Objet et développement d'applications
Cours 5 - Pattern Decorator

Patrons de conception pour POO

Florian Richoux

2014-2015

Starbucks

Fil rouge du cours

On souhaite calculer le coût de quelque chose de hautement paramétrable. Par exemple, un café chez Starbucks.

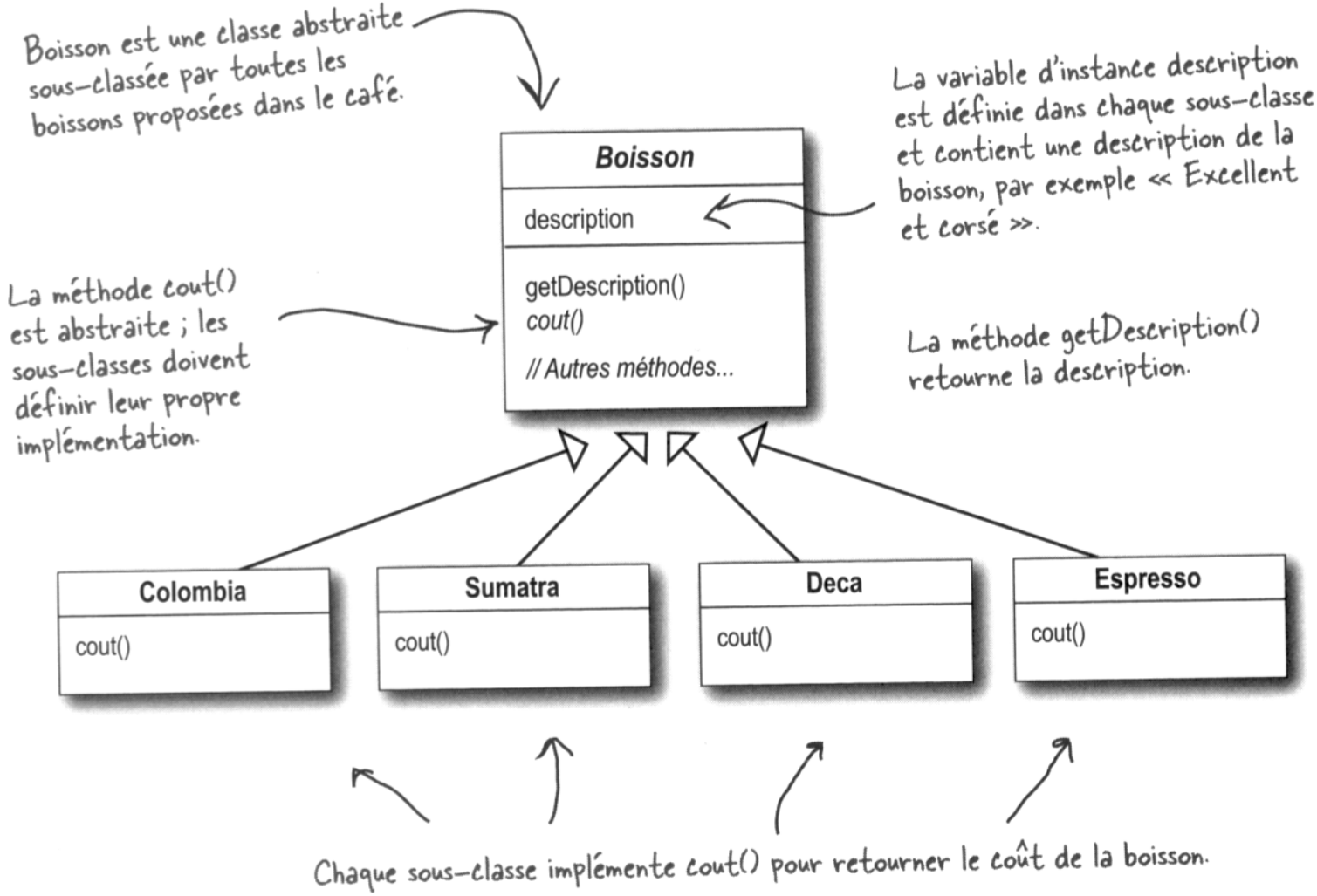
Ah c'est vrai, y'a pas de Starbucks sur Nantes...

Commande d'un café chez Starbucks : Vous choisissez votre café, puis :

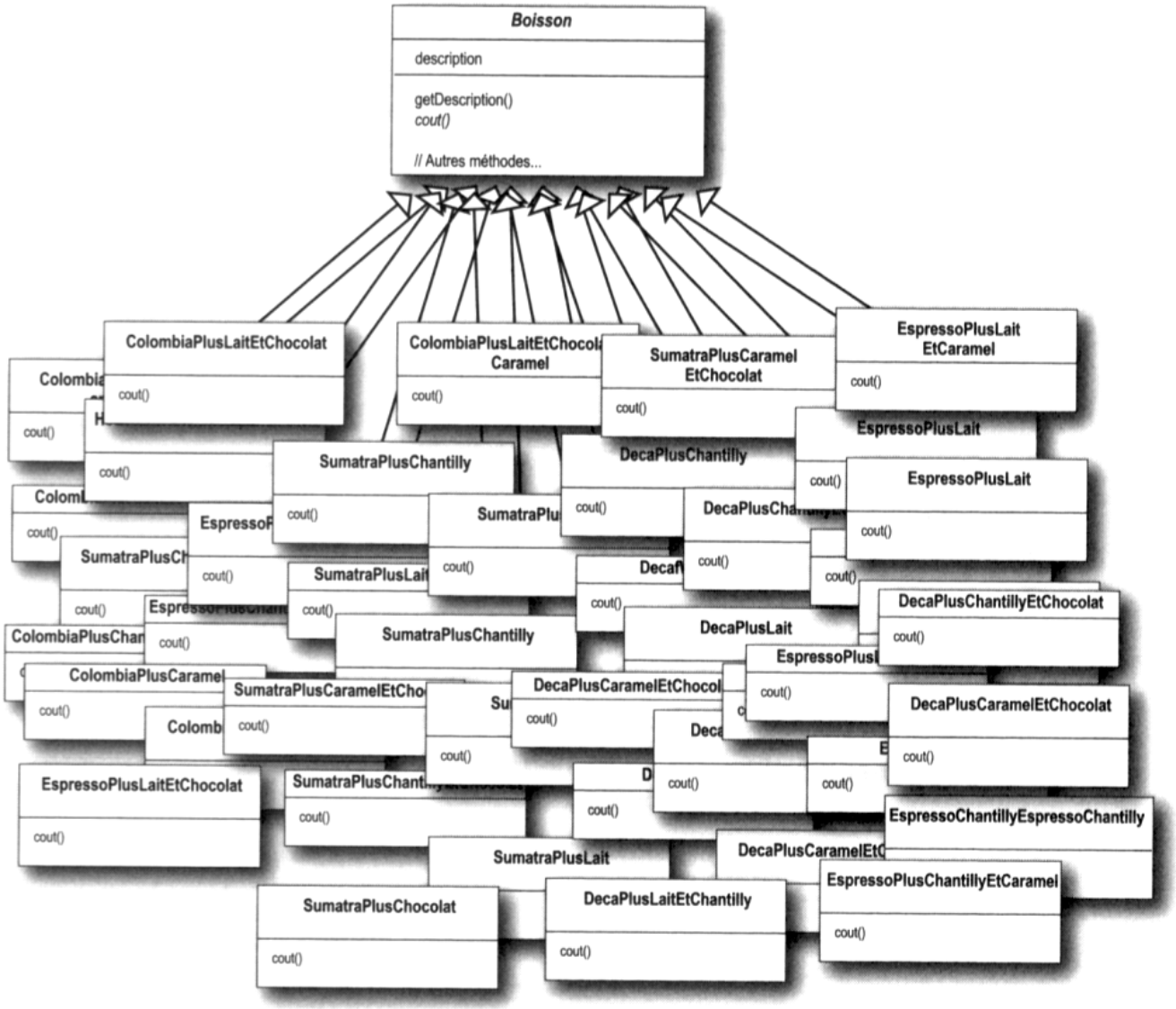
- la taille,
- avec ou sans chantilly,
- avec ou sans chocolat,
- avec ou sans caramel,
- ...

Bien sûr, ces choix ont un impact sur le prix final.

Commande de base



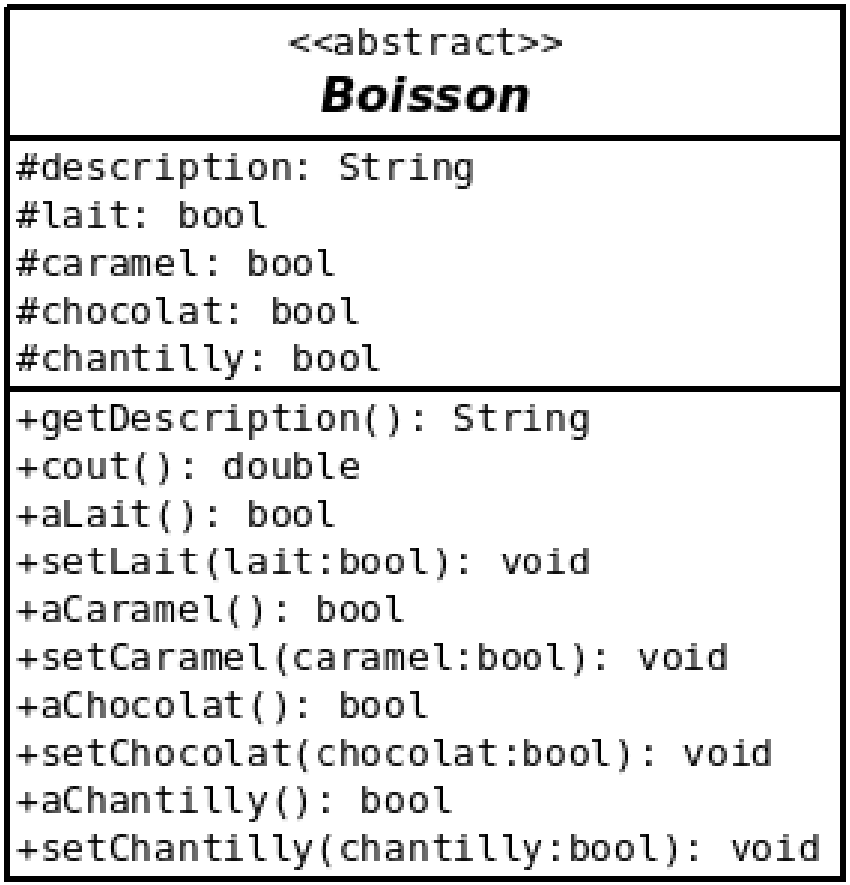
Commande complète



Première idée

Amélioration

- Utiliser des booléens pour décrire une commande.
- Hériter ensuite de la classe abstraite Boisson.



Exemple : coût d'un café colombien avec lait et caramel

Amélioration

```
class Colombia extends Boisson {
  ...
  public double cout() {
    double colombia = 1.20;

    return colombia +
    (lait ? 0.25 : 0) +
    (caramel ? 0.40 : 0) +
    (chocolat ? 0.40 : 0) +
    (chantilly ? 0.30 : 0);
  }
}
```

Est-ce la solution ?

Question

Procéder ainsi n'est pas une bonne idée. Pourquoi ?

Est-ce la solution ?

Question

Procéder ainsi n'est pas une bonne idée. Pourquoi ?

Fortes dépendances

Même problème qu'avec les canards :

- Changer les prix entraîne des modifications dans les classes.
- De même pour l'ajout/suppression d'ingrédients.
- On peut imaginer de nouvelles boissons où certains ingrédients ne sont pas possibles. Comment coder ça ?
- Et si quelqu'un veut un double chocolat, comment on fait ?

Rappelez-vous : principe ouvert-fermé

Principe ouvert-fermé

Il faut garde en tête le principe ouvert-fermé : être ouvert à l'extension et fermé à la modification.

Encore un pattern pour nous sauver

On va utiliser le pattern Decorator.

Pattern Decorator

Idée intuitive

Pour faire notre café colombien lait-caramel :

1. Instancier un objet Colombia,
2. Le décorer avec un objet Lait,
3. Le décorer avec un objet Caramel,
4. Appeler `cout` par délégations.

Décorer ?

Décorer = envelopper. Un décorateur aura le même type que l'objet qu'il décore.

Décoration

Question

Pourquoi le décorateur doit avoir le même type que l'objet qu'il décore ?

Décoration

Question

Pourquoi le décorateur doit avoir le même type que l'objet qu'il décore ?

Réponse

On veut pouvoir faire des choses comme cela :

```
Boisson cafe = new Colombia();
cafe = new Lait( cafe );
cafe = new Caramel( cafe );
```

Ce que l'on sait

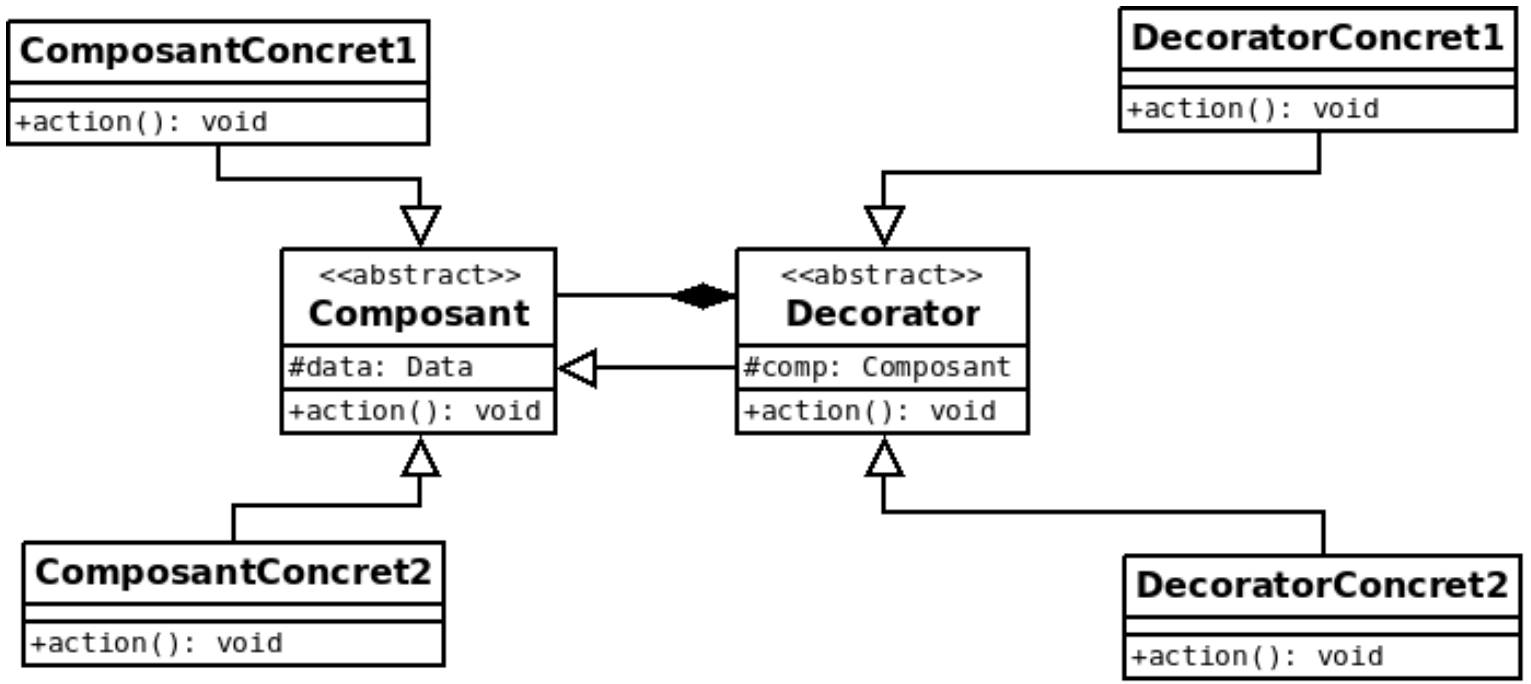
Nous savons que :

- Les décorateurs ont le même type (donc la même super-classe) que les objets qu'ils décorent.
- On peut donc manipuler et transmettre un objet décoré à la place de l'objet original.
- On peut décorer plusieurs fois un objet, éventuellement avec le même décorateur.

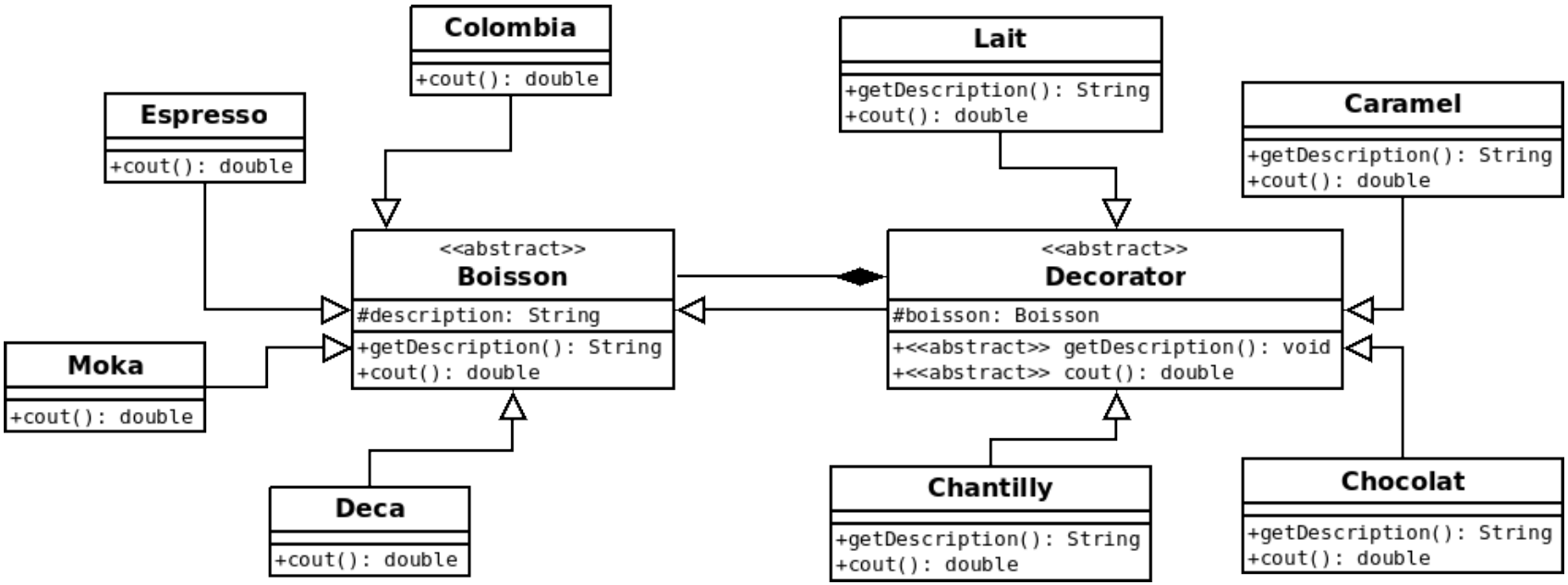
Pattern Decorator

Définition

Le pattern Decorator (Décorateur) attache dynamiquement des responsabilités supplémentaires à un objet. Il fournit une alternative souple à l'héritage, pour écrire les fonctionnalités.



Pour notre exemple



WTF?

Euh...

On devait pas éviter l'héritage et préférer la composition ?

Si !

Et c'est ce que l'on fait. L'héritage Composant $\rightarrow$ Decorator (lire "Decorator hérite de Composant") ne sert ici qu'à forcer le typage du décorateur.

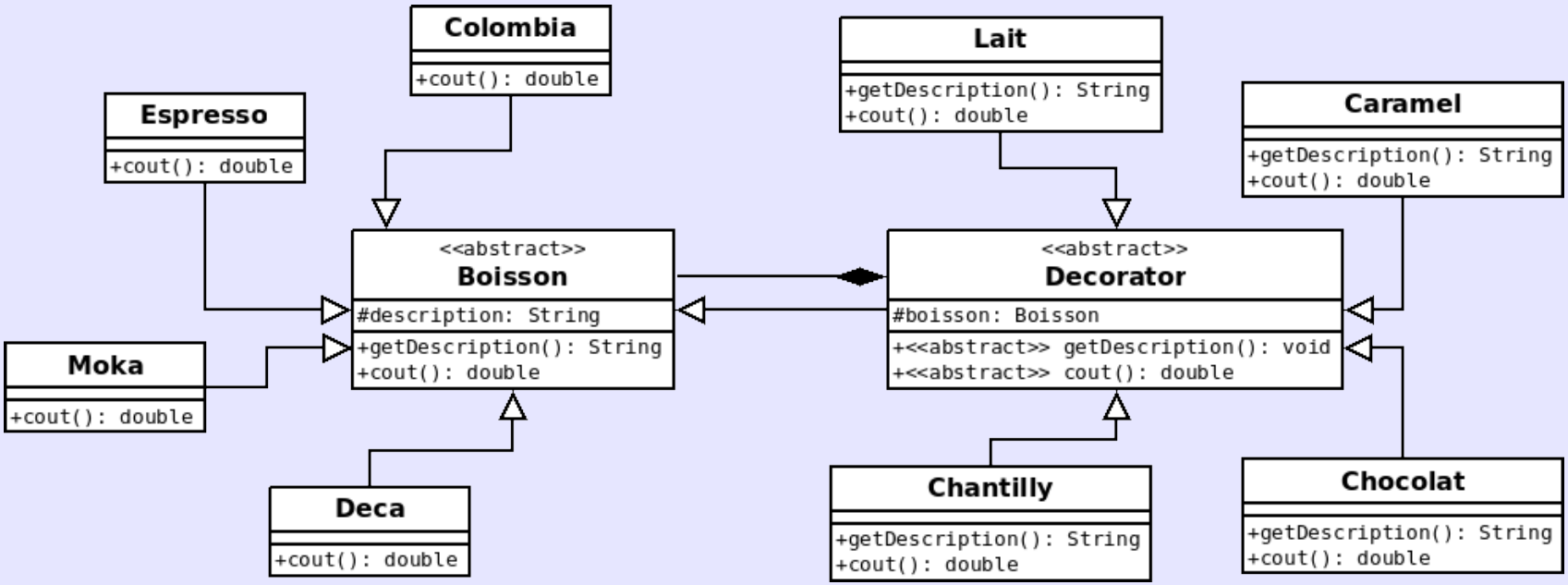
On a bien ici une composition : Decorator a une variable Composant en attribut.

En d'autres termes

Héritage vs. composition

- Decorator hérite de Composant pour qu'il y ait une correspondance de type, pas pour hériter du comportement de Composant.
- L'ajout d'un comportement se fait par composition, quand on ajoute un décorateur à un composant. (cafe = new Lait(cafe))
- Héritage = comportements déterminés statiquement, à la compilation.
- Composition = comportements ajoutés dynamiquement, lors de l'exécution.

Détails du code



Decorator vs. Strategy

Les patterns Strategy et Decorator semblent faire la même chose : ajouter des comportements. Qu'est ce qui les différencie ?

Decorator vs. Strategy

Decorator vs. Strategy

Les patterns Strategy et Decorator semblent faire la même chose : ajouter des comportements. Qu'est ce qui les différencie ?

Réponse

- Strategy permet de disposer d'une famille de comportements, et de passer facilement d'un comportement à l'autre.

Decorator vs. Strategy

Decorator vs. Strategy

Les patterns Strategy et Decorator semblent faire la même chose : ajouter des comportements. Qu'est ce qui les différencie ?

Réponse

- Strategy permet de disposer d'une famille de comportements, et de passer facilement d'un comportement à l'autre.
- Decorator permet d'activer (voire aussi de désactiver) dynamiquement un comportement.

Pour ceux qui connaissent Builder/Factory

Pattern Factory

Sert à fabriquer des objets en suivant des étapes de fabrications prédéfinis. Ex: la recette d'un gâteau.

Pattern Builder

Sert à fabriquer des objets en choisissant d'y inclure telle ou telle étape. Ex: le montage d'un PC.

Decorator vs. Factory/Builder

Factory et Builder servent à créer des objets ayant les comportements que l'on souhaite.

Decorator gère dynamiquement des objets déjà crées, et leur ajoute des comportements.

Decorator dans l'API Java

http://docs.oracle.com/javase/7/docs/api/index.html

Premier pattern de structure que l'on voit

Decorator est un pattern de **structure**

Patterns de structure : sert à structurer la composition des classes.

- Point de vue des classes : on utilise l'**héritage** pour la correspondance de type afin d'avoir une interface commune.
- Point de vue des objets : on utilise la **composition** pour avoir de nouvelles fonctionnalités.

La semaine prochaine

Deux nouveaux patterns de structure pour le prix d'un.