Cours M3105 : Conception et programmation objet avancées
Le patron de conception Décorateur

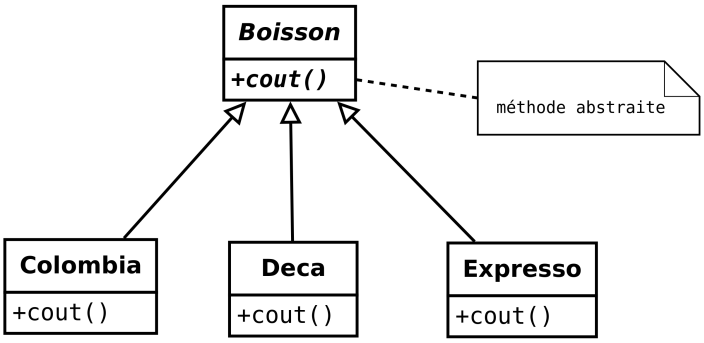
Références: cours de D. Bouthinon, livre *Design patterns — Tête la première*,
E. & E. Freeman, ed. O'Reilly

IUT Villetaneuse

2019-2020

Plan

Ajouter dynamiquement des responsabilités (options) à un objet

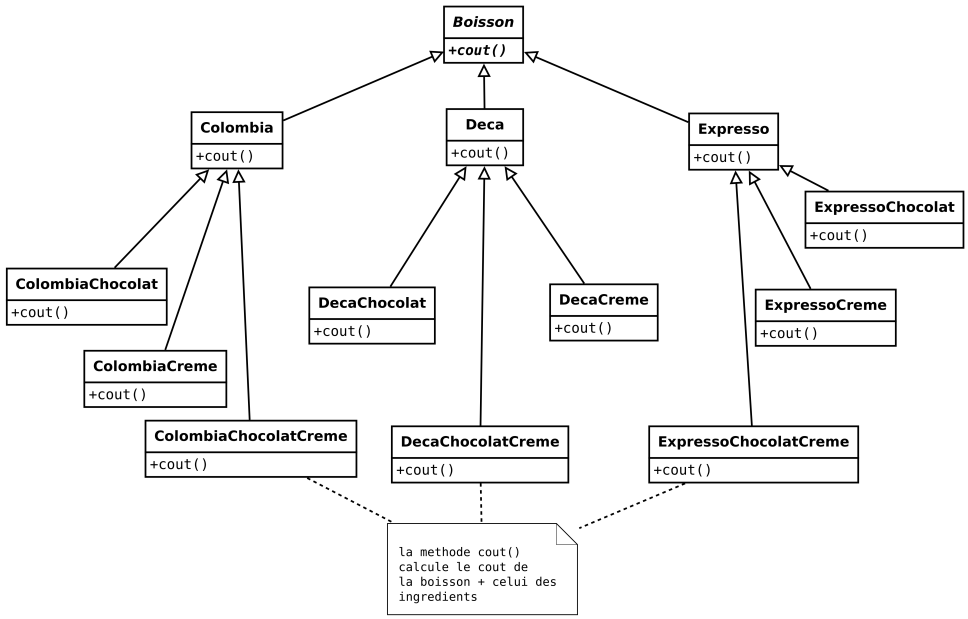


On veut pouvoir ajouter facilement et dynamiquement des ingrédients (chocolat, chantilly, crème) aux boissons.

Créer autant de classes que de combinaisons d'options

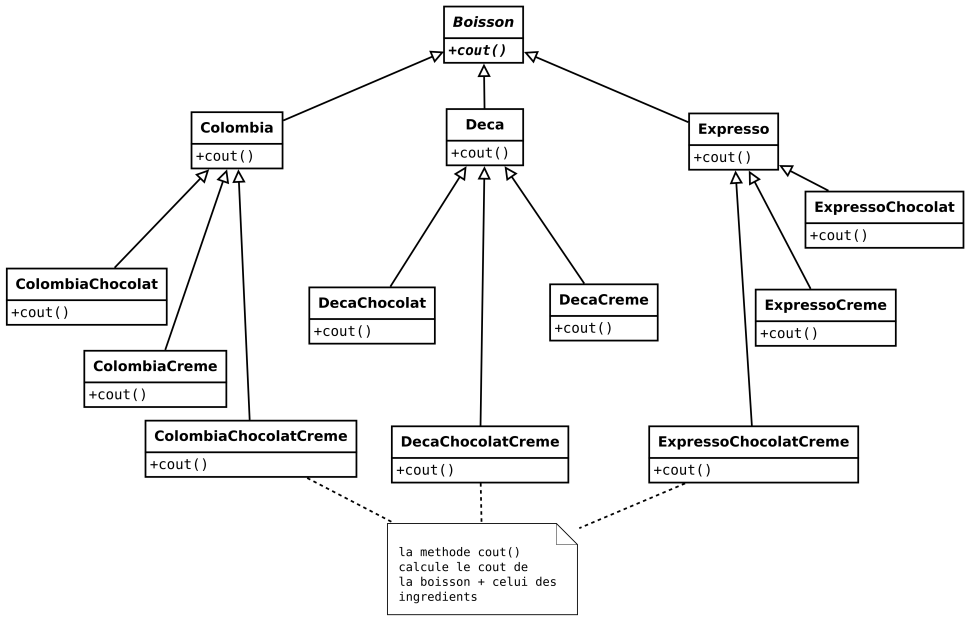
On veut pouvoir ajouter des ingrédients : (chocolat, chantilly, crème) aux boissons.

Créer autant de classes que de combinaisons d'options



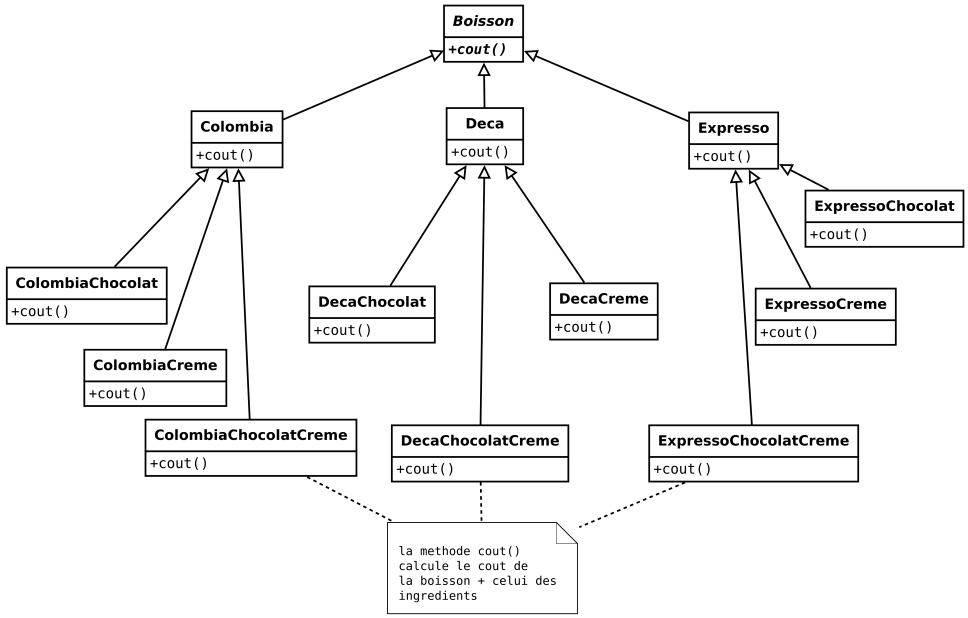
10 boissons et 10 ingrédients $\Rightarrow$ 10 230 classes.

Créer autant de classes que de combinaisons d'options



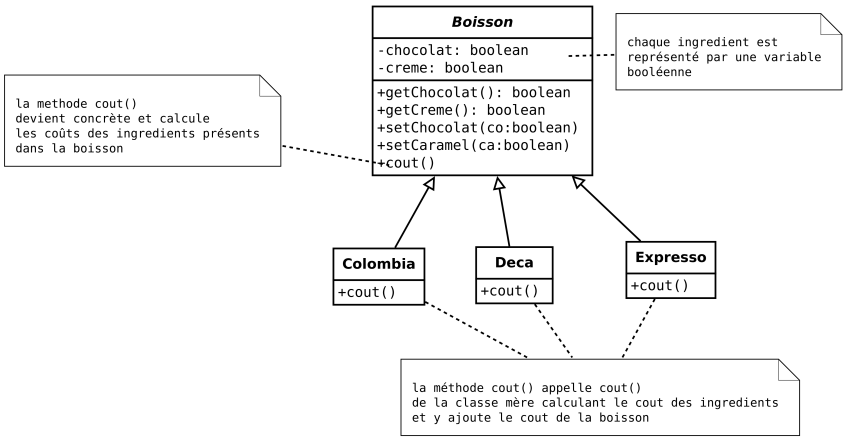
10 boissons et 10 ingrédients $\Rightarrow$ 10 230 classes.

Créer autant de classes que de combinaisons d'options

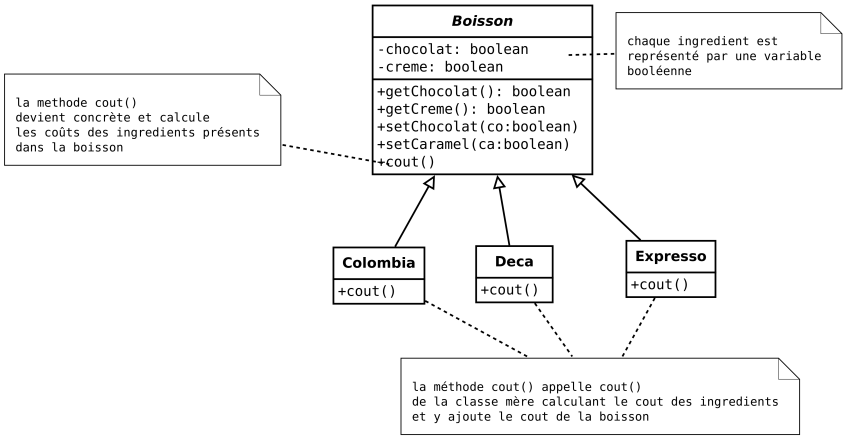


10 boissons et 10 ingrédients $\Rightarrow$ 10 230 classes.
Si le prix d'un ingrédient change, il faut modifier toutes les classes qui l'utilisent.

Créer des variables représentant les options



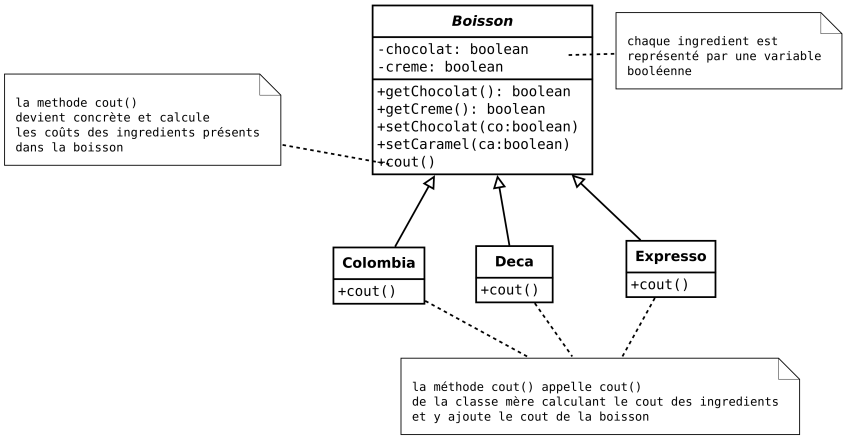
Créer des variables représentant les options



Nouveaux ingrédients ⇒ changer le code dans *Boisson*

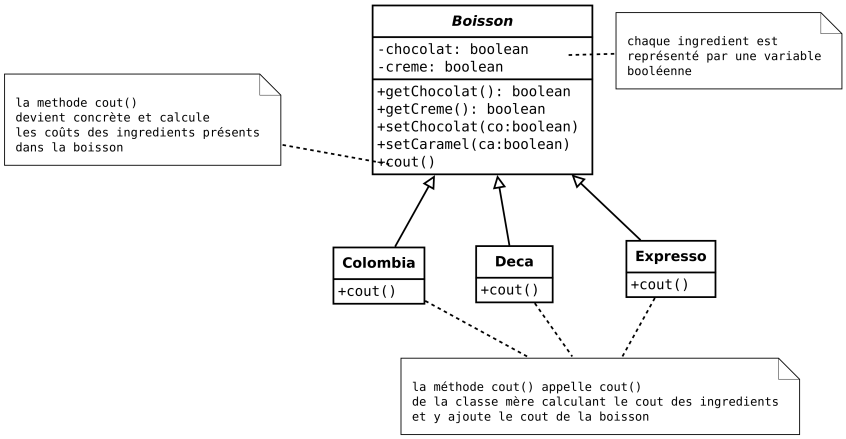
Motivations **Mauvaises conceptions** Une bonne conception Le design pattern Décorateur Principes de conception rencontrés

Créer des variables représentant les options



Nouveaux ingrédients ⇒ changer le code dans *Boisson*
Nouvelle boisson : *Thé* ⇒ l'ingrédient *Chocolat* n'est pas adapté, pourtant *Thé* hérite de *getChocolat()* et *setChocolat()*

Créer des variables représentant les options



Nouveaux ingrédients ⇒ changer le code dans *Boisson*
Nouvelle boisson : *Thé* ⇒ l'ingrédient *Chocolat* n'est pas adapté, pourtant *Thé* hérite de *getChocolat()* et *setChocolat()*
Double ration de crème ⇒ ne peut pas être représentée

Utiliser le principe d'ouverture-fermeture

Principe *open-closed* de la programmation SOLID

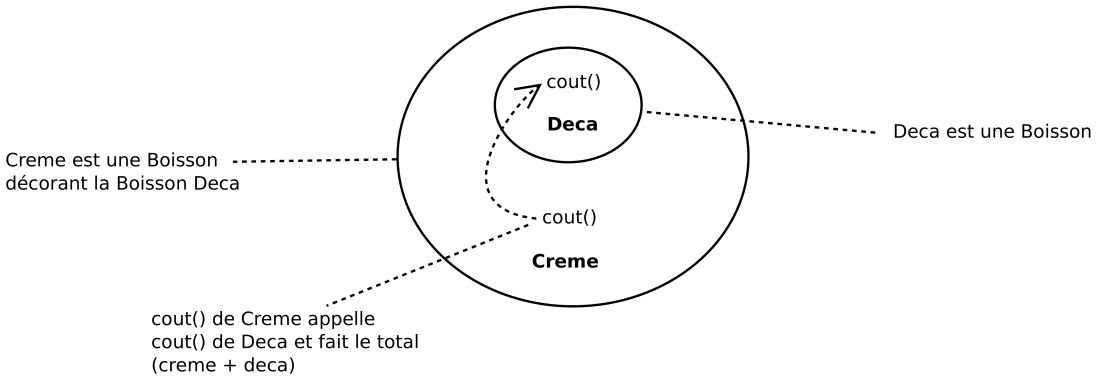
Les classes doivent être ouvertes à l'extension mais fermées à la modification.

On doit pouvoir étendre facilement une classe pour y ajouter de nouveaux comportements (ouverture) sans toucher à ce qui ne change pas (fermeture).

On ne touche pas au code (correctement testé) d'une classe (fermeture).
Les comportements ajoutés sont placés dans d'autres classes (ouverture).

Une boisson avec un décorateur

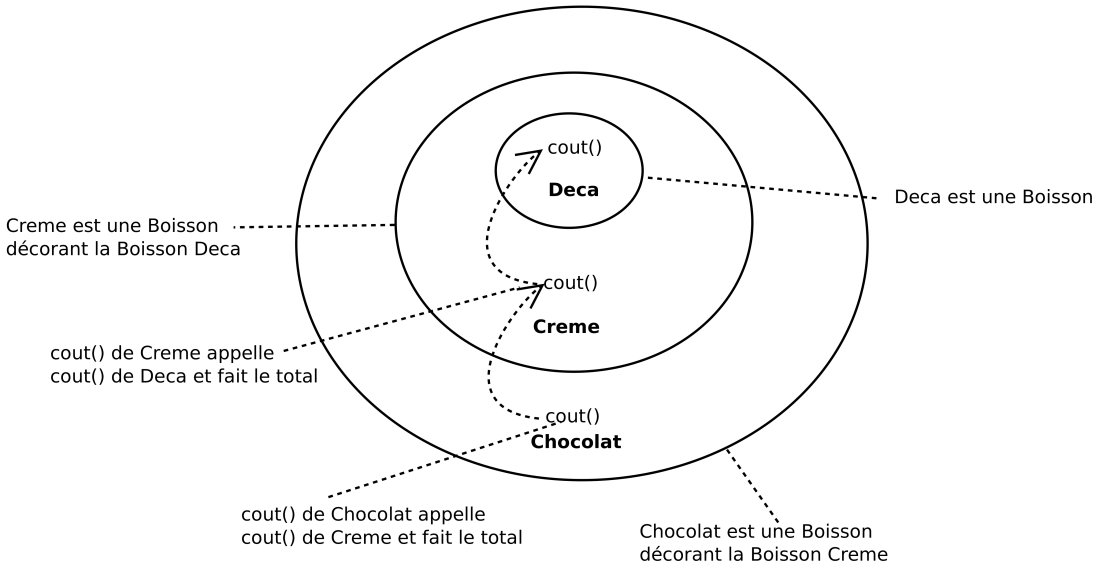
Modéliser un *Deca* à la *Creme* : chaque ingrédient (décorateur) est une *Boisson* qui encapsule une *Boisson*.



Le décorateur ajoute son comportement (calcul du coût de la crème) avant ou après avoir délégué le travail (calcul du coût du déca) à l'objet qu'il décore.

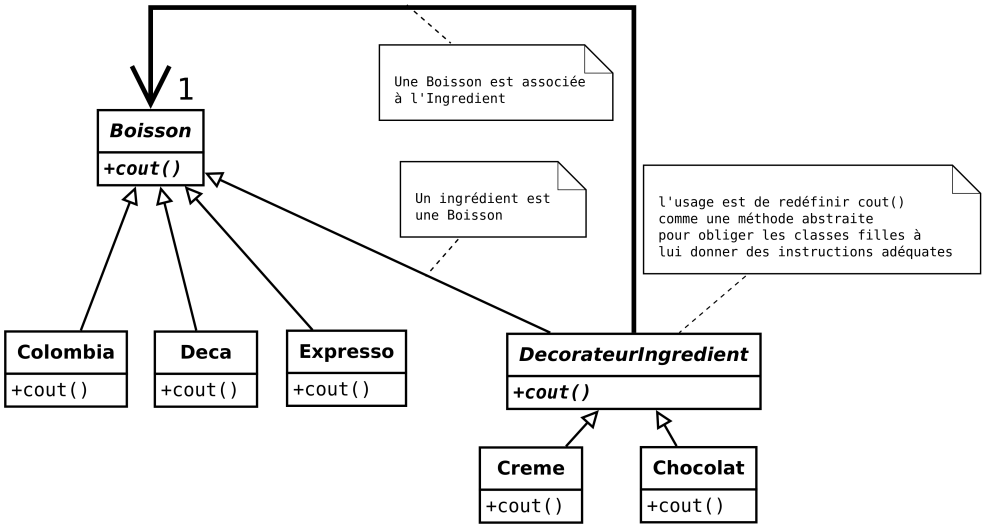
Une boisson avec plusieurs décorateurs

Modéliser un *Deca* à la *Creme* et au *Chocolat*.



Chaque décorateur est une *Boisson* qui encapsule une *Boisson* (qui peut être un décorateur).

Conception avec des décorateurs



Chaque décorateur est une *Boisson* qui encapsule une *Boisson* : double lien d'héritage et d'association entre *Decorateur* et *Boisson*.

Représentation java des boissons

```
public abstract class Boisson
{
    public Boisson() {}

    public abstract int cout();
}
```

```
public class Deca extends Boisson
{
    public Deca()
    { super(); }

    public int cout()
    { return 2; }
}
```

```
public class Colombia extends Boisson
{
    public Colombia()
    { super(); }

    public int cout()
    { return 3; }
}
```

Représentation java des décorateurs

```
public abstract class DecorateurIngredient extends Boisson
{
    private Boisson maBoisson;

    public DecorateurIngredient(Boisson uneBoisson)
    {
        super();
        this.maBoisson = uneBoisson;
    }

    public Boisson getMaBoisson()
    { return this.maBoisson; }

    public abstract int cout();
}
```

Boisson à décorer

Redéfinition de *cout()* en méthode abstraite : oblige les classes filles à redéfinir concrètement *cout()*.

```
public class Chocolat extends DecorateurIngredient
{
    public Chocolat(Boisson uneBoisson)
    { super(uneBoisson); }

    public int cout()
    { return 1 + this.getMaBoisson().cout(); }
}
```

Redéfinition concrète de *cout()*. On tient compte du coût de l'ingrédient et de la boisson décorée.

Tester les décorateurs

```
public class TestBoissons
{
    public static void main()
    {
        Boisson unDeca = new Deca();

        System.out.println(unDeca.cout());

        Boisson unDecaChoco = new Chocolat(unDeca);

        System.out.println(unDecaChoco.cout());

        Boisson un DecaChocoCreme = new Creme(unDecaChoco);

        System.out.println(unDecaChocoCreme.cout());
    }
}
```

2

$3 = 2 + 1$

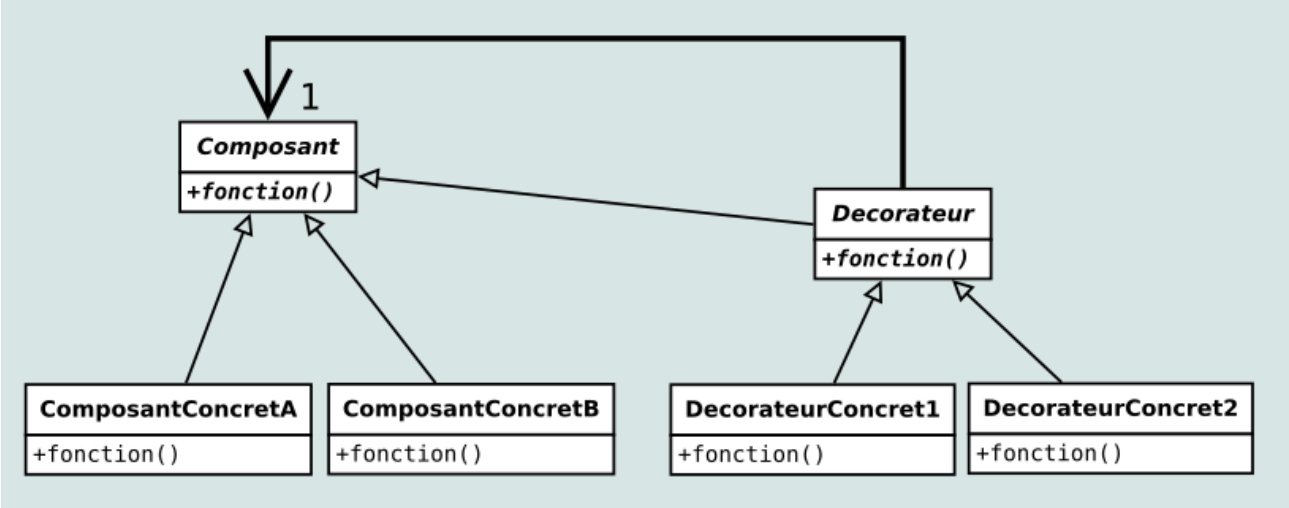
$4 = 2 + 1 + 1$

Définition

Décorateur

Le design pattern Décorateur attache dynamiquement des responsabilités supplémentaires à un objet.
Il fournit une alternative souple à l'héritage pour étendre les fonctionnalités d'un objet.

Structure du design pattern



Le *Décorateur* étend/modifie le *Composant*.
Un *Décorateur* peut être ajouté/retiré/modifié dynamiquement.

Principes généraux mis en œuvre

Les classes doivent être ouvertes à l'extension mais fermées à la modification

On doit pouvoir étendre facilement une classe pour y ajouter de nouveaux comportements (ouverture) sans changer l'existant (fermeture).

Préférer la composition à l'héritage

L'héritage manque de souplesse et impose ce qui est défini dans la classe mère à sa descendance. La composition permet de séparer les comportements.
Le décorateur fait en fait appel à la fois à la composition et à l'héritage.