Cours - Patron de conception - #3

Décorateur

Guillaume Santini

18 janvier 2024

Plan

Motivations

Objectifs

Extensible : ajouter/modifier les comportements d'une classe X,

Dynamique : le faire en cours d'exécution et non à la programmation/compilation,

Robuste : sans affecter l'exécution des programmes appelant la classe X.

Motivations

Objectifs

Extensible : ajouter/modifier les comportements d'une classe X,

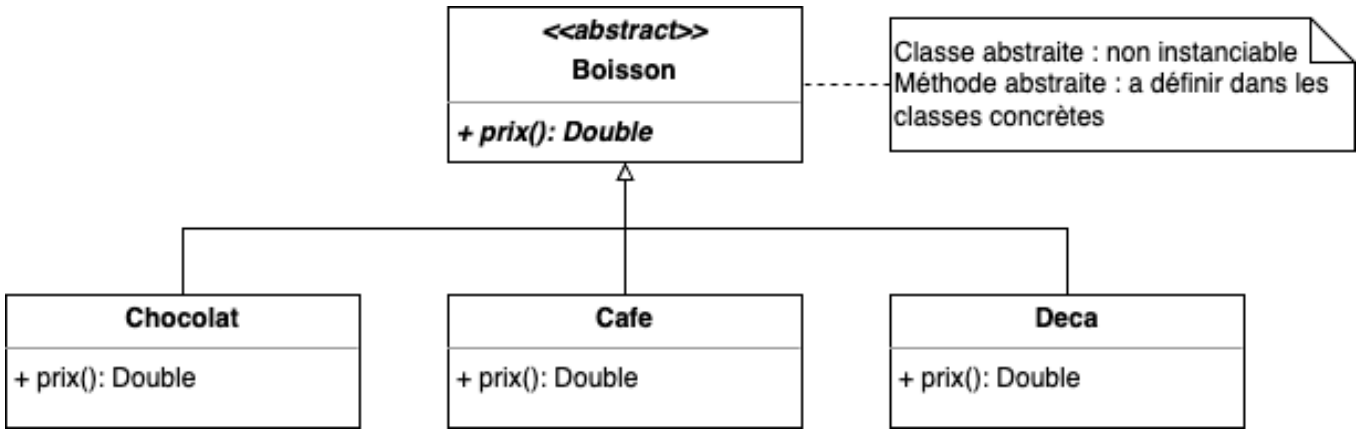
Dynamique : le faire en cours d'exécution et non à la programmation/compilation,

Robuste : sans affecter l'exécution des programmes appelant la classe X.

Principe SOLID Open/Closed

- sans modifier la classe X et
- sans modifier les classes qui utilisent la classe X.

Cas d'étude

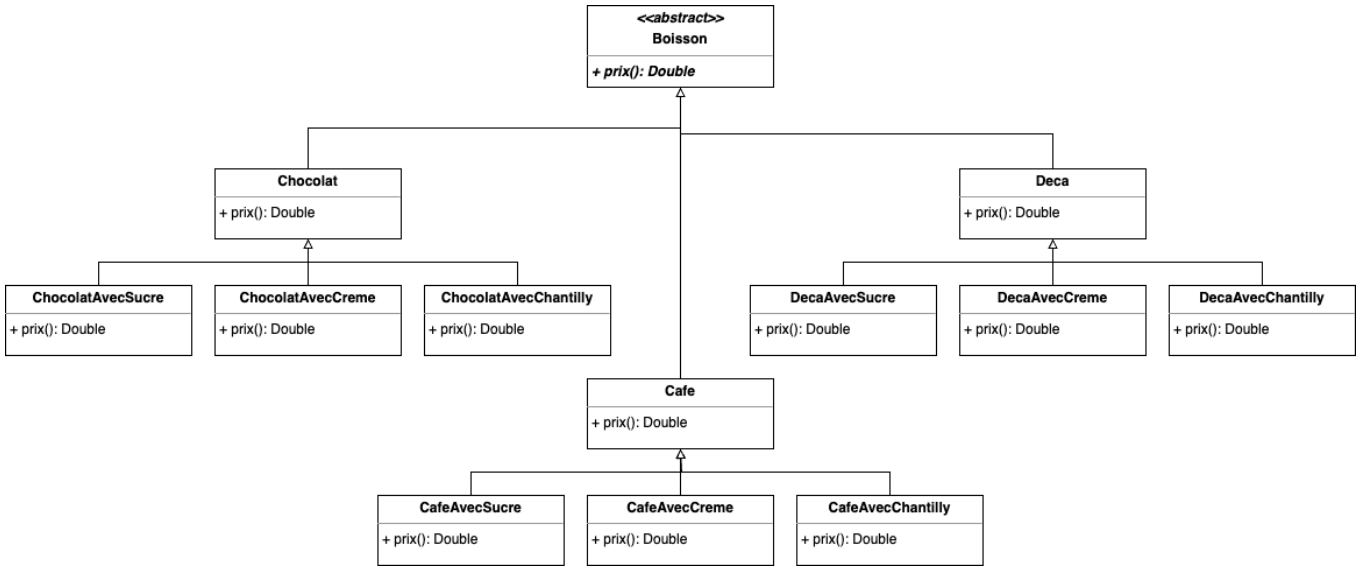


Modélisation d'un système de tarification des boissons d'un distributeur automatique

- des boissons qui ont chacun un prix de base,
- des ingrédients optionnels en nombres variables (*e.g.* `Sucre`, `Creme`, `Chantilly`, ...) qui ont chacun un coût,

$\Rightarrow$ Calculer le prix total de la boisson commandée.

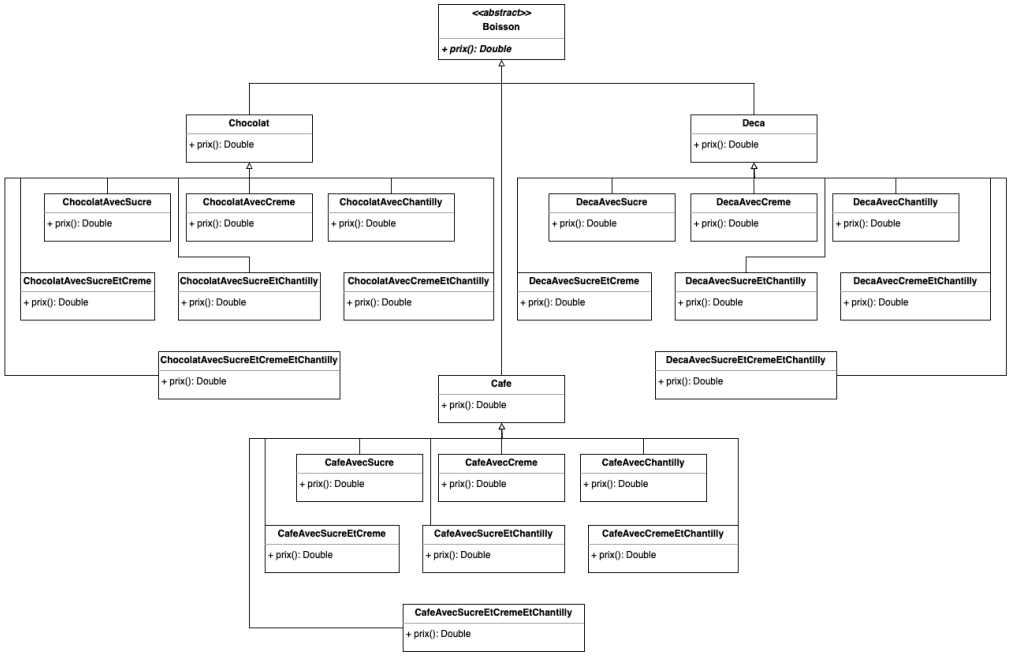
Étendre les comportements par dérivation/spécialisation



Héritage et polymorphisme de redéfinition de la méthode `prix()`

- Des classes pour redéfinir le prix de la boisson en fonction de l'ingrédient ajouté.

Étendre les comportements par dérivation/spécialisation



Héritage et polymorphisme de redéfinition de la méthode `prix()`

- Des classes pour redéfinir le prix de la boisson en fonction de l'ingrédient ajouté.
- Prévoir les combinaisons possibles d'ingrédients.

Étendre les comportements par dérivation/spécialisation

Explosion combinatoire

- pour B classes de base et N ingrédients possibles il faut définir $B * 2^N$ classes différentes,
- pour 10 classes de base et 10 ingrédients cela fait $10 * 2^{10} = 10240$ classes.
- et l'on ne prend pas en compte la possibilité d'avoir plusieurs fois le même ingrédient (*e.g.* double dose de sucre).

Étendre les comportements par dérivation/spécialisation

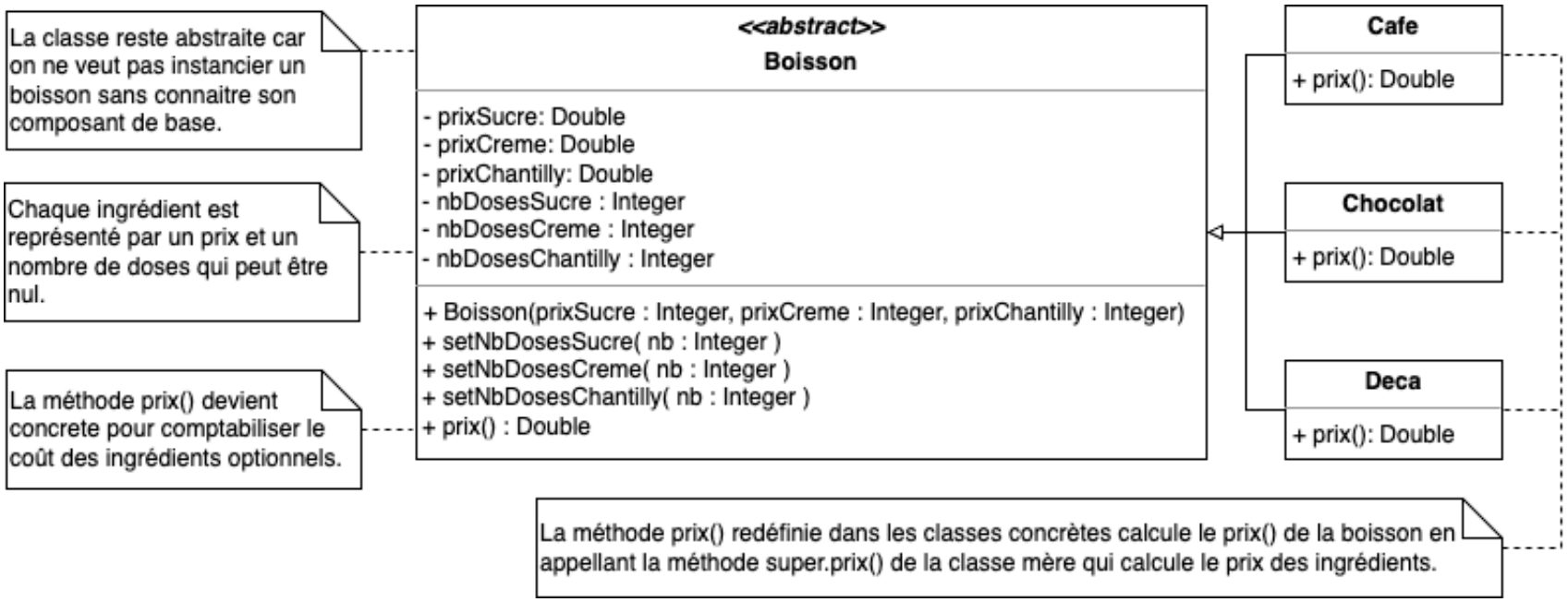
Explosion combinatoire

- pour B classes de base et N ingrédients possibles il faut définir $B * 2^N$ classes différentes,
- pour 10 classes de base et 10 ingrédients cela fait $10 * 2^{10} = 10240$ classes.
- et l'on ne prend pas en compte la possibilité d'avoir plusieurs fois le même ingrédient (*e.g.* double dose de sucre).

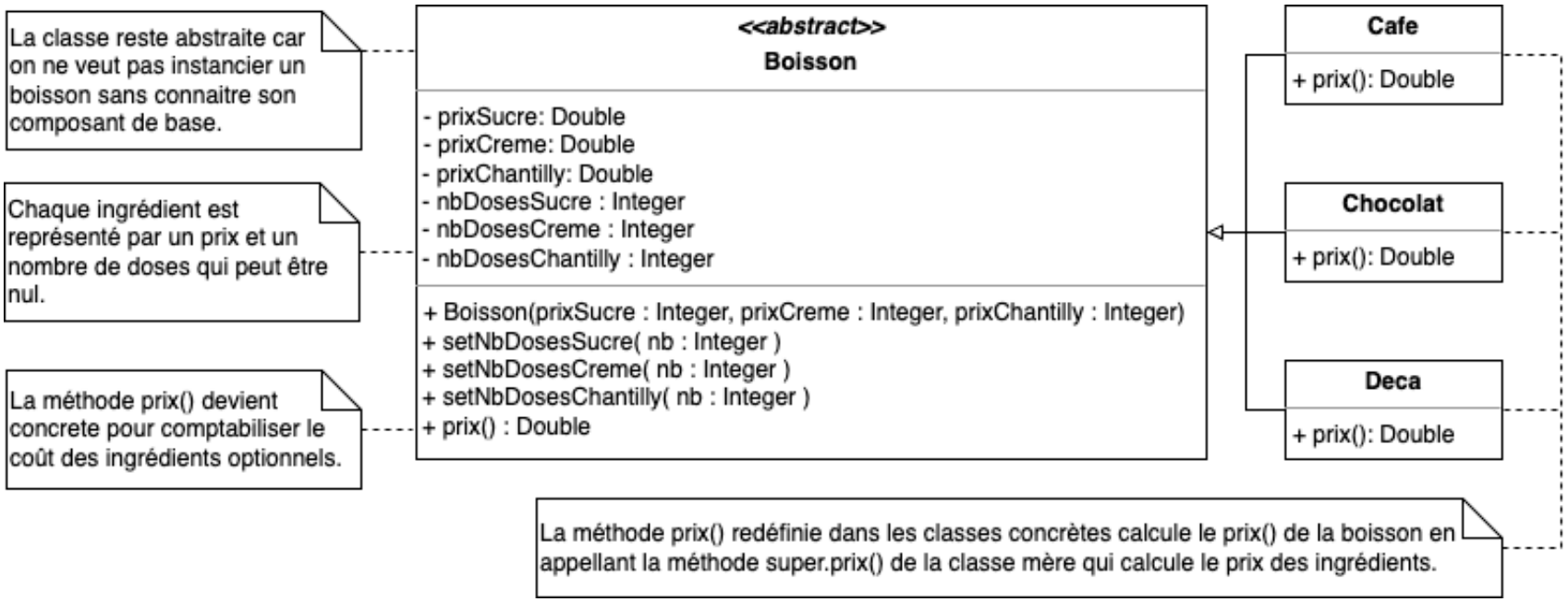
Flexibilité/Maintenabilité

- Si le prix du sucre change
  $\Rightarrow$ il faut modifier toutes les classes qui portent la responsabilité de comptabiliser cet ingrédient.
- Si on souhaite ajouter un nouvel ingrédient (*e.g.* `Chamallow`)
  $\Rightarrow$ il faut ajouter toutes les classes de combinaisons...

Complexifier la classe



Complexifier la classe



Principes SOLID

Open/Closed Ajout d'un nouvel ingrédient ⇒ modification de `Boisson`,

Interface Segregation Une nouvelle classe `The` hérite de méthodes dont elle n'a pas à dépendre (*e.g.* `nbDosesChantilly()` : pas de chantilly dans le thé).

Mise en œuvre des principes SOLID

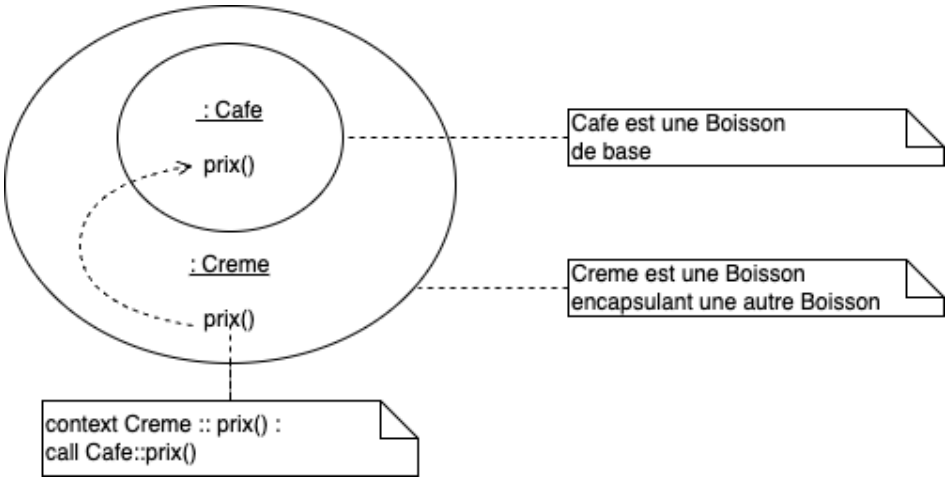
Open/Closed

- L'ajout d'un ingrédient ne doit pas demander de modifier les classes déjà développées et testées,
- On doit pouvoir étendre/modifier le comportement facilement sans toucher à ce qui ne change pas.

Interface Segregation

- Le code doit rester modulaire et
- une classe ne doit pas hériter de responsabilités (*e.g.* méthodes) dont elle n'a pas à dépendre.

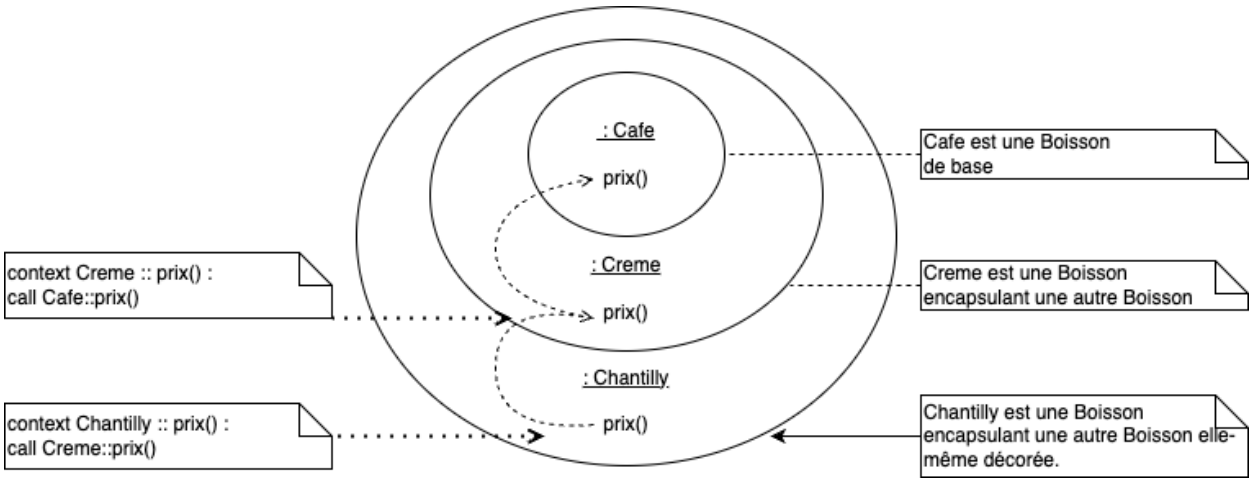
Encapsuler



Ajouter des comportements (`prix()`)

- Modéliser le `Sucre`, la `Creme` et la `Chantilly` comme des décorations de la `Boisson`,
- La `Boisson` décorée doit aussi être une `Boisson`.

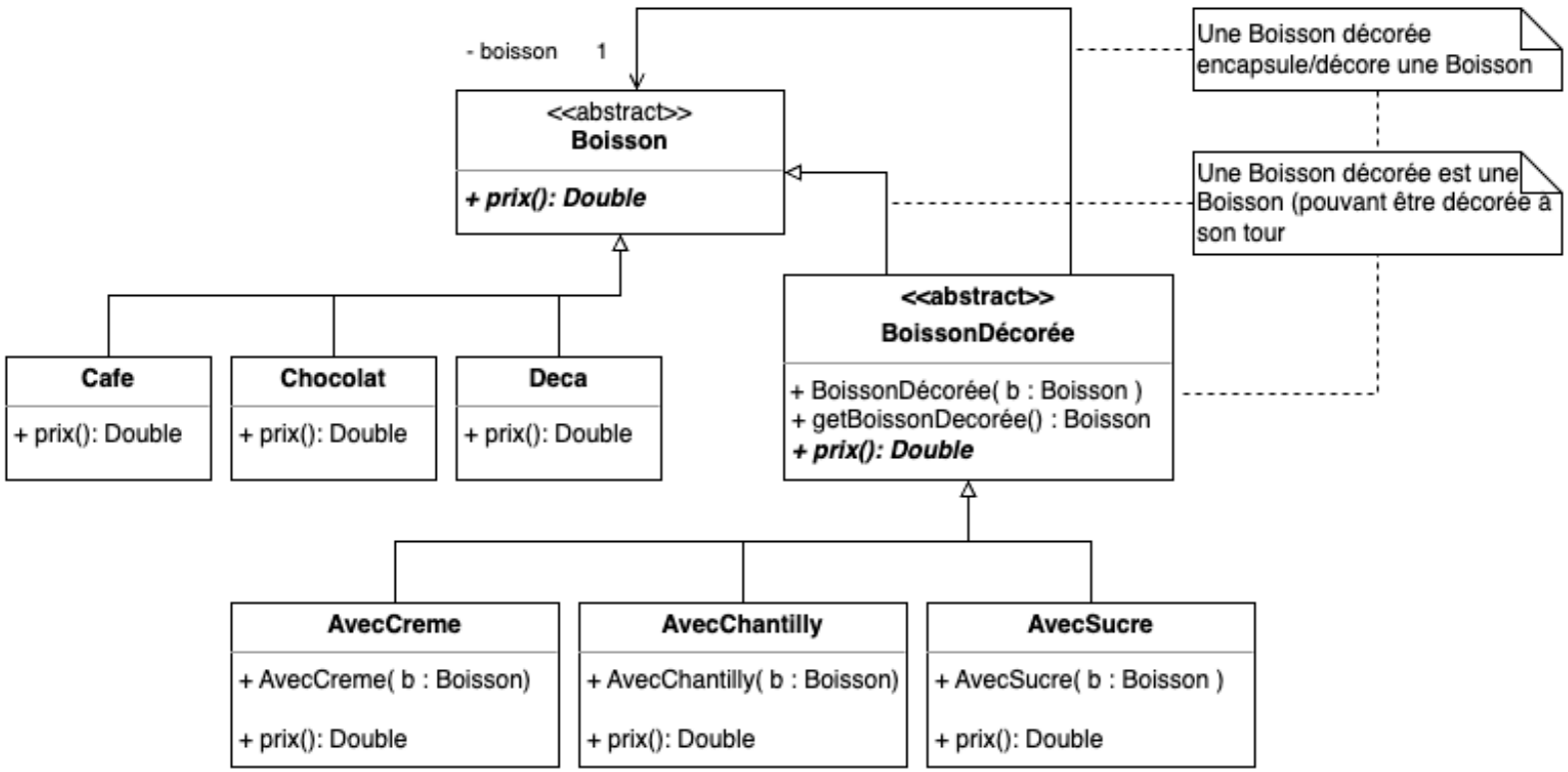
Encapsuler



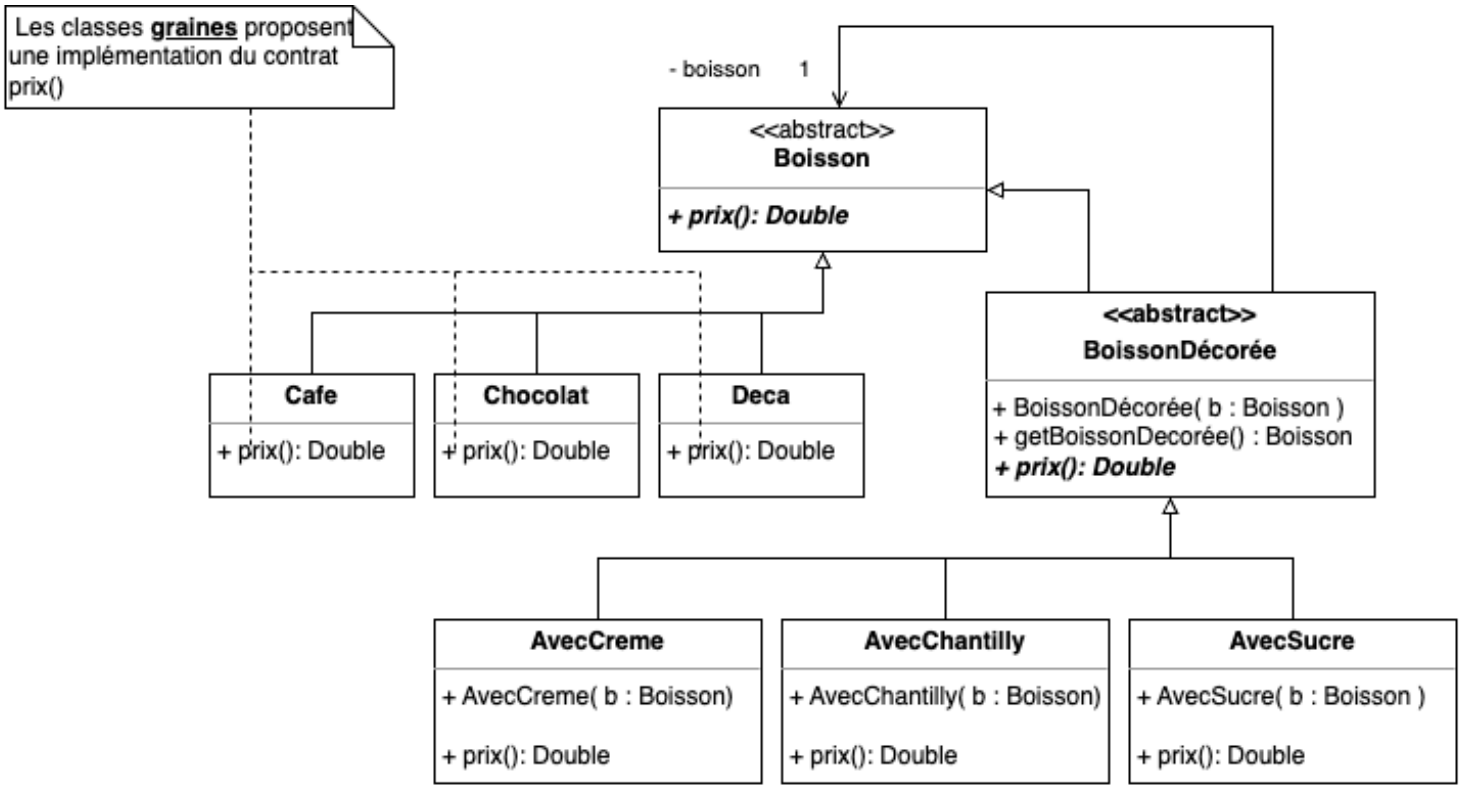
Ajouter des comportements (`prix()`)

- Les décorateurs ajoutent leur comportement à la `Boisson` décorée (ajoute leur `prix()`),
- Le calcul du `prix()` de la `Boisson` décorée délègue en partie le calcul du `prix()` à la `Boisson` qu'elle décore.

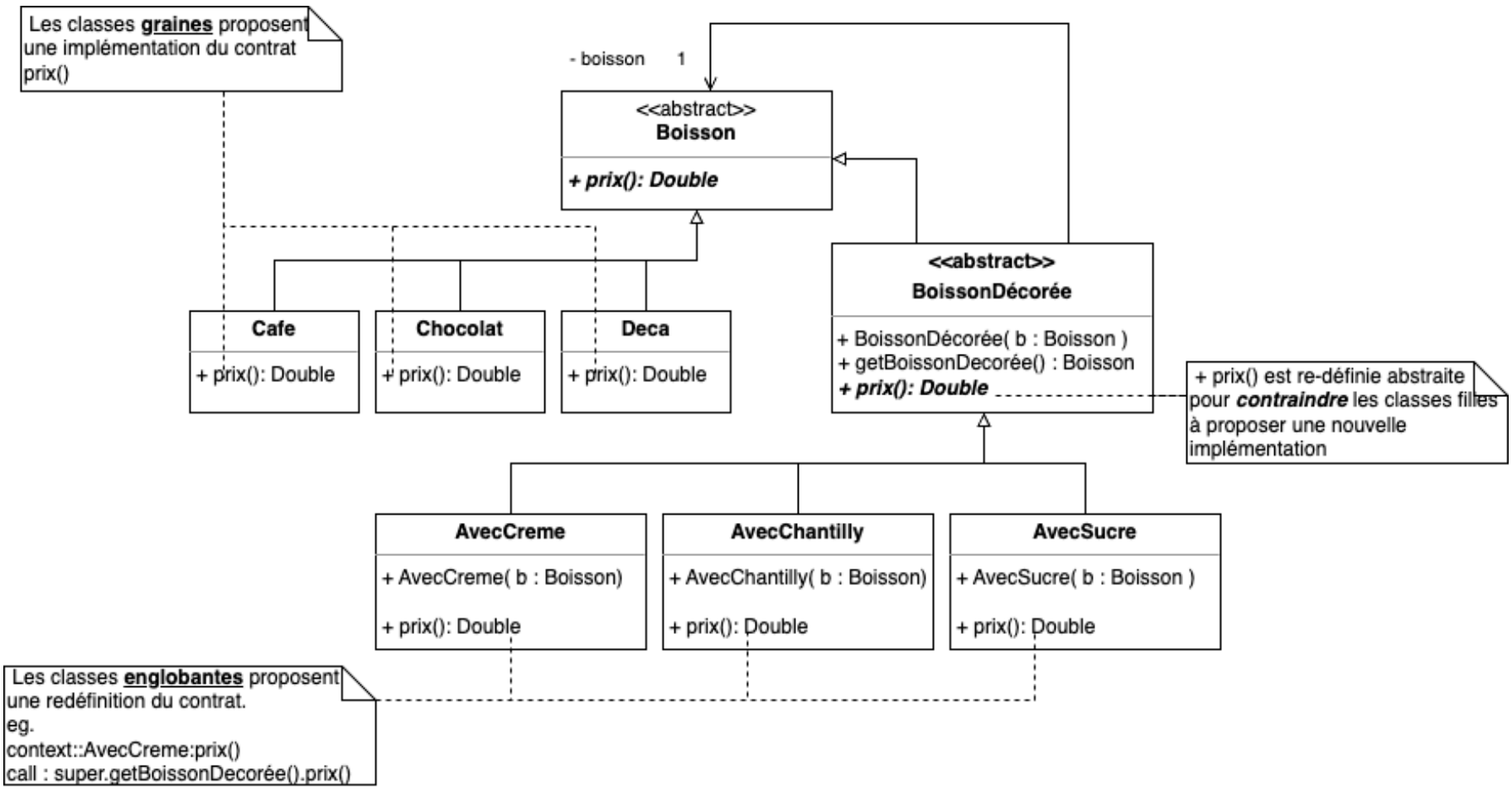
La Boisson et les ingrédients qui la décorent



La Boisson et les ingrédients qui la décorent



La Boisson et les ingrédients qui la décorent



Implémentation #1

```
public abstract class Boisson {
  public Boisson(){}
  public abstract Double prix();
}
```

Déclaration du contrat

- Le contrat est abstrait (à définir),
- La classe ne peut être instanciée.
- Le constructeur par défaut est défini et ne fait rien (surtout si la classe `Composant` propose une structure de données).

```
public class Cafe extends Boisson {
  public Cafe(){
    super();
  }
  public Double prix() {
    return 0.6 ;
  }
}
```

Classe graine

- La classe peut être instanciée,
- elle fournit une implémentation pour le contrat.

Implémentation #2

```
public abstract BoissonDecoree extends Boisson {


  private Boisson boisson ;




  public BoissonDecoree( Boisson b ) {
    super();
    this.boisson = b;
  }

  public Boisson getBoissonDecoree() {
    return this.boisson ;
  }



  public abstract Double prix() ;

}
```

Gestion de l'encapsulation

- La variable d'instance `boisson` permet la composition avec la `Boisson` qui est décorée,
- Le constructeur :
  - appel le constructeur par défaut de la classe mère qui ne fait rien,
  - initialise la composition (encapsulation),
- `getBoissonDecoree()` donne un accès à la `Boisson` décorée.
- le contrat `prix()` reste abstrait (ce n'est pas la responsabilité de cette classe).

Implémentation #3

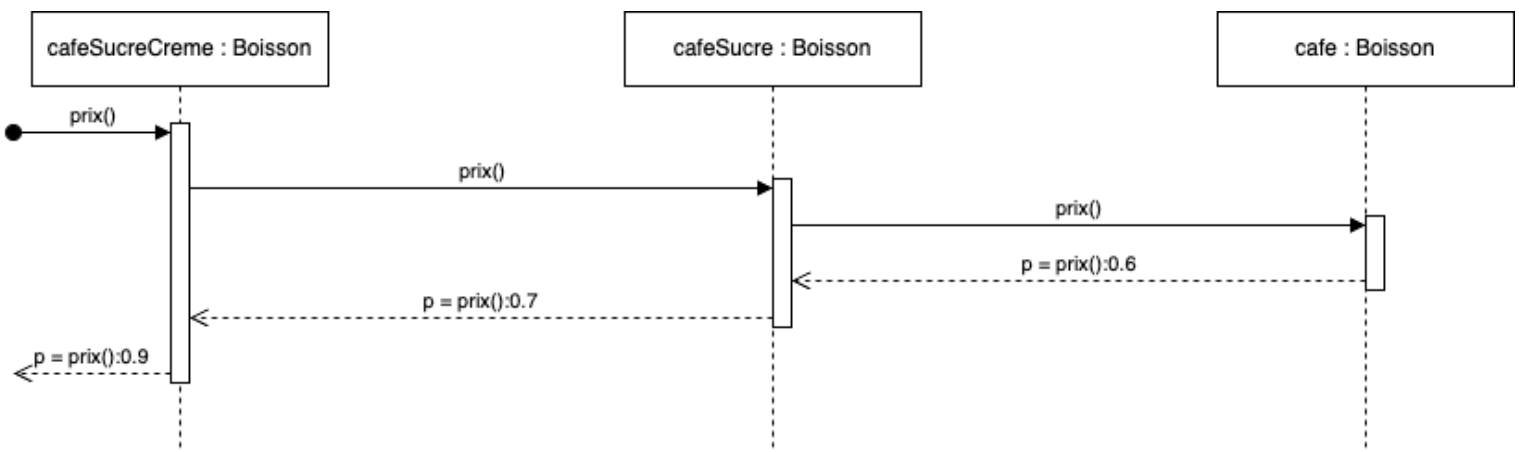
```
public class AvecSucre extends BoissonDecoree {

  public AvecSucre( Boisson b ) {
    super( b ) ;
  }

  public Double prix() {
    Double p = this.getBoissonDecoree().prix() ;
    return p + 0.1 ;
  }

}


public class AvecCreme extends BoissonDecoree {

  public AvecCreme( Boisson b ) {
    super( b ) ;
  }

  public Double prix() {
    Double p = this.getBoissonDecoree().prix() ;
    return p + 0.2 ;
  }

}
```

Appels en cascade

- Le constructeur initialise la composition (encapsulation),
- l'exécution du contrat `prix()` appelle le contrat de l'instance encapsulée et y ajoute son comportement propre.

Utilisation/Éxécution

```
public class Test {

  public static void main( String [] args ) {

  Boisson cafe = new Cafe() ;
  System.out.println( cafe.prix() ) ;

  Boisson cafeSucre = new AvecSucre( cafe ) ;
  System.out.println( cafeSucre.prix() ) ;

  Boisson cafeSucrecreme = new AvecCreme(cafeSucre) ;
  System.out.println( cafeSucrecreme.prix() ) ;
  }
}
```



Utilisation/Éxécution

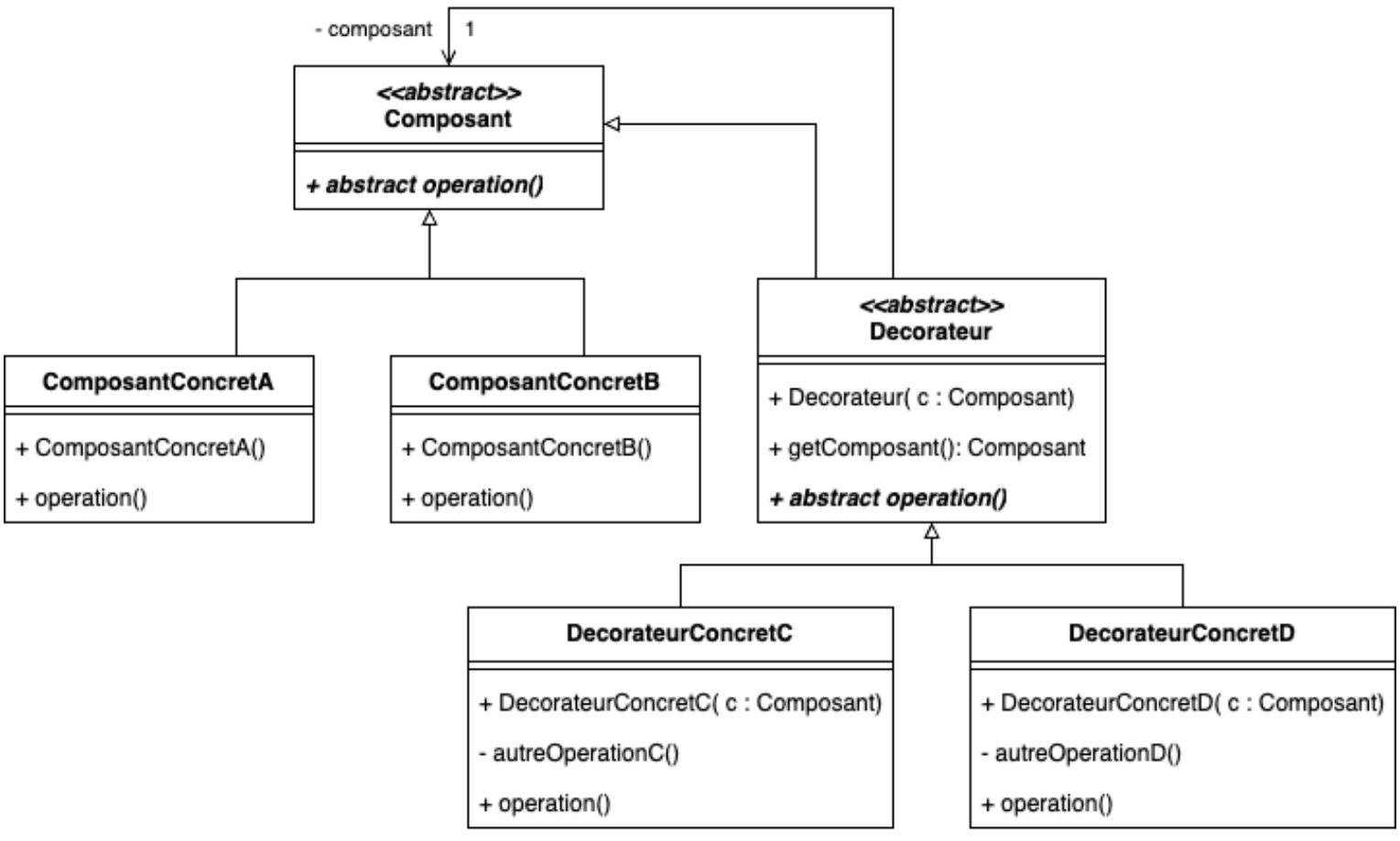
```
public class Test {

  public static void main( String [] args ) {

  Boisson cafe = new Cafe() ;
  System.out.println( cafe.prix() ) ;

  Boisson cafeSucre = new AvecSucre( cafe ) ;
  System.out.println( cafeSucre.prix() ) ;

  Boisson cafeSucrecreme = new AvecCreme(cafeSucre) ;
  System.out.println( cafeSucrecreme() ) ;
  }
}
```

Objectifs remplis

L'ajout de comportement

- est dynamique (à l'exécution et non à la compilation),
- et il ne modifie pas les programmes qui appellent la classe `Boisson` puisque la classe `BoissonDecroree` propose la même vue publique (même contrat).

Le patron de conception Décorateur



Le patron de conception Décorateur

Règles de construction

- Le `Decorateur` propose la même vue publique (le même contrat) que le `Composant`,
- Le `Decorateur` est associé par composition au `Composant` qu'il décore,
- Le constructeur du `Decorateur` se contente seulement d'initialiser la composition entre l'objet décoré et le décorateur.
- L'`opération()` :
  - dans le `ComposantConcret` est implémenté un comportement de base,
  - dans le `DecorateurConcret` le comportement appelle le comportement de l'instance qu'il encapsule auquel il ajoute ses opérations (avant ou après).
- Les relations d'héritage assurent que le `DecorateurConcret` est une instance fille de la classe décorée `Composant`.

Implémentation du patron

```
public abstract class Composant {

   public Composant(){
   }

   public abstract type operation( args ) ;

}
```

Définition du contrat abstrait du `Composant`

- Le contrat `operation()` qui sera augmenté par les décorations est déclaré de façon abstraite. Il sera implémenté par les classes concrètes :
  - `ComposantConcret` et
  - `DecorateurConcret`.
- Le constructeur par défaut de la classe est défini et ne fait rien. Si cette classe contient une ou plusieurs variables d'instance, elles sont initialisées par un autre constructeur. Le constructeur par défaut sera appelé par le constructeur du `Decorateur` qui ne doit pas surcharger la mémoire ni définir de valeurs pour ces variables d'instances.

Implémentation du patron

```
public abstract Decorateur extends Composant {

  private Composant composant ;

  public Decorateur( Composant c) {
    super();
    this.composant = c;
  }

  public Composant getComposant() {
    return this.composant ;
  }
}
```

Gestion de la composition

- La classe définit un lien de composition vers le `Composant` décoré qui peut donc être un `ComposantConcret` ou un `DecorateurConcret`.
- Le constructeur initialise cette composition et rien d'autre. Il appelle le constructeur par défaut de la classe mère `Composant` qui ne fait rien (surtout pas d'initialisation de variables structurelles qui sont laissées aux classes concrètes de `Composant`.
- La méthode `getComposant` permet d'accéder à l'instance décorée.

Implémentation du patron

```
public class ComposantConcretA extends Composant {

  public ComposantConcretA() {
    super();
    ...
  }

  public abstract type operation( args){
    # comportement de base
  }

}
```

Redéfinition de l'`operation()`

- Le contrat de la classe `Composant` est redéfini en
- ajoutant avant ou après l'appel de l'`operation()` de l'objet décoré, les comportements du `DecorateurConcret`.

Implémentation du patron

```
public DecorateurConcretC extends Decorateur {

  public DecorateurConcretC( Composant c){
    super(c);
  }

  public type operation() {
    super.getComposant().operation() ;
    # autres instructions ajoutees
    # a l'operation
  }
}
```

Définition de l'operation() graine

- Le DecorateurConcret propose une implémentation *graine* du contrat operation().

Principes de conception

Principes généraux mis en œuvre

- Préférer la composition à l'héritage :
  - L'héritage manque de souplesse et impose ce qui se définit dans la classe mère à sa dépendance,
  - La composition permet de séparer les comportements.
  - Le patron de conception Décorateur utilise *à la fois* la composition et l'héritage.

Principe SOLID Open/Closed

Sans modifier les classes développées (et testées), on peut facilement ajouter une classe concrète :

- fille de `Composant` pour ajouter une classe de base,
- fille de `Decorateur` pour ajouter une classe de décoration (d'ajout de comportements).

Principes de conception

Principe SOLID Interface segregation

Les contrats sont déclarés de façon abstraite dans des classes séparées :

- la vue publique dans la classe `Composant`,
- la gestion de la composition dans la classe `Decorateur` (appel de l'opération de l'instance encapsulée).

Principe SOLID Single Responsability

Chaque classe n'a qu'une seule responsabilité

`Composant` déclarer le contrat,

`ComposantConcret` implémenter le contrat,

`Decorateur` gérer la composition d'encapsulation,

`DecorateurConcret` ajouter des comportements.

Credits

- Support soumis à copyleft : 
- Le cours de D. Bouthinon,
- **Design patterns — Tête la première**, E. & E. Freeman, ed. O'Reilly.