Design Patterns
Part 3

Mohamed Youssfi

Laboratoire Signaux Systèmes Distribués et Intelligence Artificielle (SSDIA)

ENSET, Université Hassan II Casablanca, Maroc

Email : med@youssfi.net

Supports de cours : http://fr.slideshare.net/mohamedyoussfi9

Chaîne vidéo : http://youtube.com/mohamedYoussfi

Recherche : http://www.researchgate.net/profile/Youssfi_Mohamed/publications

Pattern Décorateur

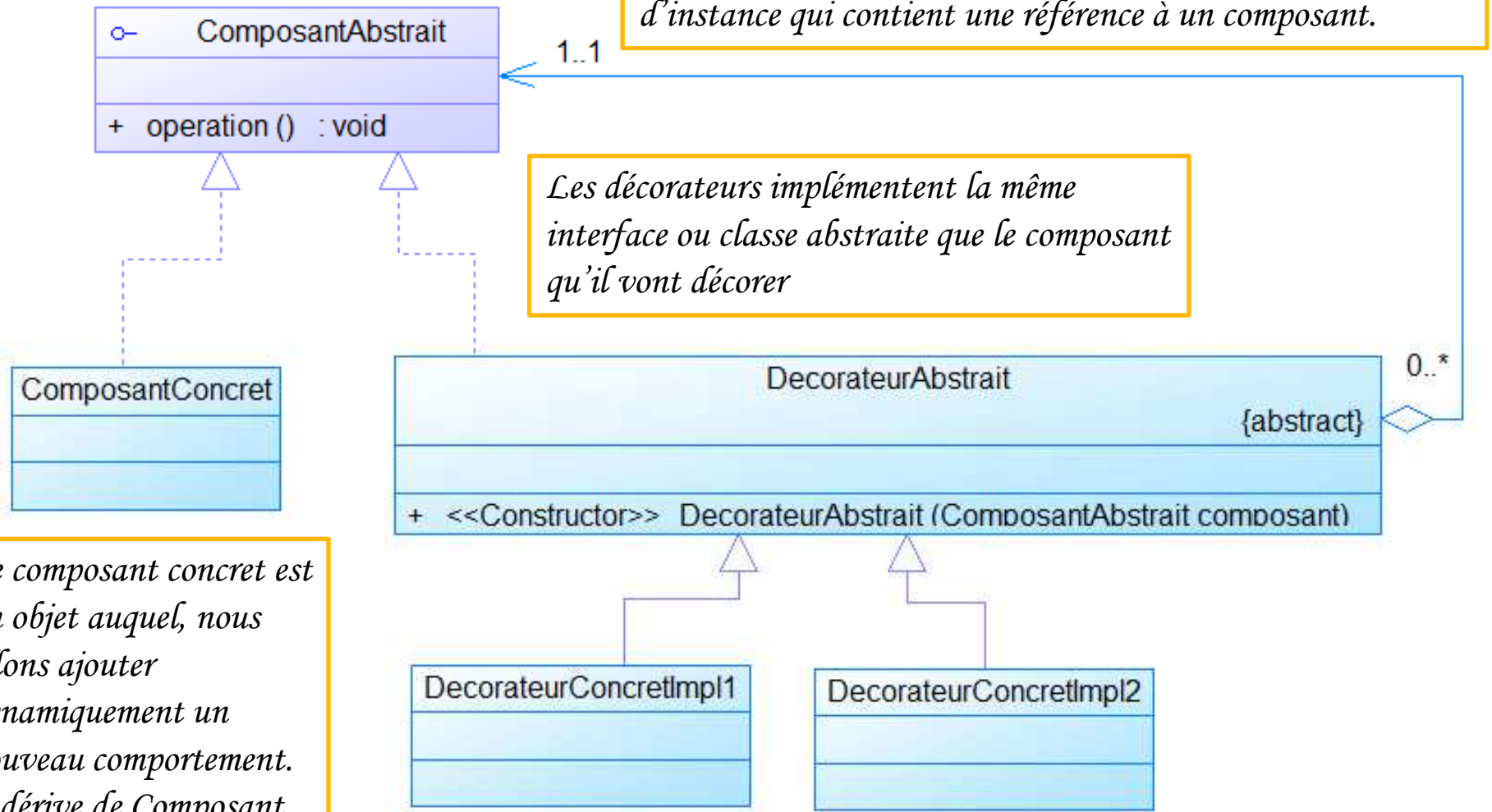
Pattern Décorateur

- Catégorie :
  - Structure
- Définition du 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.*
- Résultat :
  - Le Design Pattern permet d'isoler les responsabilités d'un objet.

Diagramme de classes

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'il vont décorer

Le composant concret est un objet auquel, nous allons ajouter dynamiquement un nouveau comportement. Il dérive de Composant

Les décorateurs ajoute généralement le nouveau comportement en effectuant un traitement avant ou après une méthode existante dans le composant

Responsabilités

- **ComposantAbstrait :**
  - Définit l'interface ou une classe abstraite qui représente le composant abstrait à décorer.
- **ComposantConcret** :
  - Implémentation de l'interface qui représente le composant concret à décorer et qui correspondant aux fonctionnalités souhaitées à la base.
- **DecorateurAbstrait :**
  - Interface ou classe abstraite qui définit le décorateur abstrait et contient une référence vers un objet Abstraction.
- **DecorateurConcretImpl1 et DecorateurConcretImpl2 :**
  - Représentent les décorateurs concrets des composants
  - Les décorateurs ont un constructeur acceptant un objet ComposantAbstrait.
  - Les méthodes des décorateurs appellent la même méthode de l'objet qui a été passée au constructeur.
  - La décoration ajoute des responsabilités en effectuant des opérations avant et/ou après cet appel.

Responsabilités

- La partie cliente manipule un objet Abstraction.
- En réalité, cet objet Abstraction peut être
  - un objet ComposantConcret
  - ou un objet DecorateurConcret.
  - Ainsi, des fonctionnalités supplémentaires peuvent être ajoutées à la méthode d'origine.
  - Ces fonctionnalités peuvent être par exemple des traces de log ou une gestion de buffer pour des entrées/sorties.

Implémentation

ComposantAbstrait.java :

```
public interface ComposantAbstrait {
   public void operation();
}
```

ComposantConcretImpl1.java :

```
public class ComposantConcretImpl1 implements ComposantAbstrait {
  @Override
  public void operation() {
    System.out.println("Je sais faire uniquement ça Version 1");
  }
}
```

ComposantConcretImpl2.java :

```
public class ComposantConcretImpl2 implements ComposantAbstrait {
  @Override
  public void operation() {
    System.out.println("Je sais faire uniquement ça Version 2");
  }
}
```

Implémentation

DecorateurAbstrait.java :

```
public abstract  class DecorateurAbstrait implements ComposantAbstrait {
  protected ComposantAbstrait composantAbstrait;
  public DecorateurAbstrait(ComposantAbstrait composantAbstrait) {
   super();
   this.composantAbstrait = composantAbstrait;
  }
}
```

DecorateurConcretImpl1.java :

```
public class DecorateurConcretImpl1 extends DecorateurAbstrait {
 public DecorateurConcretImpl1(ComposantAbstrait composantAbstrait) {
   super(composantAbstrait);
 }
@Override
public void operation() {
  System.out.println("Décorateur 1 : avant, je je fais X");
  composantAbstrait.operation();
  System.out.println("Décorateur 1 : après, je je fais Y");
}
}
```

Implémentation

DecorateurConcretImpl2.java :

```
public class DecorateurConcretImpl2 extends DecorateurAbstrait {
 public DecorateurConcretImpl1(ComposantAbstrait composantAbstrait) {
   super(composantAbstrait);
 }
@Override
public void operation() {
  System.out.println("Décorateur 2 : avant, je fais A");
  composantAbstrait.operation();
  System.out.println("Décorateur 2 : après, je fais B");
}
}
```

Utilisation du pattern Décorateur

```
ComposantAbstrait c=new ComposantConcretImpl1();
c.operation();
```


```
c=new DecorateurConcretImpl1(c);
c.operation();
```


```
c=new DecorateurConcretImpl2(c);
c.operation();
```

Utilisation du pattern Decorateur

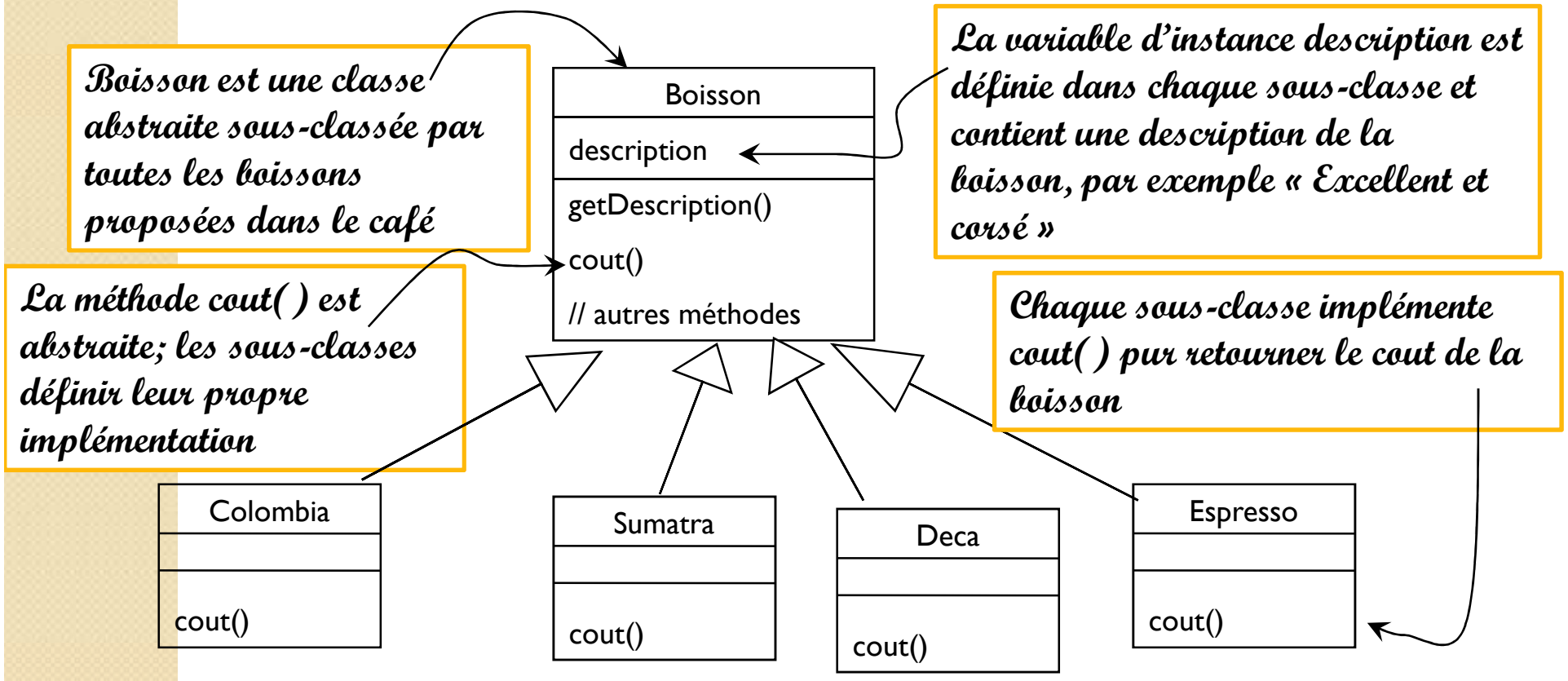
```
public class Application {
public static void main(String[] args) {
    ComposantAbstrait composantAbstrait=new ComposantConcretImpl1();
    composantAbstrait.operation();
    System.out.println("--------------------"); System.out.println("Première décoration");
    System.out.println("--------------------");
    composantAbstrait=new DecorateurConcretImpl1(composantAbstrait);
    composantAbstrait.operation();
    System.out.println("--------------------"); System.out.println("Deuxième décoration");
    System.out.println("--------------------");
    composantAbstrait=new DecorateurConcretImpl2(composantAbstrait);
    composantAbstrait.operation();
}}
```

```
Je sais faire uniquement ça Version1
--------------------
Première décoration
--------------------
Décorateur 1 : avant, je je fais X
Je sais faire uniquement ça Version1
Décorateur 1 : après, je je fais Y
--------------------
Deuxième décoration
--------------------
Décorateur 2 : avant, je je fais A
Décorateur 1 : avant, je je fais X
Je sais faire uniquement ça Version1
Décorateur 1 : après, je je fais Y
Décorateur 2 : après, je je fais B
```

Application

Bienvenue chez StarbuzzCoffee

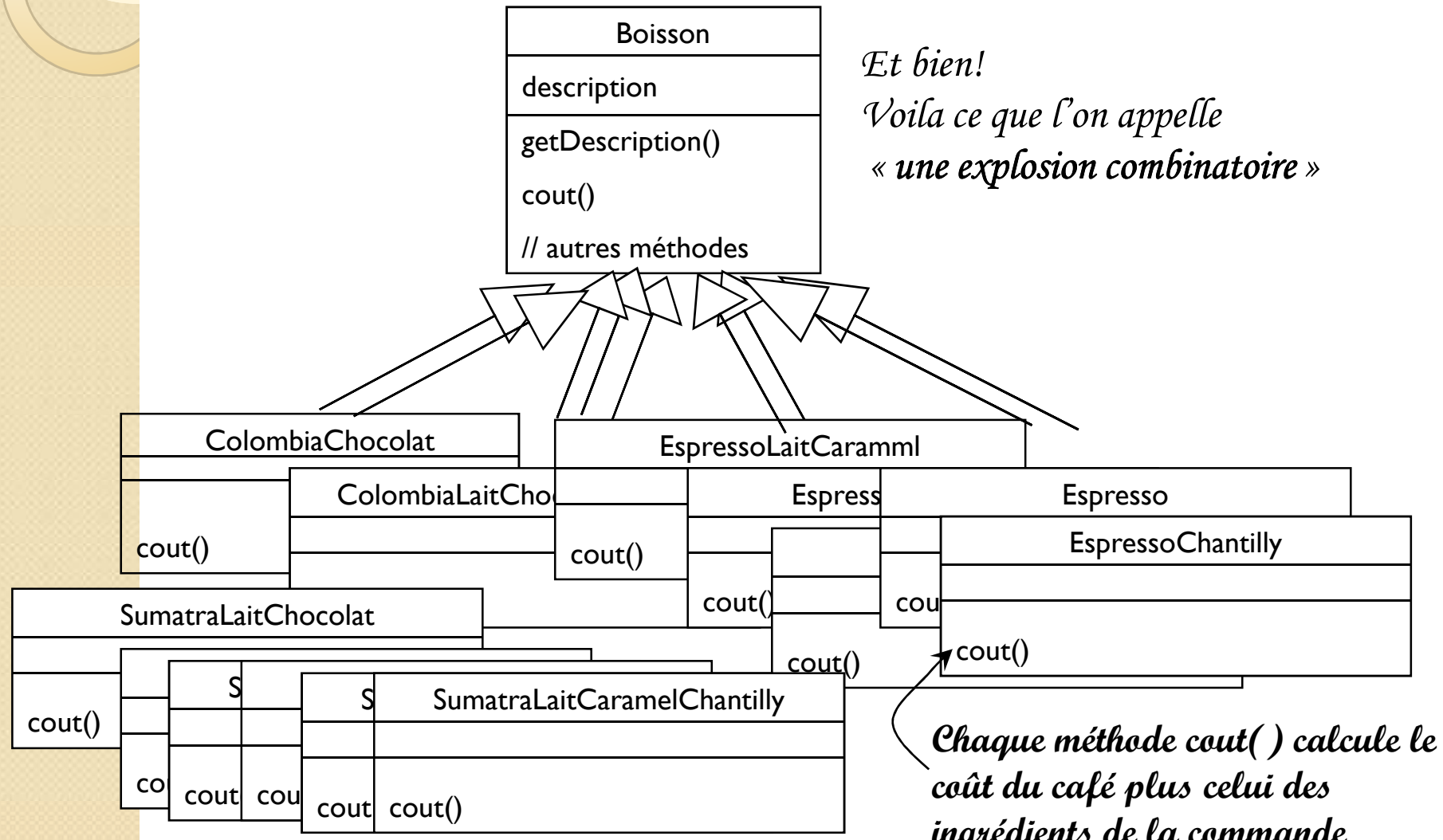
- Starbuzz Coffee s'est fait un nom en devenant la plus importante chaîne de « salons de café » aux états unis.
- Quand ils ont commencé, ils on conçu leurs classes comme ceci:



Bienvenue chez StarbuzzCoffee

- En plus de votre café, vous pouvez également demander plusieurs ingrédients, comme
  - de la mousse de lait,
  - du caramel,
  - du chocolat,
  - du sirop,
  - de la vanille
  - ou noisette
  - et couronner le tout avec de la crème chantilly.
- Starbuzz Coffee, facturant chacun de ces suppléments, ils ont besoin de les intégrer à leur système de commande

Voici leur premier essai



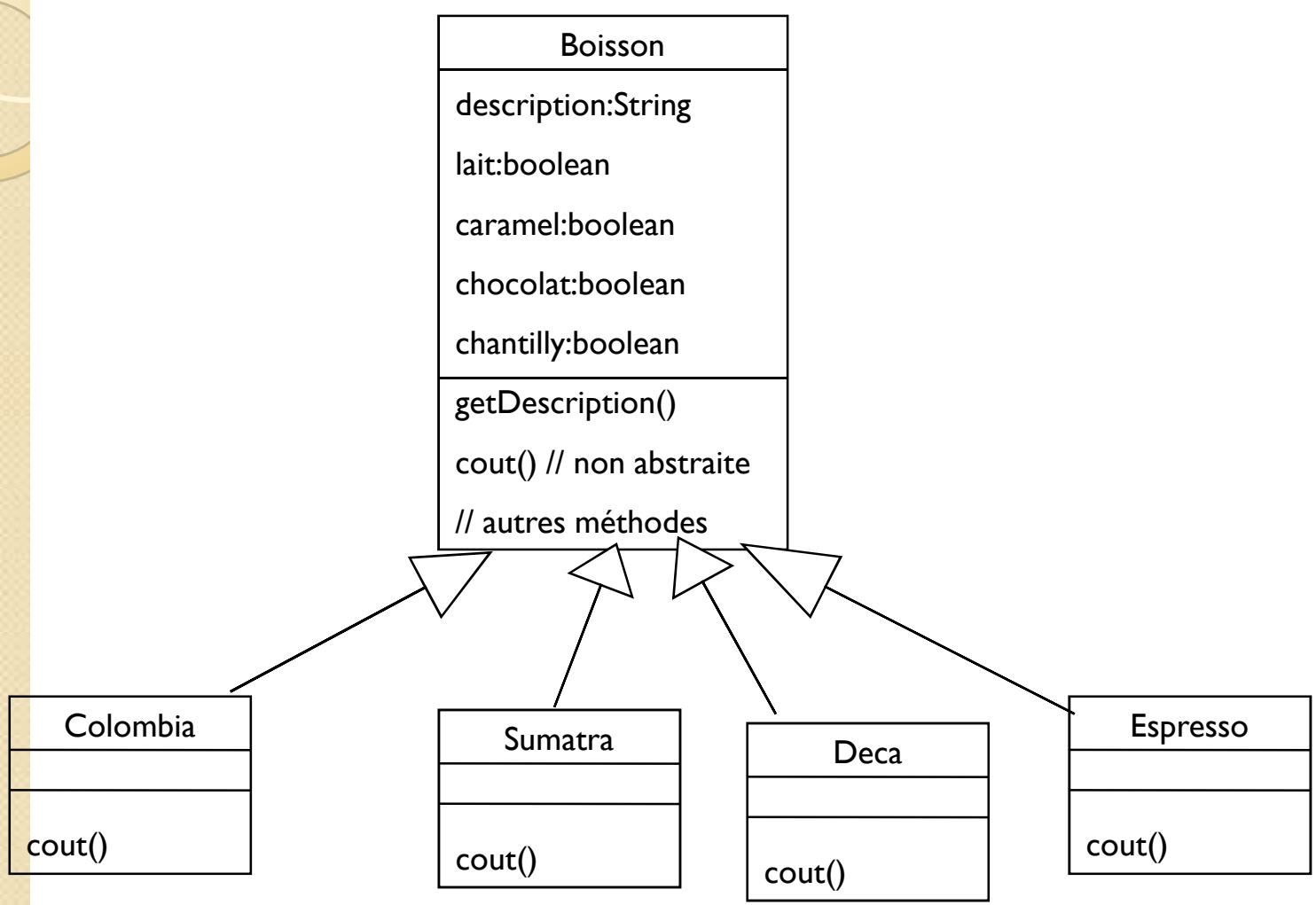
Et bien!
Voila ce que l'on appelle
*« **une explosion combinatoire** »*

Chaque méthode cout( ) calcule le coût du café plus celui des ingrédients de la commande

Musclez vos neurones

- De toute évidence, Starbuzz s'est fourré tout seul dans un vrai cauchemar.
- Que va-t-il se passer si le prix du lait change?
- Que feront-ils quand il y'a un supplément de vanille?
- Au-delà du problème de maintenance, ils enfreignent l'un des principes de la conception que nous avons déjà vus.
  - Lequel ?
    - -
  - Quelle conception proposez-vous?
    - -

Une proposition pour éviter l'explosion de classes



Ecrivez les méthodes cout() pour les deux classes suivantes: (pseudo code uniquement)

```java
/*Boisson.java*/
public class Boisson {
  protected String description;
  protected boolean lait;
  protected boolean caramel;
  protected boolean chocolat;
  protected boolean chantilly;

  public double cout(){
  double prixIngerdients=0;
  if(lait==true) prixIngerdients+=2;
  if(caramel==true) prixIngerdients+=1;
  if(chocolat==true) prixIngerdients+=1.5;
  if(chantilly==true) prixIngerdients+=0.5;
  return prixIngerdients;
  }
  // Getters et Setters
}
```

```java
/*Espresso.java*/
public class Espresso extends
Boisson {
  public Espresso() {
    description="Espresso";
  }
  @Override
public double cout() {
  return super.cout()+6;
}
}
```

```java
/*Application.java*/
 Espresso boisson=new Espresso();
 System.out.println(boisson.cout());
 boisson.setCaramel(true);
 boisson.setChantilly(true);
 System.out.println(boisson.cout());
```

A vos crayons!

- Quelles sont les exigences et les autres facteurs qui pourraient changer et avoir un impact négatif sur cette conception?
  - Application ouverte à la modification pour chaque ajout de nouveau ingrédients
  - Attribuer des fonctionnalités non appropriées aux classes dérivées
  - Impossibilité de multiplier l'application des ingrédients
  - -
  - -
  - -
  - -
  - -
  - -

Principe Ouvert - Fermé

Principe de conception:

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

5ème Principe de conception

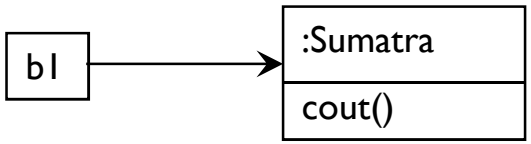
- Notre but est de permettre d'étendre facilement les classes pour incorporer de nouveaux comportements sans modifier le code existant.
- Qu'obtiendrons-nous, si nous y parvenons?
  - Des conceptions résistantes au changement et suffisamment souples pour accepter de nouvelles fonctionnalités répondant à l'évolution des besoins.

Faites connaissance du « Pattern Décorateur »

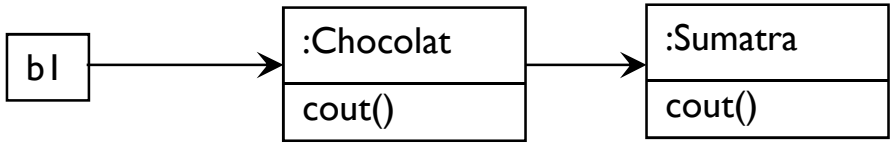
- Nous avons vu que représenter notre boisson plus le schéma de tarification des ingrédients au moyen de l'héritage n'a pas bien fonctionné; nous obtenons
  - une explosion de classes,
  - une conception rigide,
  - ou nous ajoutons à la classe de base des fonctionnalités non appropriée pour certaines sous-classes.
- Voici ce que nous allons faire.
  - 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:
  - Créer un objet Sumatra
  - Le décorer avec un objet Chocolat
  - Le décorer avec un objet Chantilly
  - Appeler la méthode cout() et nous appuyer sur la délégation pour ajouter les coûts des ingrédients.

Construire une boisson avec des décorateurs

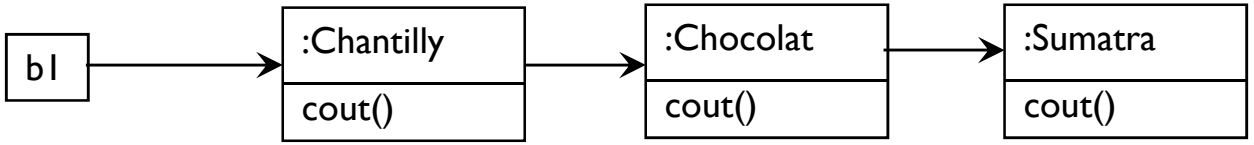
- Commençons par créer un objet Sumatra
  - Boisson b1=new Sumatra();



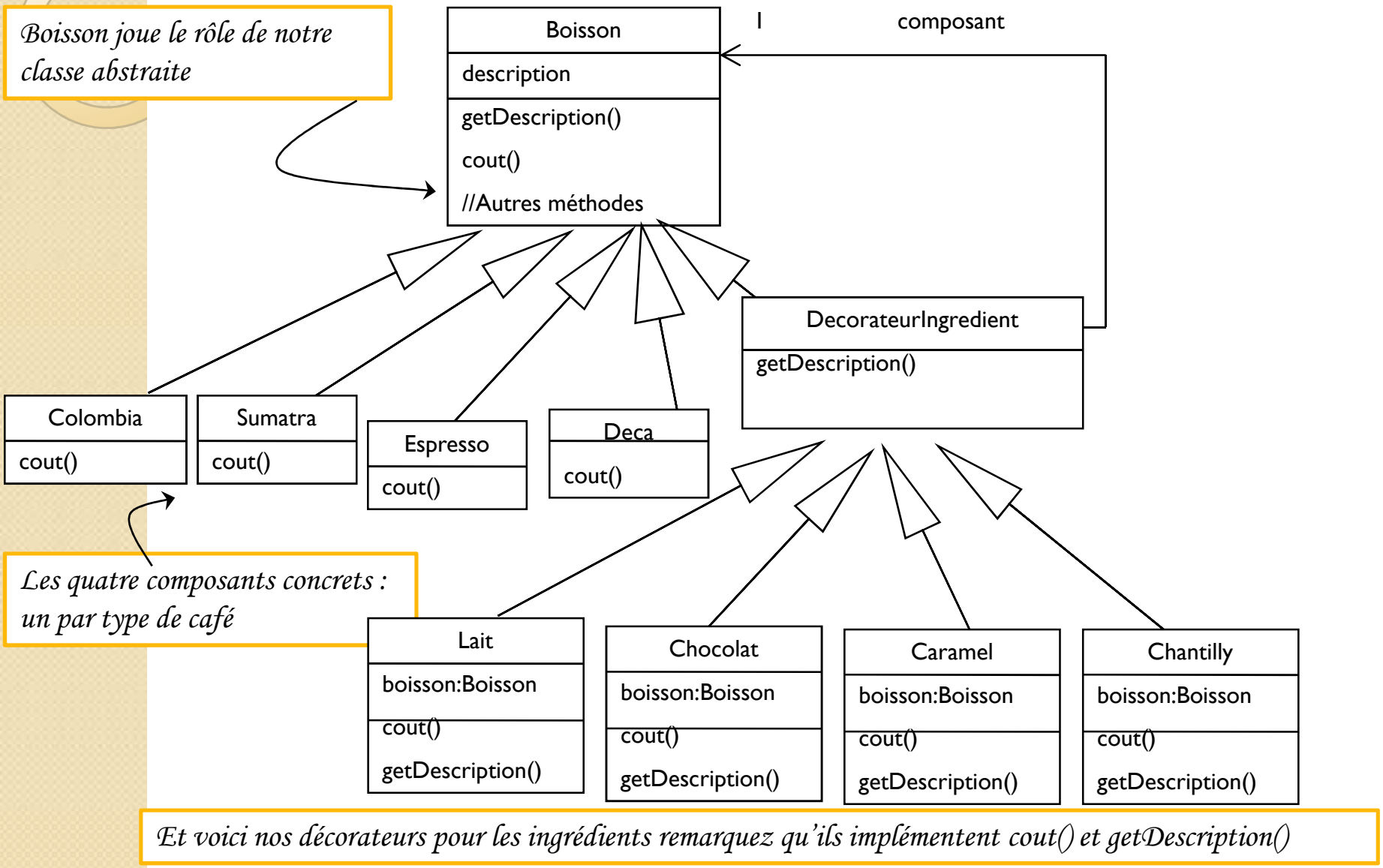
- Le client veut ajouter du chocolat. Nous créer donc un objet Chocolat et nous enveloppons le Sumatra dedans.
  - b1=new Chocolat(b1);



- Le Client veut aussi du Chantilly. Nous créons un décorateur Chantilly et nous enveloppons Chocolat dedans.
  - B1=new Chantilly(b1);



Application du pattern Décorateur à notre problème



Implémentation de Starbuzz

Boisson.java

```
public abstract class Boisson {
 String description;

public String getDescription() {
 return description;
}
public abstract  double cout();
}
```

DecorateurIngredient.java

```
public abstract class DecorateurIngredient extends Boisson {
 protected Boisson boisson;
 public DecorateurIngredient(Boisson boisson) {
  this.boisson = boisson;
 }
 public abstract String getDescription();
}
```

Coder les boissons

Espresso.java

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

Columbia.java

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

Coder les ingrédients (Décorateurs)

Chocolat.java

```
public class Chocolat extends DecorateurIngredient {
public Chocolat(Boisson boisson) {  super(boisson); }
public double cout() {  return 0.20 +boisson.cout() ;
}
public String getDescription() {
 return boisson.getDescription()+", Chocolat";
}
}
```

Caramel.java

```
public class Caramel extends DecorateurIngredient {
public Caramel(Boisson boisson) { super(boisson); }
public double cout() { return 0.22 +boisson.cout() ; }
public String getDescription() {
 return boisson.getDescription()+", Caramel";
}
}
```

Le café est servi

```
public class ApplicationDecorateur {

public static void main(String[] args) {
 Boisson b1=new Espresso();
 System.out.println(b1.getDescription()+" € "+b1.cout());

 Boisson b2=new Colombia();
 b2=new Caramel(b2);
 b2=new Chocolat(b2);
 b2=new Chocolat(b2);
 System.out.println(b2.getDescription()+" € "+b2.cout());
}
}
```

A vos crayons : Dessinez vos objets

Exercice

- InputStream du package ja.io est une classe abstraite dont plusieurs implémentations concrètes définissent une méthode read() qui permet de lire un entier à partir d'une entrée quelconque.
- La classe Concrète FileInpustream est un InputStream qui permet de lire des entiers à partir d'un fichier :
  - Pour créer l'objet:
    - InputStream is=new FileInputStream("original.txt");
  - Pour lire un entier du fichier:
    - int c=is.read();
- Nous souhaitons créer un décorateur de InputStream qui nous permet de décrypter un fichier qui contient une suite de nombres paires. Le décryptage consiste à récupérer du fichiers deux nombre par deux nombres en faisant la différence entre les deux.
  - Exemple : Si le fichier contient le texte : bacadb
  - Le nombre décrypté est obtenu comme suit:
    - code ascii (b)-code ascii(a) = 1
    - code ascii (c)-code ascii(a) = 2
    - code ascii (d)-code ascii(b) = 2
  - Le nombre décrypté est 122.
  - Pour créer l'objet DecrypteInputStream, on peut écrire
    - is=new DecrypteurInputStream(is);
  - Pour lire un nombre décrypté
    - int c=is.read();

Travail à faire

- Créer un diagramme de classes
- Créer un décorateur abstrait de InpuStream : DecorateurInputStream.java
- Créer un Décorateur concret de InputStream, qui permet de décrypter un InputStream : DecrypteInputStream.java
- Créer un fichier texte « original.txt » qui contient le texte crypté
- Créer une application qui permet de:
  - Créer un FileInputStream représentant le fichier « Original.txt »
  - Décorer l'objet créé par un objet de DecrypteInputStream
  - Afficher le texte décrypté