HAI913I
Evolution et restructuration des logiciels

Rendu TP Refactoring

Auteur :

Canta Thomas (21607288)
Fontaine Quentin (21611404)

Master 2 - Génie Logiciel
Faculté des sciences de Montpellier
Année universitaire 2021/2022

Question 1 & 2

Duplication

Le premier refactoring se base sur la duplication de code. Un exemple simple pouvant le décrire est le calcul du prix hors taxes (HT) et toutes taxes confondues (TTC) lors d'un passage en caisse par exemple.

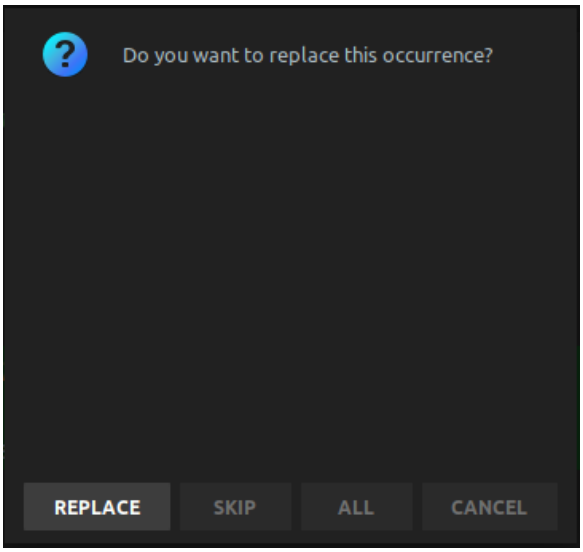
```java
// Calcul le prix total toute taxes comprises (TTC)
public static double getTTC(double tva, double... sweets) {
  double total = 0;
  for (double s : sweets) total += s;
  return total + (total * tva);
}
```

Listing 1 – Calcul du prix TTC

```java
// Calcul la somme des produits hors taxes (HT)
public static double getHT(double ...sweets) {
  double total = 0;
  for (double s : sweets) total += s;
  return total;
}
```

Listing 2 – Calcul du prix HT

On constate que la somme des produits, ici de magnifiques confiseries car on est gourmands, a été dupliquée. En utilisant l'outil de refactorisation fourni par Intellij (Refactor > Find And Replace Code Duplicates) il propose de modifier une occurrence de notre méthode *getHT()* dans *getTTC()*.



Après remplacement notre méthode *getTTC()* à été modifié de la façon suivante :

```java
// Calcul le prix total toute taxes comprises (TTC)
public static double getTTC(double tva, double... sweets) {
  double total = getHT(sweets);
  return total + (total * tva);
}
```

Listing 3 – Calcul du prix TTC

Invert Boolean

Le second refactoring est celui de l'"*invert boolean*". Dans notre exemple, nous avons une classe avec un attribut double, nommé *a*. Une méthode renvoie faux si *a* est compris entre 15 et 100, sinon vrai. Notre but est justement inverse. Nous voulons vrai si *a* est bel et bien dans cette intervalle et faux sinon.

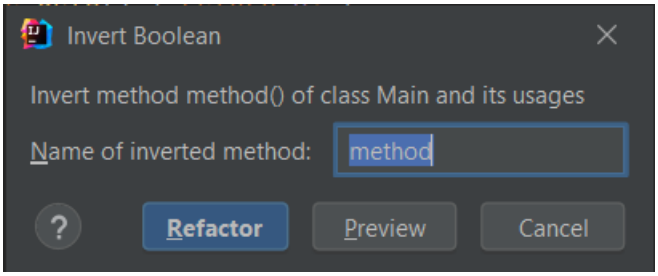
```
0  public class Main {
1    private double a;

3    public boolean method() {
4      if (a > 15 && a < 100) {
5        return false;
6      }
7      return true;
8    }

10   //...getter and setters

12   public static void main(String[] args) {
13     Main m = new Main();
14     m.setA(25);
15     System.out.println("a = " + m.getA() + "\n 15 < a < 100 ? " + m.method());
16   }
17 }
```

Listing 4 – Avant Invert boolean



Le nouveau code obtenu. Après exécution, le main affiche bel et bien true pour a = 25.0.

```
0  public class Main {
1    private double a;

3    public boolean method() {
4      if (a > 15 && a < 100) {
5        return true;
6      }
7      return false;
8    }

10   // getter and setters

12   public static void main(String[] args) {
13     ...
14   }
15 }
```

Listing 5 – Après Invert boolean

Question 3

👉 Substitute Algorithm

La substitution d'algorithme est une méthode permettant de factoriser une partie ou une méthode d'un programme afin d'améliorer la qualité, que ce soit en terme de lisibilité ou, potentiellement, d'optimisation via des fonctions java plus adéquate. Cette re-factorisation n'est pas disponible sur Intellij.

```
function foundPerson(people) {
 for(let i=0; i<people.length; i++) {
  if (people[i] === "Don") {
   return "Don";
  }
  if (people[i] === "John") {
   return "John";
  }
  if (people[i] === "Kent") {
   return "Kent";
  }
 }
 return "";
}
```

```
function foundPerson(people) {
 const candidates = ["Don", "John", "Kent"];
 return people.find(p =>
    candidates.includes(p)) || '';
}
```

👉 Invert boolean

Cette re-factorisation est disponible sur Intellij mais pas dans le catalogue d'Eclipse. Cette refactorisation permet la permutation des booléens à l'intérieur d'une méthode voire même d'une classe comme illustré précédemment.

Question 4

👉 Il est possible d'extraire une interface d'une unique classe, en revanche il est encore impossible de sélectionner tout son code et d'extraire une interface globale à implémenter automatiquement dans les autres méthodes sélectionnées. Voici un exemple d'exécution (cf. Figure 1, 2, 3 page 4). Ceci induit donc que nos interfaces restent dupliquées si ajoutées une par une, en plus d'être pénible à faire.

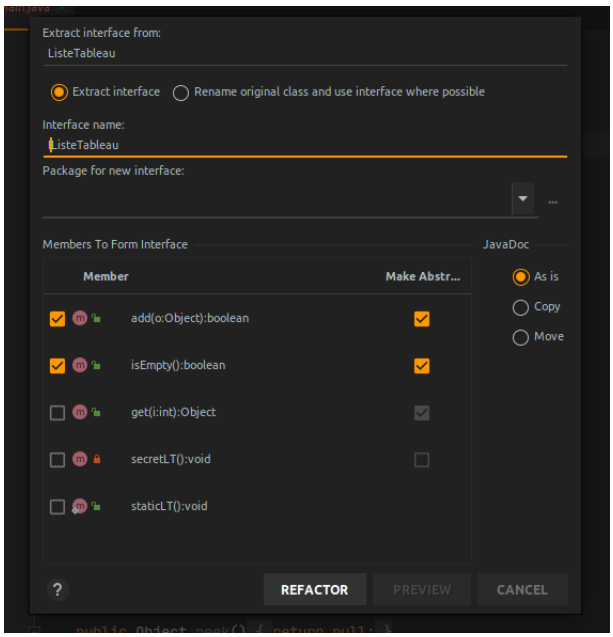


Fig. 1 – Step 1

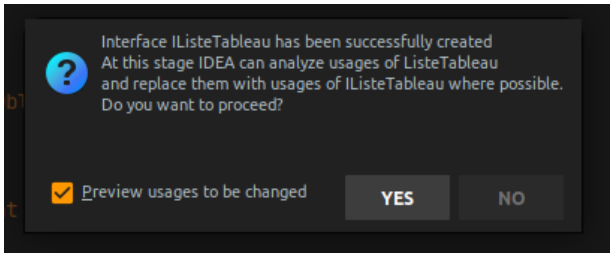


Fig. 2 – Step 2

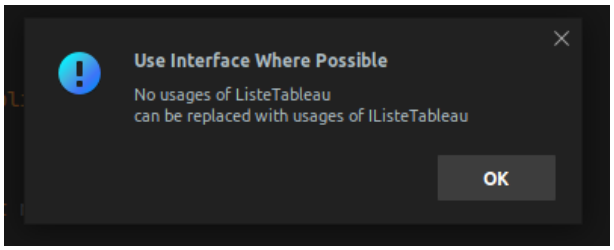


Fig. 3 – Step 3

👉 Voici notre hiérarchie d'interfaces sans duplication.

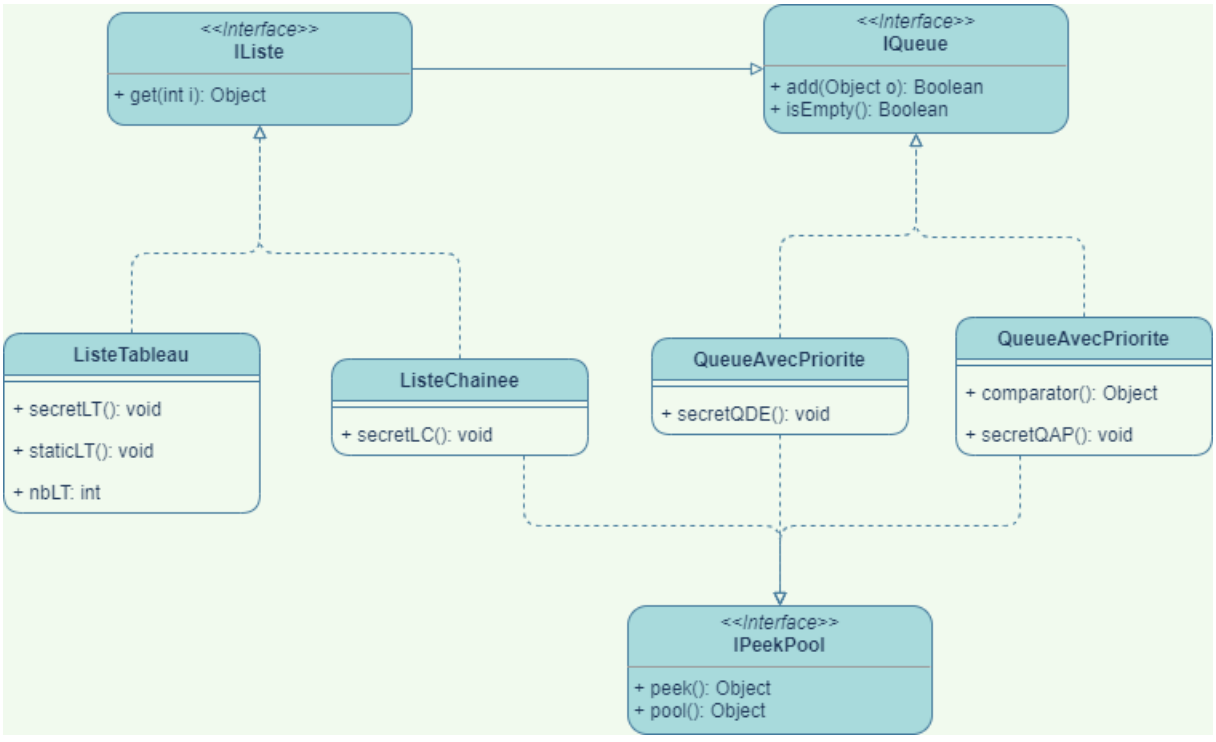


Fig. 4 – Implémentation naïve

👉 Résultat obtenus suite à l'utilisation du "big refactoring"

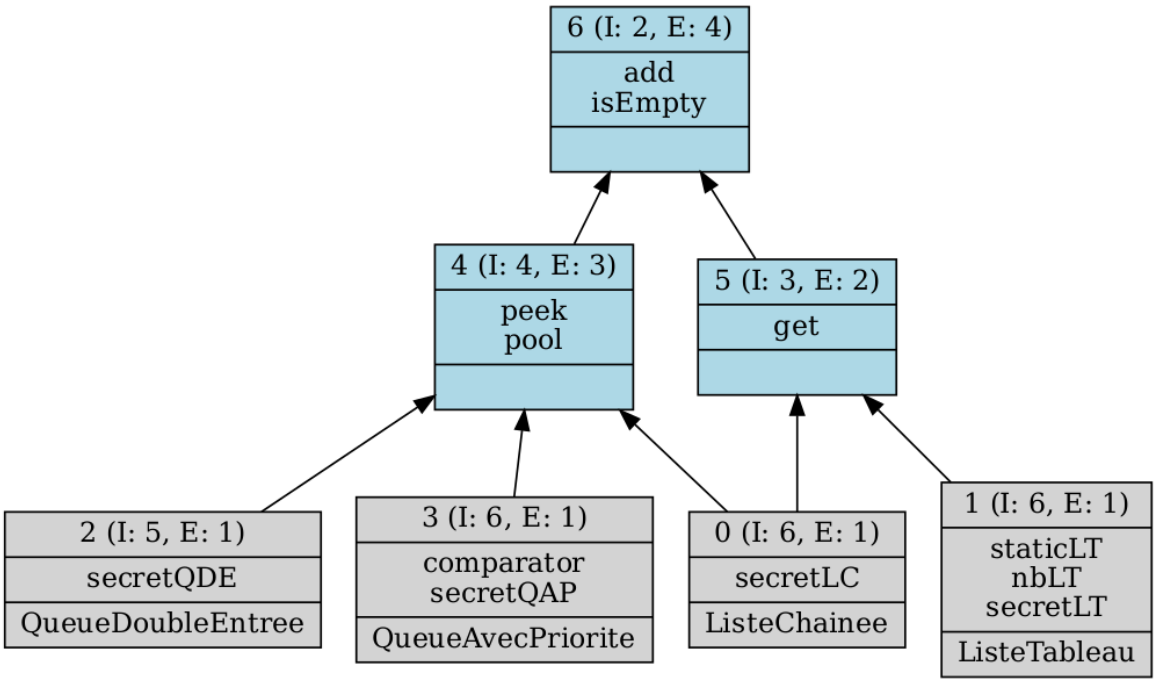


Fig. 5 – big refactoring sur nos classes de tableau

👉 Après comparaison, nous constatons que nous avions quasiment la même architecture, à une exception près, nous n'avons pas étendu la classe IPeekPool avec IQueue. En effet, c'est bien plus malin, mais il était 8h32 du matin quand on a fait notre interface et je n'ai pas pu prendre de café noisettes 😭. Cette méthode permet de faciliter grandement l'analyse formelle de concepts et donc simplifier la maintenance système ou encore la migration logiciel.