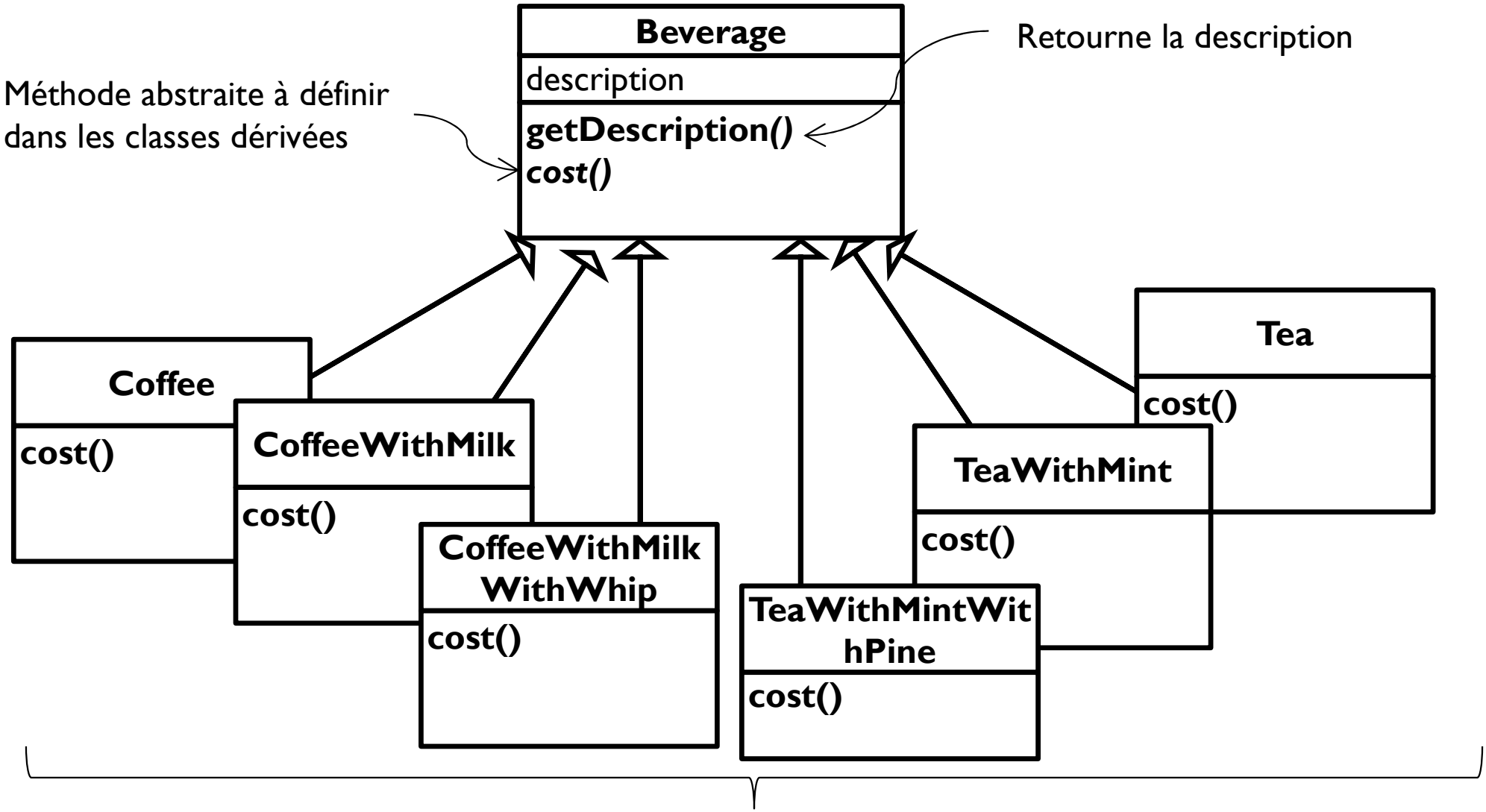
Le patron "Decorator"

StarCoffee : Spécification (1/10)

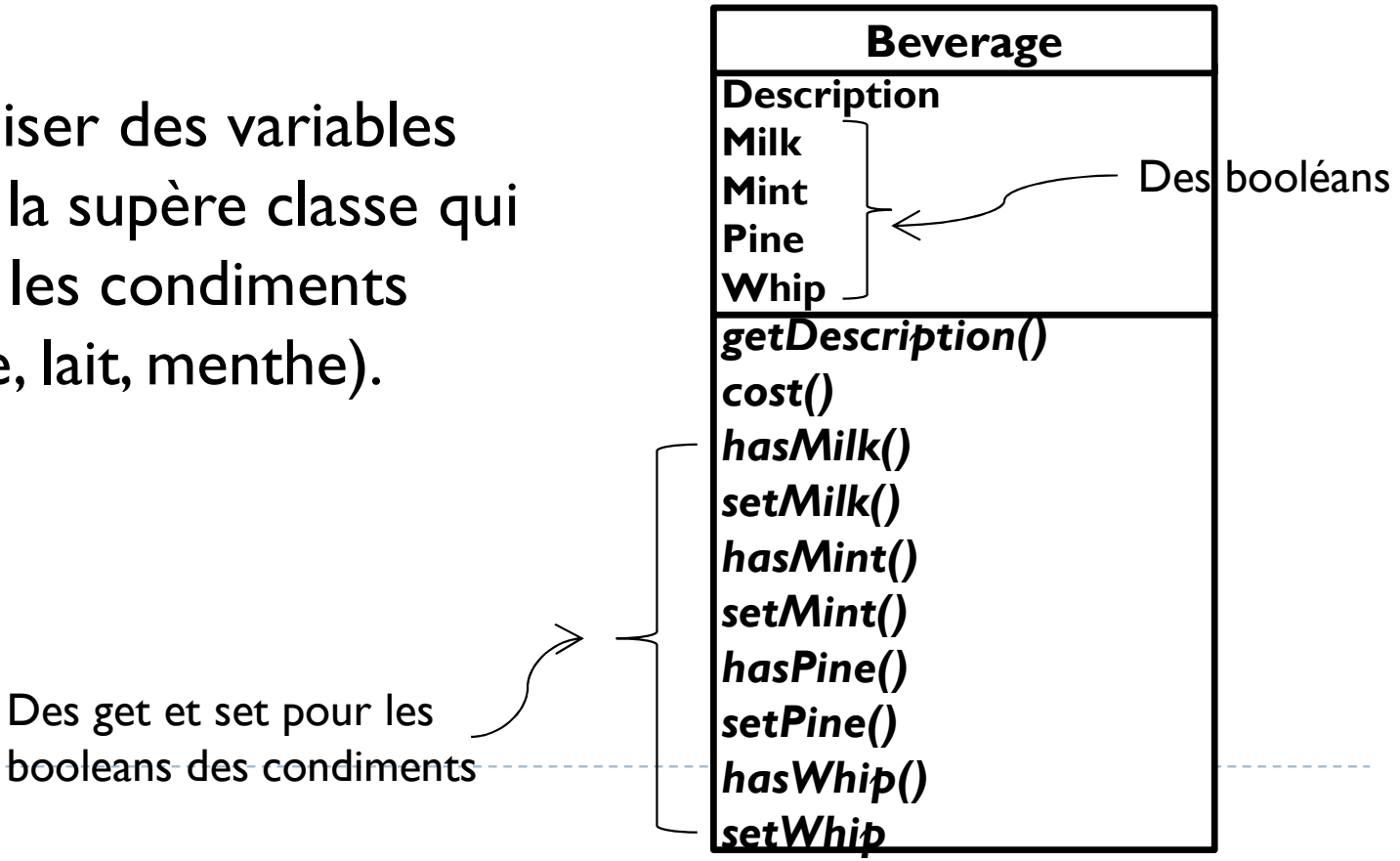
- Objectif: Mettre en œuvre un système de gestion des offres de boisson pour la clientèle de StarCoffee
- Besoin: Décrire les ajouts en extra, et calculer le prix total
  - Thé, Thé-à-la-menthe, Thé-à-la-mente-aux-pignons, etc..
  - Café, Café-au-Lait, Café-au-Lait-à-la-mousse, etc..
- Conception: OO
  - Concevoir une supère classe Boisson que toutes les autres classes héritent.
  - Définir autant de classes qu'il y en a de types de boissons :
    - Beverage.java, Coffee.java, CoffeeWithMilk.java, CoffeeWithMilkWithWhip.java, Tea.java, TeaWithMint.java, TeaWithMintWithPine.java, etc.

StarCoffee : Conception (2/10)

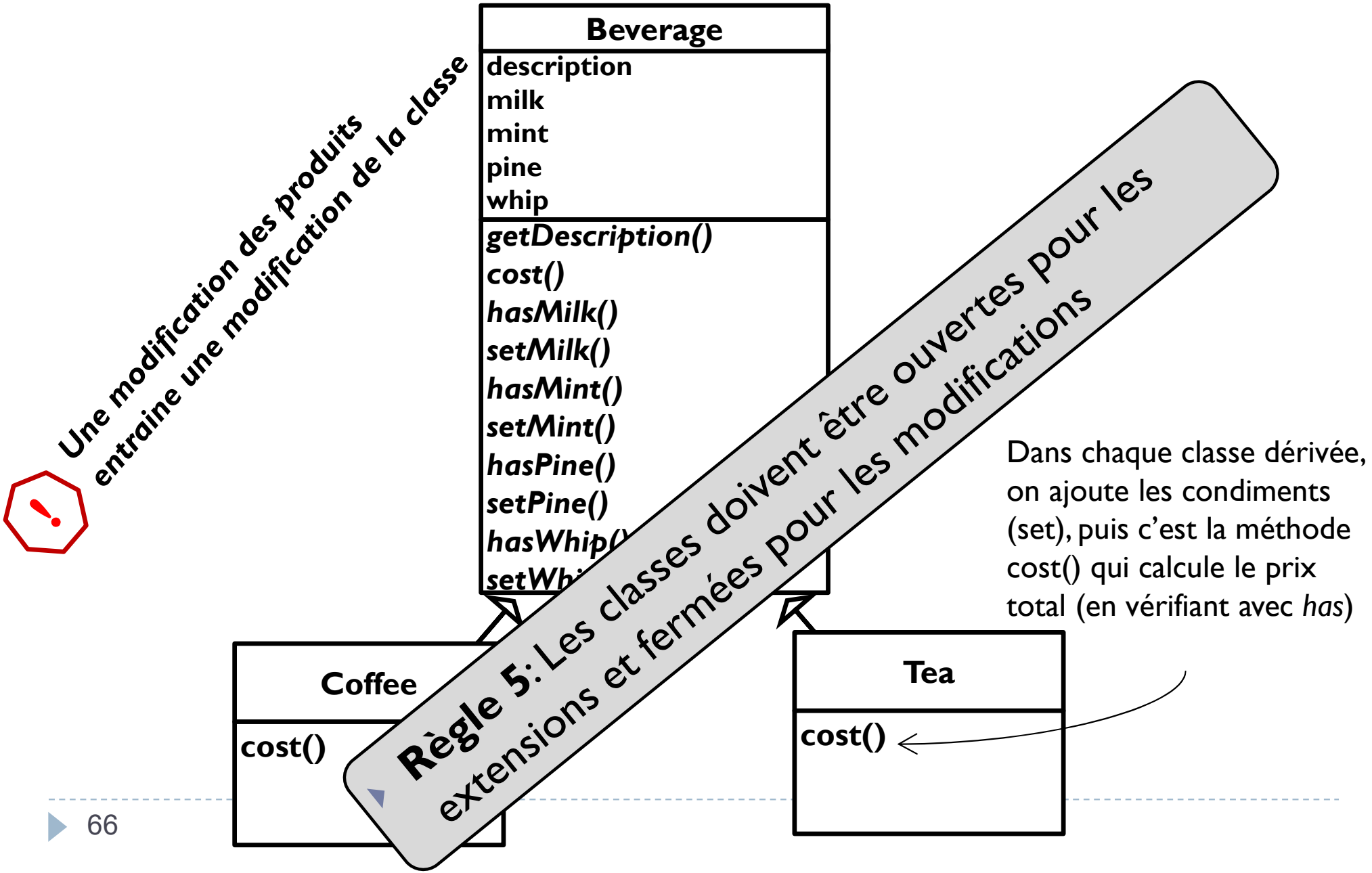


StarCoffee : Problème (3/10)

- **Problème** : En plus des deux produits présentés précédemment, StarCoffee offre une variété d'autres boissons et condiments: (Si on offre 2 types de Thé, on doit ajouter plusieurs autres classes (selon les condiments possibles), etc.
  - Constat: éclatement du diagramme de classes par un nombre ingérable de classes
- **Solution** : Utiliser des variables d'instance dans la supère classe qui représenteront les condiments (pignon, mousse, lait, menthe).

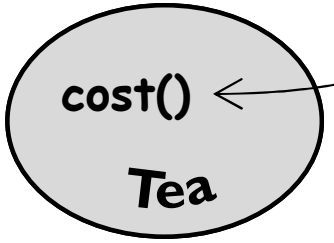


StarCoffee : Nouvelle conception (4/10)



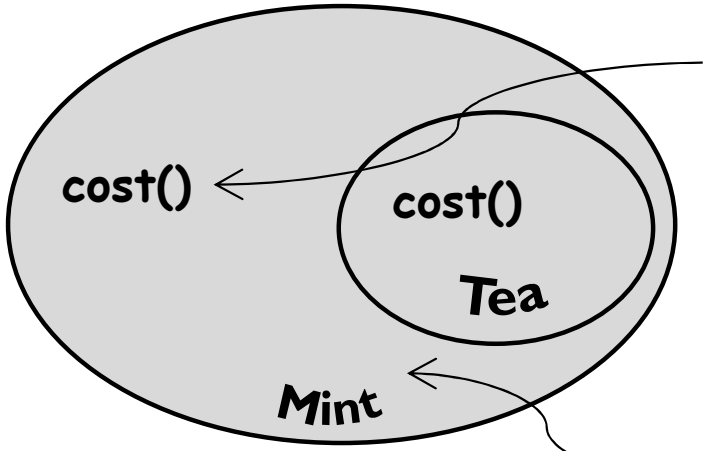
StarCoffee : Un boisson décoré (5/10)

1. On commence par l'objet Tea



La classe Tea hérite de Beverage et possède une méthode cost() qui calcule le prix du boisson

2. Le client choisit la menthe, alors on crée un objet Mint qui enveloppe le Tea



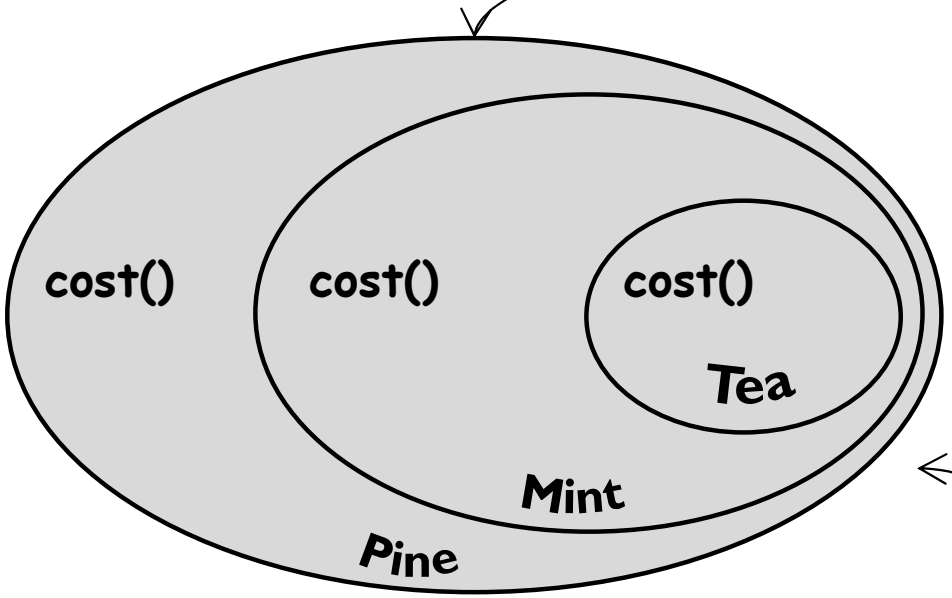
L'objet Mint possède une méthode cost() et à travers le polymorphisme, on peut traiter chaque Beverage enveloppé dans le Mint comme un Beverage aussi.

L'objet Mint est un décorateur. Son type est un miroir de l'objet qu'il décore (Beverage).

StarCoffee : Un boisson décoré (6/10)

3. Le client veut aussi des pignons, alors on crée un objet Pine qui emballe le Mint

L'objet Pine est un décorateur, donc il est un miroir du type Tea et inclut une méthode cost()



L'objet Tea, enveloppé dans un Mint et un Pine, reste toujours un Beverage. Alors on peut faire avec lui ce qu'on peut faire avec les Beverage, y compris l'invocation de sa méthode cost()

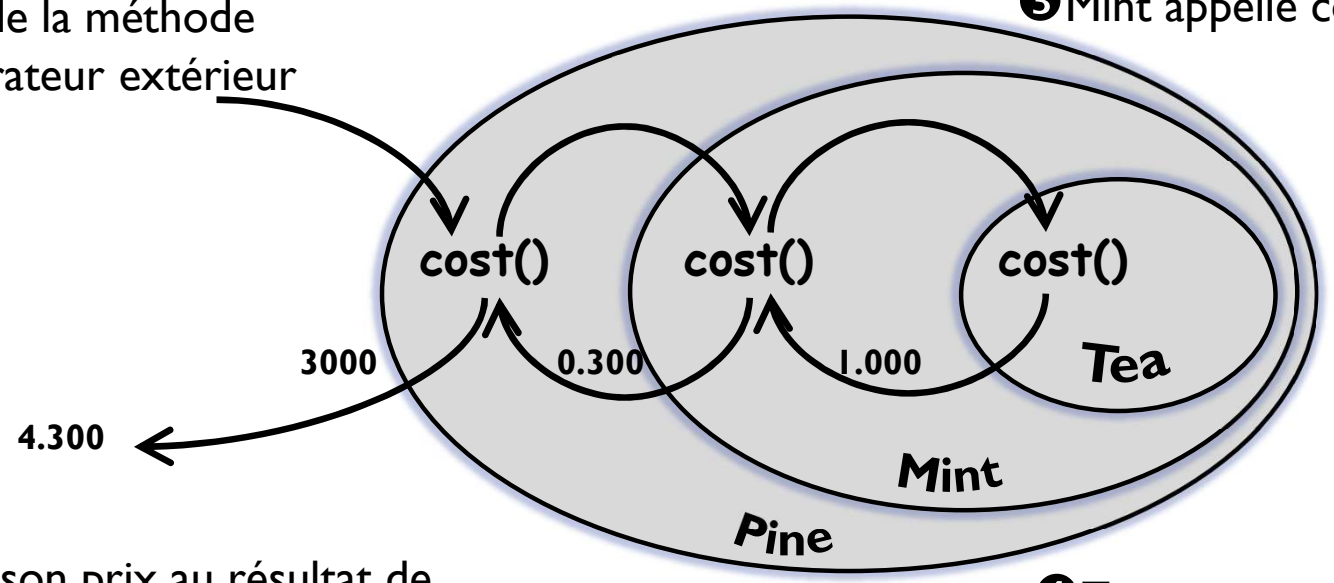
StarCoffee : Le coût du boisson décoré (7/10)

- L'idée est de calculer le coût en partant du décorateur le plus extérieur (Pine) et puis, ce dernier délègue le calcul à l'objet décoré, etc.

❷Pine appelle cost() de Mint

❶ Invocation de la méthode cost() du décorateur extérieur

❸Mint appelle cost() de Tea



❻Pine ajoute son prix au résultat de Mint, et retourne le résultat total: 4.300

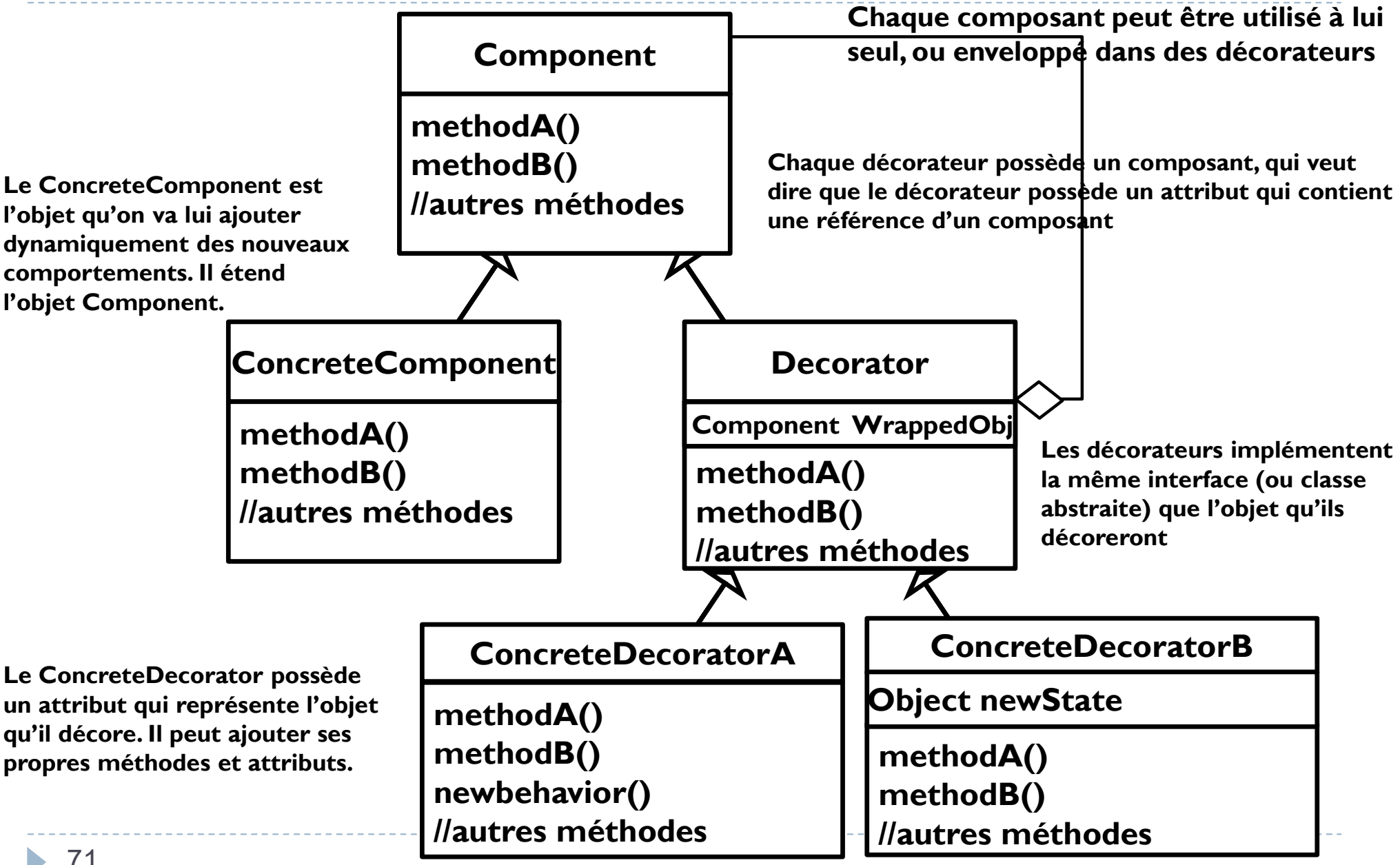
❹Tea retourne son prix: 1.000

❺Mint ajoute son prix au résultat de cost() de Tea, et retourne le nouveau total: 1.300

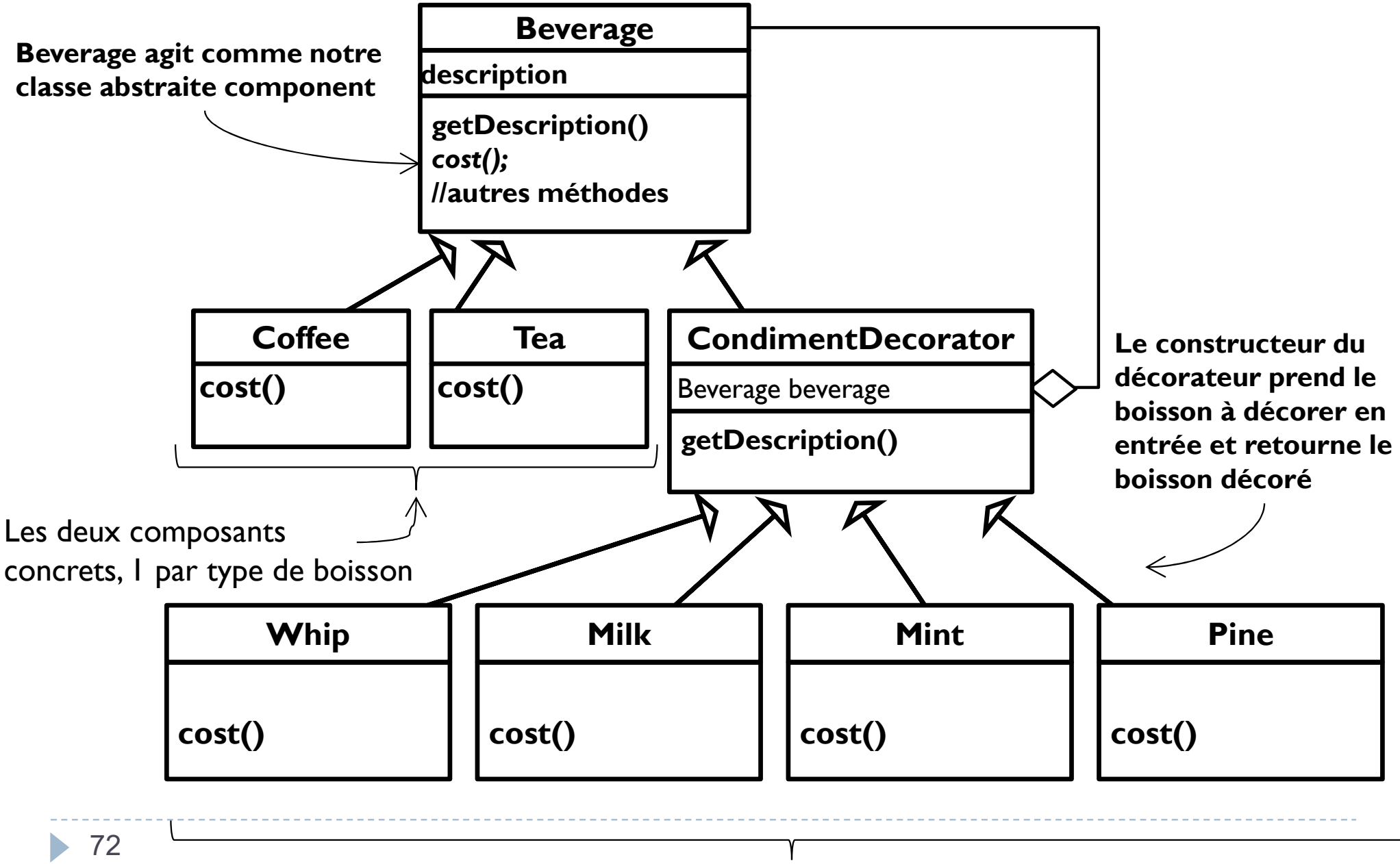
StarCoffee : Le patron Decorator (8/10)

- Définition: **Decorator**
  - Le **patron decorator** attache des responsabilités additionnelles à un objet dynamiquement. Les décorateurs offrent une alternative flexible de sous-classement afin d'étendre les fonctionnalités.

StarCoffee : Le diagramme de classes du patron(9/10)



StarCoffee : La conception finale (10/10)



Récapitulatif (1/2)

- Bases de l'OO: Abstraction, Encapsulation, Polymorphisme & Héritage
- Principes de l'OO
  - Encapsuler ce qui varie
  - Favoriser la composition sur l'héritage
  - Programmer avec des interfaces et non des implémentations
  - Opter pour une conception faiblement couplée
  - Les classes doivent être ouvertes pour les extensions et fermées pour les modifications
- Patron de l'OO
  - Strategy: définit une famille d'algorithmes interchangeables
  - Observer: définit une dépendance 1-à-plusieurs entre objets.
  - **decorator**: attache des responsabilités additionnelles à un objet dynamiquement. Les décorateurs offrent une alternative flexible de sous-classement afin d'étendre les fonctionnalités.

Récapitulatif (2/2)

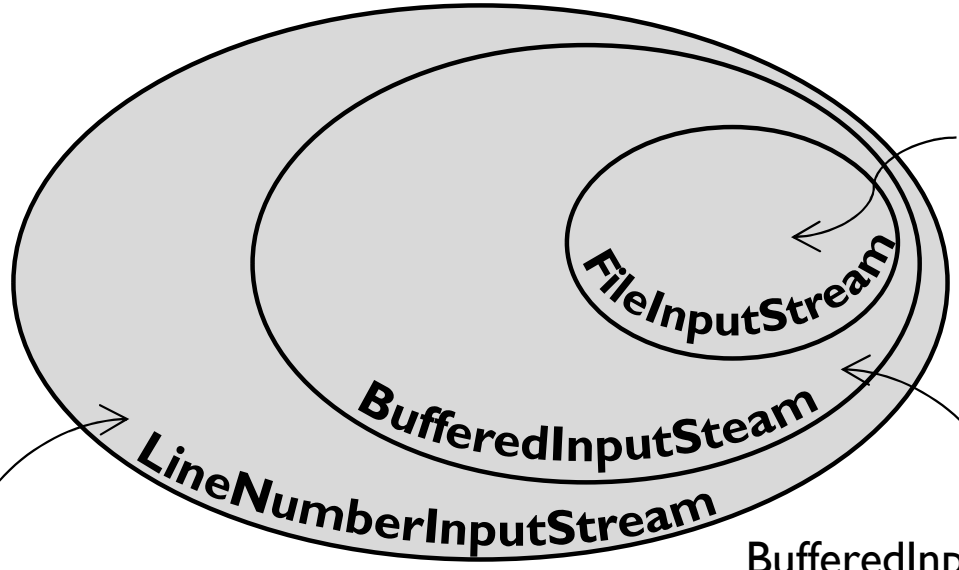
- L'héritage est une forme d'extension, mais il n'est pas nécessairement la meilleure manière pour obtenir la flexibilité dans notre conception
- Le patron decorator implique un ensemble de classes de décorations qui sont utilisées pour envelopper les composants concrets.
- Les classes décorateurs reflètent le type de composant qu'ils décorent.
- Les décorateurs changent le comportement de leurs composants tout en ajoutant des nouvelles fonctionnalités après/avant (ou à la place de) l'appel des méthodes des composants
- On peut envelopper un composant dans n'importe quel nombre de décorateurs
- Les décorateurs sont transparents par rapport au client du composant

Exercice (1/5)

1. Comment faire pour obtenir un café avec "double mousse"?
2. StarCoffee a ajouté un nouveau boisson (Citronnade) au système, comment procéder pour l'inclure dans la conception actuelle?
3. StarCoffee veut introduire des tailles pour ses menus: SMALL, MEDIUM et LARGE. Comment prendre en charge cette nouvelle spécification, si la taille modifie seulement les prix des composants concrets?

Exercice (2/5)

- Avec le patron decorator, le package java.io doit donner plus de sens, puisqu'il se base largement sur ce patron.



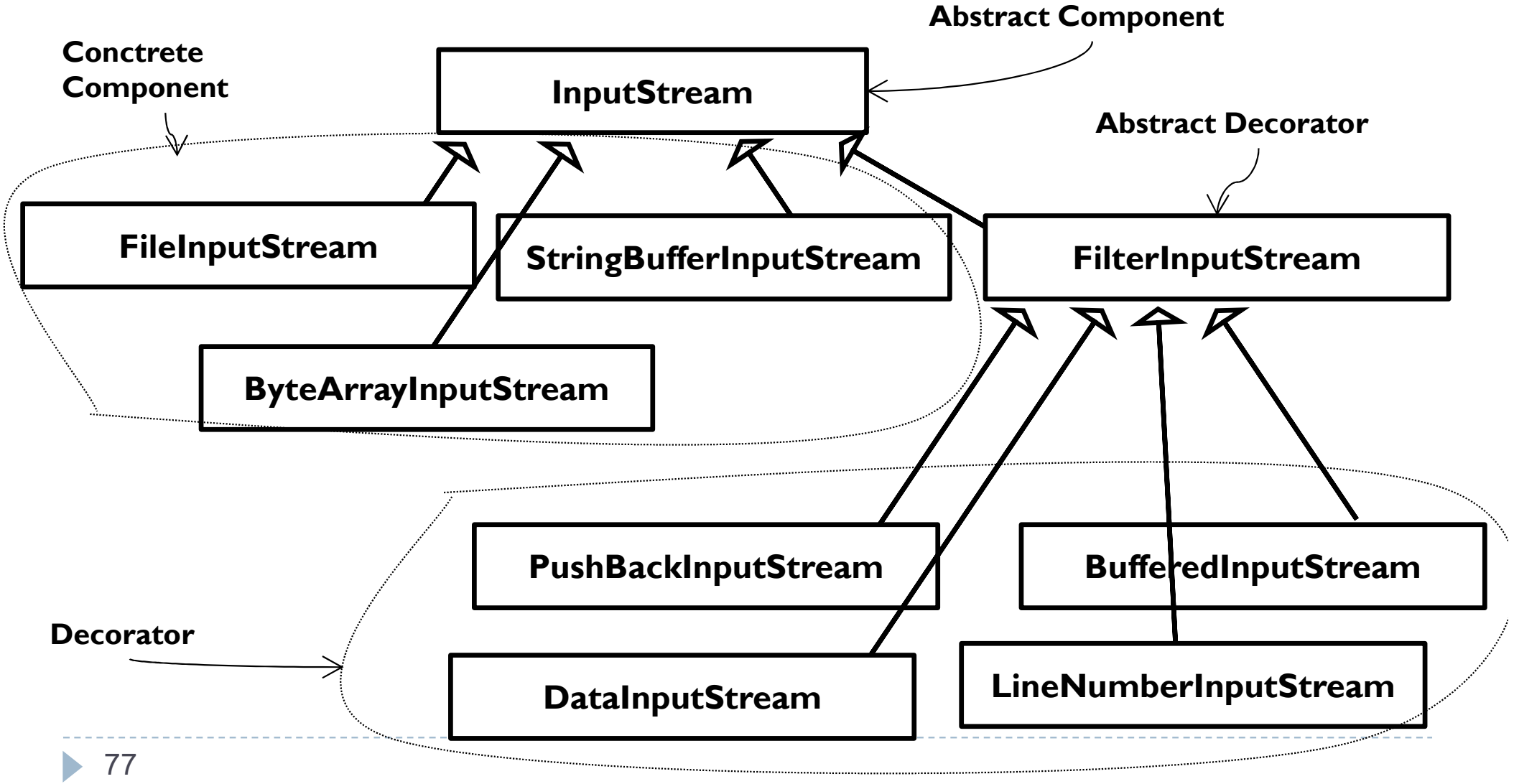
FileInputStream est un composant décoré. Le package java.io offre plusieurs objets pour la lecture des *octets*: FileInputStream, StringBufferInputStream, ByteArrayInputStream, etc.

LineNumberInputStream est aussi un décorateur concret. Il ajoute la capacité de compter les nombres de lignes en lisant les données.

BufferedInputStream est un décorateur concret. Il ajoute deux comportements: (i) il tamponne les entrées afin d'améliorer la performance, et (ii) ajoute la méthode readLine() pour lire une ligne de *caractères* à la fois.

Exercice (3/5): Décoration de java.io

1. Ecrire un décorateur qui convertit tous les caractères majuscules en minuscules dans le flux d'entrée (InputStream).



Exercice (3/5): Décoration de java.io

1. On vous accorde ici le code de la classe FilterInputStream:

```
package java.io;

public class FilterInputStream extends InputStream {
protected volatile InputStream in;
protected FilterInputStream(InputStream in) {
this.in = in;
}
public int read() throws IOException {...}
public int read(byte b[]) throws IOException {...}
public int read(byte b[], int off, int len) throws
IOException {...}
public long skip(long n) throws IOException {...}
public int available() throws IOException {...}
public void close() throws IOException {...}
public synchronized void mark(int readlimit) {...}
public synchronized void reset() throws IOException {...}
public boolean markSupported() {...}
}
```

Exercice (4/5)

- L'objectif de cet exercice est de mettre en œuvre un système flexible de gestion des offres de voiture pour la clientèle de StarCar. Le besoin de cette société se résume à décrire les options demandées par le client (VitreElectrique, AirBag et ABS) et inclure son cout au prix total de la voiture choisie. Deux types de voiture sont gérés par la société, à savoir, camionnette et berline. Chaque voiture est caractérisée par un cout et une description textuelle. Le prix de chaque type de voiture ainsi que celui de chaque option est à fixer au moment de la création.
- En utilisant le patron Decorator, donnez le diagramme de classes de l'application StarCar. (Précisez les méthodes et les attributs, correspondant au bout de code présenté dans le slide suivant)

Exercice (5/5)

```
public static void main(String[] args) {
    Voiture v1=new Camionnette ("P404",10000);
Voiture v2=new Berline ("P407",20000);
    v1=new ABS(v1, 800);//800 représente le prix de l'option ABS
    v2=new VitreElectrique(v2, 1000); // 1000 représente le prix de l'option
    v2=new AirBag(v2, 1200); // 1200 représente le prix de l'option
System.out.println("La voiture est une "+v1.getDescription());
//affiche: La voiture est une P404 avec ABS
   System.out.println("Son prix est:"+ v1.cost());
//affiche: Son prix est 10800
System.out.println("La voiture est une "+v2.getDescription());
//affiche: La voiture est une P407 avec VitreElectrique avec AirBag
   System.out.println("Son prix est:"+ v2.cost());
//affiche: Son prix est 22200
}
```