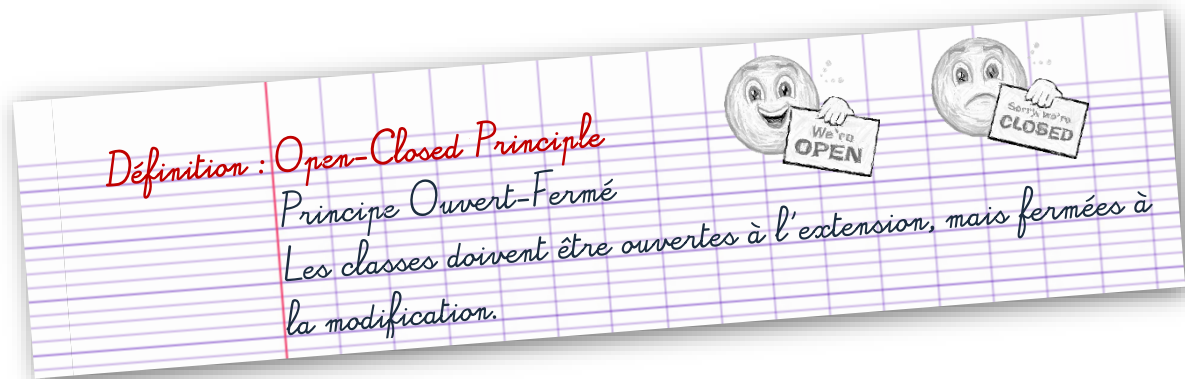


TD 10 – Premiers Principes de Conceptions Orientée Objet

Cours 1. Open-Closed Principle

1.1 Définition

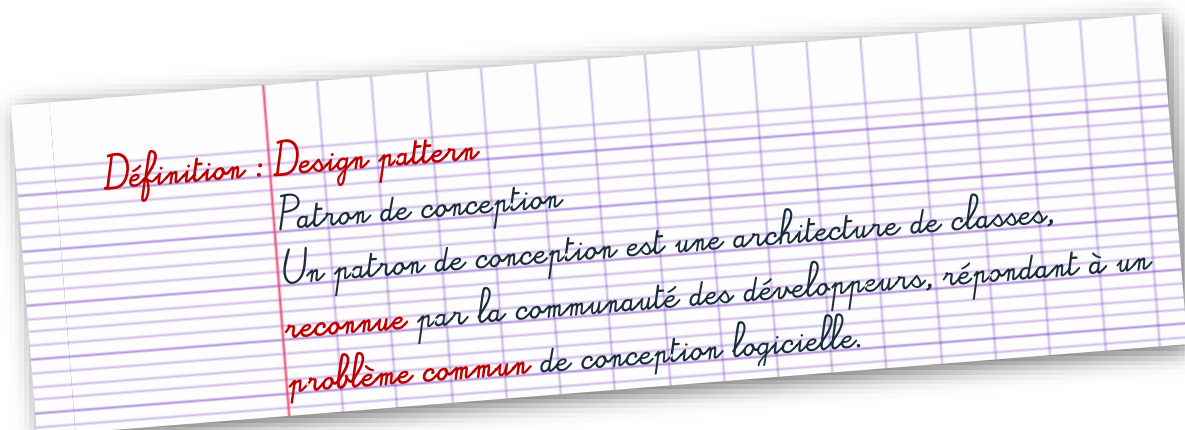



- ➡ Le but est de permettre d'étendre facilement les classes pour incorporer de nouveaux comportements sans modifier le code existant.
- ➡ Malgré la contradiction apparente, il existe des techniques qui permettent d'étendre le code sans le modifier directement

🐛 Comme pour tout principe de conception, appliquer le principe Ouvert-Fermé PARTOUT est inutile, peu rentable et susceptible de déboucher sur un code complexe et difficile à comprendre.

Cours 2. Les designs patterns

2.1 Définition



 Vous savez déjà réutiliser du code (de vos anciens projets, copié-collé d'internet, ...), les designs patterns vous permettent de réutiliser de l'expérience !

Cours 3. Le pattern Décorateur

Sur l'exemple de l'exercice 1, partons de la boisson **Deca**. Supposons que le client veuille du lait, alors créons un objet de type **Lait** et enveloppons l'objet de type **Deca** dedans. L'objet boisson est alors un objet de type **Lait** qui possède un membre boisson **Deca**. Si maintenant il souhaite ajouter de la **Chantilly** à sa boisson, alors enveloppons l'objet précédent dans un objet de type **Chantilly** :



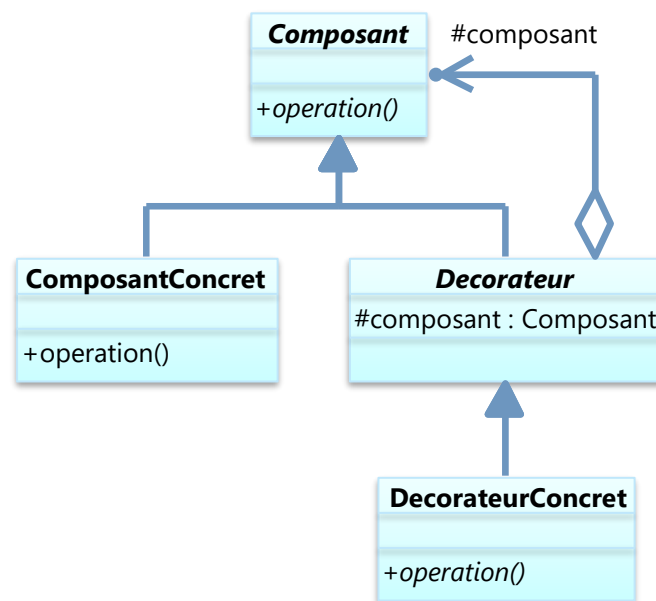
Lorsqu'il faudra calculer le coût de la boisson, alors la fonction **cout()** de l'objet **Chantilly** appellera la fonction **cout()** de l'objet **Lait** qui appellera la fonction **cout()** de l'objet **Deca**. A chaque fois la fonction **cout()** ajoutera son résultat au résultat précédent.

Le design pattern Décorateur permet de réaliser cela, en tirant partie de l'héritage et de l'agrégation. En effet dans l'exemple ci-dessus, les objets sont agrégés entre eux, mais le résultat doit rester une boisson. Il est temps maintenant de vous donner la définition du design pattern Décorateur.

Définition : Le design pattern Décorateur

Le design pattern Décorateur permet d'attacher dynamiquement des responsabilités supplémentaires à un objet. C'est une alternative souple à l'héritage pour proposer de nouvelles fonctionnalités.

Un dessin vaut mieux qu'un long discours, voici sa représentation UML :

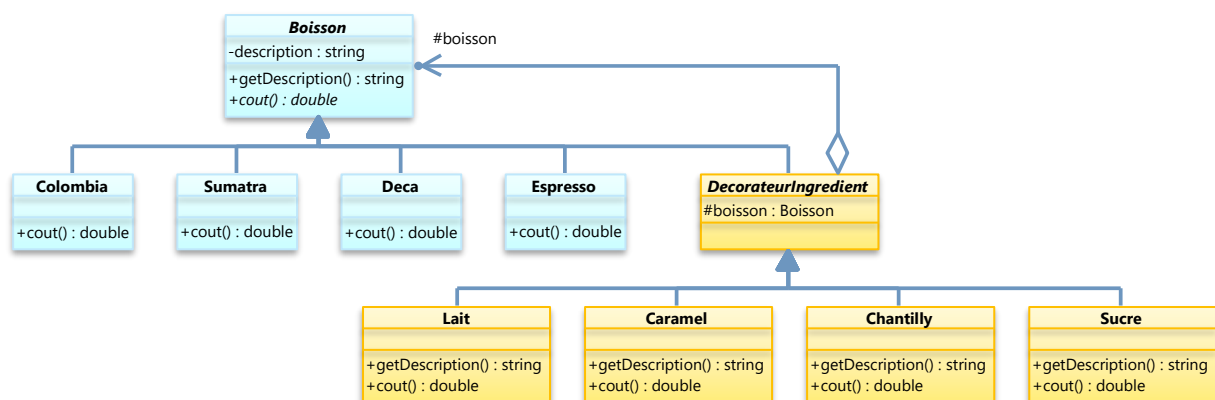


Dans le cas de la BAKoffee, les boissons sont les composants à décorer avec les ingrédients supplémentaires.

- Le design pattern Décorateur applique le principe Ouvert-Fermé, mais ce n'est pas le seul. Appliquer le principe Ouvert-Fermé peut passer par d'autres types d'implémentations. Ce pattern vous est présenté car il correspond au problème de la BAKoffee.
- Ce pattern applique-t-il le principe SRP – Single Responsibility Principle vu au dernier TD ?

Exercice 2. Appliquons le pattern Décorateur

Appliquez ce pattern au cas de la BAKoffee de l'exercice 1.



- Implémentez totalement en C++ un des décorateurs concrets
- Comment coderiez-vous la préparation complète de la boisson (l'appuie sur la touche Go du service marketing) ?
 - Attention, certains ingrédients doivent absolument être mis en début de préparation (le sucre par exemple) et d'autre à la fin (la Chantilly).

- ➡ Solution possible : La description n'est plus une chaîne de caractères, mais une liste de paires <int priorité, string description>. La fonction getDescription des décorateurs ajoute un élément à la liste contenue dans la description de la boisson. Une priorité élevée correspond à un ingrédient à mettre en début, une priorité basse pour un ingrédient en fin de préparation. La préparation se fera en récupérant la liste complète par getDescription de la boisson à préparer puis en la triant par priorité décroissante. Cette solution est implémentée dans le fichier zip.