



ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT
TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

المدرسة العليا لأساتذة التعليم التقني المحمدية

Design Patterns

2^{ème} année Cycle Ingénieur

GLSID 2, ICCN 2 & IIBDCC 2

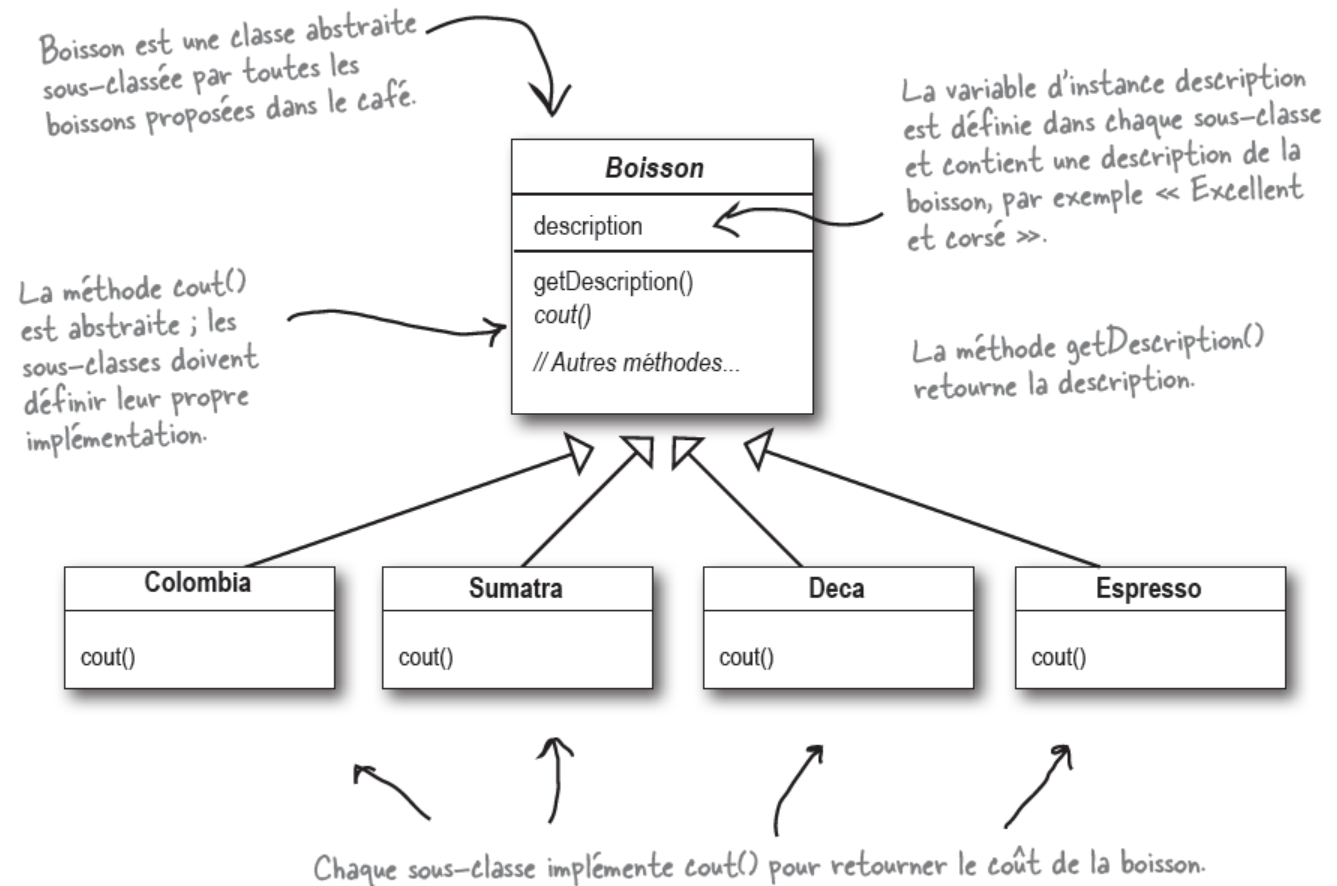
Pr. SARA RETAL



Problèmes de conception



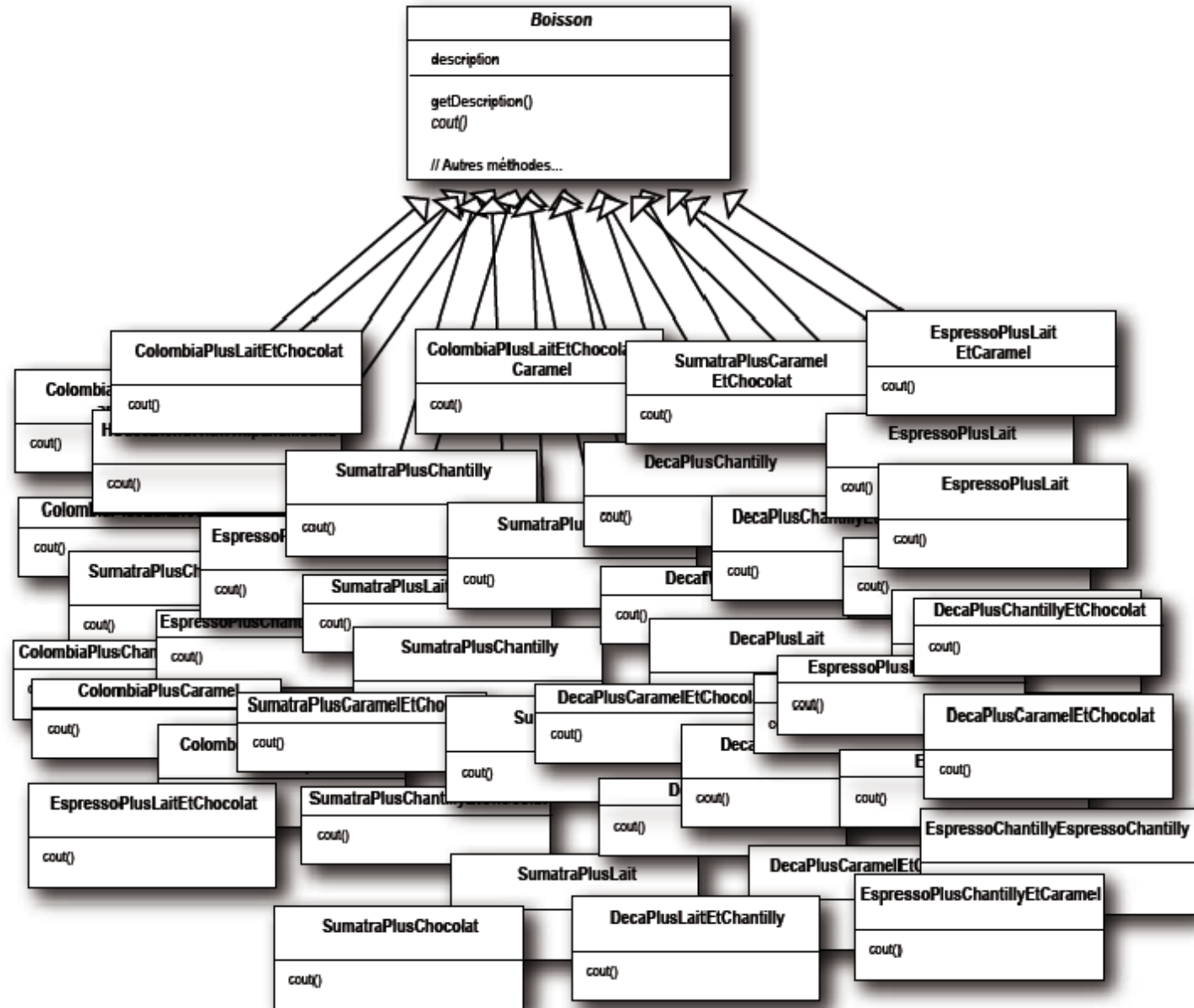
Exemple Starbuzz Coffee



Problèmes de conception



Exemple Starbuzz Coffee



Problèmes de conception



Exemple Starbuzz Coffee

- Que se passe-t-il quand le prix du lait augmente ? Que font-ils quand ils proposent un nouveau supplément de vanille ?
- Au-delà du problème de maintenance, ils enfreignent l'un des principes de conception que nous avons déjà vus. Lequel ?

Problèmes de conception



Exemple Starbuzz Coffee

<i>Boisson</i>
description lait caramel chocolat chantilly
getDescription() cout() aLait() setLait() aCaramel() setCaramel() aChocolat() setChocolat() aChantilly() setChantilly() // Autres méthodes...

Nouvelles valeurs booléennes pour chaque ingrédient.

Maintenant, nous implémentons `cout()` dans `Boisson` (au lieu qu'elle demeure abstraite), pour qu'elle puisse calculer les coûts associés aux ingrédients pour une instance de boisson donnée. Les sous-classes redéfiniront toujours `cout()`, mais elles appelleront également la super-version pour pouvoir calculer le coût total de la boisson de base plus celui des suppléments.

Ces méthodes lisent et modifient les valeurs booléennes des ingrédients.

Problèmes de conception

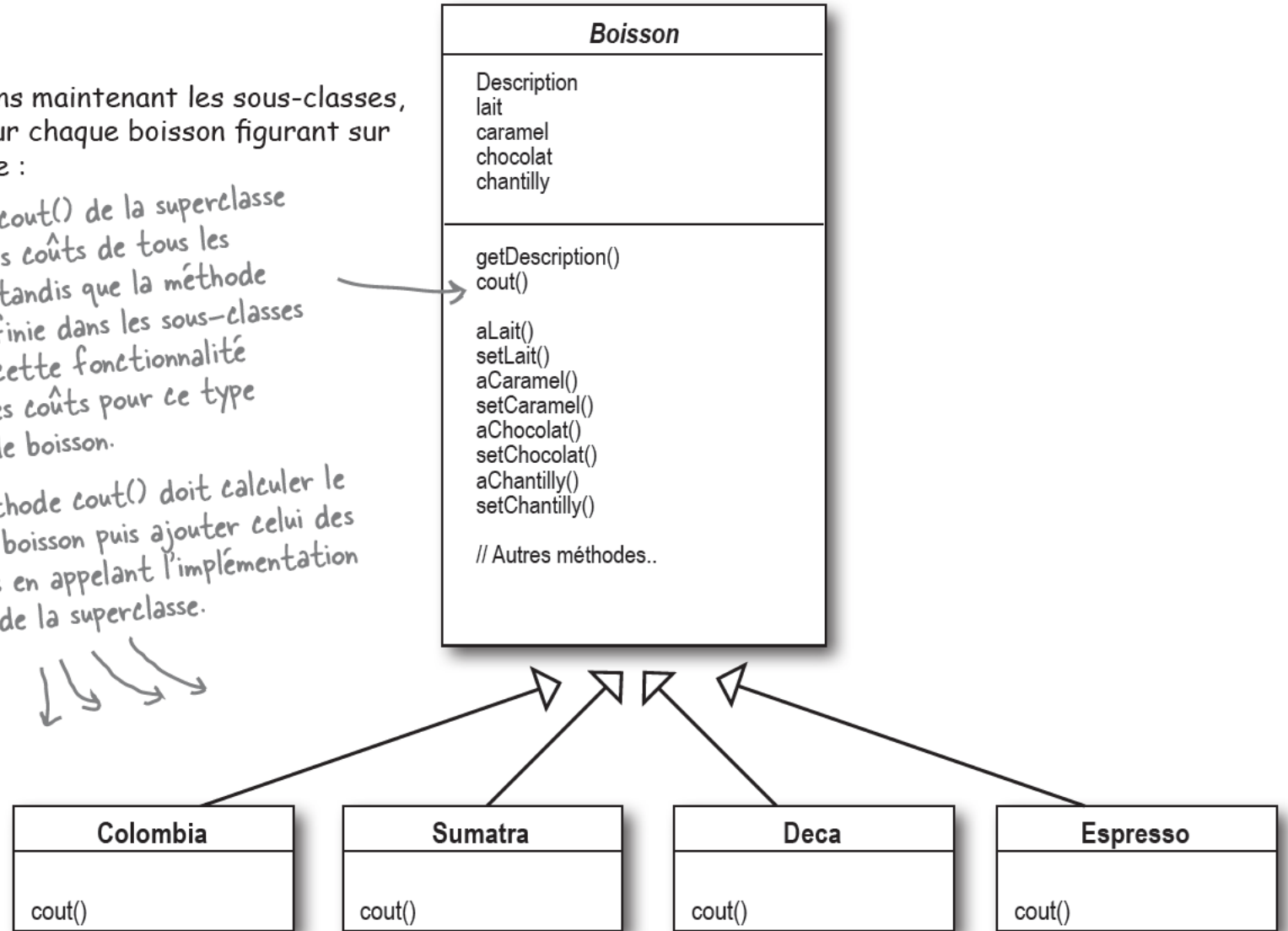


Exemple Starbuzz Coffee

Ajoutons maintenant les sous-classes, une pour chaque boisson figurant sur la carte :

La méthode `cout()` de la superclasse va calculer les coûts de tous les ingrédients, tandis que la méthode `cout()` redéfinie dans les sous-classes va étendre cette fonctionnalité et inclure les coûts pour ce type spécifique de boisson.

Chaque méthode `cout()` doit calculer le coût de la boisson puis ajouter celui des ingrédients en appelant l'implémentation de `cout()` de la superclasse.



Problèmes de conception



Exemple Starbuzz Coffee

Quelles sont les exigences ou les autres facteurs qui pourraient changer et avoir un impact sur cette conception ?

Principe de conception

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

Principe de conception

- *Notre but est de permettre d'étendre facilement les classes pour incorporer de nouveaux comportements sans modifier le code existant.*

- *Qu'obtenons-nous si nous y parvenons ?*

Des conceptions résistantes au changement et suffisamment souples pour accepter de nouvelles fonctionnalités répondant à l'évolution des besoins.

Problèmes de conception



Exemple Starbuzz Coffee

Nous allons commencer par une boisson et nous allons la « décorer » avec des ingrédients au moment de l'exécution.

Si par exemple le client veut un Sumatra avec Chocolat et Chantilly

Nous allons ?

Problèmes de conception



Exemple
Starbuzz Coffee

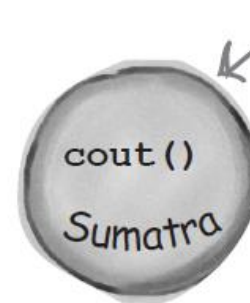
- ➊ **Prendre un objet Sumatra.**
- ➋ **Le décorer avec un objet Chocolat.**
- ➌ **Le décorer avec un objet Chantilly.**
- ➍ **Appeler la méthode `cout()` et nous appuyer sur la délégation pour ajouter les coûts des ingrédients.**

Problèmes de conception



Exemple Starbuzz Coffee

1 Commençons par notre objet Sumatra.



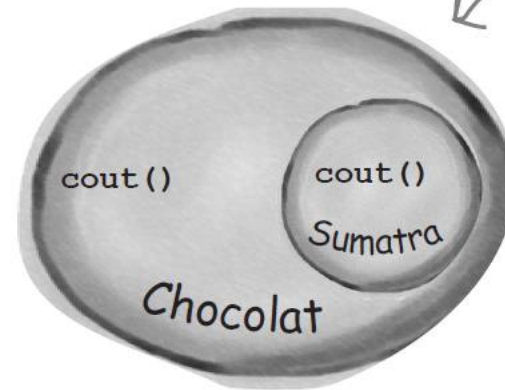
Souvenez-vous que Sumatra hérite de Boisson et a une méthode `cout()` qui calcule le prix de la boisson.

Problèmes de conception



Exemple Starbuzz Coffee

- ② **Le client veut du chocolat. Nous créons donc un objet Chocolat et nous enveloppons le Sumatra dedans.**



L'objet Chocolat est un décorateur. Son type reflète l'objet qu'il décore, en l'occurrence une Boisson. (Par « reflète », nous voulons dire qu'il est du même type.)

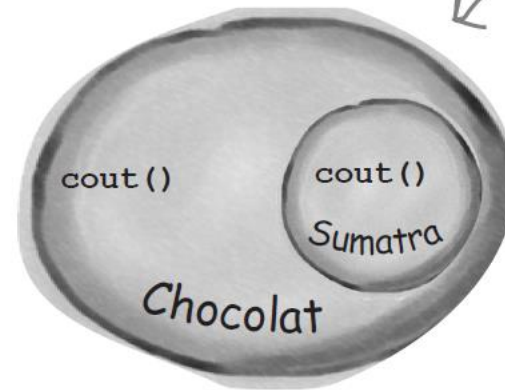
Chocolat a donc également une méthode `cout()`. Grâce au polymorphisme, nous pouvons traiter n'importe quelle Boisson enveloppée de Chocolat comme une Boisson (parce que Chocolat est un sous-type de Boisson).

Problèmes de conception



Exemple Starbuzz Coffee

- ② **Le client veut du chocolat. Nous créons donc un objet Chocolat et nous enveloppons le Sumatra dedans.**



L'objet Chocolat est un décorateur. Son type reflète l'objet qu'il décore, en l'occurrence une Boisson. (Par « reflète », nous voulons dire qu'il est du même type.)

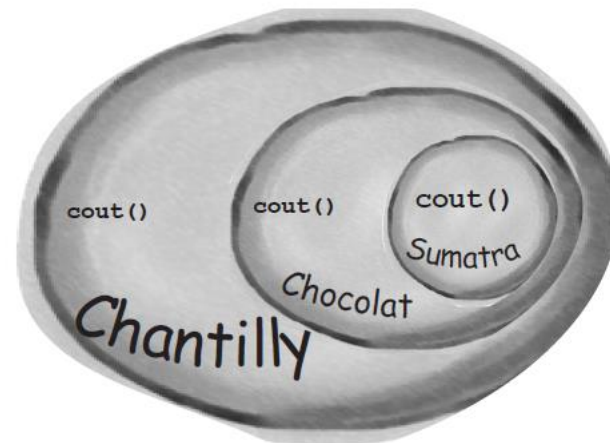
Chocolat a donc également une méthode `cout()`. Grâce au polymorphisme, nous pouvons traiter n'importe quelle Boisson enveloppée de Chocolat comme une Boisson (parce que Chocolat est un sous-type de Boisson).

Problèmes de conception



Exemple Starbuzz Coffee

- ③ **Le client veut aussi de la Chantilly. Nous créons un décorateur Chantilly et nous enveloppons Chocolat dedans.**



Chantilly est un décorateur. Il reflète également le type de Sumatra et possède une méthode `cout()`.

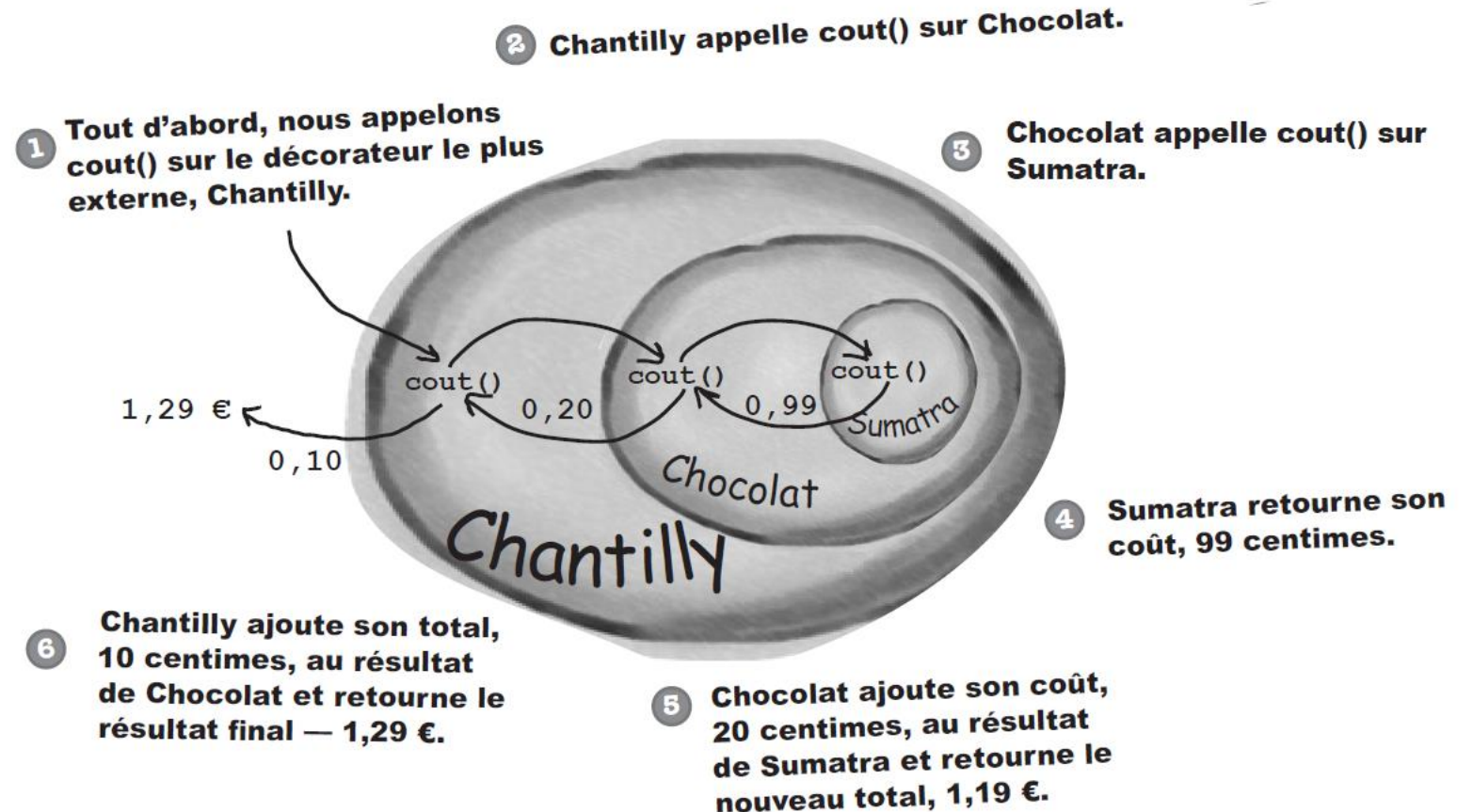
Ainsi, un Sumatra enveloppé de Chocolat et de Chantilly est toujours une Boisson. Nous pouvons lui faire tout ce que nous ferions avec un Sumatra, notamment appeler sa méthode `cout()`.

Problèmes de conception



Exemple Starbuzz Coffee

- ④ Il est temps de calculer le coût pour le client. Pour ce faire, nous appelons `cout()` sur le décorateur le plus externe, Chantilly, et Chantilly va déléguer le calcul du coût à l'objet qu'il décore. Une fois qu'il aura le coût, il ajoutera le coût de Chantilly.



Problèmes de conception



Exemple Starbuzz Coffee

- Les décorateurs ont le même supertype que les objets qu'ils décorent.
- Vous pouvez utiliser un ou plusieurs décorateurs pour envelopper un objet.
- Comme le décorateur a le même supertype que l'objet qu'il décore, nous pouvons transmettre un objet décoré à la place de l'objet original (enveloppé).
- Le décorateur ajoute son propre comportement soit avant soit après avoir délégué le reste du travail à l'objet qu'il décore.
- Les objets pouvant être décorés à tout moment, nous pouvons les décorer dynamiquement au moment de l'exécution avec autant de décorateurs que nous en avons envie.

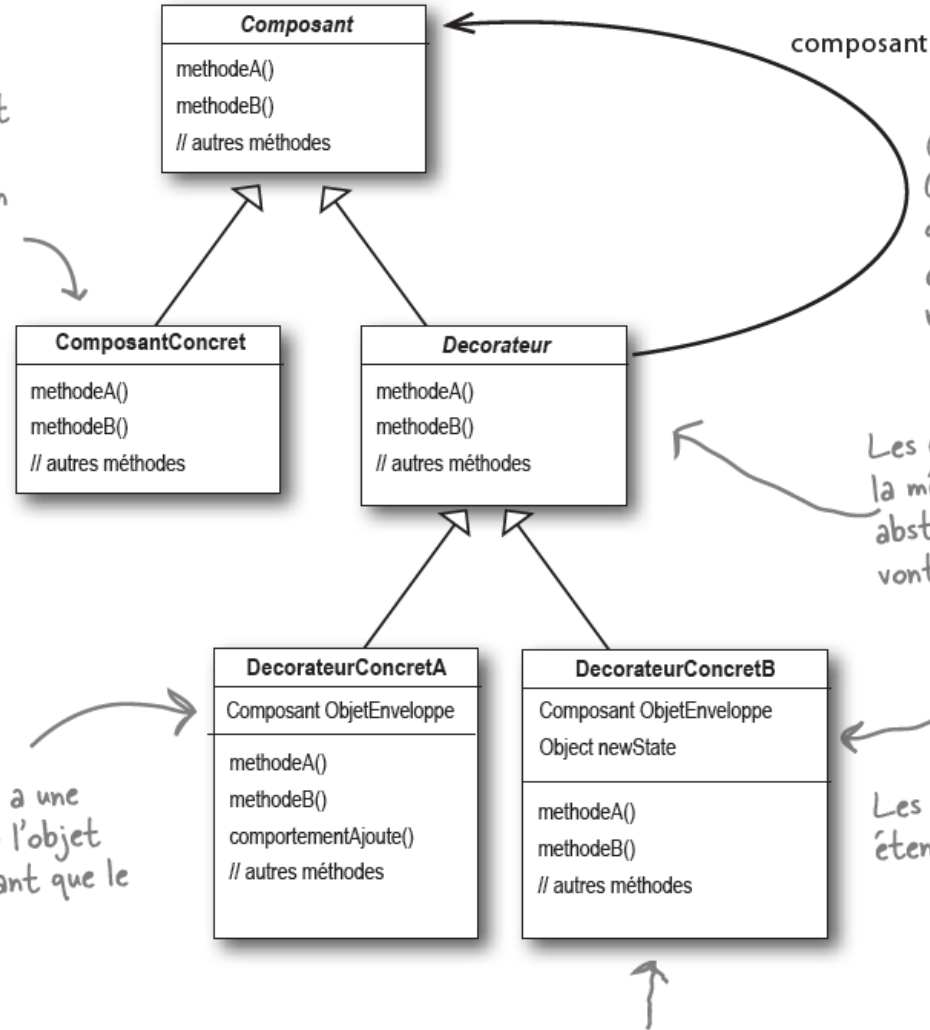
Le pattern Décorateur

Le pattern Décorateur attache dynamiquement des responsabilités supplémentaires à un objet. Il fournit une alternative souple à la dérivation, pour étendre les fonctionnalités.

Le pattern Décorateur

Le ComposantConcret est l'objet auquel nous allons ajouter dynamiquement un nouveau comportement. Il dérive de Composant.

Le DecorateurConcret a une variable d'instance pour l'objet qu'il décore (le Composant que le Décorateur enveloppe).



Chaque composant peut être utilisé seul ou enveloppé par un décorateur.

Chaque décorateur A-UN (enveloppe) un composant, ce qui signifie qu'il a une variable d'instance qui contient une référence à un composant.

Les décorateurs implémentent la même interface ou classe abstraite que le composant qu'ils vont décorer.

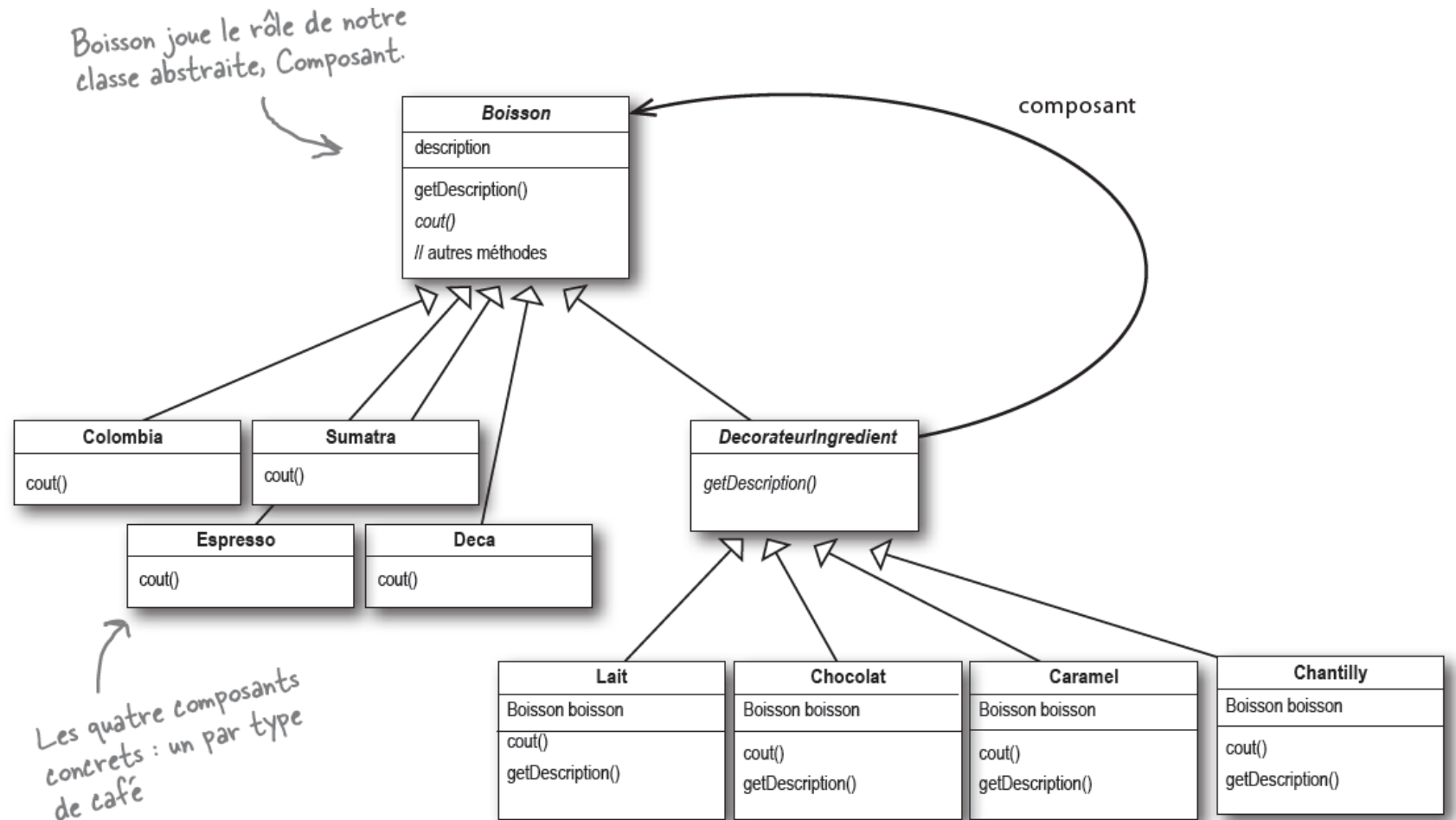
Les décorateurs peuvent étendre l'état du composant.

Les Décorateurs peuvent ajouter de nouvelles méthodes, mais on ajoute généralement le nouveau comportement en effectuant un traitement avant ou après une méthode existant dans le composant.

Problèmes de conception



Exemple Starbuzz Coffee



héritage vs. composition

- Le DecorateurIngredient étend toujours la classe Boisson. C'est de l'héritage, ça, non ?
- Si on n'a besoin de rien d'autre que d'hériter le type du composant, Pourquoi ne pas utiliser une interface au lieu d'une classe abstraite pour la classe Boisson ?

Le code de Starbuzz

```
public abstract class Boisson {  
    String description = "Boisson inconnue";  
    public String getDescription() {  
        return description;  
    }  
    public abstract double cout();  
}
```

Boisson est une classe abstraite qui possède deux méthodes : `getDescription()` et `cout()`.

`getDescription` a déjà été implémentée pour nous, mais nous devons implémenter `cout()` dans les sous-classes.

Le code de Starbuzz

```
public abstract class DecorateurIngredient extends Boisson {  
    public abstract String getDescription();  
}
```

D'abord, comme elle doit être interchangeable avec une Boisson, nous étendons la classe Boisson.

Nous allons aussi faire en sorte que les ingrédients (décorateurs) réimplémentent tous la méthode getDescription(). Nous allons aussi voir cela dans une seconde...

Le code de Starbuzz

```
public class Espresso extends Boisson {  
    public Espresso() {  
        description = "Espresso";  
    }  
    public double cout() {  
        return 1.99;  
    }  
}
```

Nous étendons d'abord la classe Boisson, puisqu'il s'agit d'une boisson.

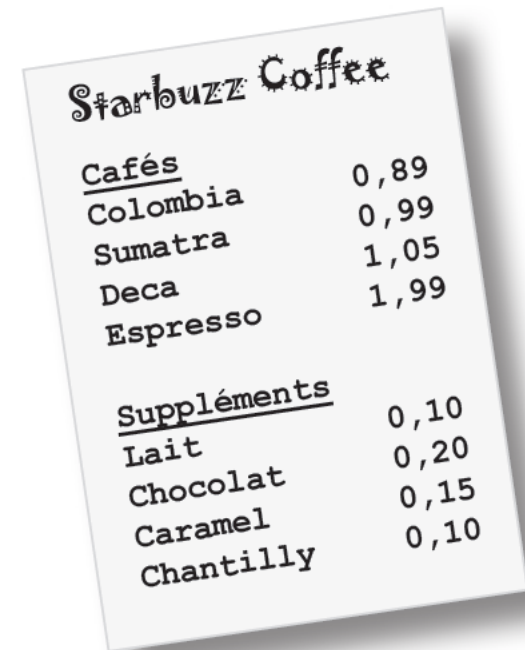
Nous gérons la description dans le constructeur de la classe. Souvenez-vous que la variable d'instance description est héritée de Boisson.

Enfin, nous calculons le coût d'un Espresso. Maintenant que nous n'avons plus besoin d'ajouter les ingrédients dans cette classe, il suffit de retourner le prix d'un Espresso : 1,99 €.

Le code de Starbuzz

```
public class Colombia extends Boisson {  
    public Colombia() {  
        description = "Pur Colombia";  
    }  
  
    public double cout() {  
        return .89;  
    }  
}
```

Bien. Voici une autre Boisson. Il suffit de spécifier la description appropriée, << Pur Colombia >>, puis de retourner le prix correct : 89 centimes.

A tilted image of a Starbuzz Coffee menu. It lists various coffee drinks and their prices, categorized into 'Cafés' and 'Suppléments'.

Starbuzz Coffee	
<u>Cafés</u>	
Colombia	0,89
Sumatra	0,99
Deca	1,05
Espresso	1,99
<u>Suppléments</u>	
Lait	0,10
Chocolat	0,20
Caramel	0,15
Chantilly	0,10

Le code de Starbuzz

Chocolat est un décorateur : nous étendons DecorateurIngredient.

Souvenez-vous que DecorateurIngredient étend Boisson.

Nous allons instancier Chocolat avec une référence à une Boisson en utilisant :

```
public class Chocolat extends DecorateurIngredient {  
    Boisson boisson;  
  
    public Chocolat(Boisson boisson) {  
        this.boisson = boisson;  
    }  
  
    public String getDescription() {  
        return boisson.getDescription() + ", Chocolat";  
    }  
  
    public double cout() {  
        return .20 + boisson.cout();  
    }  
}
```

(1) Une variable d'instance pour contenir la boisson que nous enveloppons.

(2) Un moyen pour affecter à cette variable d'instance l'objet que nous enveloppons. Ici, nous allons transmettre la boisson que nous enveloppons au constructeur du décorateur.

La description ne doit pas comprendre seulement la boisson – disons « Sumatra » – mais aussi chaque ingrédient qui décore la boisson, par exemple, « Sumatra, Chocolat ». Nous allons donc déléguer à l'objet que nous décorons pour l'obtenir, puis ajouter « Chocolat » à la fin de cette description.

Nous devons maintenant calculer le coût de notre boisson avec Chocolat. Nous déléguons d'abord l'appel à l'objet que nous décorons pour qu'il calcule son coût. Puis nous ajoutons le coût de Chocolat au résultat.

Le code de Starbuzz

```
public class StarbuzzCoffee {
```

```
    public static void main(String args[]) {
```

```
        Boisson boisson = new Espresso();
```

```
        System.out.println(boisson.getDescription()
            + " €" + boisson.cout());
```

```
        Boisson boisson2 = new Sumatra();
```

```
        boisson2 = new Chocolat(boisson2);
```

```
        boisson2 = new Chocolat(boisson2);
```

```
        boisson2 = new Chantilly(boisson2);
```

```
        System.out.println(boisson2.getDescription()
            + " €" + boisson2.cout());
```

```
        Boisson boisson3 = new Colombia();
```

```
        boisson3 = new Caramel(boisson3);
```

```
        boisson3 = new Chocolat(boisson3);
```

```
        boisson3 = new Chantilly(boisson3);
```

```
        System.out.println(boisson3.getDescription()
            + " €" + boisson3.cout());
```

```
    }
```

```
}
```

Commander un espresso, pas d'ingrédients et afficher sa description et son coût.

Créer un objet Sumatra.

L'envelopper dans un Chocolat.

L'envelopper dans un second Chocolat.

L'envelopper de Chantilly.

Enfin nous servir un Colombia avec Caramel, Chocolat et Chantilly.

Question

- Starbuzz ont ajouté des tailles à leur carte. Vous pouvez maintenant commander un café en taille normale, grande et petite (traduction : petit, moyen et grand).
- Comme Starbuzz a vu cela comme une partie intrinsèque de la classe Café, ils ont ajouté deux méthodes à la classe Boisson : `setTaille()` et `getTaille()`. Ils voudraient également que le prix des ingrédients varie selon la taille. Par exemple, le Caramel coûterait respectivement 10, 15 et 20 centimes pour un petit café, un moyen et un grand.

Question

```
public class Caramel extends DecorateurIngredient {
    Boisson boisson;

    public Caramel(Boisson boisson) {
        this.boisson = boisson;
    }

    public int getTaille() {
        return boisson.getTaille();
    }

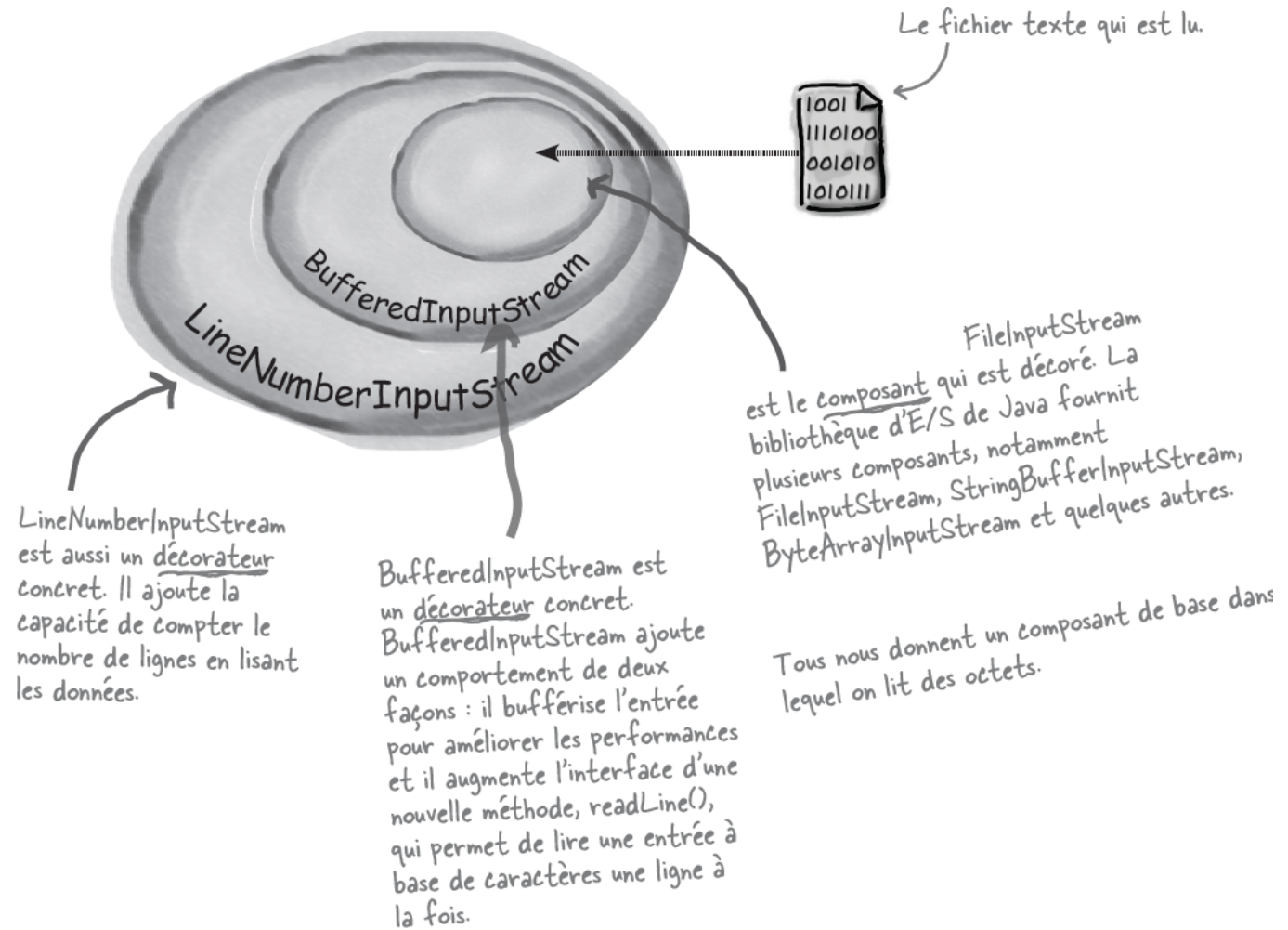
    public String getDescription() {
        return boisson.getDescription() + ", Caramel";
    }

    public double cout() {
        double cout = boisson.cout();
        if (getTaille() == Boisson.NORMAL) {
            cout += .10;
        } else if (getTaille() == Boisson.GRANDE) {
            cout += .15;
        } else if (getTaille() == Boisson.VENTI) {
            cout += .20;
        }
        return cout;
    }
}
```

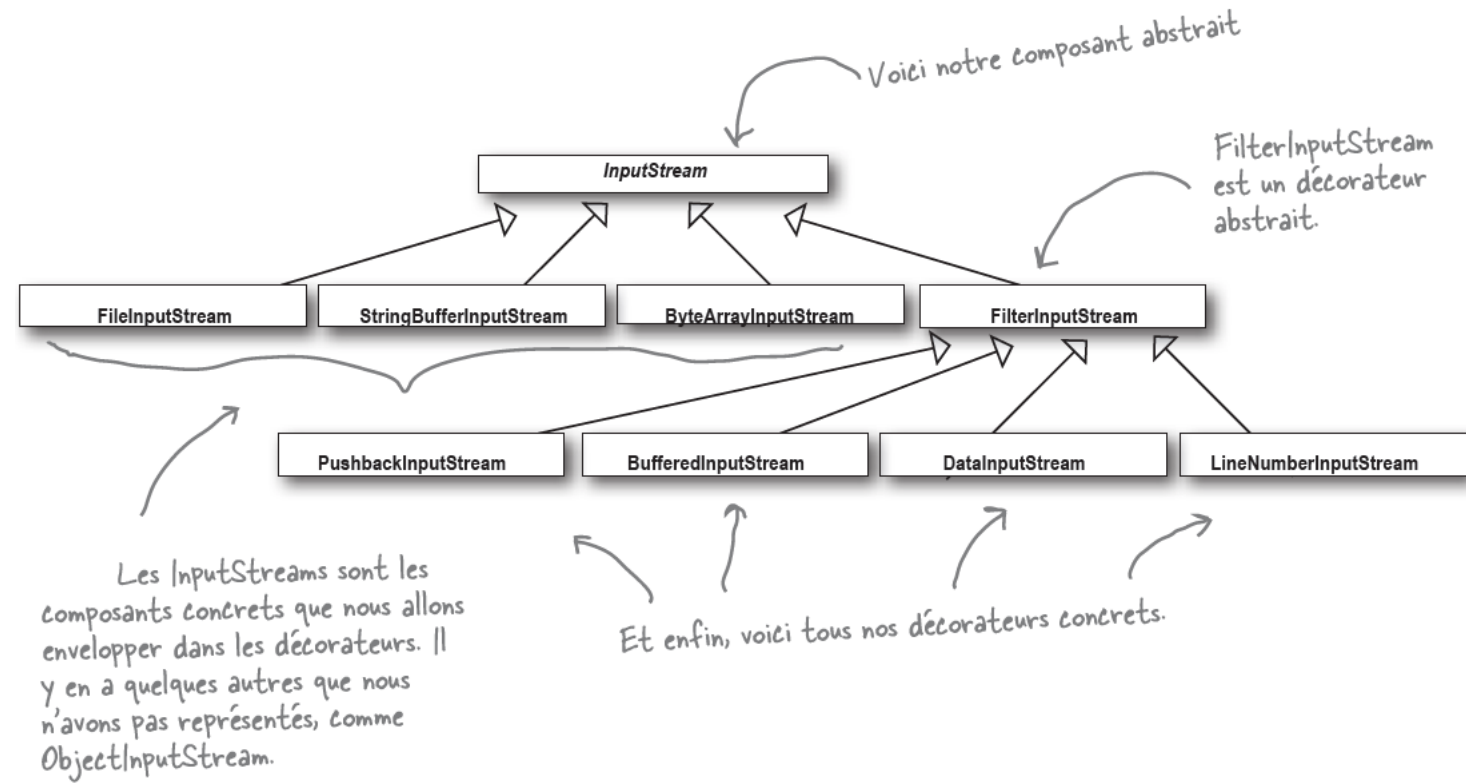
Nous devons maintenant propager la méthode `getTaille()` à la boisson enveloppée. Nous devons également transférer cette méthode dans la classe abstraite puisqu'elle sera utilisée dans tous les ingrédients décorateurs.

Ici nous obtenons la taille (qui se propage tout du long jusqu'à la boisson concrète) et nous ajoutons le coût approprié.

Les E/S Java



Les E/S Java



Exercice : Personnalisation d'armure dans un jeu vidéo

Contexte :

Vous êtes chargé de développer un module pour un jeu vidéo dans lequel un personnage peut porter une armure de base. Cette armure peut être améliorée progressivement avec différentes extensions (ex : bouclier supplémentaire, plaque de résistance au feu, camouflage, amélioration anti-projectiles, etc.). Chaque amélioration ajoute un bonus spécifique (comme une meilleure résistance ou une nouvelle capacité) ainsi qu'un coût supplémentaire en termes de poids de l'armure ou de ressources.

Objectif :

Appliquer le design pattern Décorateur pour modéliser le système d'extensions de l'armure. Le système doit permettre d'ajouter dynamiquement des fonctionnalités sans modifier la classe de base. Afficher les propriétés finales d'une armure, comme : Description complète de l'armure Résistance totale Coût total en poids