Université de Montpellier
Master 2 Informatique Génie Logiciel

Évolution et restructuration des logiciels

TP Refactoring

Rapport du TP4

Étudiants :
ENGEL Arthur
ROMERO Gaétan
GARCIA Léa

2. Application des opérations de refactoring sur les programmes d'une autre personne du groupe

Move one or more static method (Léa Garcia)

L'opération de refactoring qu'il y avait besoin d'appliquer est la suivante : Move One Or More Static Method. Les captures d'écrans ci-dessous nous montrent les différentes étapes et résultats du refactoring appliqués au projet.

Cette opération de refactoring permet de déplacer une méthode statique d'une certaine classe jusqu'à une autre classe, dans laquelle elle aura plus de sens. Utiliser cette méthode permet de déplacer la méthode statique directement d'une classe à l'autre sans avoir à faire un copier-coller qui peut être mal fait et causer des erreurs par la suite, mais elle permet également de remplacer correctement toutes les références à cette méthode dans notre projet. Par exemple dans notre cas, dans la méthode main() de notre classe Main, avant le refactoring la méthode choisirStrategie() était appelée sur la classe Joueur et après le refactoring, elle est appelée sur la classe Entraîneur.

La mise en oeuvre du refactoring sous IntelliJ est bien réalisée, il suffit de choisir le refactoring puis nous avons uniquement besoin de choisir la classe destinataire de la méthode statique. Enfin, tout se fait sans qu'on le voit, on peut voir uniquement la méthode disparaître et c'est en allant dans le code des classes que l'on peut remarquer là où des changements ont eu lieu.

Tout s'est bien passé, je n'ai pas eu à toucher au code du main, le refactoring s'est occupé de modifier la classe, sur laquelle la méthode choisirStrategie() est appelée depuis la méthode main() de la classe Main.

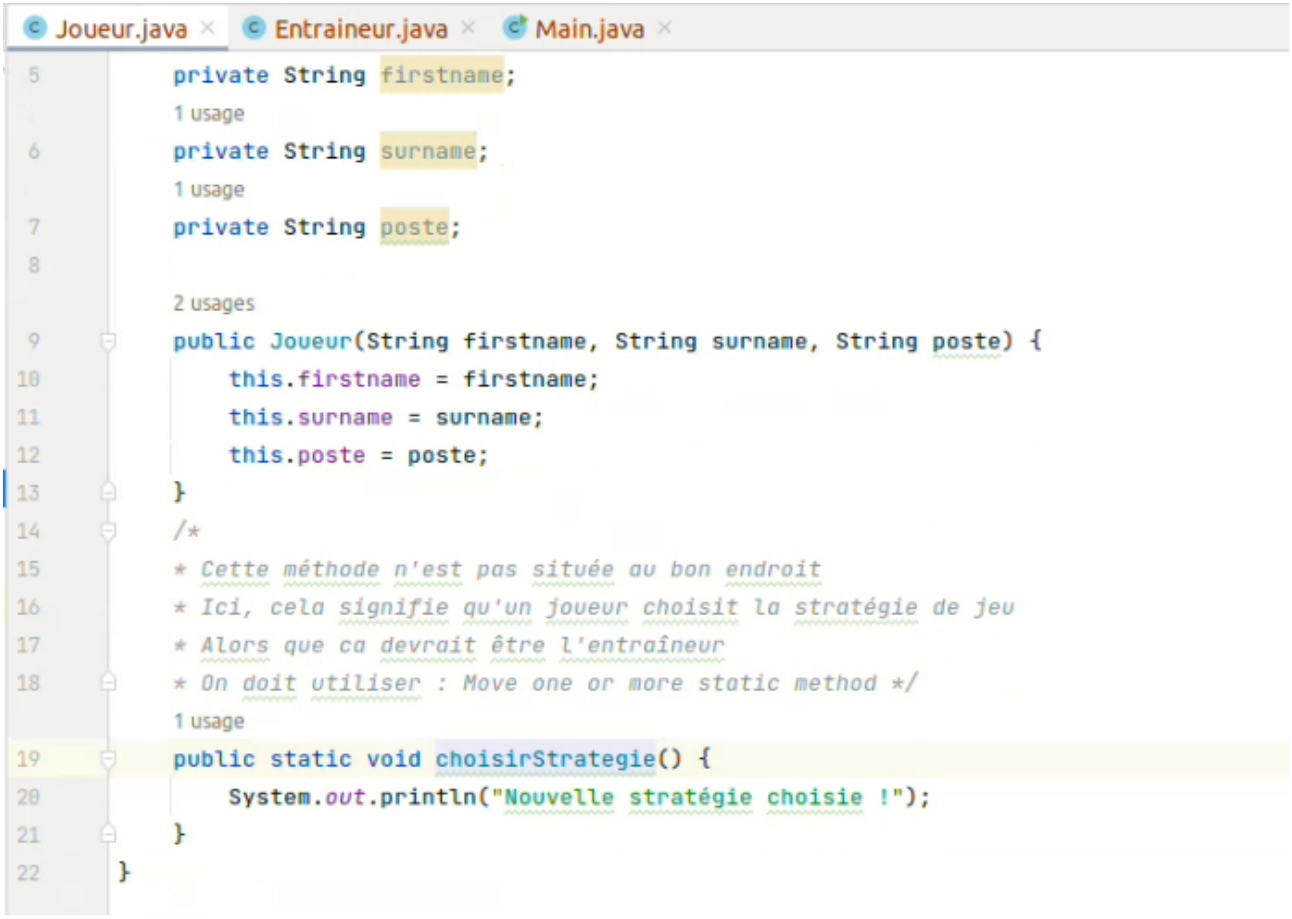

FIGURE 1 – Méthode choisirStratégie() présente à tord dans la classe Joueur, avant refactoring

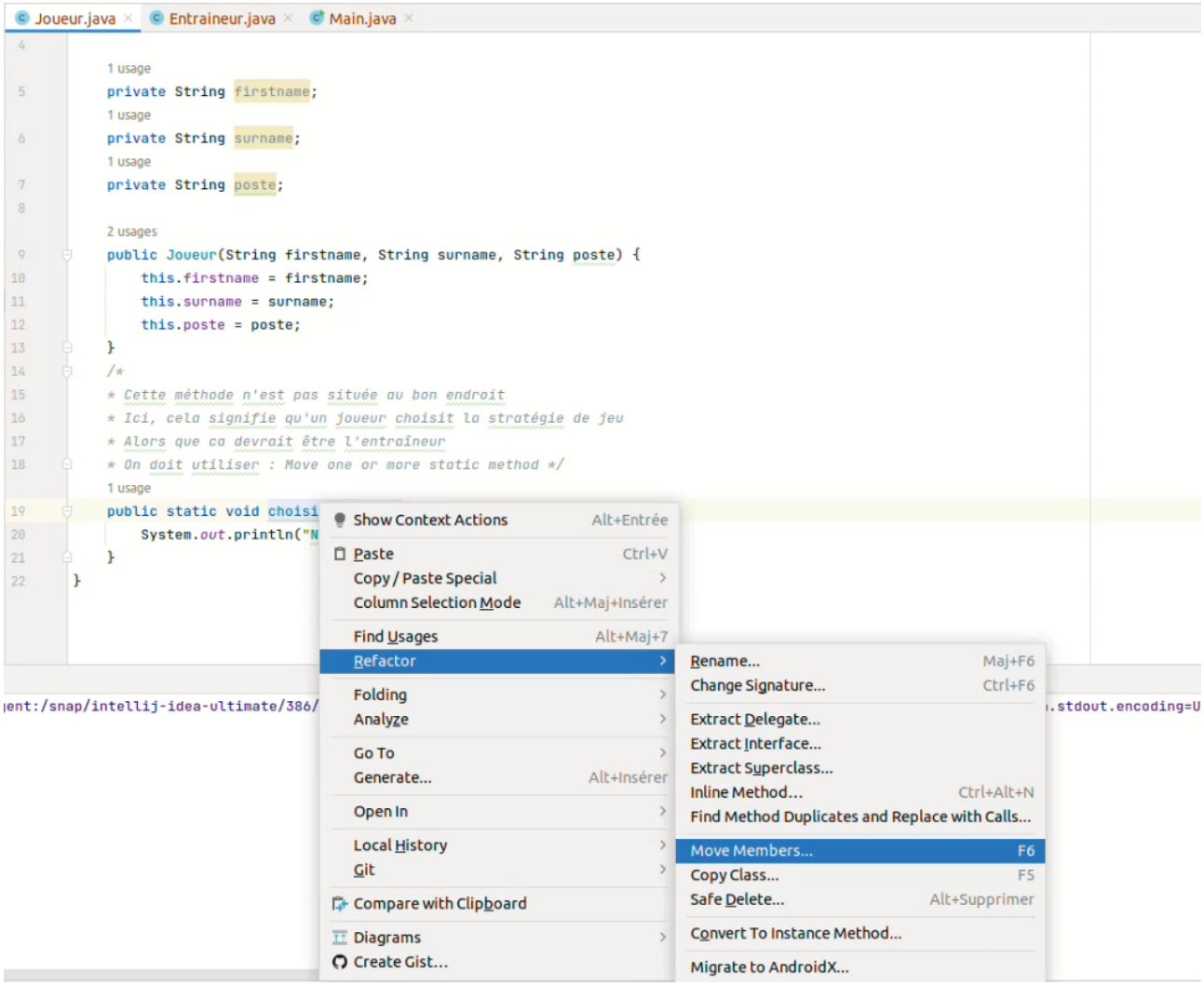

FIGURE 2 – Choix du refactoring

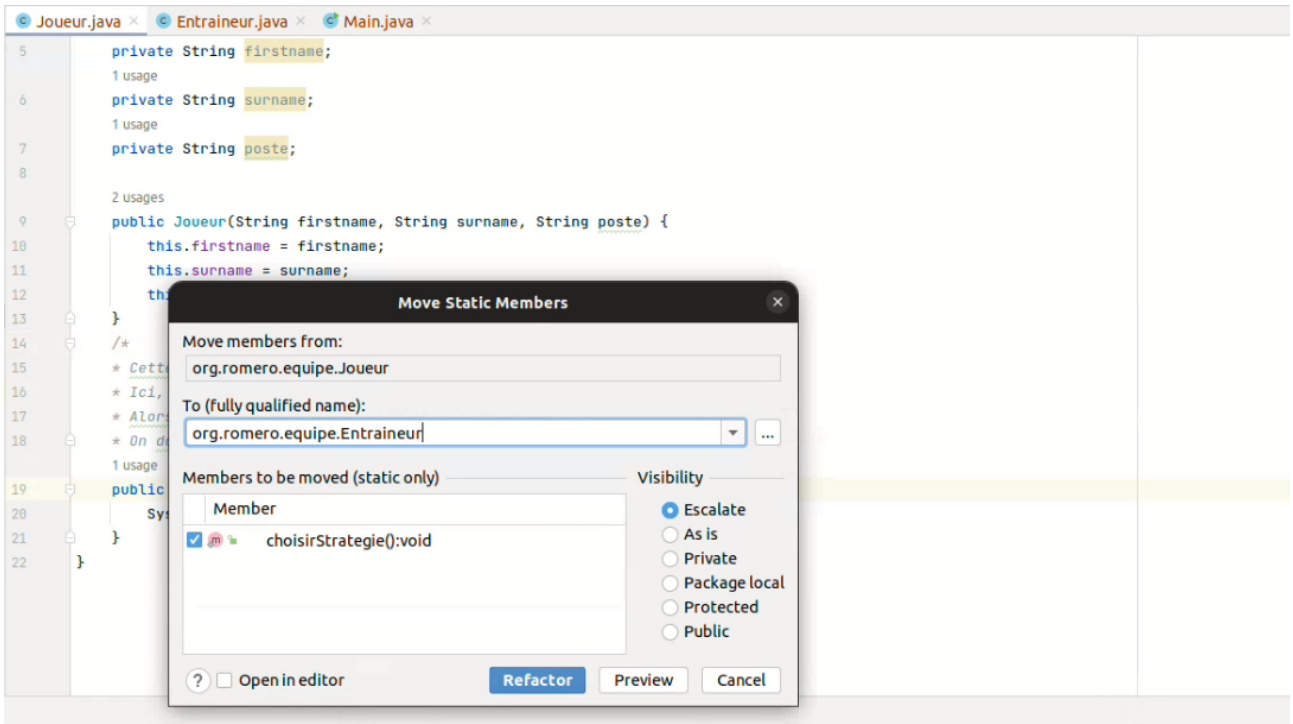

FIGURE 3 – Choix de la classe pour le refactoring

```
Joueur.java × Entraineur.java × Main.java ×
7       private String composition;
8
        1 usage
9       public Entraineur(String firstname, String surname) {
10          this.firstname = firstname;
11          this.surname = surname;
12          this.composition = "";
13      }
14
15      /*
16      * Cette méthode n'est pas située au bon endroit
17      * Ici, cela signifie qu'un joueur choisit la stratégie de jeu
18      * Alors que ca devrait être l'entraîneur
19      * On doit utiliser : Move one or more static method */
        1 usage
20      public static void choisirStrategie() {
21          System.out.println("Nouvelle stratégie choisie !");
22      }
23
24      public String newComposition(String newComposition) {
25          this.composition = newComposition;
26          return this.composition;
27      }
28  }
```

FIGURE 4 – Résultat dans la classe Entraîneur

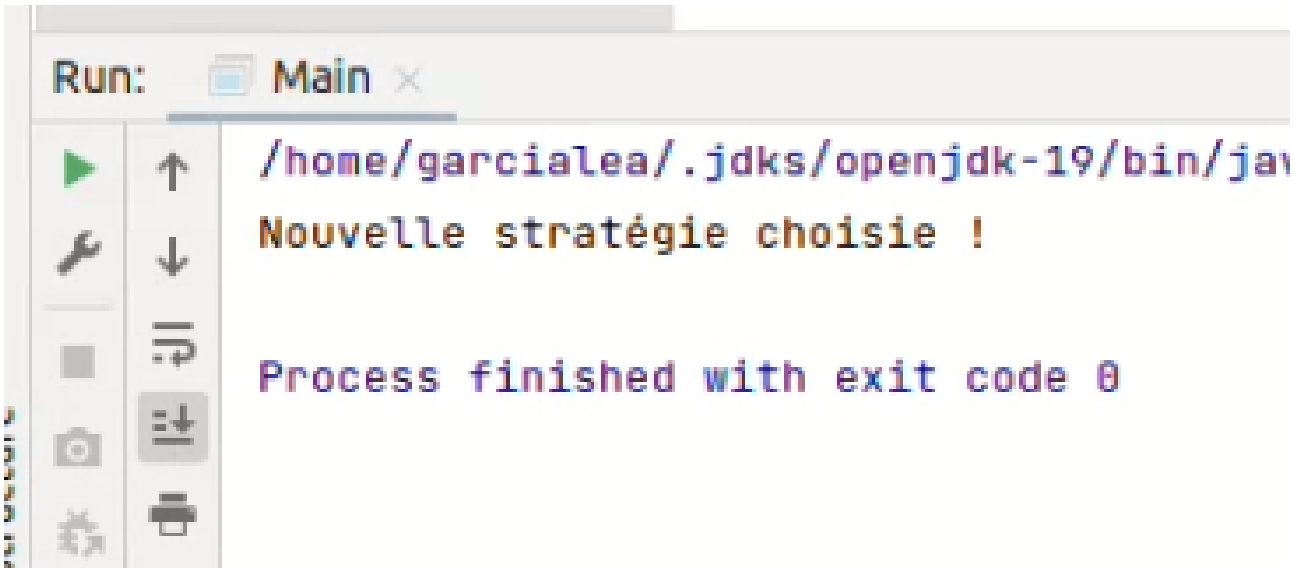

FIGURE 5 – Résultat dans le main

Extract class (Gaétan Romero)

L'opération de refactoring que nous devions réaliser est la suivante : Extract Class. La procédure appliquée ainsi que les résultats obtenus lors cette opération de refactoring sont décrits par les captures d'écrans affichées par la suite.

On trouve un intérêt à cette procédure lorsque plusieurs éléments contenus dans une classe devraient faire partie d'une autre classe. C'est à dire, lorsque des objets d'une classe possèdent des caractéristiques qu'ils ne devraient, logiquement, pas avoir. Par exemple, imaginons une classe Voiture. Si les objets de cette classe ont comme attributs "pneus" et "jantes", logiquement, il serait plus optimal et intelligent de faire une classe Roue à part, et avoir un objet Roue dans la classe Voiture.

Cette méthode permet surtout de grandement améliorer la qualité du code pour la relecture et l'amélioration future du code, mais aussi de modulariser notre code afin de rendre des classes plus spécifiques et moins grandes.

La mise en oeuvre sous IntelliJ est bien réalisée, dans le sens où l'interface est assez intuitive, les informations sont détaillées et facilement accessibles. On peut donner un nom à la classe, sélectionner rapidement les attributs ou méthodes que l'on souhaite déplacer, ainsi qu'une case nous permettant de choisir si l'on veut des accesseurs. Un détail supplémentaire qui n'est pas précisé dans l'interface du menu est la création d'un constructeur vide dans la nouvelle classe. Dans notre cas, il a été supprimé car il était inutile mais cela peut être appréciable dans certains cas.

Le test s'est très bien déroulé et aucun changement n'a été apporté au main().

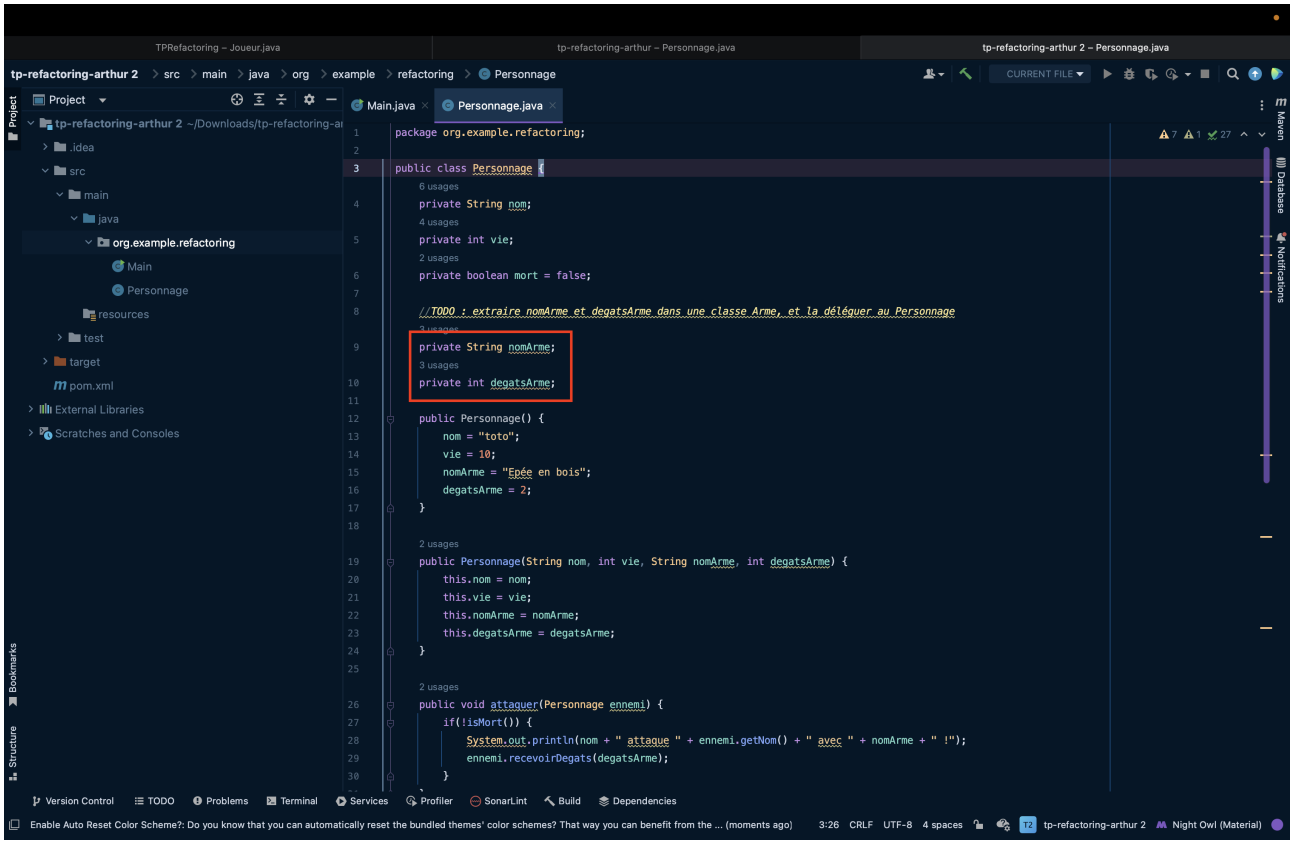

FIGURE 6 – Classe avant le refactoring

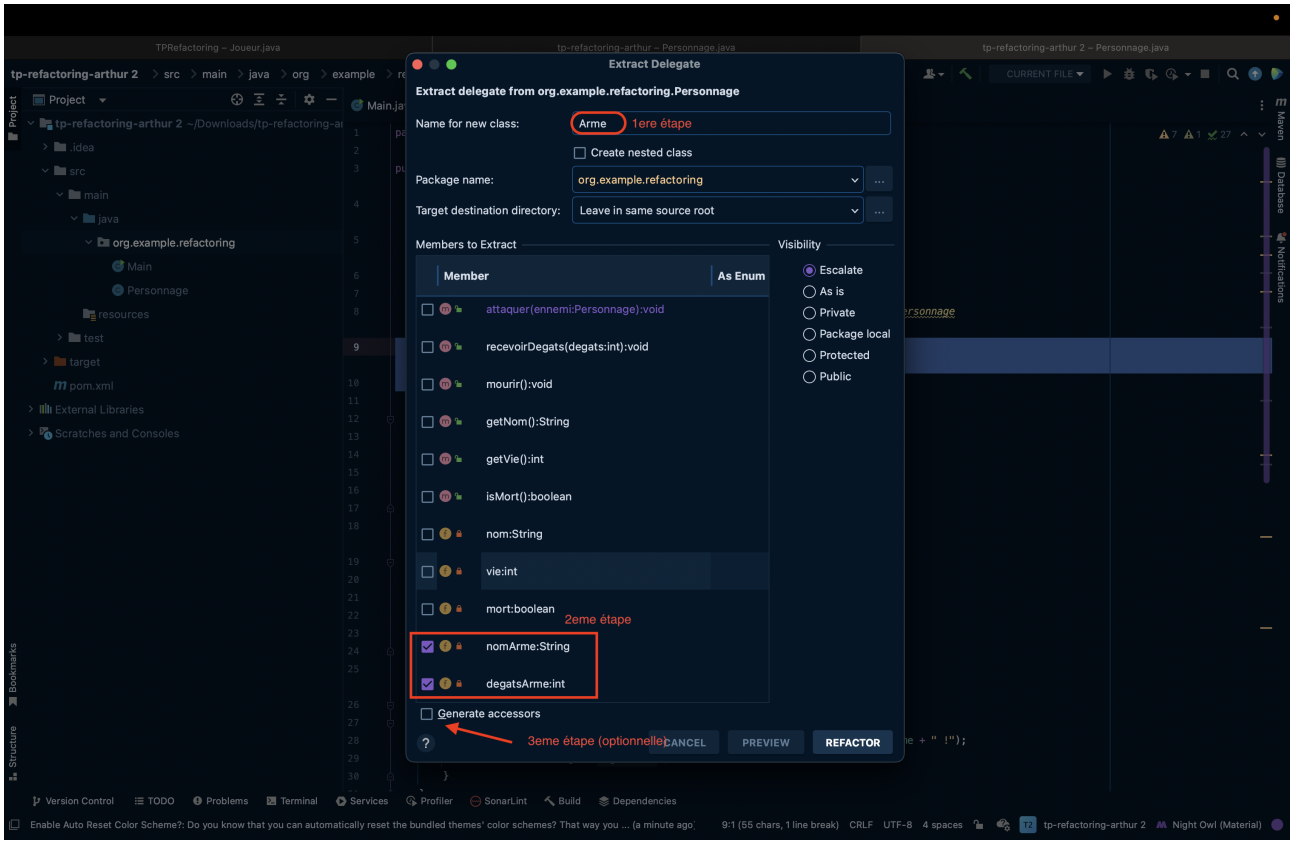

FIGURE 7 – Menu du refactoring "extract class"

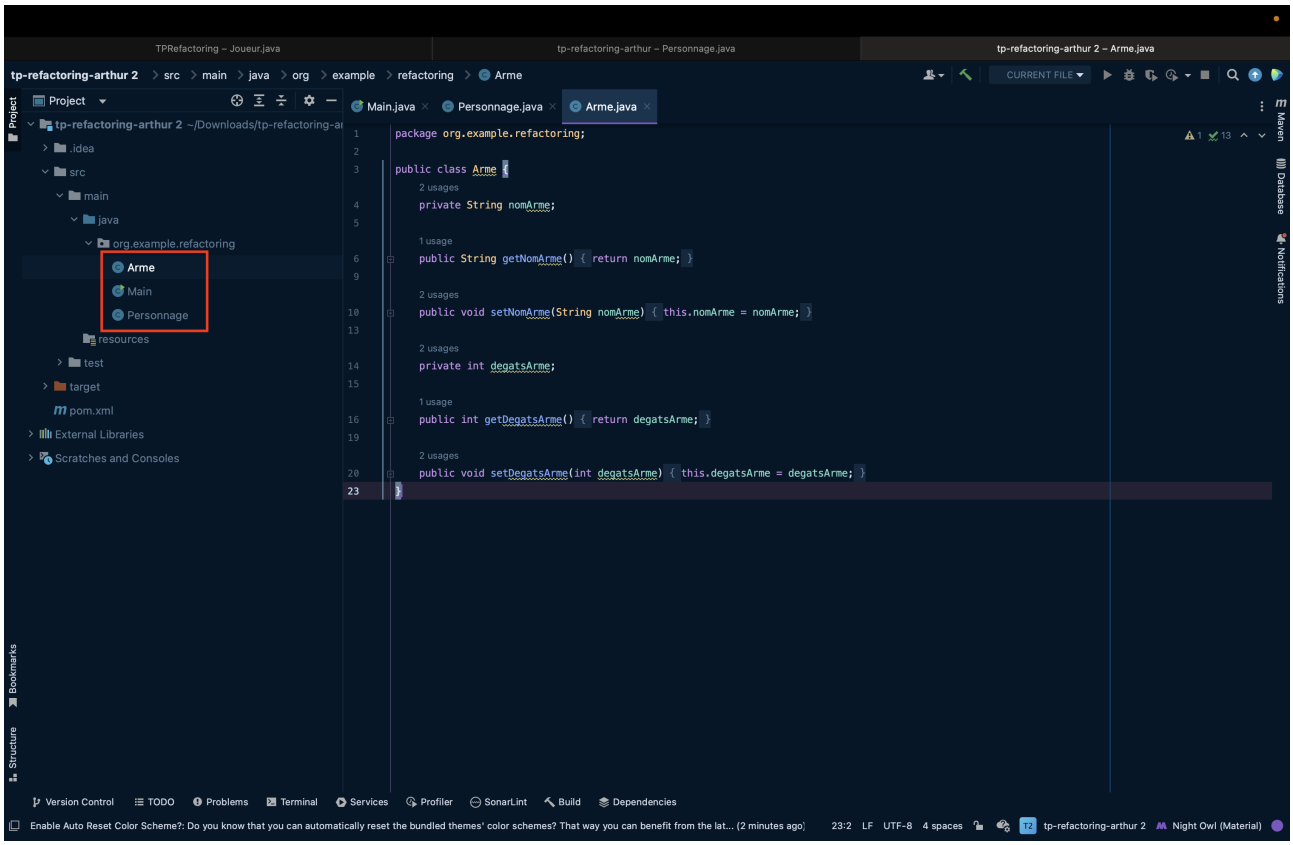

FIGURE 8 – Nouvelle classe créée

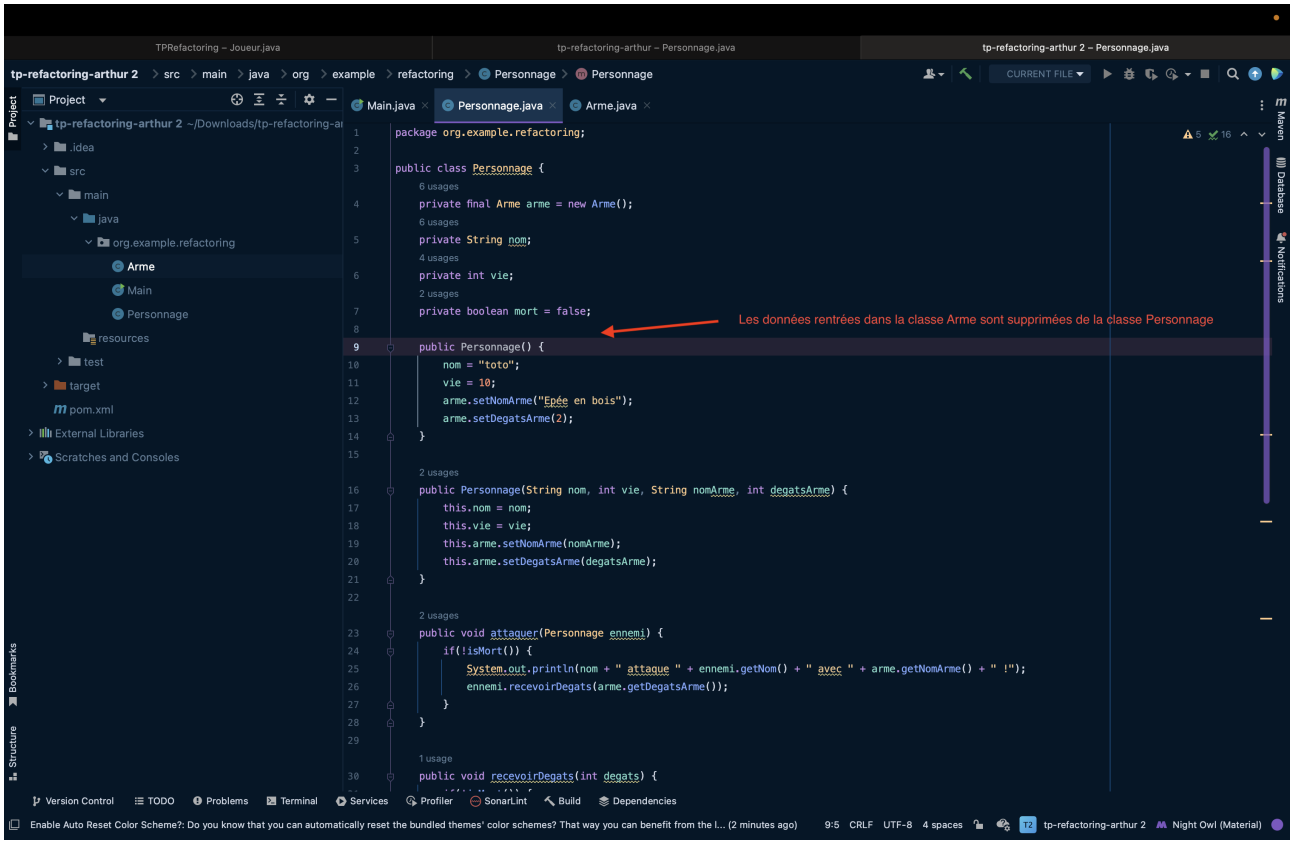

FIGURE 9 – Après le refactoring

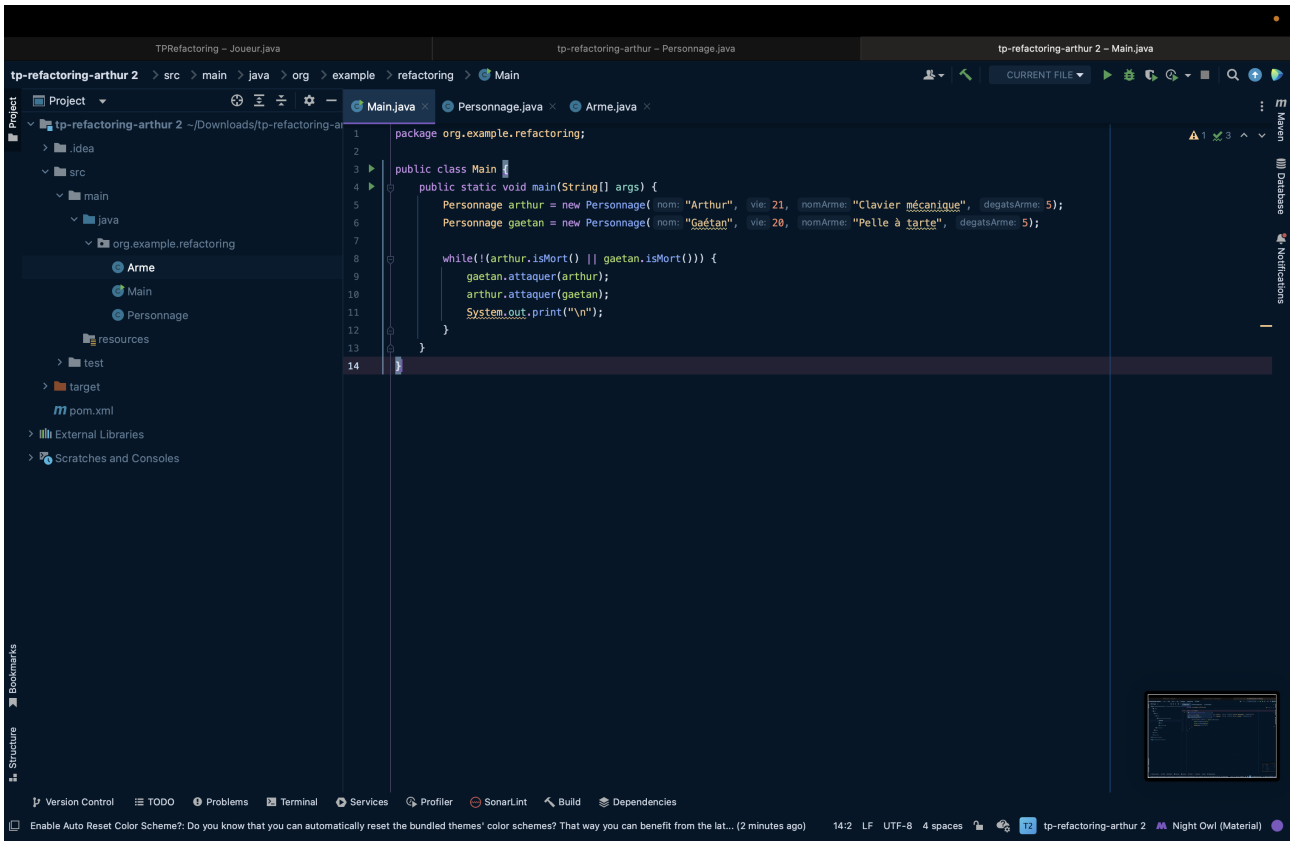

FIGURE 10 – main avant et après

Encapsulate field (Arthur Engel)

Ce refactoring est pratique pour changer la visibilité d'un champ facilement, et pour générer automatiquement des accesseurs appropriés à celui-ci. On peut aussi utiliser ce refactoring dès la création d'un champ pour éviter d'avoir à écrire soi-même des accesseurs en premier lieu.

Les intérêts de ce refactoring, à mon sens :

- Le temps gagné, particulièrement si on encapsule plusieurs champs à la fois.
- La sécurité par rapport à l'encapsulation manuelle : l'IDE remplace automatiquement toute référence au champ dans le projet par les accesseurs nouvellement créés, alors qu'un changement de ces références à la main crée un risque d'erreur. On évite aussi un rare cas ou un développeur écrirait mal un accesseur, puisque celui-ci est généré automatiquement. Attention cependant, il est toujours possible de passer un attribut en privé sans lui générer de getter, ou de générer les accesseurs en privé/protected, les rendant potentiellement inutilisables dans d'autres parties du programme. Dans ce cas l'IDE signalera si cela pose un problème dans le reste du code et où.

Je pense que la mise en oeuvre (sous IntelliJ) est bien réalisée, on a toutes les options qu'on pourrait attendre : choix de la visibilité de l'attribut, choix des accesseurs (get/set/les deux) à générer, visibilité de ces accesseurs, nom des accesseurs. De plus, on peut appliquer ce refactoring sur plusieurs champs à la fois. Également, le refactoring préfixe les getters pour les booléens par "is" au lieu de "get", un détail appréciable. Je pense qu'éventuellement l'IDE devrait prévenir d'un éventuel changement du comportement du programme (comme expliqué précédemment) avant d'appliquer le refactoring, mais on pourrait aussi dire qu'il revient au développeur de faire preuve de jugeotte.

Tout s'est bien déroulé et je n'ai rien eu à changer dans le main.

Voici les différentes étapes pour encapsuler un champ dans IntelliJ en image :

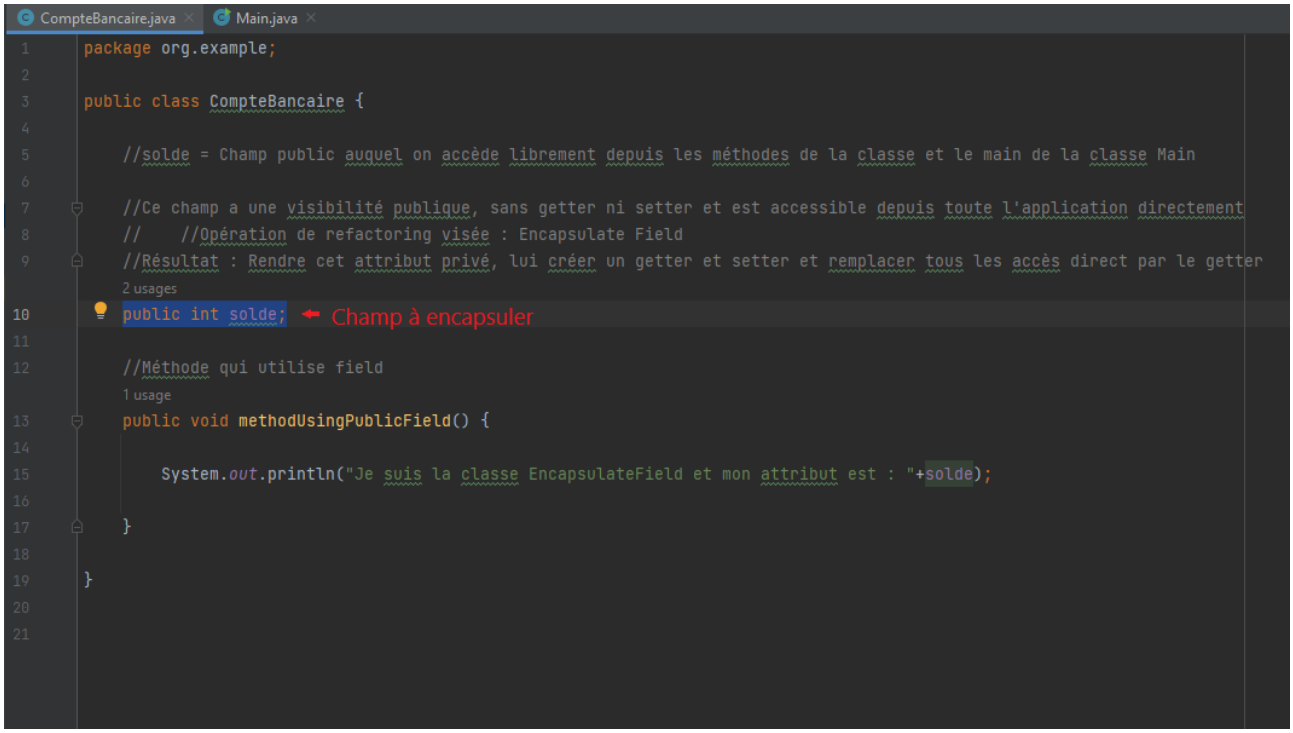

Figure 11 – Sélection du champ à encapsuler

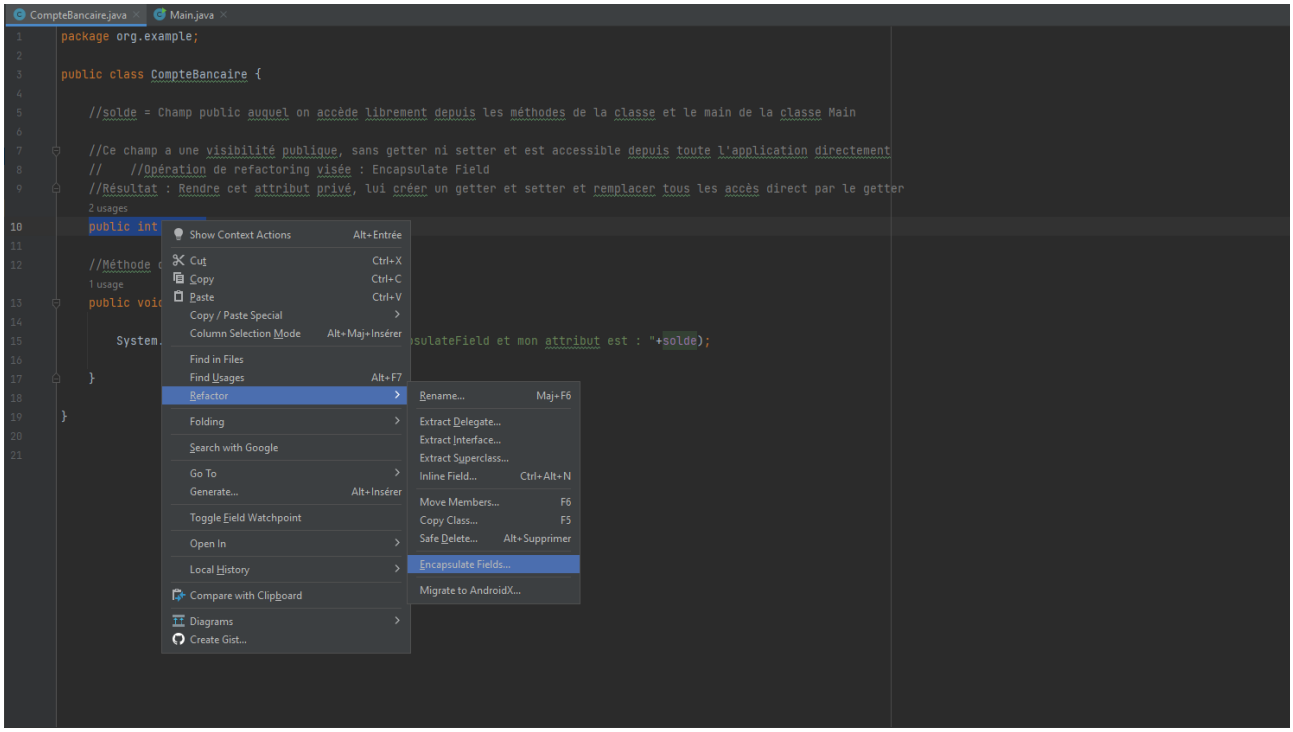

FIGURE 12 – Choix fonction de refactoring dans le menu déroulant

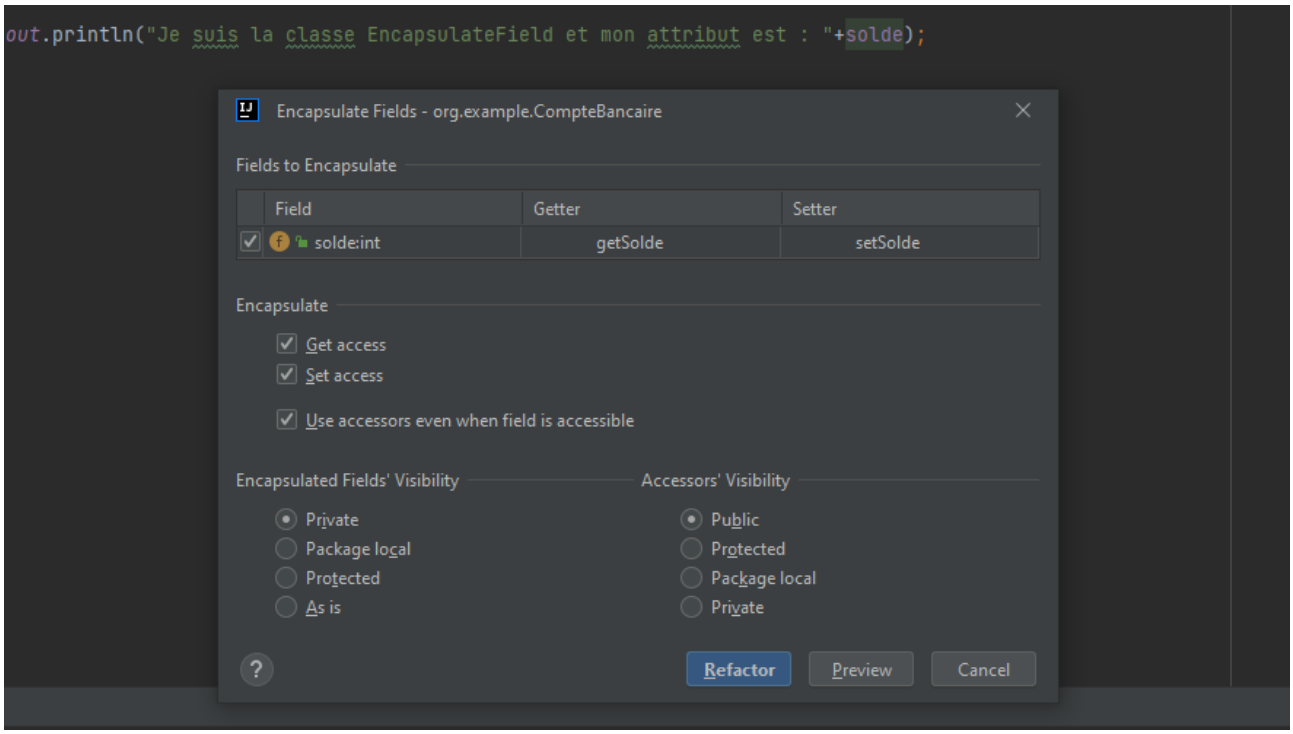

FIGURE 13 – Menu de la fonction de refactoring

FIGURE 14 – Changements dans la classe

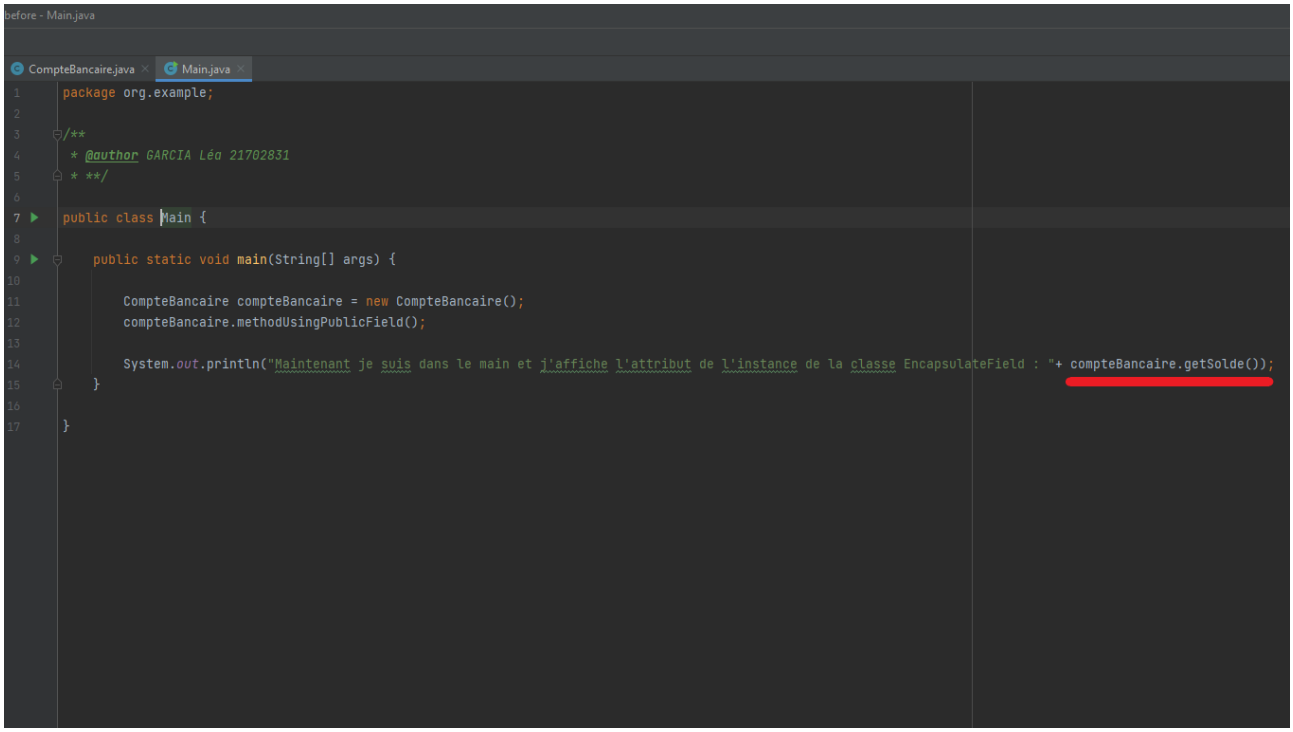

FIGURE 15 – Changements dans le main

3. Comparaison de 2 catalogues de refactorings

Refactoring proposé dans le catalogue en ligne mais pas sous IntelliJ : Combine Functions into Transform

Ce refactoring permet de prendre plusieurs fonctions ayant un paramètre en commun et de les combiner en une seule. Cela permet de créer des opérations complexes à partir d'opérations simples. Lien vers ce refactoring : https://www.refactoring.com/catalog/combineFunctionsIntoTransform.html

Refactoring proposé sous IntelliJ mais pas dans le catalogue en ligne : Use Interface where possible

Ce refactoring ne modifie ni le code des classes ni le code de l'interface, mais remplace les références à des classes par leur interface parente là ou c'est possible. Cela permet de rendre le code moins spécifique, et donc plus flexible.

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

Question 1

Après avoir essayé de différentes manières d'appliquer l'opération de refactoring "extract interface" sur l'ensemble des classes du sujet de TP, nous n'avons pas réussi à extraire une interface commune à toutes les classes, on peut seulement extraire une interface à partir d'une seule classe à la fois. Si l'on extrait les interfaces de chaque classe une à une, on crée une interface pour chaque classe contenant toutes les méthodes extraites.

Question 2

Non, IntelliJ n'a pas factorisé les méthodes entre interface. Cela ne serait pas idéal dans tous les cas, car chaque classe implémente son interface spécifique. On préfèrerait que les classes aient une interface commune avec un nombre maximal de méthodes communes extraites, et éventuellement des sous-interfaces pour les spécificités de certaines classes.

Question 3

```java
package classes;

import interfaces.SubListe1;
import interfaces.SubListe2;

class ListeChainee implements SubListe1, SubListe2 {
    @Override
    public boolean add(Object o) {return true;}
    @Override
    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(){}
}
```

```java
package classes;

import interfaces.SubListe2;

class ListeTableau implements SubListe2 {
    @Override
    public boolean add(Object o) {return true;}
    @Override
    public boolean isEmpty() {return true;}
    public Object get(int i) {return null;}
    private void secretLT(){}
    public static void staticLT() {}
    int nbLT;
}
```

```java
package classes;

import interfaces.SubListe1;

class QueueAvecPriorite implements SubListe1 {
    @Override
    public boolean add(Object o) {return true;}
    @Override
    public boolean isEmpty() {return true;}
    public Object peek() {return null;}
    public Object poll() {return null;}
    public Object comparator() {return null;}
    private void secretQAP(){}
}
```

```java
package classes;

import interfaces.SubListe1;

class QueueDoubleEntree implements SubListe1 {
    @Override
    public boolean add(Object o) {return true;}
    @Override
    public boolean isEmpty() {return true;}
    public Object peek() {return null;}
    public Object poll() {return null;}
    private void secretQDE(){}
}
```

```java
package interfaces;

public interface Liste {
    boolean add(Object o);

    boolean isEmpty();
}
```

```java
package interfaces;

public interface SubListe1 extends Liste{
    public Object peek();
    public Object poll();
}
```

```java
package interfaces;

public interface SubListe2 extends Liste{
    public Object get(int i);
}
```

Question 4

Nous avons créé comme indiqué un fichier csv pour les classes ListeTableau, ListeChainee, QueueDoubleEntree, QueueAvecPriorite, avec les méthodes add, isEmpty, get, peek et poll. Voici le csv et le AOC-Poset résultant :

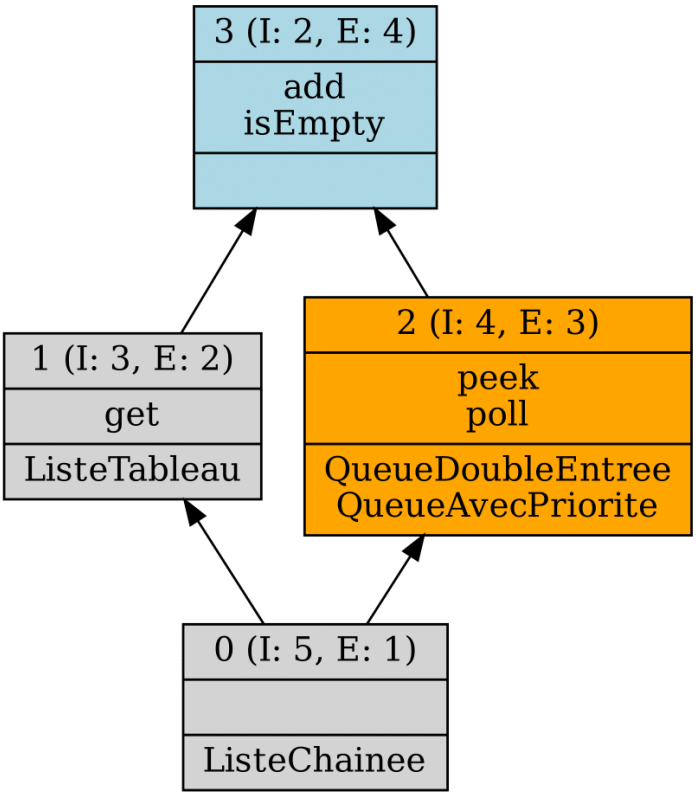

FIGURE 16 – AOC-Poset généré grâce au CSV

```
;add;isEmpty;get;peek;poll
ListeTableau;1;1;1;0;0
ListeChainee;1;1;1;1;1
QueueDoubleEntree;1;1;0;1;1
QueueAvecPriorite;1;1;0;1;1
```

Question 5

Le résultat produit par l'application de fca4j avec notre fichier csv nous a donné un AOC-poset pouvant se traduire en une hiérarchie d'interfaces, factorisée, et indiquant quelle interface chaque classe devrait implémenter.

En comparaison avec ce que nous avons implémenté à la main dans les questions précédentes, nous pouvoir voir que la hiérarchie des interfaces est la même. Cependant si nous avions eu besoin d'extraire à la main des interfaces depuis un plus grand nombre de classes, nous aurions mis beaucoup plus de temps et probablement obtenu un résultat peu optimisé et/ou avec des erreurs.

Cette approche fonctionne aussi bien car ce problème précis est exactement le genre de problèmes que l'analyse formelle de concepts permet de résoudre. Les classes sont équivalentes aux objets et les méthodes aux attributs. Il n'est donc pas surprenant qu'on obtienne un treillis se traduisant immédiatement en une hiérarchie d'interfaces optimisée. ListeChainee possédant toutes les méthodes il est logique qu'elle soit dans le concept tout en bas, par exemple. Toutes les classes ont add et empty, donc il y a un concept tout en haut les contenant, mais aucune ne possède uniquement ces deux méthodes, donc il n'y a pas d'objets dans ce concept. Le treillis se sépare en deux pour gérer les spécificités entre les deux classes Queue et ListeTableau.