Université de Montpellier
M2 Génie Logiciel

TP : Refactoring

HAI913I : Evolution et restructuration des logiciels

Étudiants :
Denis Beauget Hayaat Hebiret

Encadrant :
Marianne Huchard

Année : 2021 – 2022

Table des matières

1 Création de programmes nécessitant un refactoring

1.1 Programme Denis :

Dans le code mis en œuvre par Denis l'objecif est de créer des affichages sur la sortie standard faisant un petit calcul et affichant des éléments. Ici, l'objectif est d'appliquer la méthode de refactoring **Extract method**

```
public class refactMethod {

    public static void main(String[] args) {
        int res = 0;
        double variable = 0.25;
        System.out.println("Ce message aussi court ou
        long soit-il n'a pas grand-chose à faire ici");
        System.out.println("Le résultat est" + (res+variable));
    }
}
```

1.2 Programme Hayaat :

Dans le code mis en œuvre par Hayaat l'objectif est de créer une classe Personne à l'intérieur d'une classe Famille pour y appliquer la méthode de refactoring **Move type to file**

```
public class Famille {

    public String idFoyer;
    public ArrayList<String> roleListe = new ArrayList<String>();
    public ArrayList<Personne> personnes = new ArrayList<Personne>();

    public Famille(String id, ArrayList<Personne> p, ArrayList<String> role){
        this.idFoyer = id;
        this.personnes = p;
        this.roleListe = role;
        }
    public Famille() {}


    public Personne newPersonne(String n, String p, String a) {
        Personne personne = new Personne(n,p,a);
        return personne;
    }

    public class Personne {
        String nom;
        String prenom;
```

```
23          String anniversaire;
24
25          public Personne(String n , String p,String a) {
26              this.nom = n;
27              this.prenom = p;
28              this.anniversaire = a;
29          }
30          public String getNom() {return nom;}
31          public String getPrenom() {return prenom;}
32          public String getAnniversaire() {return anniversaire;}
33      }
34
35      public ArrayList<Personne> getPersonnes(){
36          return this.personnes;
37      }
38
39      public String toString() {
40          String rslt= null;
41          for (int i=0 ; i<personnes.size();i++) {
42              rslt+="Nom : " + personnes.get(i).getNom()+"\n";
43              rslt+="Prenom : " +  personnes.get(i).getPrenom()+"\n";
44              rslt+="Role : " + roleListe.get(i)+"/n";
45              rslt+="Anniversaire : " + personnes.get(i).getAnniversaire()
46              .toString()+"\n";
47              rslt+="idFoyer : " + idFoyer+"\n \n";
48          }
49          return rslt;
50      }
51
52  }
```

2 Application des opérations de refactoring sur les programmes de l'autre personne

2.1 Hayaat sur le code de Denis : Extract method

L'opération appliqué ici est **Extract method** [1] disponible dans le menu "Refactor" d'Eclipse.

On peut également avoir un mécanisme de "Preview" avant de lancer l'opération, cela permet d'avoir un aperçu du résultat avant de valider l'opération.

1. Extract Method : `https://refactoring.com/catalog/extractFunction.html`

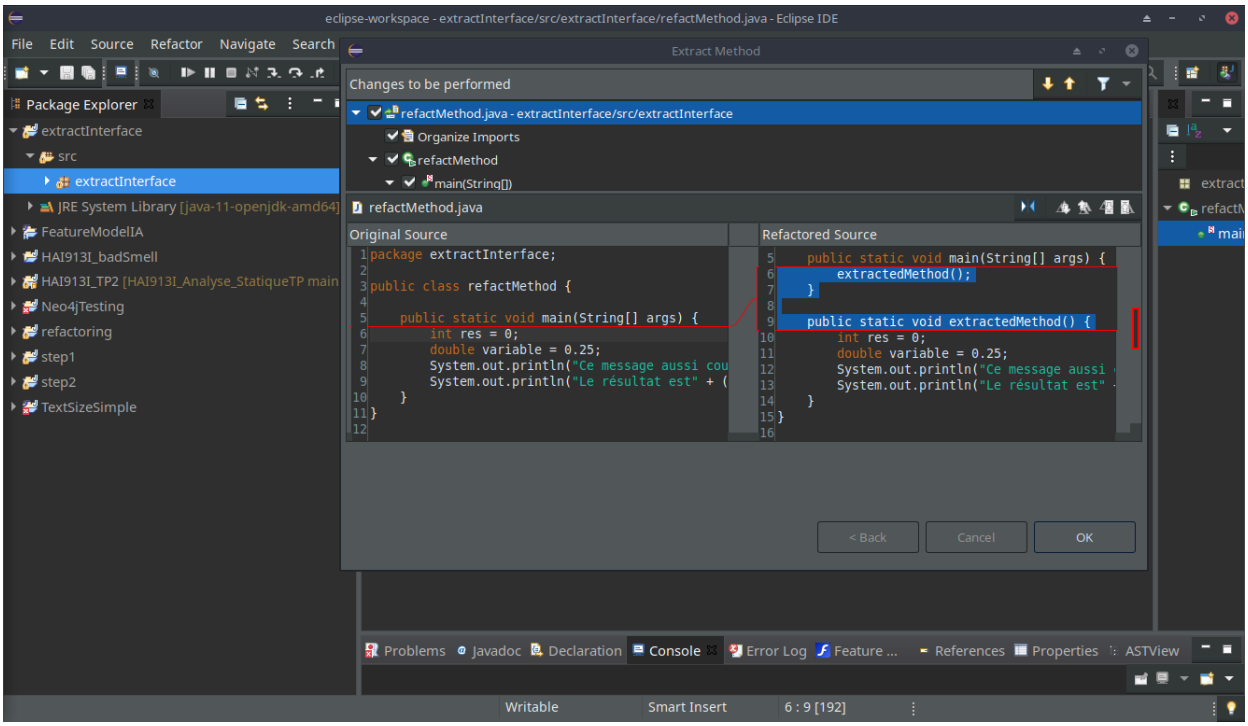

FIGURE 1 – Preview de l'opération sur Eclipse

Avec cette opération nous n'avons pas eu de problème ou de modification à faire après l'application de l'opération. Le main avait le même comportement que précédemment.

```java
//Ici on observe bien que les différents calcul
//et affichage présent dans le main on bien été déplacé dans une méthode.
public class refactMethod {

    public static void main(String[] args) {
        extractedMethod();
    }

    public static void extractedMethod() {
        int res = 0;
        double variable = 0.25;
        System.out.println("Ce message aussi court ou
        long soit-il n'a pas grand-chose à faire ici");
        System.out.println("Le résultat est" + (res+variable));
    }
}
```

2.2 Denis sur le code d'Hayaat : Move type to new file

L'opération appliquée ici est **Move type to new file**[2] disponible dans le menu "Refactor" d'Eclipse.

2. Move type to new file : `https://www.visualstudiotips.co.uk/tip/move-type-to-new-file/`

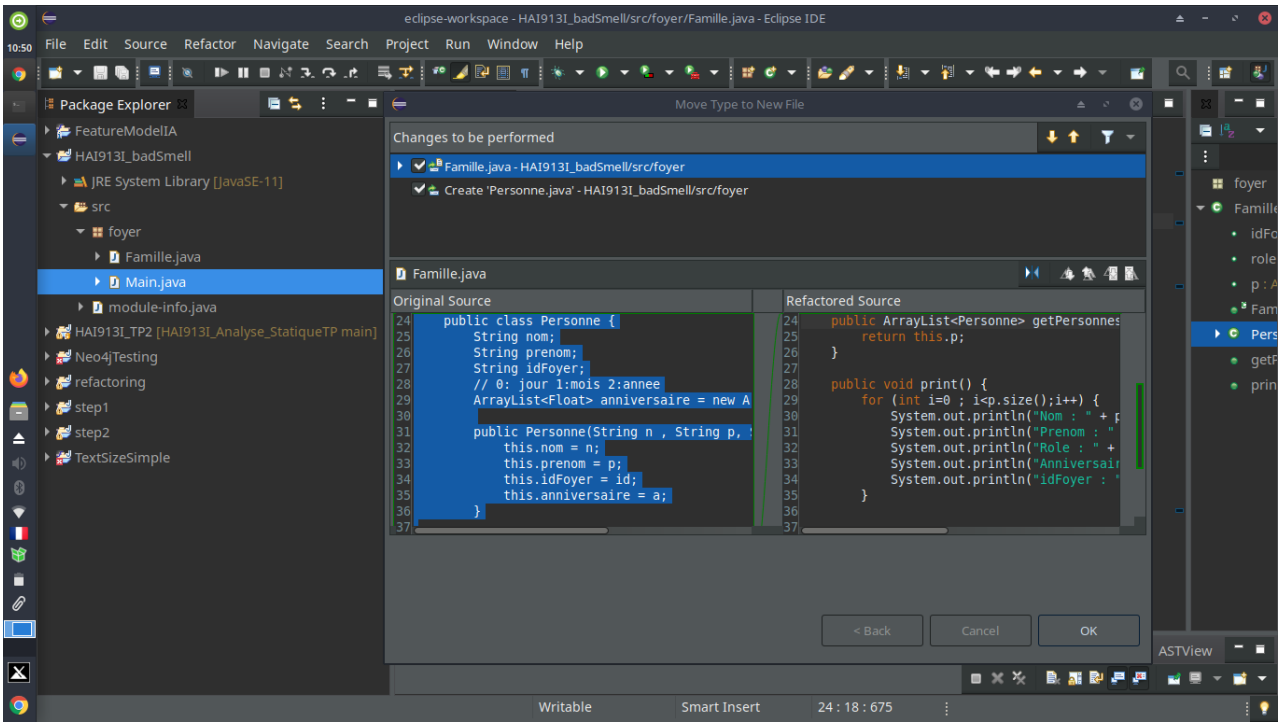

FIGURE 2 – Preview de l'opération sur Eclipse

On peut également avoir un mécanisme de "Preview" avant de lancer l'opération comme l'opération présentée précédemment, cela permet d'avoir un aperçu du résultat avant de valider l'opération.

```
1  // Dans la classe Famille la classe Personne
2  // devient une méthode get renvoyant une ArrayList<Personne>
3
4  public ArrayList<Personne> getPersonnes(){
5      return this.p;
6  }
```

```
1  // La classe Personne initialement dans la classe Famille est
2  // déplacé dans un nouveau fichier Personne.java
3
4  public class Personne {
5          String nom;
6          String prenom;
7          String anniversaire;
8
9          public Personne(String n , String p,String a) {
10             this.nom = n;
11             this.prenom = p;
12             this.anniversaire = a;
```

```
13          }
14          public String getNom() {return nom;}
15          public String getPrenom() {return prenom;}
16          public String getAnniversaire() {return anniversaire;}
17  }
```

3 Comparaison de 2 catalogues de refactorings

3.1 Le catalogue Refactoring

Pour mettre le catalogue d'Eclipse en échec nous avons crée le GEBTB (le "Good Enough But Too Bad") un exemple de 2 classes un peu difficile à comprendre avec un main() peu compréhensible. L'objectif était de mettre en avant plusieurs problèmes et d'autre un petit peu plus compliquer.

La première c'est la gestion des codes d'erreurs [3] qu'on retrouve dans la Classe 1 ligne 18 à travers la méthode "NOMELEMENT".

Cette opération n'est pas proposée sur Eclipse, mais elle permet de remplacer des codes d'erreurs numérique (à la C) avec des Exception (à la Java).

FIGURE 3 – Exemple de l'opération 'Replace Error Code With Exception'

3. Replace Error Code With Exception `https://refactoring.com/catalog/replaceErrorCodeWithException.html`

Classe 1 :

```java
public class goodEnough {

    private String NOMelement;
    private double NUMelement;

    public final static int A = 1, B=2, C=3;
    public int NUMlettre;



    public goodEnough(String NOM, double NUM) {
        this.NOMelement = NOM;
        this.NUMelement = NUM;
        this.NUMlettre = A;
    }


    int NOMELEMENT(int num) {
        if(this.NUMelement > 10) {
            return -1;
        }
        else {
            return 0;
        }
    }
}
```

Classe 2 :

```java
public class goodEnoughtFILS extends goodEnough {

    private String NOMelement;
    private static final int NUMelement = 8;


public goodEnoughtFILS(String NOM,int NUM) {
        super(NOM,NUM);
        this.NOMelement = NOM;
    }


public static void main(String[] args) {
    goodEnoughtFILS FILS = new goodEnoughtFILS("LENOM",21);

    System.out.println(FILS.NUMlettre);
    System.out.println(FILS.NOMELEMENT(45));
```

```
18      }
19  }
```

3.2 Le catalogue Eclipse

Finalement, on se rend compte qu'aucune opération de refactoring issu du catalogue d'Eclipse (à part celle simpliste comme Rename) ne peut-être appliqué à se code. Nous avons donc mis en place un autre exemple pour illustrer une méthode de refactoring qui n'apparaît pas dans le catalogue mais bien sur Eclipse : **Introduce indirection**.

Cette opération permet d'extraire l'appel d'une méthode dans une nouvelle méthode. Cette opération est utile lorsque :

— La méthode initial ne peut pas être changé car elle est issu d'une librairie.
— La méthode initial ne doit pas être changé car c'est une méthode issu d'une API par exemple.

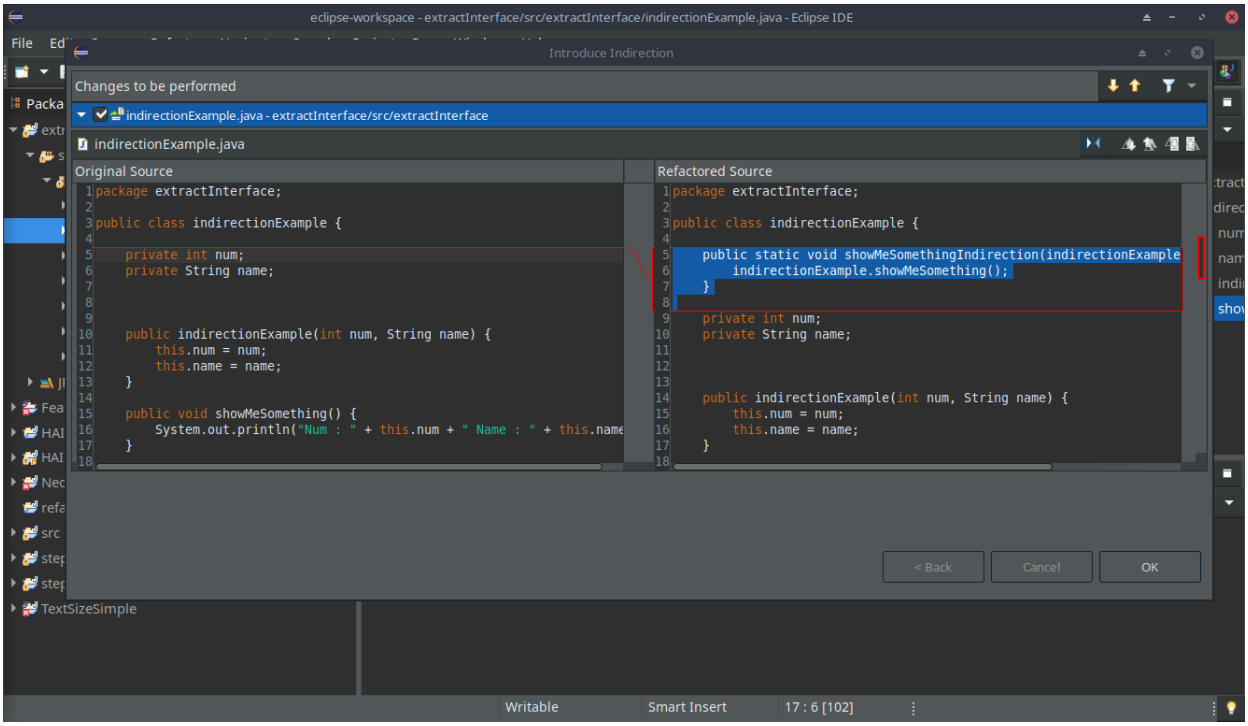

FIGURE 4 – Preview de l'opération sur Eclipse

```
1   public class indirectionExample {
2
3       private int num;
4       private String name;
5
6
```

```
7
8       public indirectionExample(int num, String name) {
9           this.num = num;
10          this.name = name;
11      }
12
13      public void showMeSomething() {
14          System.out.println("Num : " + this.num + " Name : " + this.name);
15      }
16
17  }
```

4 Etude de l'opération de refactoring 'Extract Interface'

```
1   //Rappel des classes de l'exercice
2
3
4   class ListeTableau{
5   public boolean add(Object o) {return true;}
6   public boolean isEmpty() {return true;}
7   public Object get(int i) {return null;}
8   private void secretLT(){}
9   public static void staticLT() {}
10  int nbLT;
11  }
12
13  class ListeChainee{
14  public boolean add(Object o) {return true;}
15  public boolean isEmpty() {return true;}
16  public Object get(int i) {return null;}
17  public Object peek() {return null;}
18  public Object poll() {return null;}
19  private void secretLC(){}
20  }
21
22  class QueueDoubleEntree{
23  public boolean add(Object o) {return true;}
24  public boolean isEmpty() {return true;}
25  public Object peek() {return null;}
26  public Object poll() {return null;}
27  private void secretQDE(){}
28  }
29
30  class QueueAvecPriorite{
31  public boolean add(Object o) {return true;}
32  public boolean isEmpty() {return true;}
33  public Object peek() {return null;}
34  public Object poll() {return null;}
```

```
35  public Object comparator() {return null;}
36  private void secretQAP(){}
37  }
```

4.1 Comportement 1 : Tout dans le même fichier

Si toutes les classes sont présentes dans le même fichier, même en sélectionnant toutes les classes du fichier Eclipse ne proposera l'opération de refactoring '**Extract Interface** uniquement sur la première classe de l'interface, et donc en prenant en compte seulement les méthodes présentes dans cette classe. En validant l'opération, on obtient donc 1 interface qui n'est implémentée que par la première classe de notre fichier. Ce n'est évidemment pas le comportement souhaité et il ne résout pas la problématique.

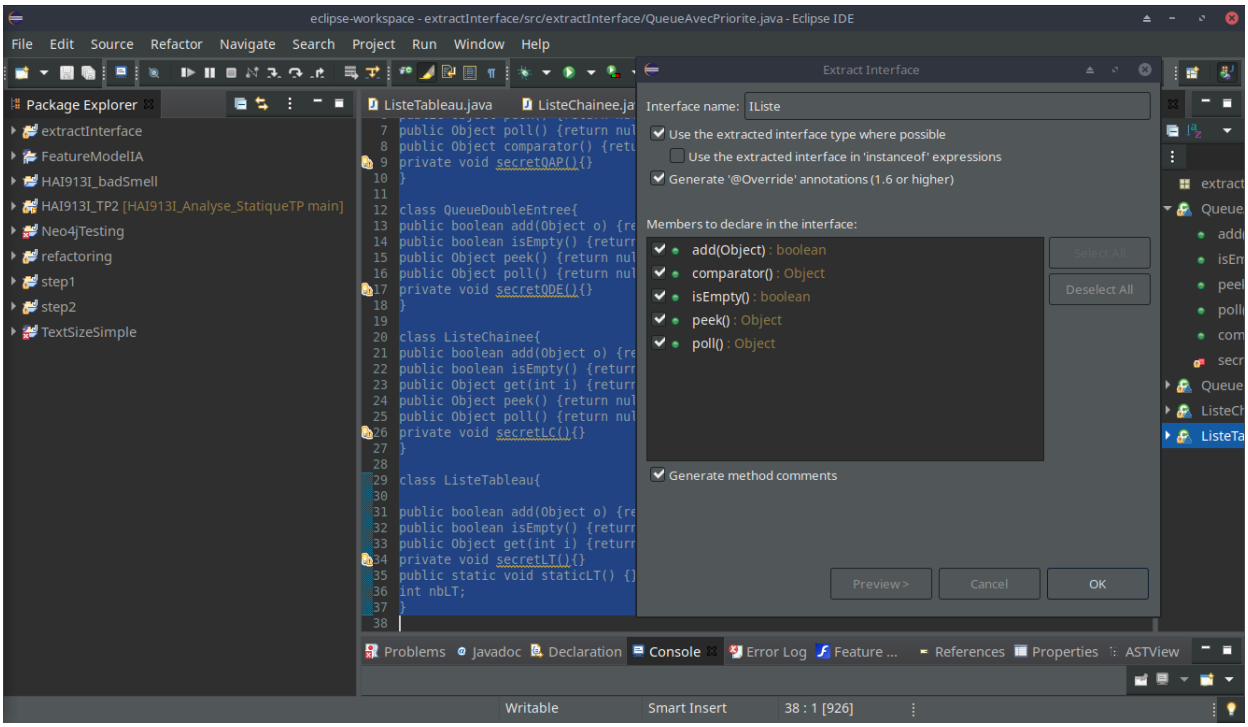

FIGURE 5 – Preview de l'opération sur Eclipse

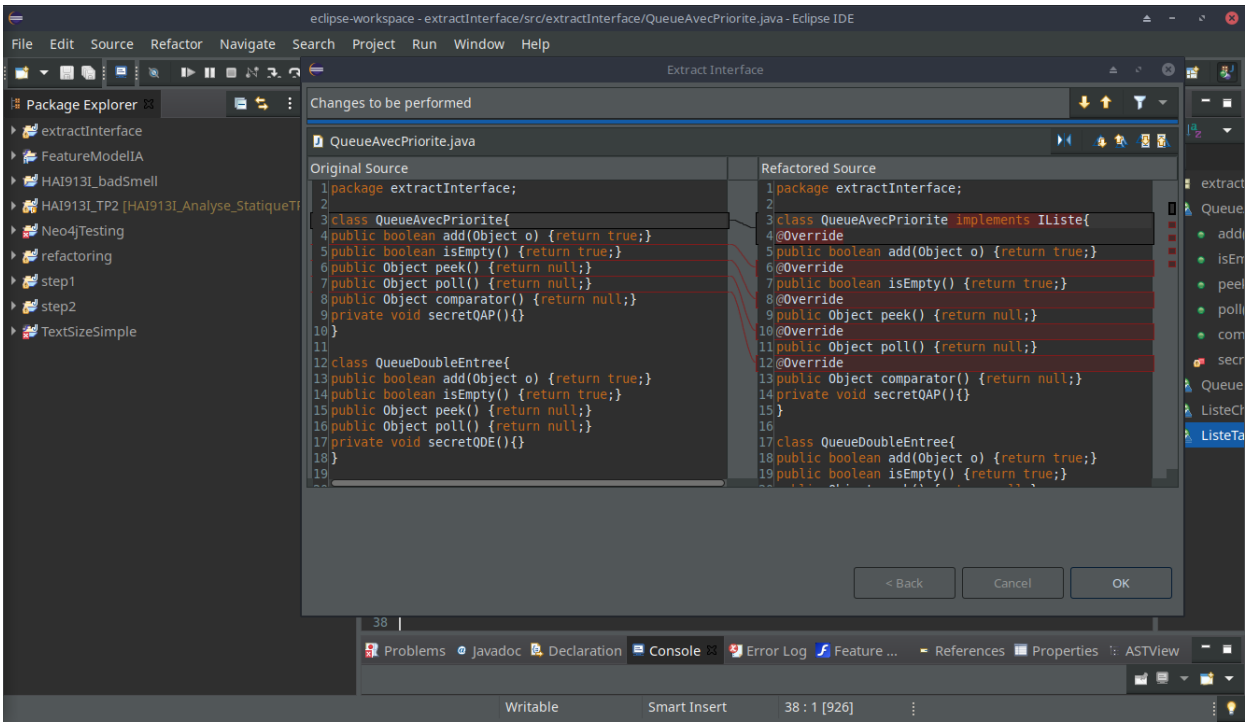

FIGURE 6 – Preview de l'opération sur Eclipse

4.2 Comportement 2 : 1 fichier par classe

Si les classes sont séparées avec 1 classe par fichier d'une part, on ne peut pas sélectionner toutes les classes (l'opération Extract Interface est grisée dans le menu.) donc on est obligé de choisir 1 seul fichier et finalement, on retombe sur le comportement décrit précédemment, on obtient 1 seule interface implémentée par 1 seule classe (celle choisit) et donc on ne résout en rien le problème posé.

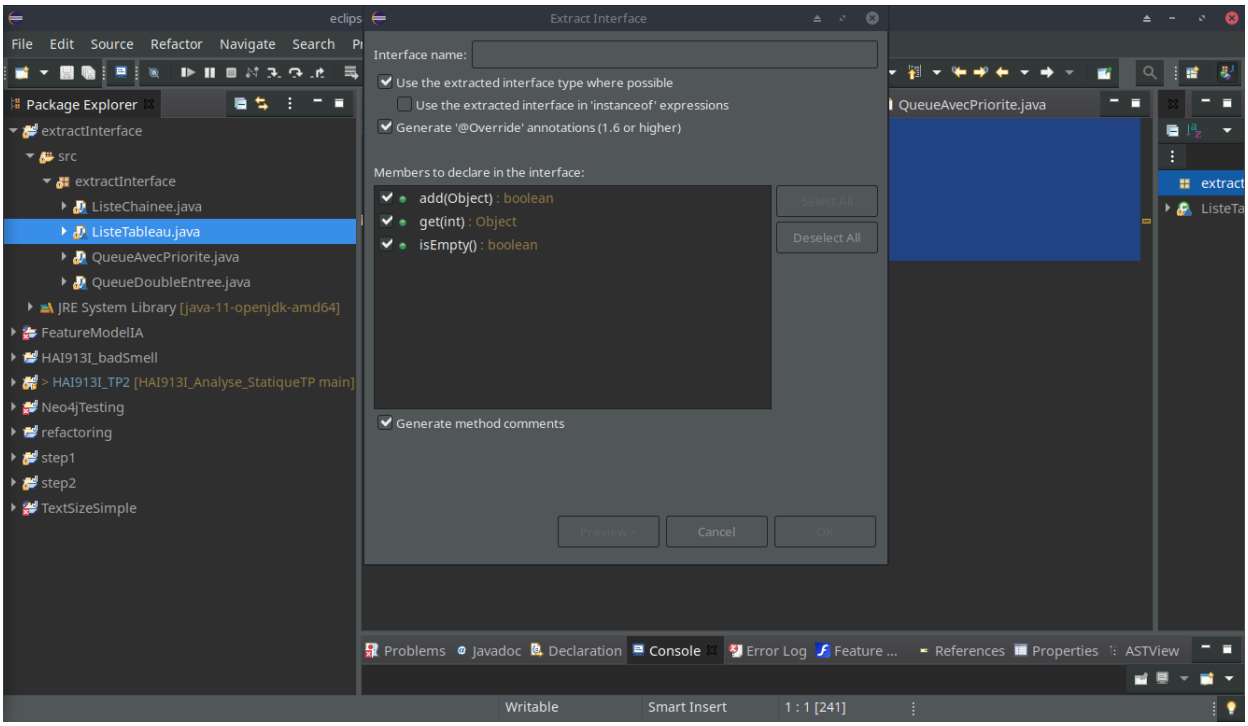

FIGURE 7 – Preview de l'opération sur la classe ListeTableau

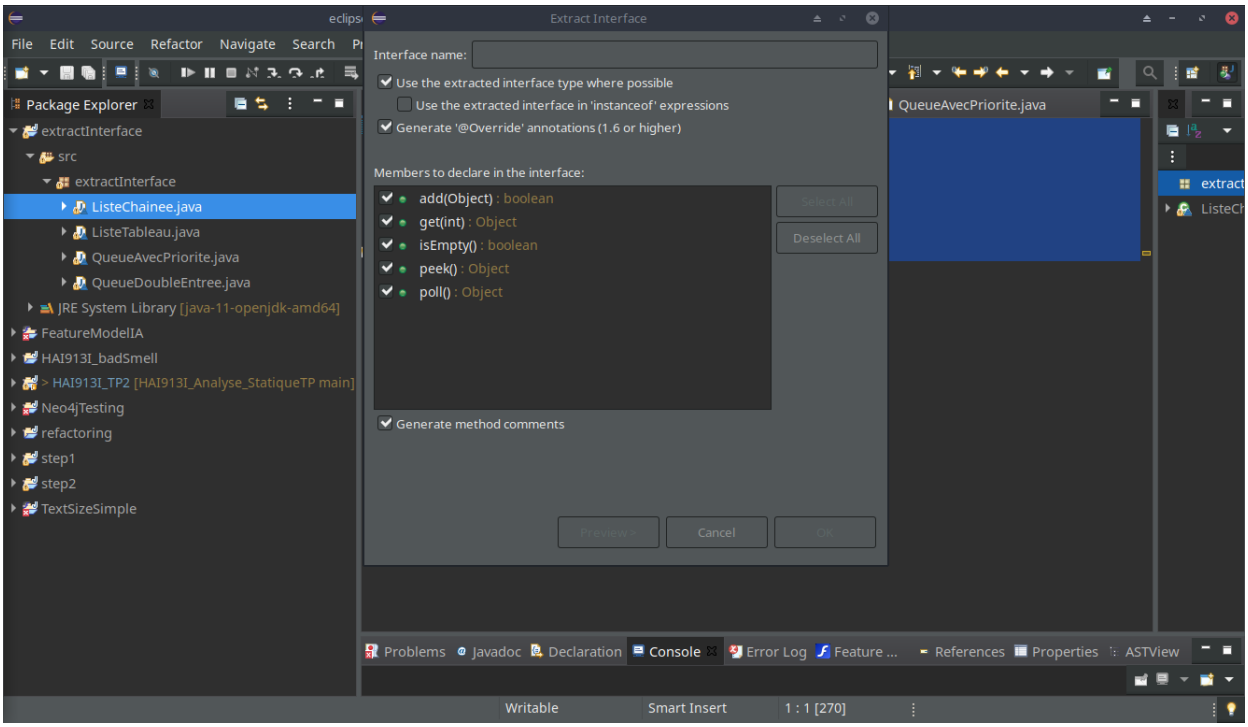

FIGURE 8 – Preview de l'opération sur la classe ListeChainee

4.3 Bilan : Que faire ?

4.4 Big Refactoring... à la main !

4.4.1 Solution Naïve : 4 interfaces

Dans un premier temps (dans l'optique de faire un "Big refactoring" sans utiliser d'outils comme l'AOCPoset) nous avons décidé d'appliquer une méthode "algorithmique", cet à dire chercher une solution naïve pour s'orienter vers une solution plus spécialisée.

La solution naïve (trop ?) qui parait évidente dans ce problème, c'est la création d'une interface pour chacune des classes, chacune reprenant les différentes méthodes de chaque classe.

Point positif :

- La facilité d'implémentation
- La compréhension du code plus facile

Point négatif :

- La répétition inutile des mêmes méthodes
- La quantité de fichiers
- Très peu de réuitilisation possible
- Ne résout pas la problématique de base

```
1  interface IQueueDoubleEntree {
2
3      boolean add(Object o);
4      boolean isEmpty();
5      Object peek();
6      Object poll();
7
8  }
9
10 interface IQueueDoubleAvecPriorite {
11
12     boolean add(Object o);
13     boolean isEmpty();
14     Object peek();
15     Object poll();
16     Object Comparator
17 }
18
19 interface IListeChainee {
20
21     boolean add(Object o);
22     boolean isEmpty();
23     Object get(int i);
```

```
24      Object peek();
25      Object poll();
26
27  }
28
29  interface IListeTableau {
30      boolean add(Object o);
31      boolean isEmpty();
32      Object get(int i);
33  }
34
35
36  //Exemple d'utilisation de cette solution
37
38  class ListeChainee implements IListeChainee {
39
40  public boolean add(Object o) {return true;}
41  public boolean isEmpty() {return true;}
42  public Object get(int i) {return null;}
43  public Object peek() {return null;}
44  public Object poll() {return null;}
45  private void secretLC(){}
46  }
```

4.4.2 Solution imparfaite : 2 interfaces

Pour continuer notre démarche algorithmique, nous avons cherché une solution plus spécialisée (qui répond vraiment au problème). En regardant un petit peu le code nous avons remarqué que les 2 classes de Liste avait plusieurs méthodes en commun et pareil pour les 2 classes Queue. Si on crée 2 interfaces contenant le minimum de méthodes requises pour la plus petite classe Liste et pour la plus petite Queue, on obtient "seulement" la répétition de 2 méthodes (au lieu de 6 initialement) et il manquera 1 méthode : comparator qui n'est présente que dans 1 classe Queue.

Finalement, on voit que 2 interfaces permettent de satisfaire 3 classes en réduisant le nombre de répétitions, mais cette solution reste néanmoins imparfaite.

Point positif :

- Réduction des répétitions de méthode
- Factorisation du code
- Réponse partiel à la problématique

Point négatif :

- Solution encore imparfaite
- Nécessite plus de temps que la solution naïve

```
interface IQueue {

    boolean add(Object o);

    boolean isEmpty();

    Object peek();

    Object poll();

}

interface IListe {

    boolean add(Object o);

    boolean isEmpty();

    Object get(int i);

}


//Exemple d'utilisation de cette solution

class ListeChainee implements IListe,IQueue {

public boolean add(Object o) {return true;}
public boolean isEmpty() {return true;}
public Object get(int i) {return null;}
public Object peek() {return null;}
public Object poll() {return null;}
private void secretLC(){}
}
```

5 Annexe : application de l'analyse formelle de concepts

5.1 Le fichier CSV obtenu

On peut voir ci-dessous que le CSV est assez simple, il reprend le modèle vu en cours. Chaque classe représentée par une ligne et chaque colonne représente une méthode public, l'attribut 1 signifie que la méthode est bien dans la classe de la ligne correspondante et l'attribut 0 si la classe ne possède pas la méthode.

	Standard	Standard	Standard	Standard	Standard	Standard	Standard
1		add	isEmpty	get	peek	poll	comparator
2	ListeTableau	1	1	1	0	0	0
3	ListeChainee	1	1	1	1	1	0
4	QueueDoubleEntree	1	1	0	1	0	0
5	QueueAvecPriorite	1	1	0	1	1	1

FIGURE 9 – Fichier CSV

5.2 AOCPoset obtenu

Finalement, on obtient un AOCPoset assez simple qui "valide" (en tout cas qui ne contredit pas) la solution "imparfaite" des 2 interfaces. (On retrouve le même problème avec la méthode comparator).

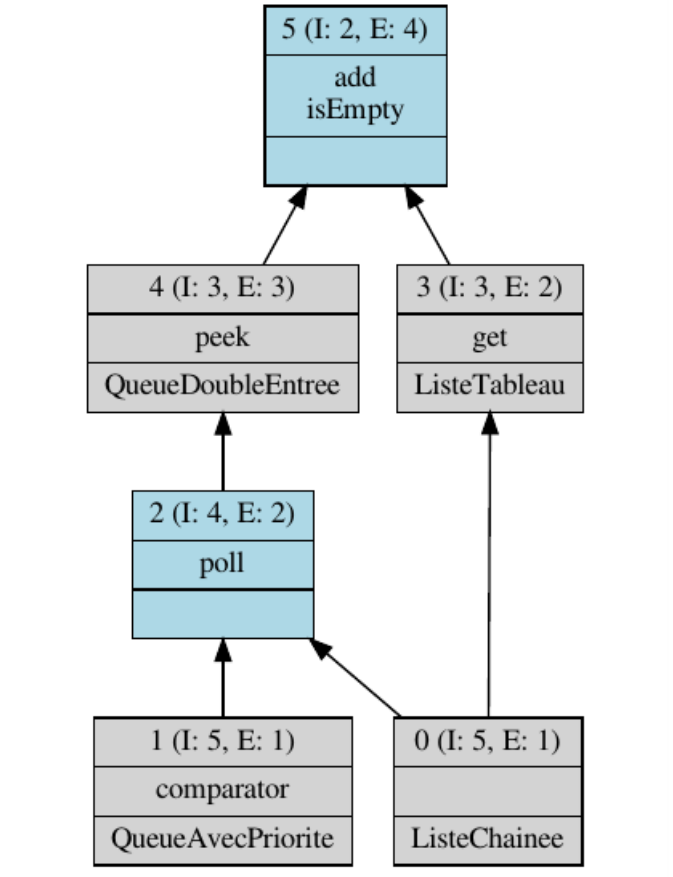

FIGURE 10 – AOCPoset obten

5.3 Bilan

Néanmoins, l'application et l'utilisation de l'AOCPoset dans ce cas précis est très intéressant. D'une part, on voit une des applications possibles de l'AOCPoset à une problématique de programmation et d'autre part l'AOCPoset permet d'appuyer une solution programmatique apportée par un tiers, on pourrait donc imaginer un refactoring conséquent réaliser par une équipe de développeur et l'utilisation d'outils comme l'AOCPoset permettrai de soutenir leur solution et de comprendre leur choix de conception plus facilement, c'est donc un outil possiblement très utile.