Devoir Maison JAVA - Rapport

Gün, Alexis Cordier

Avril 2022

1 Nos choix de modélisation

Résumons d'abord ce qui était demandé dans le DM. Il nous fallait créer divers Sandwich : des sandwich **"normaux"**, **végétarien** et **vegan**. On doit pouvoir chercher l'ingrédient le plus calorique d'un sandwich, ajouter un Ingrédient, savoir si deux sandwich ont un ingrédient en commun, ainsi qu'échanger l'ingrédient d'un sandwich avec celui d'un autre.

1.1 Comprendre les problèmes

Si l'on a pris végétarien et vegan ce n'est pas pour rien. En effet un vegan a plus de restrictions qu'un végétarien qui a lui même plus de restriction qu'un non végétarienne.

Il y a donc un certaine hiérarchie a respecter. Par exemple un sandwich végétarien acceptera volontiers des ingrédients vegans et végétarien. Alors que les sandwich vegans eux n'accepterons pas les ingrédients végétariens car un oeuf par exemple est bon pour un végétarien mais pas pour un vegan.

1.2 Résolution des problèmes

Ainsi au moment de la création d'un Ingrédient (Pain ou Sauce compris), il faudra toujours veiller a se poser la question de sa classification. ***Exemple :*** *le boeuf est un ingrédient, la salade est un ingrédientVegan et un oeuf est un ingredientVegetarien*

Nous nous sommes donc basé sur le concept d'héritage et de vérification du type d'interface pour vérifier la compatibilité entre les ingrédients en fonction du sandwich.

2 Nos choix de programmation

2.1 Des interfaces

Pour créer la hiérarchie nécessaire à la résolutions des problèmes posés, nous avons créé trois interfaces : Aliment, AlimentVégétariens et AlimentVegan, chacune des ces interfaces implémentes respectivement une classe au moins (l'UML permet de bien le visualiser). Les interfaces sont extrêmement utiles, par exemple, quand on arrive à la méthode "moveIngTo", les "instanceof" de java permettent de checker si oui ou non un aliment est compatible pour aller vers le sandwich cible.

```
public void moveIngTo(I i, Sandwich<?, ?, ? super I> s) {
  if (!this.ingredients.contains(i)) {
    System.err.println(x: "Ingredient n'est pas dans le sandwich !");
  } else {
    if ((s instanceof SandwichVegetarien)
        && (!(i instanceof IngredientVegetarien) || !(i instanceof IngredientVegan))) {
      System.err.println(x: "Ingredient n'est pas compatible avec le sandwich !");
    } else if ((s instanceof SandwichVegan) && !(i instanceof IngredientVegan)) {
      System.err.println(x: "Ingredient n'est pas compatible avec le sandwich !");
    } else {
      s.ingredients.add(i);
      this.ingredients.remove(i);
      System.out.println(x: "Moved!");
    }
  }
}
```

Figure 1: Le besoin d'interfaces

2.2 Des classes génériques

Avec la création de classes génériques comme l'exemple ci-après, et avec l'extension de chaque type, nous pouvons vérifier qu'un sandwich est au minimum bien composé de pain et d'une sauce au moins, qui ne sont pas des ingrédients quelconque. Le dernier type doit être un Ingredient que l'on a créé par exemple.

```
public class Sandwich<P extends Pain, S extends Sauce, I extends Ingredient> implements Iterable<Ingredient> {
  protected String nom;
  protected P pain;
  protected S sauce;
  protected ArrayList<I> ingredients;

  public Sandwich(String nom, P pain, S sauce) {
    this.nom = nom;
    this.pain = pain;
    this.sauce = sauce;
    this.ingredients = new ArrayList<>();
  }
```

Figure 2: Exemple de classe générique

2.3 Des intérateurs et streams

Pour la méthode qui cherchait l'ingrédient le plus calorique, les itérateurs et stream étaient vraiment utiles. Comme on peut le voir dans les deux image ci-dessous. Dans les deux cas le but est de parcourir tout les Ingrédients d'un sandwich (pain et sauce comprise) et quand on a trouvé un ingrédient plus calorique que les autres on le renvoie.

L'avantage des itérateurs ou streams est de pouvoir itérer une collection sans avoir a faire le compte dans la fonction. Les streams eux fonctionnent a peu près de la même manière, sauf qu'ici, il faut créer une liste qui contient tous les éléments puis grâce a certaines méthode on peut chercher dans tous les éléments lequel possède me max en kcal.

```
public Ingredient ingredientPlusCaloriqueIterateur() {
  Ingredient i = (this.pain.kcal > this.sauce.kcal) ? this.pain : this.sauce;

  for (Ingredient j : this) {
    if (j.kcal > i.kcal) {
      i = j;
    }
  }

  return i;
}
```

Figure 3: Méthode qui utilise un itérateur

```
public Ingredient ingredientPlusCaloriqueStream() {
  ArrayList<Ingredient> tousIngredients = new ArrayList<>();
  for (int counter = 0; counter < this.ingredients.size(); counter++) {
    tousIngredients.add(ingredients.get(counter));
  }
  tousIngredients.add(pain);
  tousIngredients.add(sauce);

  int maxKcal = tousIngredients.stream().mapToInt(x -> x.kcal).max().getAsInt();
  return tousIngredients.stream().filter(x -> x.kcal == maxKcal).findFirst().get();
}
```

Figure 4: Méthode qui utilise stream