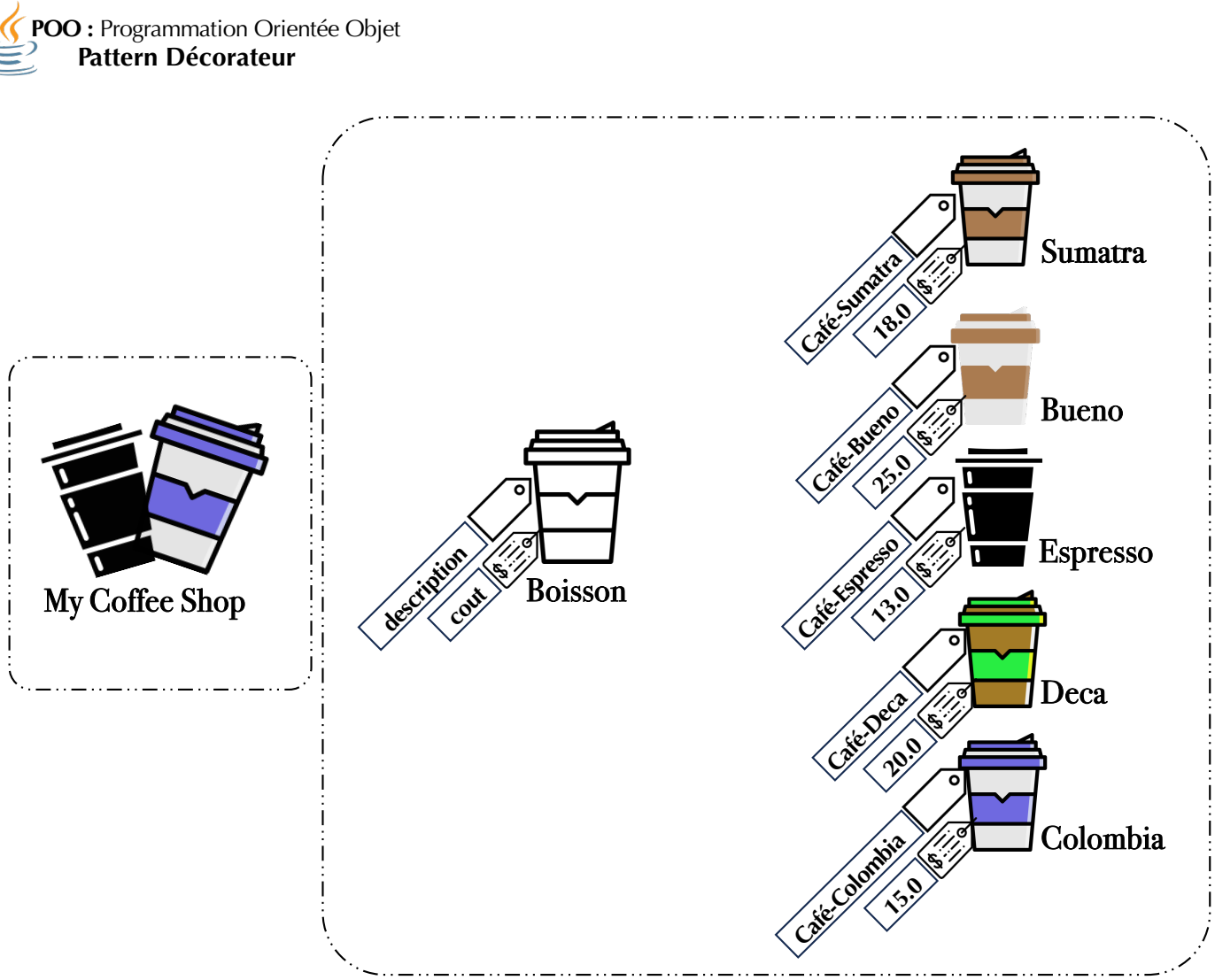
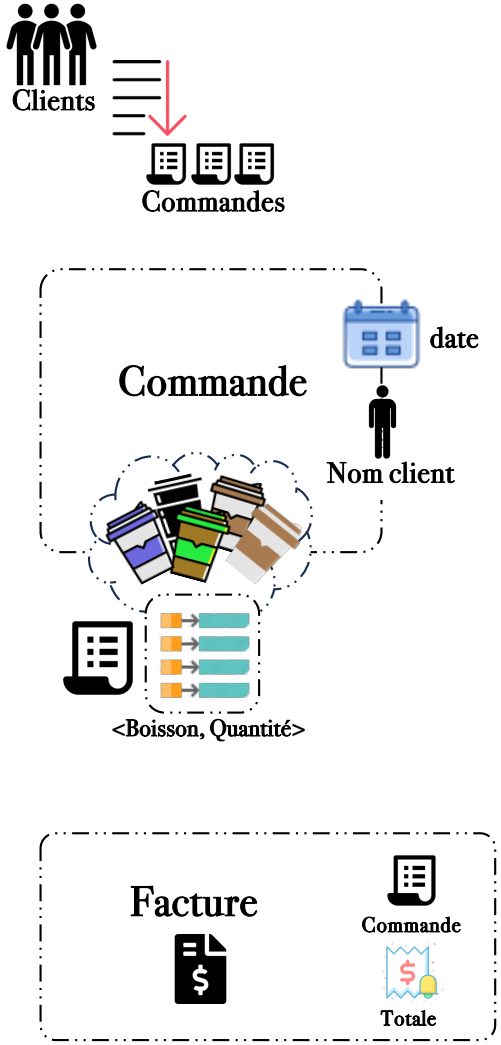
Pattern
Decorator

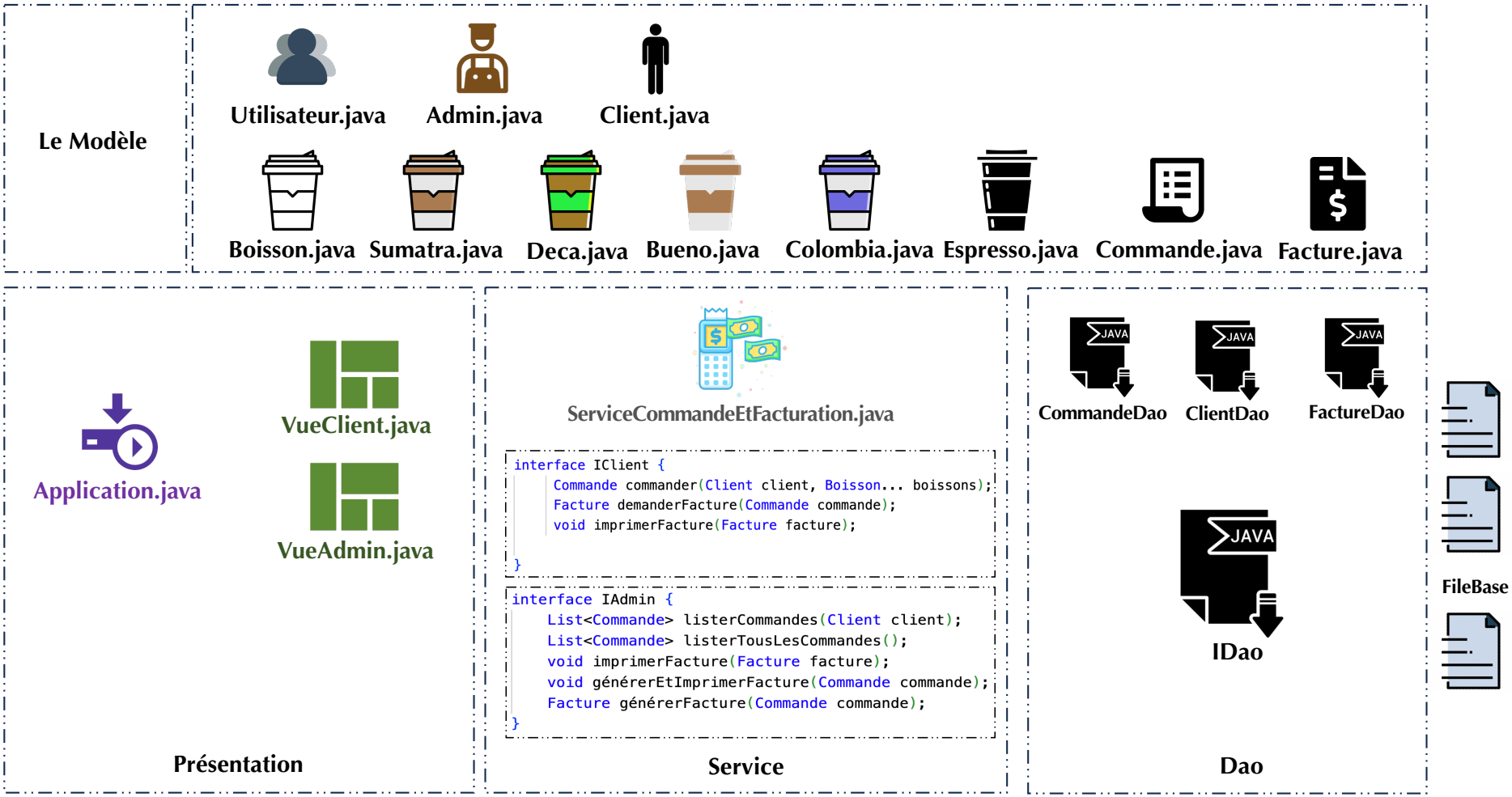
POO : Programmation Orientée Objet
Pattern Décorateur

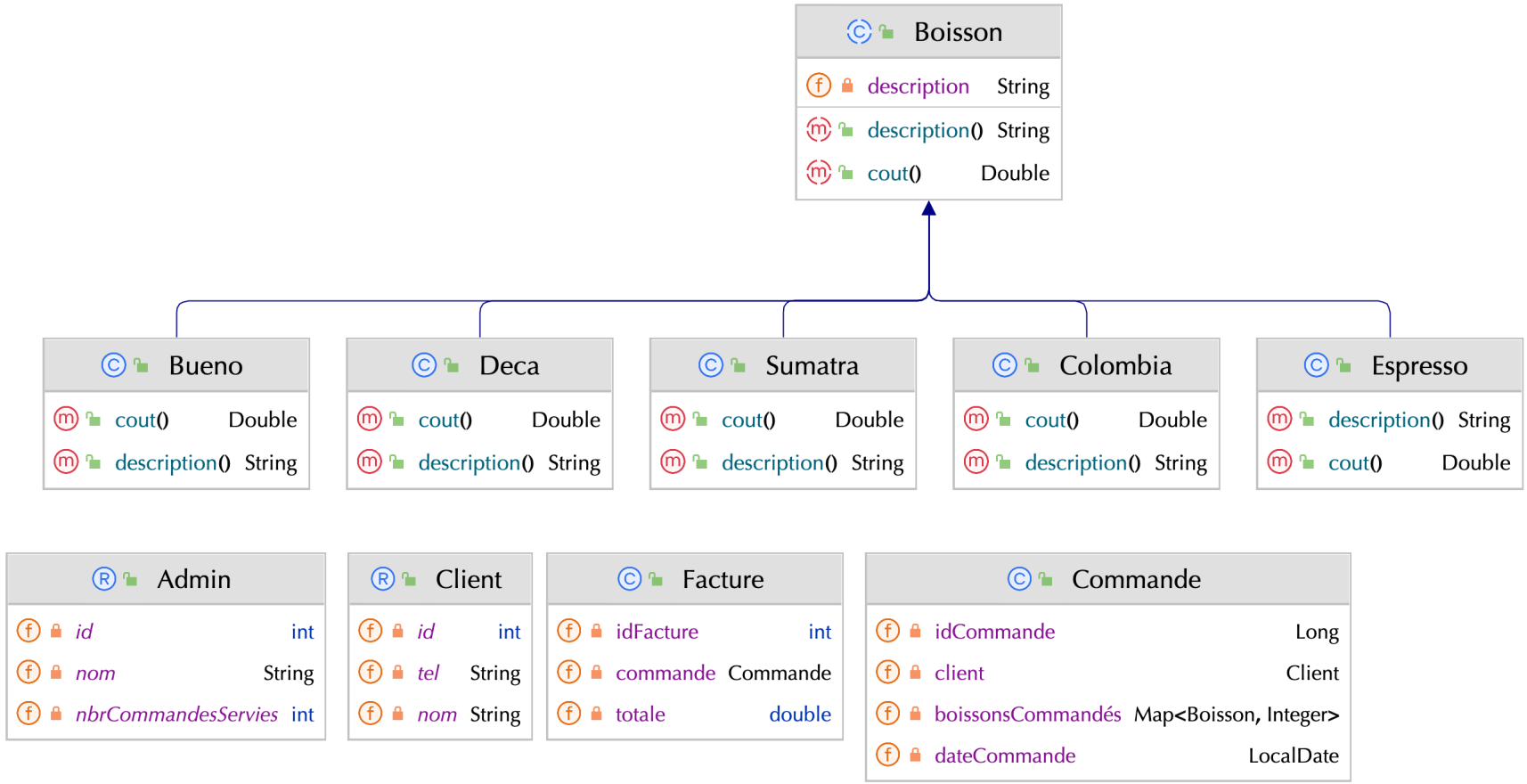




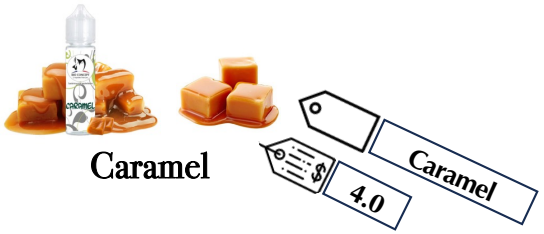
POO : Programmation Orientée Objet

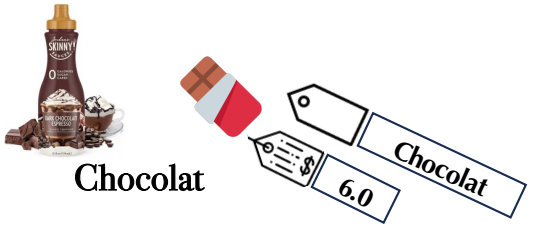
Pattern Décorateur

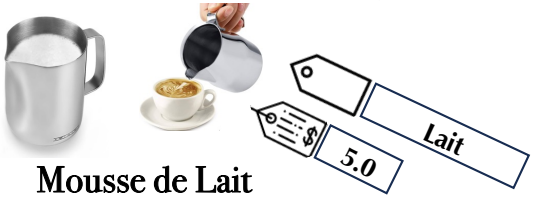



Boisson
description String
description() String
cout() Double
Bueno
cout() Double
description() String
Deca
cout() Double
description() String
Sumatra
cout() Double
description() String
Colombia
cout() Double
description() String
Espresso
description() String
cout() Double
Admin
id int
nom String
nbrCommandesServies int
Client
id int
tel String
nom String
Facture
idFacture int
commande Commande
totale double
Commande
idCommande Long
client Client
boissonsCommandés Map<Boisson, Integer>
dateCommande LocalDate

POO : Programmation Orientée Objet
Pattern Décorateur



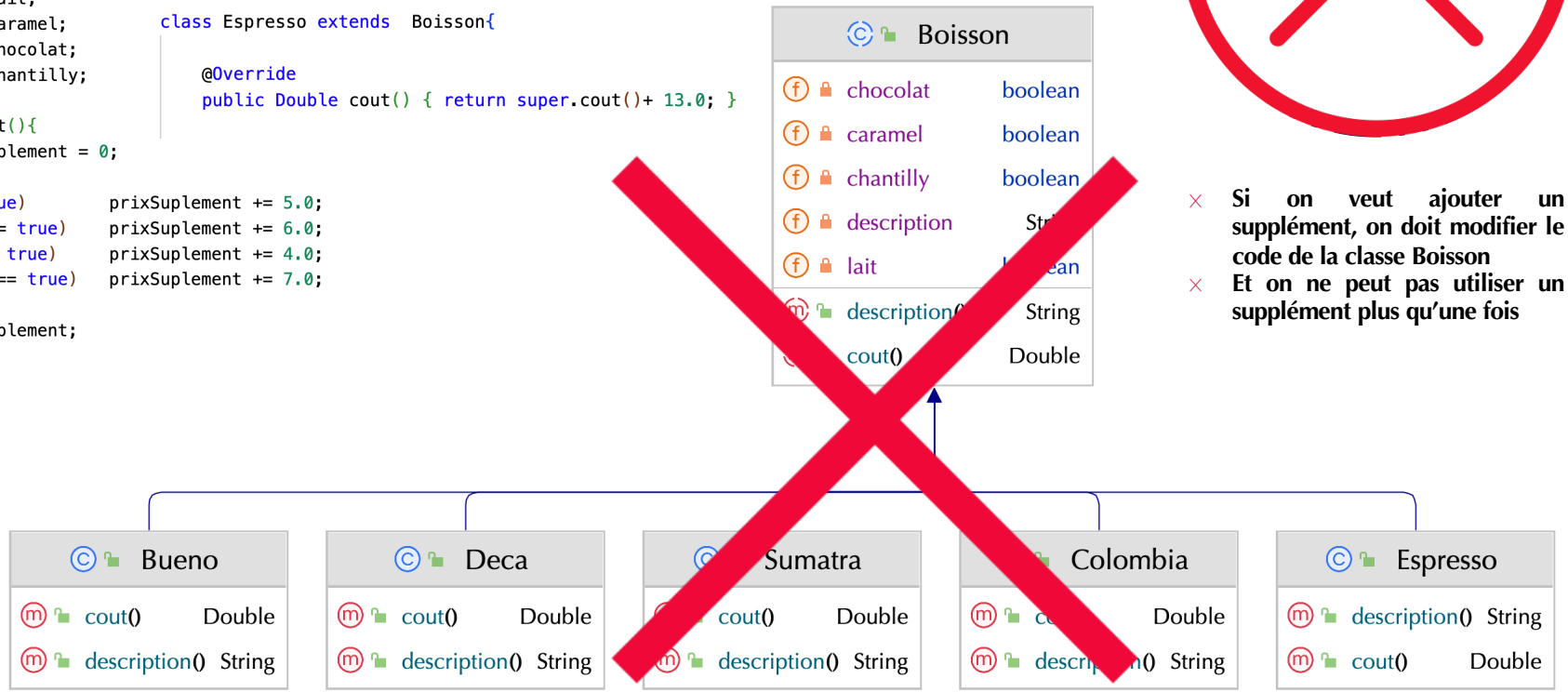




Explosion Combinatoire

```java
public class Boisson {

    private String description;
    private boolean lait;
    private boolean caramel;
    private boolean chocolat;
    private boolean chantilly;

    public Double cout(){
        double prixSuplement = 0;

        if(lait == true)        prixSuplement += 5.0;
        if(chocolat == true)    prixSuplement += 6.0;
        if(caramel == true)     prixSuplement += 4.0;
        if(chantilly == true)   prixSuplement += 7.0;

        return prixSuplement;
    }

    //Getters Setters
}
```

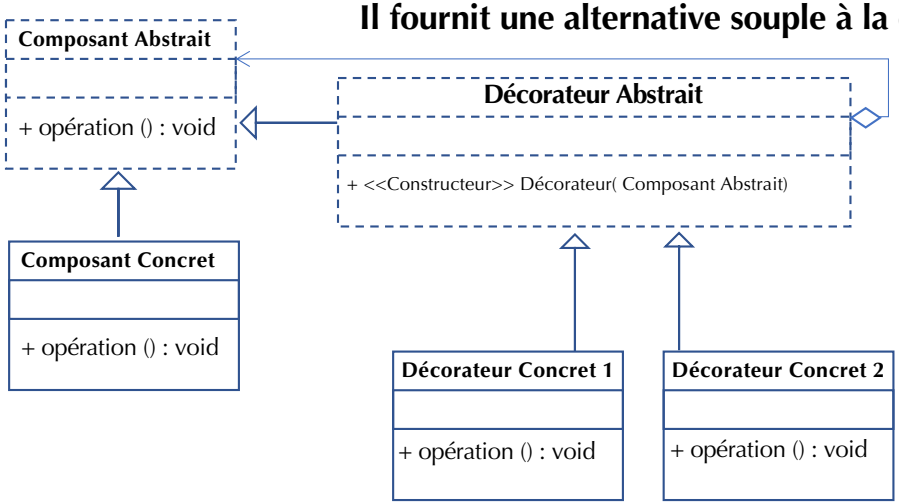
```java
class Espresso extends  Boisson{

    @Override
    public Double cout() { return super.cout()+ 13.0; }
```



- Si on veut ajouter un supplément, on doit modifier le code de la classe Boisson
- Et on ne peut pas utiliser un supplément plus qu'une fois

Pattern Décorateur

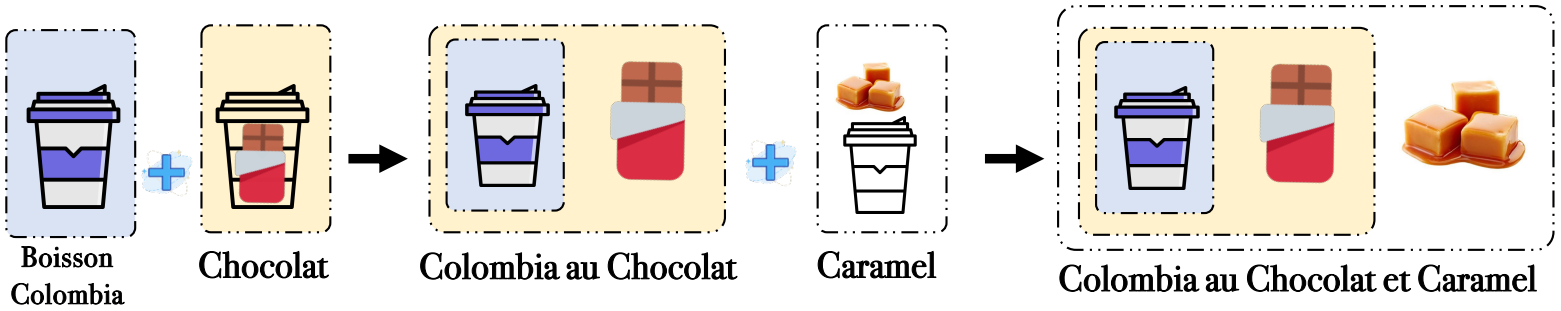
Le pattern Décorateur attache dynamiquement des responsabilités supplémentaires a un objet.
Il fournit une alternative souple à la dérivation, pour étendre les fonctionnalités.



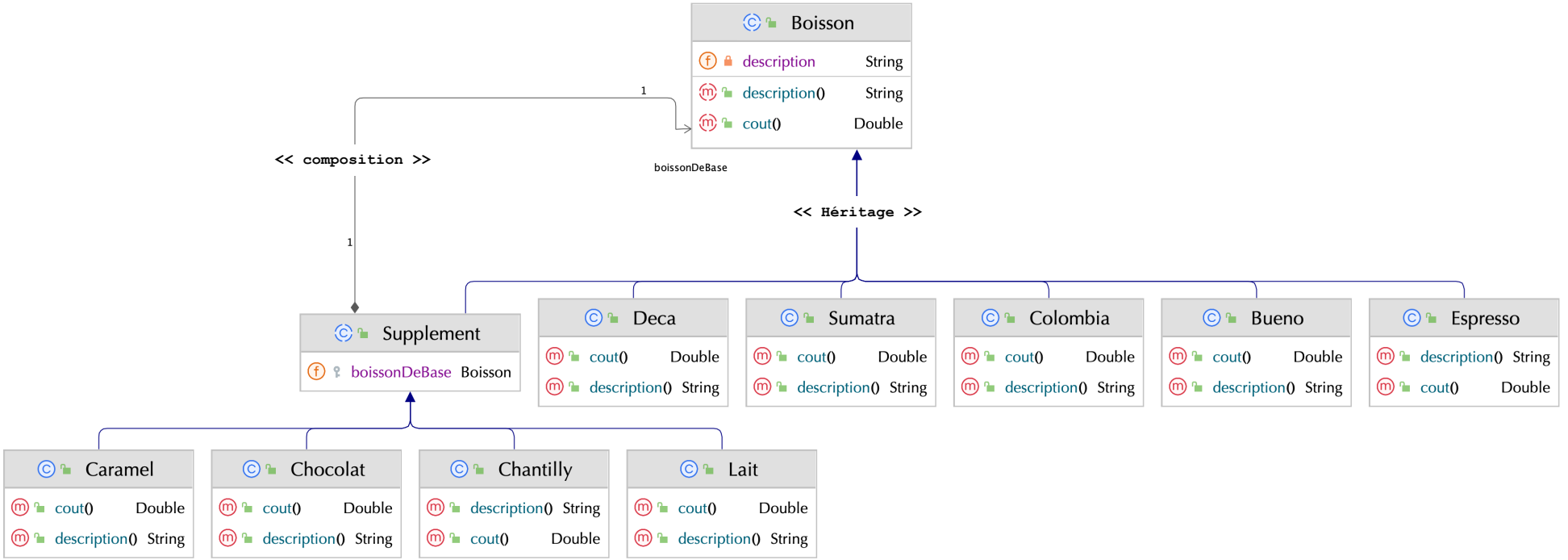
Le principe est simple : l'objet de base et les objets qui le décorent doivent être du même type, et ce, toujours pour la même raison, le polymorphisme.

les objets qui vont décorer notre **Boisson** posséderont la même méthode **cout()** et **description()** que notre objet principal, et nous allons faire fondre cet objet dans les autres.
Cela signifie que nos objets qui vont servir de décorateurs comporteront une instance de type **Boisson**; ils vont englober les instances les unes après les autres et du coup, nous pourrons appeler la méthode **cout**() et **description()** de manière récursive !

Vous pouvez voir les décorateurs comme des poupées russes : il est possible de mettre une poupée dans une autre.
Cela signifie que si nous décorons notre **Boisson** avec un objet **Supplément-Chocolat** et un objet **Supplément-Caramel**, la situation pourrait être symbolisée par la figure suivante :

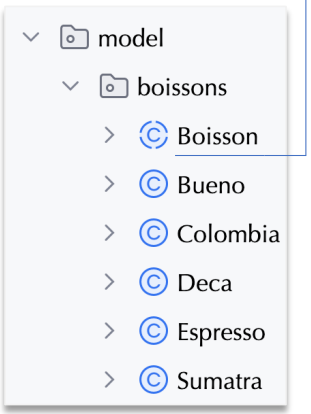


Pattern Décorateur



POO : Programmation Orientée Objet
Pattern Décorateur

Implémentation




```java
public abstract class Boisson {
    no usages
    private String description;
    9 implementations
    public abstract Double cout();
    4 usages   9 implementations
    public abstract String description();
    @Override
    public final String toString() { return "| " + this.description(); }
    @Override
    public boolean equals(Object autre) {...}
    @Override
    public int hashCode() { return Objects.hash(description()); }
}
```

```java
public class Bueno extends  Boisson{
    @Override
    public Double cout() { return 25.0; }
    @Override
    public String description() { return "BUENO-CAFE"; }
}
```

```java
public class Colombia extends  Boisson{
    @Override
    public Double cout() { return 15.0; }
    @Override
    public String description() { return "COLOMBIA-CAFE"; }
}
```

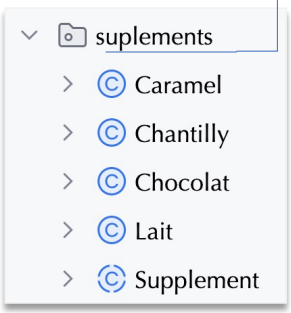
```java
public class Deca extends  Boisson{
    @Override
    public Double cout() { return 20.0; }
    @Override
    public String description() { return "DECA-CAFE"; }
}
```

```java
public class Sumatra extends  Boisson{
    @Override
    public Double cout() { return 18.0; }
    @Override
    public String description() { return "SUMATRA-CAFE"; }
}
```

```java
public class Espresso extends  Boisson{
    @Override
    public Double cout() { return 13.0; }
    @Override
    public String description() { return "ESPRESSO-CAFE"; }
}
```

POO : Programmation Orientée Objet
Pattern Décorateur

Implémentation




```java
public abstract class Supplement extends Boisson {
    9 usages
    protected Boisson boissonDeBase;
    4 usages
    public Supplement(Boisson boissonDeBase){

            this.boissonDeBase = boissonDeBase;
    }

}
```

```java
public class Caramel extends Supplement{
    4 usages
    public Caramel(Boisson boissonDeBase){super(boissonDeBase);}
    @Override
    public Double cout() { return 4.0 + boissonDeBase.cout(); }
    @Override
    public String description() { return "Caramel(" + boissonDeBase.description() + ")"; }
}
```

```java
public class Chocolat extends Supplement{
    10 usages
    public Chocolat(Boisson boissonDeBase){ super(boissonDeBase);}
    @Override
    public Double cout() { return 6.0 + boissonDeBase.cout(); }
    @Override
    public String description() { return "Chocolat(" + boissonDeBase.description()+")"; }
}
```

```java
public class Lait extends Supplement{
    no usages
    public Lait(Boisson boissonDeBase){ super(boissonDeBase); }
    @Override
    public Double cout() { return 5.0 + boissonDeBase.cout(); }

    @Override
    public String description() { return "Lait(" + boissonDeBase.description() + ")"; }
}
```

```java
public class Chantilly extends Supplement{
    no usages
    public Chantilly(Boisson boissonDeBase){super(boissonDeBase);}
    @Override
    public Double cout() { return 7.0 + boissonDeBase.cout(); }
    @Override
    public String description() { return "CrèmeChantilly(" + boissonDeBase.description()+")"; }
}
```