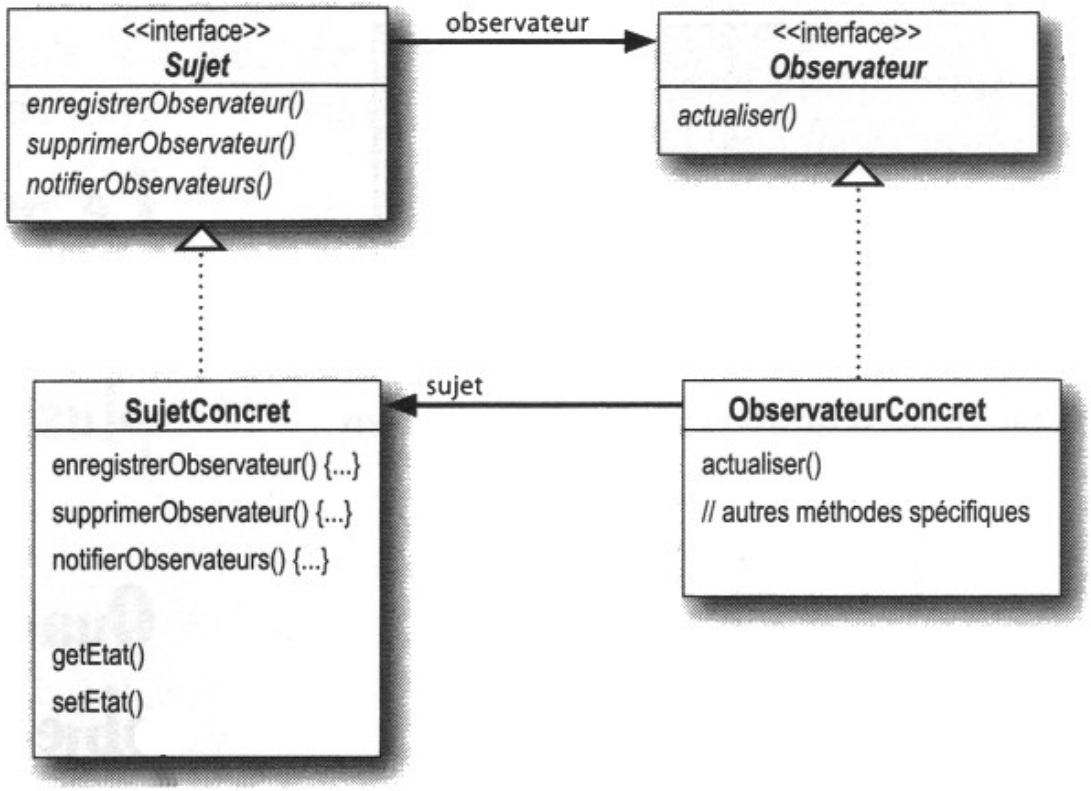
Exemple : un bouton à la suite d'un click



Pattern Observateur / Observer

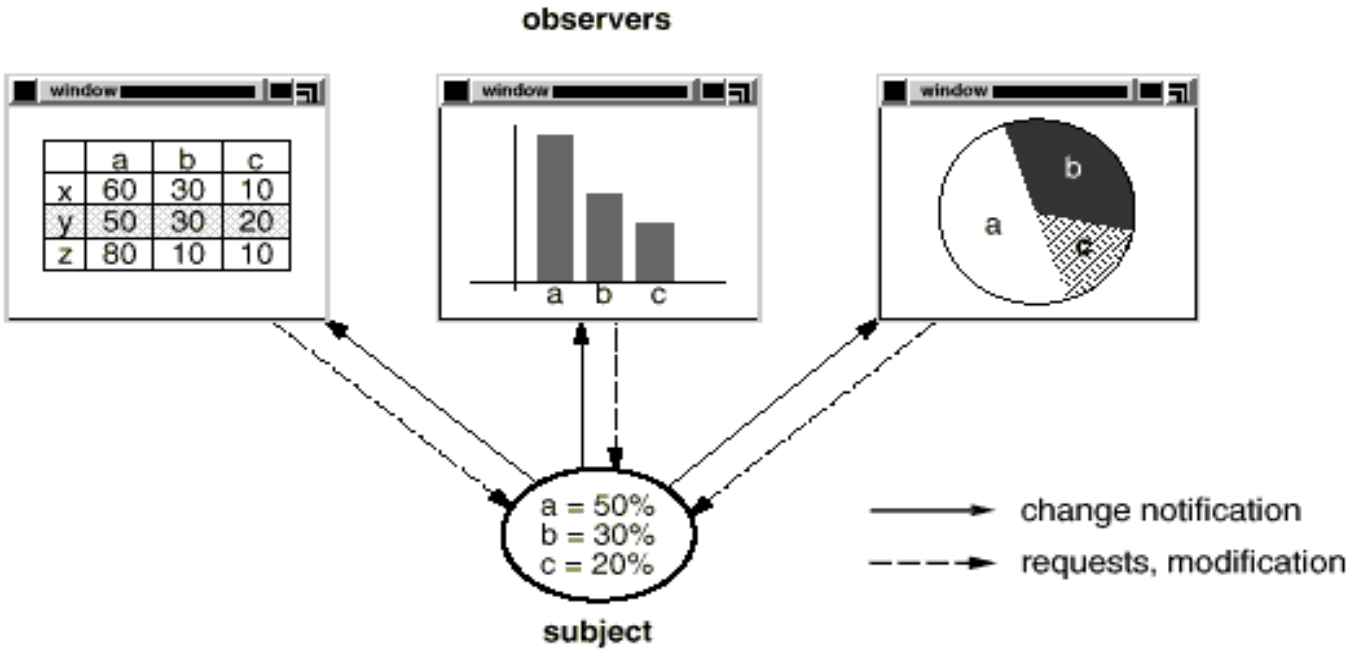
Objectif : permettre à un objet d'informer d'autres objets qu'il ne connaît pas de l'évolution de son état interne

Exemple : un bouton à la suite d'un click

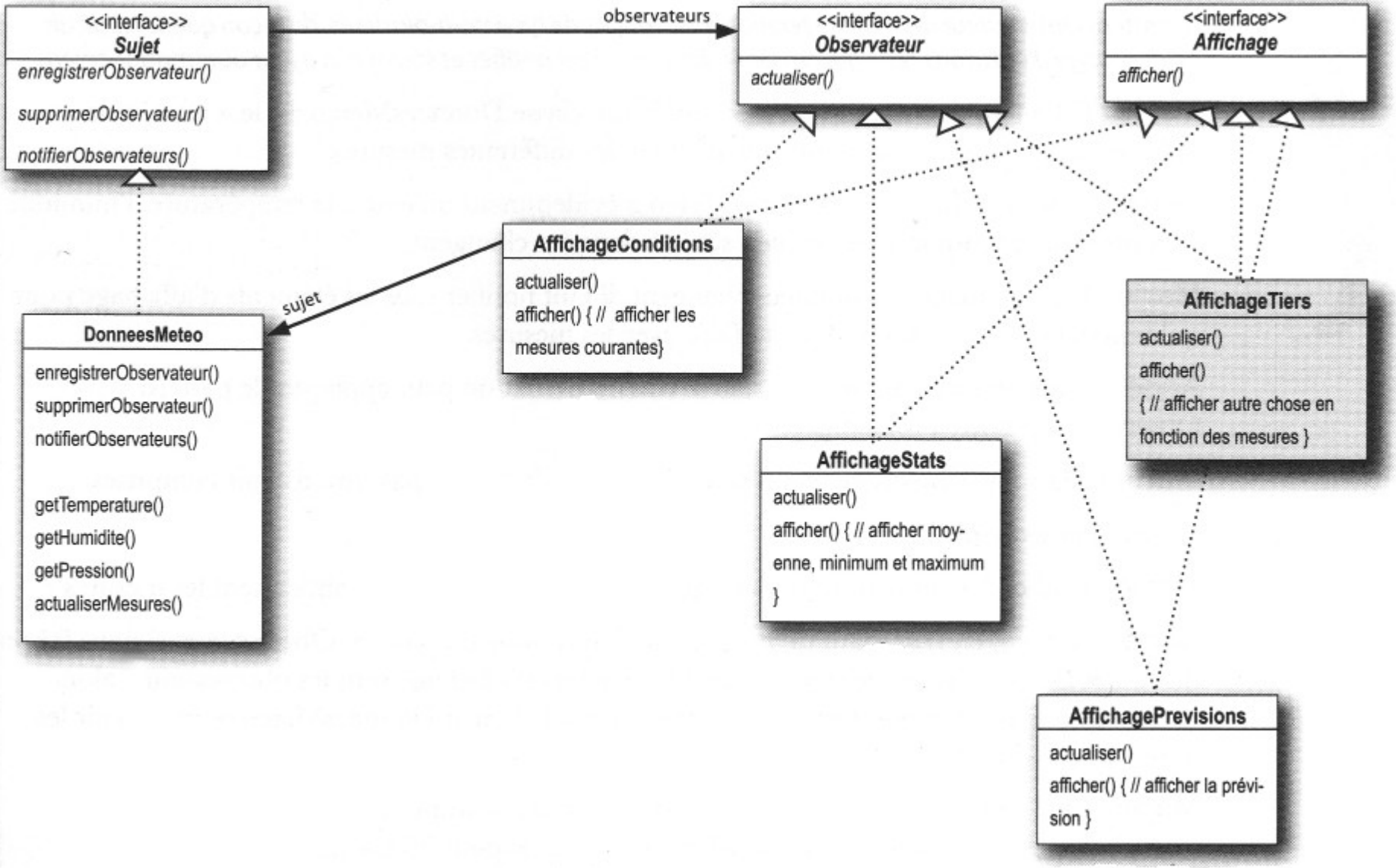


Observateur / Observer

Une dépendance 1-N entre les objets : le changement d'un objet sera envoyé à tous les observateurs



Solution 2 avec patron Observateur



Solution 2 - implémentation


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

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

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

Solution 2 - implémentation


```
public class DonneesMeteo implements Sujet {
    private List<Observateur> observateurs;
    private float temperature;
    private float humidite;
    private float pression;

    public DonneesMeteo() {
      observateurs = new ArrayList();
    }
    public void enregistrerObservateur(Observateur o) {
      observateurs.add(o);
    }
    public void supprimerObservateur(Observateur o) {
       observateurs.remove(i);
    }
    public void notifierObservateurs() {
      for (Observateur o : observateurs) {
       o.actualiser(temperature, humidite, pression);
      }
    }
    public void actualiserMesures() {
      notifierObservateurs();
    }
```

Solution 2 - implémentation

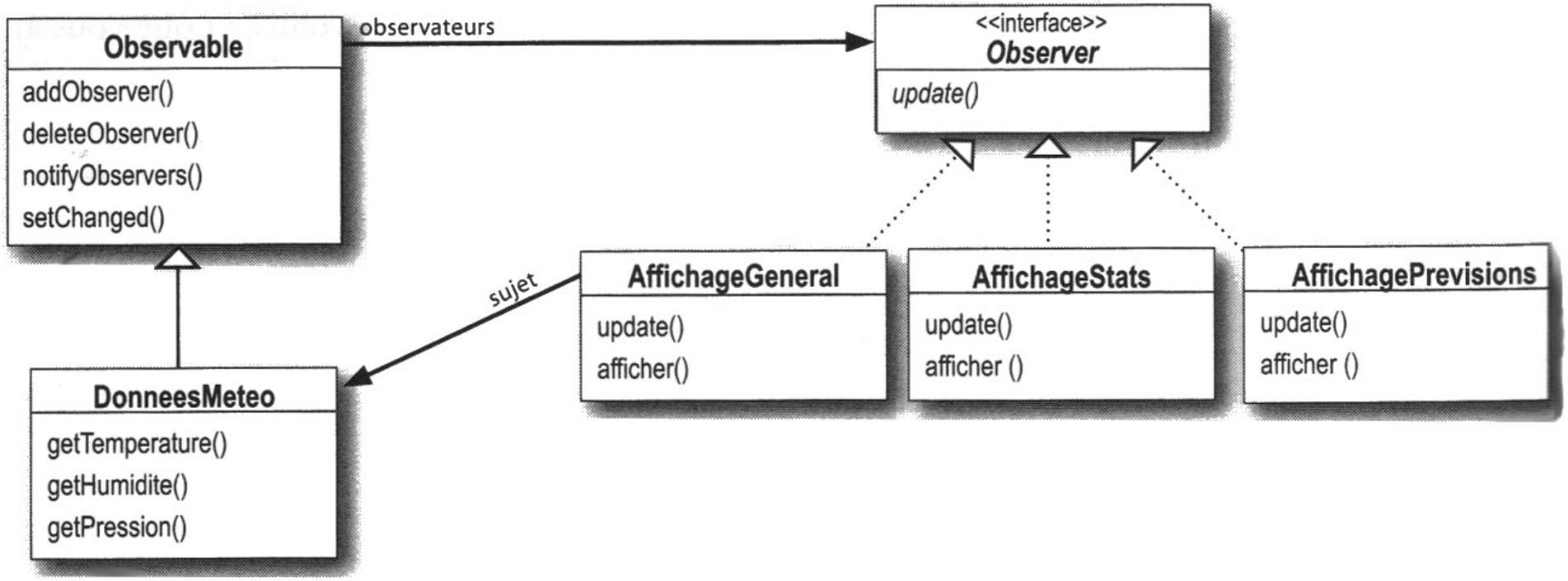

```
public class AffichageConditions implements Observateur, Affichage {
    private float temperature;
    private float humidite;
    private Sujet donneesMeteo;

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

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

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

Version JAVA de Observateur



Version JAVA de Observateur


```
public class DonneesMeteo extends Observable {
   private float temperature;
   private float humidite;
   private float pression;

   public DonneesMeteo() { }

   public void measurementsChanged() {
      setChanged();
      notifyObservers();
   }

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

Version JAVA de Observateur


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

   public AffichageConditions(Observable observable) {
      // this.observable = observable;
      observable.addObserver(this);
   }
   public void update(Observable obs, Object arg) {
      if (obs instanceof DonneesMeteo) {
       DonneesMeteo donneesMeteo = (DonneesMeteo)obs;
       this.temperature = donneesMeteo.getTemperature();
       this.humidite = donneesMeteo.getHumidite();
       afficher();
      }
   }
   public void afficher() {
       System.out.println("Conditions actuelles: " + temperature
       + " degrés C et " + humidite + "% d'humidité");
   }
}
```

Conclusions +

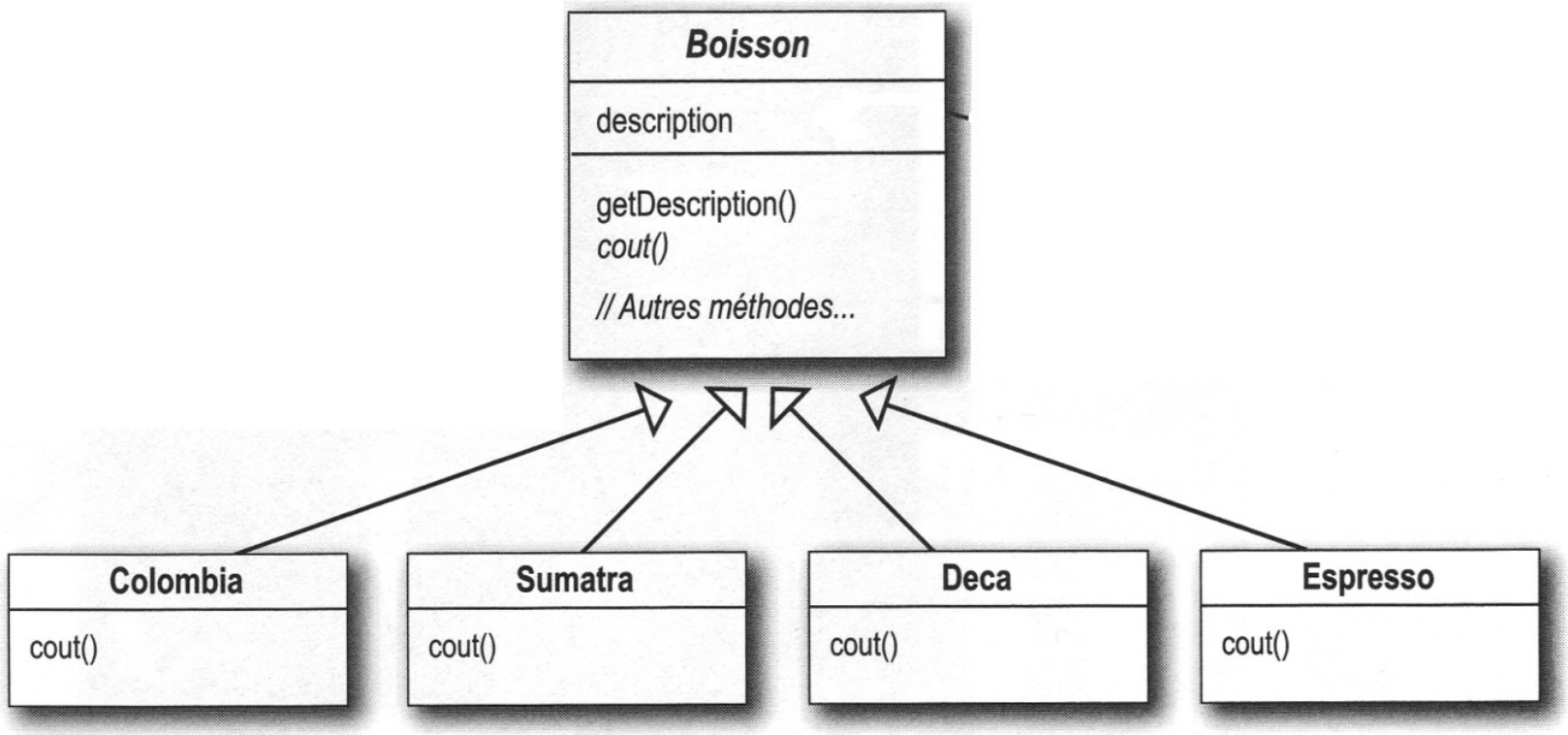
- Principes de conception
  - Encapsulez ce qui varie
  - Programmez des interfaces et non des implémentations
  - Préférez la composition à l'héritage
  - Inversion des dépendances : dépendez d'abstractions et non de classes concrètes
  - Efforcez vous de coupler faiblement les objets qui interagissent
- Patron de conception
  - Stratégie…
  - Fabrique, fabrique abstraite, Monteur
  - Observateur : définit une relation 1-n entre objets. Lorsqu'un objet change d'état, tous ceux qui en dépendent en sont notifiés et se mettent à jour.

Votre pote Jojo

Où comment vous devez assumer vos paroles de la dernière soirée au V&B avec Jojo en lui modélisant son SI en moins de 30 minutes...

Jojo lance son commerce...

- Cafés :



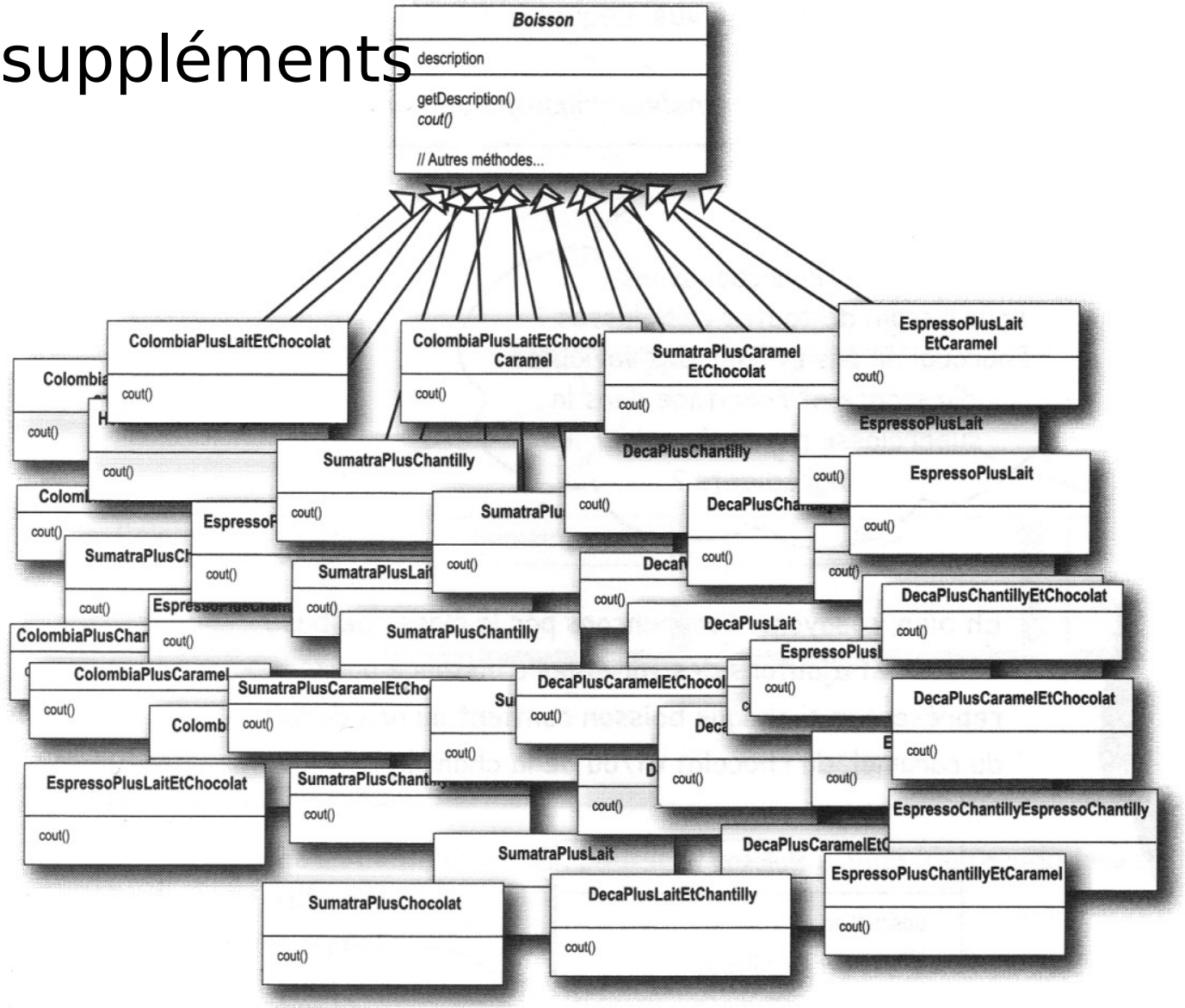
Nouveau problème…

- Cafés avec suppléments

- Lait
- Chocolat
- Chantilly
- Caramel
- …

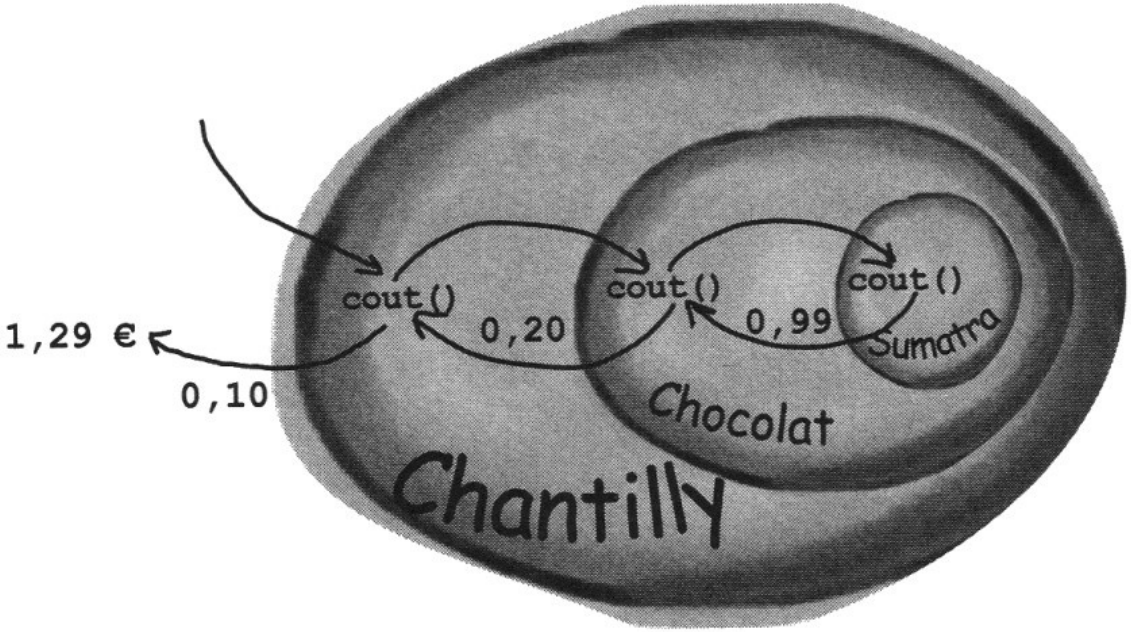
Nouveau problème...

- Avec suppléments



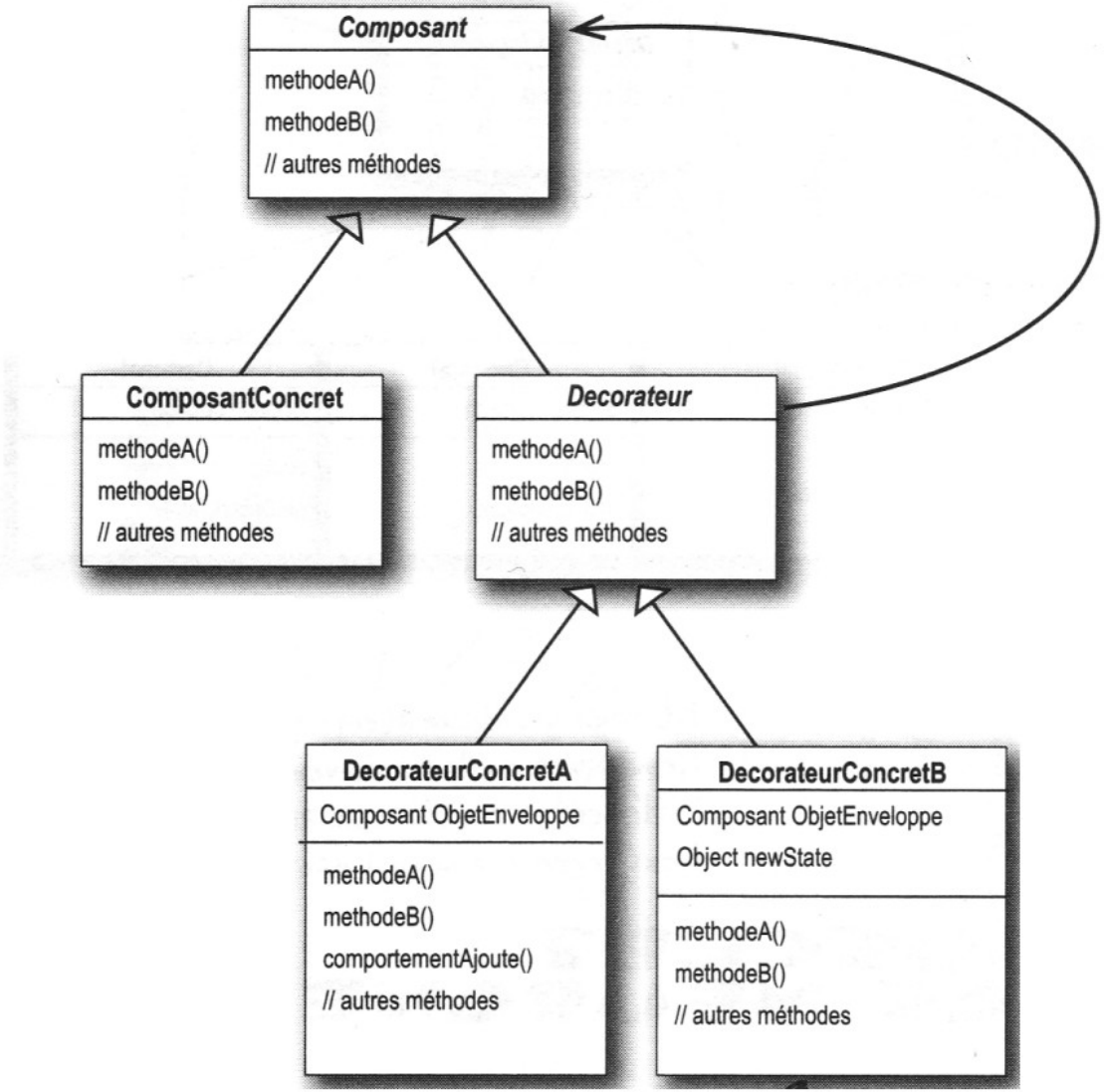
Principe à appliquer

- On « décore » le café avec ses suppléments



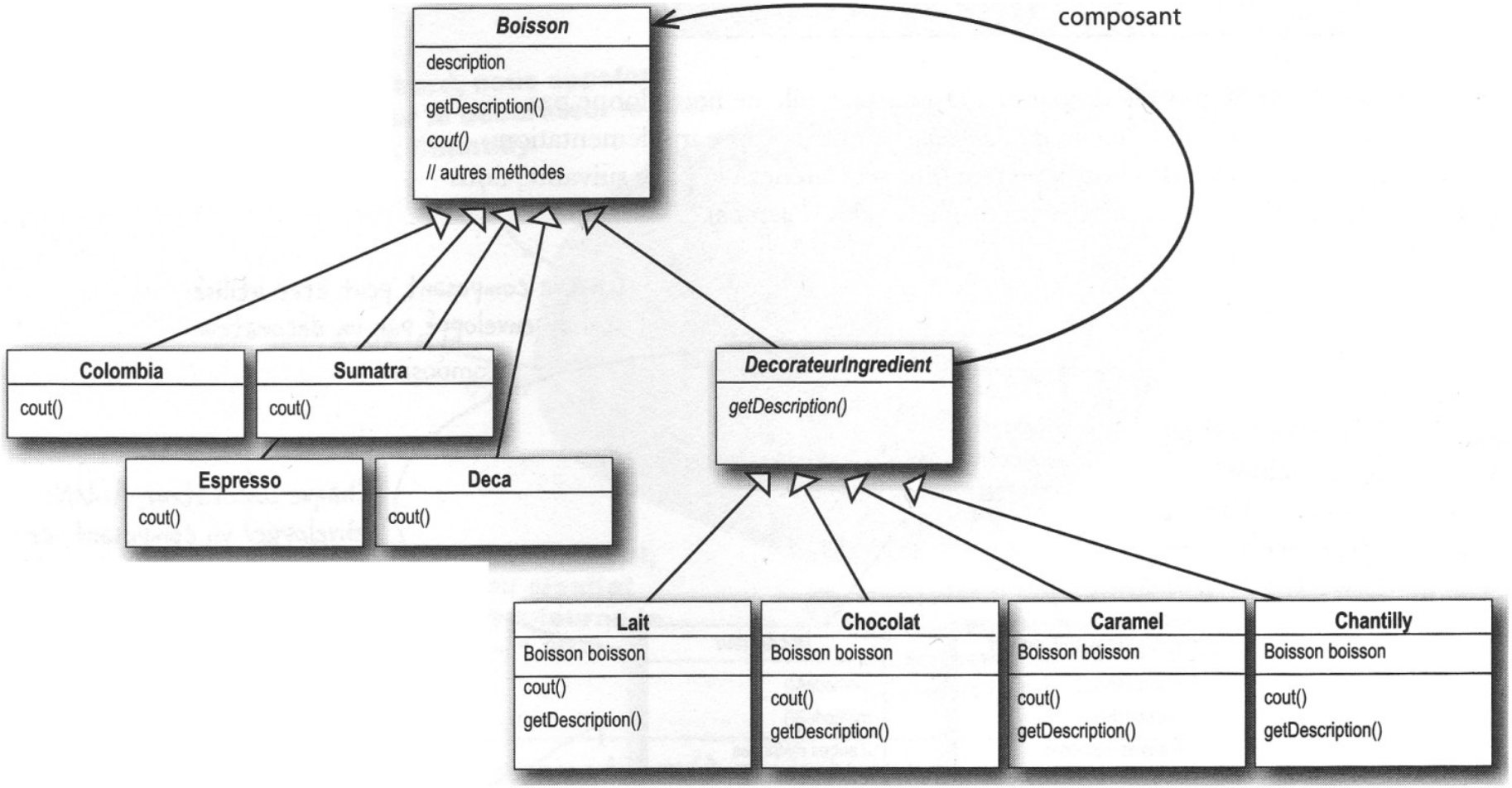
C’est le principe utilisé pour les E/S Java
C’est un patron de conception !

Patron décorateur / decorator



Patron décorateur / decorator

- Solution Cafés avec décorateur



Conclusions +

- Principes de conception
  - Encapsulez ce qui varie
  - Programmez des interfaces et non des implémentations
  - Préférez la composition à l'héritage
  - Efforcez vous de coupler faiblement les objets qui interagissent
  - Les classes doivent être ouvertes à l'extension mais fermées à la modification
- Patron de conception
  - Stratégie
  - Fabriques
  - Observateur
  - Décorateur : attache des responsabilités supplémentaires à un objet de façon dynamique. Alternative à la dérivation pour étendre les fonctionnalités

Le décorateur 2

Le retour,

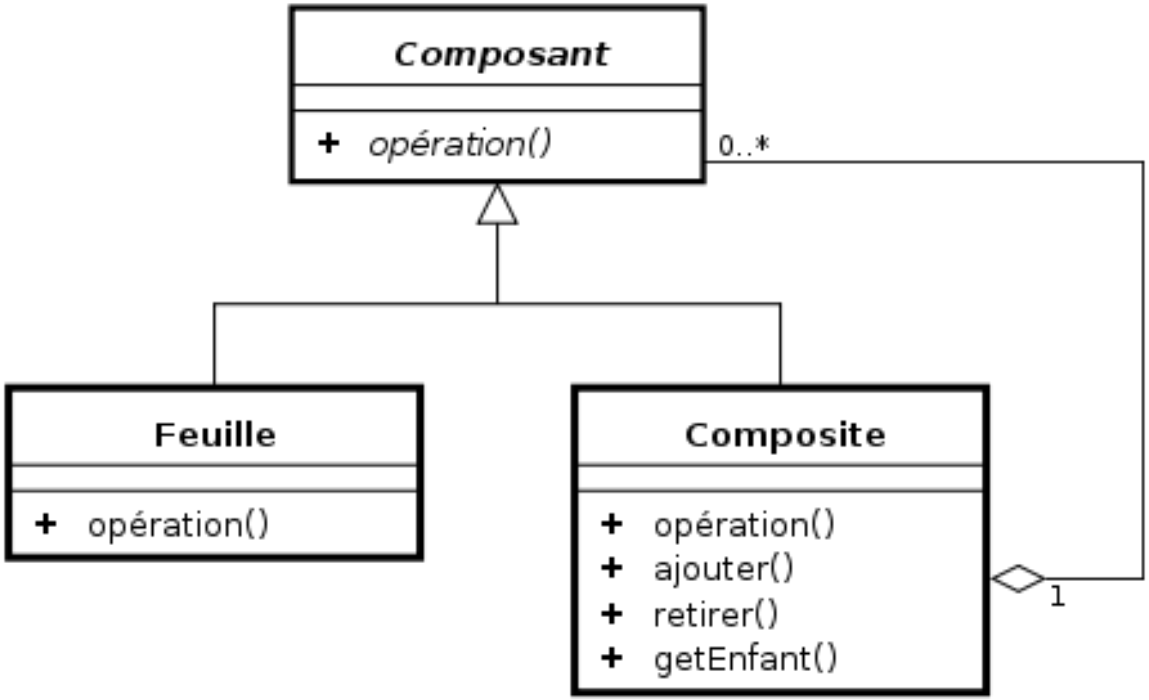
Et il est pas content !

Un système de fichier...

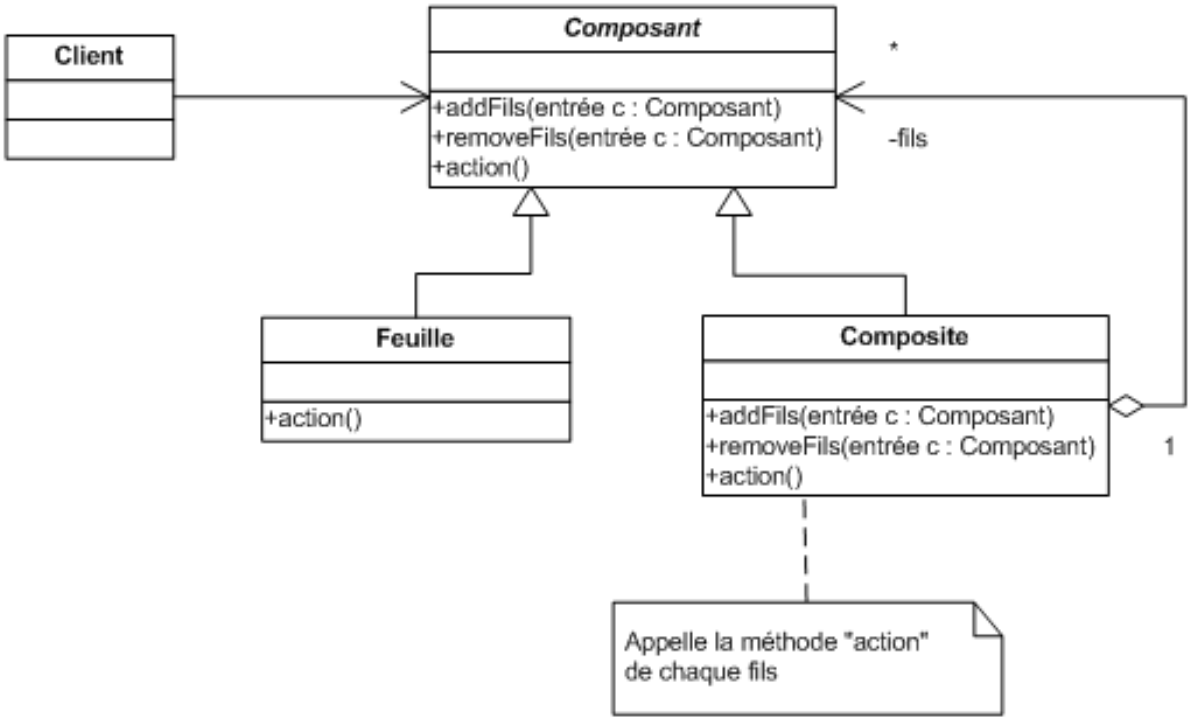
- Trois types d’objets :
  - Un répertoire
  - Un fichier
  - Un lien

- Quelle structure pour les représenter ?

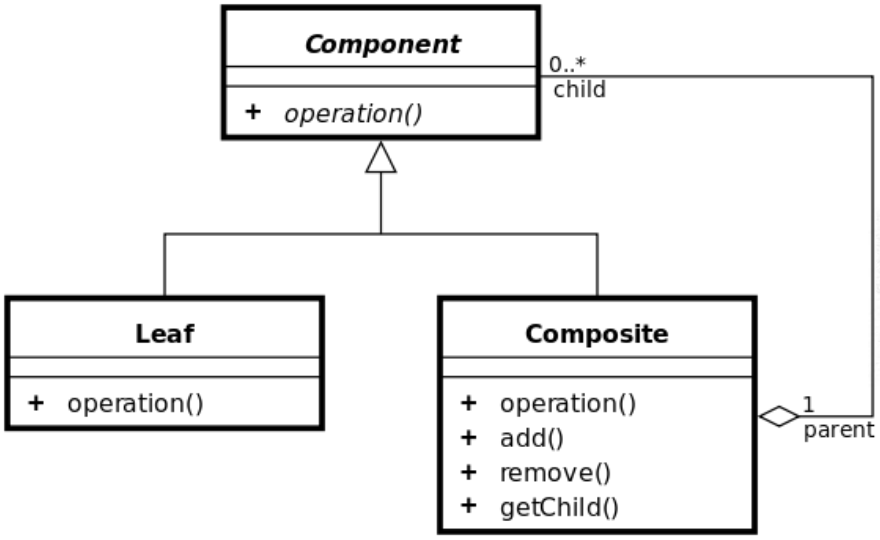
Pattern Composite

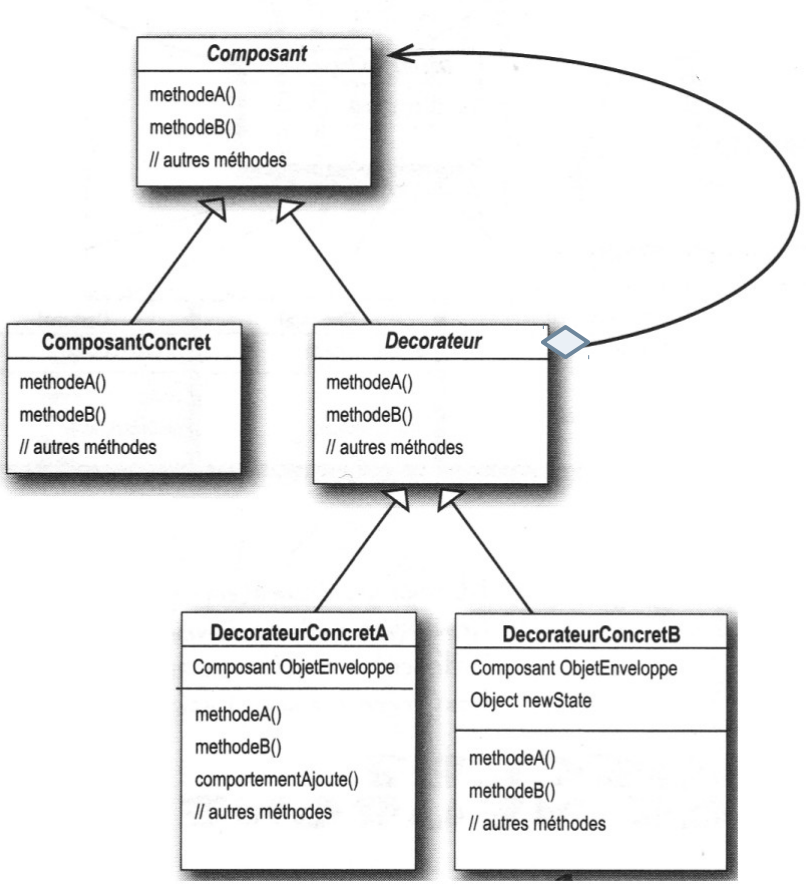


Pattern Composite : variante bof



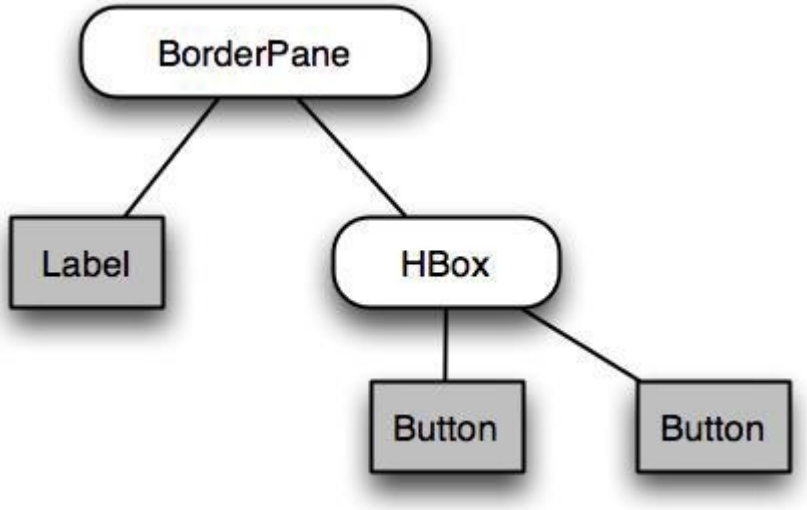
WTF !


Component
+ operation()
0..*
child
Leaf
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent


Composant
methodeA()
methodeB()
// autres méthodes
ComposantConcret
methodeA()
methodeB()
// autres méthodes
Decorateur
methodeA()
methodeB()
// autres méthodes
DecorateurConcretA
Composant ObjetEnveloppe
methodeA()
methodeB()
comportementAjoute()
// autres méthodes
DecorateurConcretB
Composant ObjetEnveloppe
Object newState
methodeA()
methodeB()
// autres méthodes

Exemple de composite

- En java ...



Exemple de composite

- Autre exemple ?

  (1+2)*3

Les copaings

- Visitor