

# TP3 : Opérations de refactoring

ADOLPHE Benjamin n°22008346


LAURET Nicolas n°22002804

## Exercice 1

Le code source ci-dessous est également disponible dans le dépôt de rendu.

GitHub - benjaminAd/Evolution-Et-Restructuration-des-Logiciels-TP3-Refactoring

Contribute to benjaminAd/Evolution-Et-Restructuration-des-Logiciels-TP3-Refactoring development by creating an account on GitHub.

 <https://github.com/benjaminAd/Evolution-Et-Restructuration-des-Logiciels-TP3-Refactoring>

benjaminAd/**Evolution-Et-Restructuration-des-...**



Contributor 0 Issues 0 Stars 0 Forks 0

## Exercice 2

### Refactoring du code de Nicolas

Pour refactoriser le code de Nicolas sur IntelliJ, j'ai utilisé l'opération de refactoring "*Move Class*" qui est traduite par "*Move type to new file*" dans Eclipse. Cette opération consiste à déplacer une classe dans un autre fichier. Une fois l'opération effectuée, un fichier `Carre.java` a été créé contenant le code de la classe `Carre`. J'ai ensuite exécuté la méthode `main` du programme et elle fonctionnait toujours.

```
public class Rectangle {
    protected int longueur;
    protected int largeur;

    public Rectangle(int longueur, int largeur) {
        this.longueur = longueur;
        this.largeur = largeur;
    }

    public int getLongueur() {
        return longueur;
    }

    ...

    @Override
    public String toString() {
        return "Rectangle{ longueur=" + longueur + ", largeur=" + largeur + " }";
    }
}

//TODO Move Carre to another file (Move type to new file)
class Carre extends Rectangle {

    public Carre(int longueur) {
        super(longueur, longueur);
    }

    @Override
    public String toString() {
        return "Carre{ longueur=" + longueur + ", largeur=" + largeur + " }";
    }
}
```

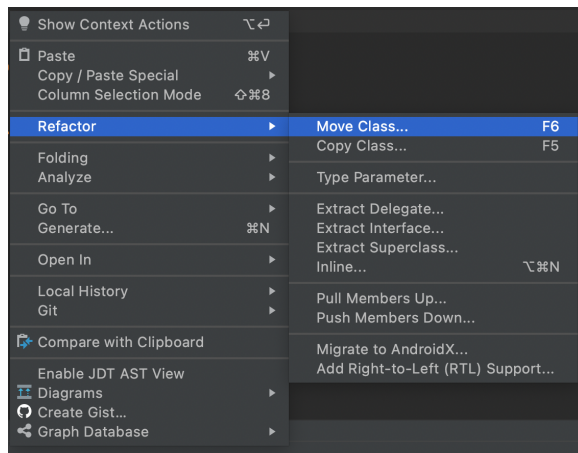


Figure 1 - Opérations de refactoring

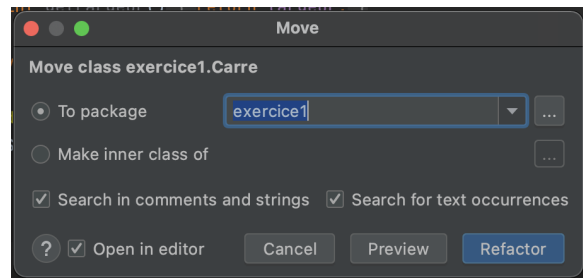


Figure 2 - Interface "Move Type to new File"

```
public class Rectangle {
    protected int longueur;
    protected int largeur;

    public Rectangle(int longueur, int largeur) {
        this.longueur = longueur;
        this.largeur = largeur;
    }

    public int getLongueur() {
        return longueur;
    }

    ...

    @Override
    public String toString() {
        return "Rectangle{ longueur=" + longueur + ", largeur=" + largeur + " }";
    }
}
```

```
//TODO Move Carre to another file (Move type to new file)
class Carre extends Rectangle {

    public Carre(int longueur) {
        super(longueur, longueur);
    }

    @Override
    public String toString() {
        return "Carre{ longueur=" + longueur + ", largeur=" + largeur + " }";
    }
}
```

### Refactoring du code de Benjamin

Pour refactor le code de Benjamin sur Eclipse, j'ai utilisé l'opération de refactoring "*Extract Superclass*" qui consiste à créer et déplacer dans une super-classe des attributs et méthodes de la classe courante. Une fois l'opération effectuée sur la classe `Hexagone`, une super-classe `Polygone` contenant les attributs et méthodes communs aux autres classes. J'ai ensuite exécuté la méthode `main` du programme et il fonctionnait toujours.

Cependant, j'ai pu constater les limites d'Eclipse en essayant de refactor une autre sous-classe de `Polygone` (`Rectangle`). En effet, il était impossible d'extraire la super-classe depuis `Rectangle` car la super-classe `Polygone` existait déjà. On peut donc en conclure qu'il est impossible sur Eclipse de refactor plusieurs classe en même temps afin d'y extraire une super-classe commune.

```
public class Hexagone {

    protected int nb_cotes;
    protected int longueur_cote;
    protected int angle;

    public Hexagone(int longueur_cote, int angle) {
        this.nb_cotes = 6;
        this.longueur_cote = longueur_cote;
        this.angle = angle;
    }

    public int getNb_cotes() {
        return nb_cotes;
    }

    ...

    @Override
    public String toString() {
        return "Hexagone{" +
            "nb_cotes=" + nb_cotes +
            ", longueur_cote=" + longueur_cote +
            ", angle=" + angle +
            "}";
    }
}
```

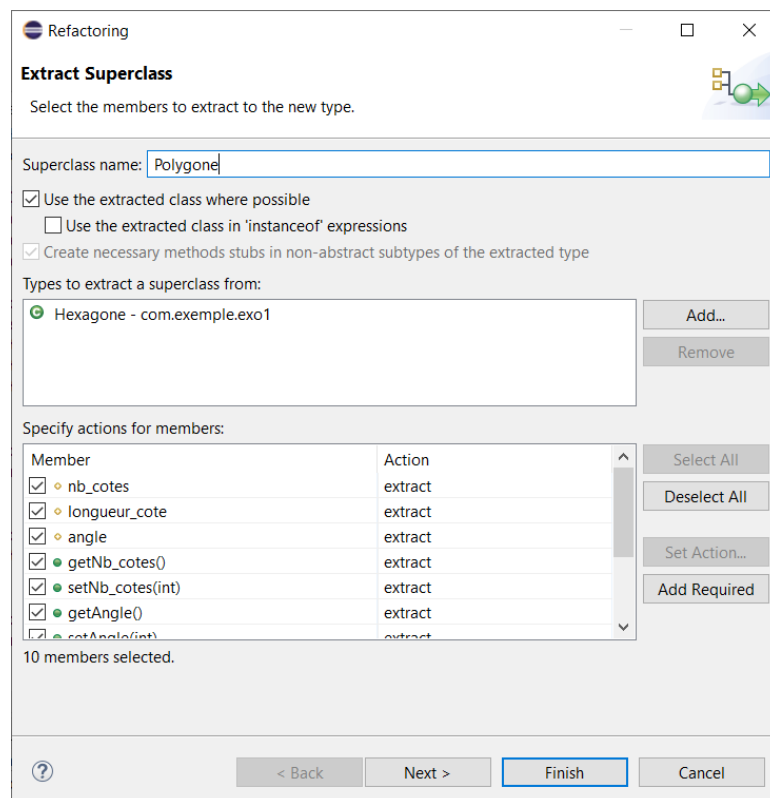


Figure 3 - Opérations de refactoring

```

public class Hexagone extends Polygone {
    public Hexagone(int longueur_cote, int angle) {
        super();
        this.nb_cotes = 6;
        this.longueur_cote = longueur_cote;
        this.angle = angle;
    }

    @Override
    public String toString() {
        return "Hexagone{" +
            "nb_cotes=" + nb_cotes +
            ", longueur_cote=" + longueur_cote +
            ", angle=" + angle +
            "}";
    }
}

```

```

public class Polygone {

    protected int nb_cotes;
    protected int longueur_cote;
    protected int angle;

    public Polygone() {
        super();
    }

    public int getNb_cotes() {
        return nb_cotes;
    }

    ...

    public void setLongueur_cote(int longueur_cote) {
        this.longueur_cote = longueur_cote;
    }
}

```

### Exercice 3

L'opération **Remove Dead Code** est présente dans le catalogue mais pas sur Eclipse/IntelliJ. Cette opération permet de supprimer le code mort. Le code mort est un code qui ne sera jamais appelé par le programme. Le code mort va rendre le code plus difficile à lire et à comprendre.

```

if(false) {
    doSomethingThatUsedToMatter();
}
    ↓
//Rien -> Le code est effacé

```

L'opération **Generalize Declared Type** est proposée par Eclipse mais pas par le catalogue. Cette opération de refactoring permet de vérifier les possibles généralisations des types déclarés en détectant la non-utilisation des abstractions appropriées.

```

HashSet set = new HashSet();
    ↓
Set set = new HashSet();

```

## Exercice 4

En sélectionnant toutes les classes et en utilisant l'opération "*Extract Interface*", IntelliJ décide d'appliquer le refactoring uniquement sur la première classe du fichier c'est-à-dire `ListeTableau`.

Cependant si on sélectionne les classes une par une, IntelliJ pourra créer une interface par classe. On remarque cependant que les signatures de méthodes communes restent dupliquées entre les différentes interfaces lors du refactoring.

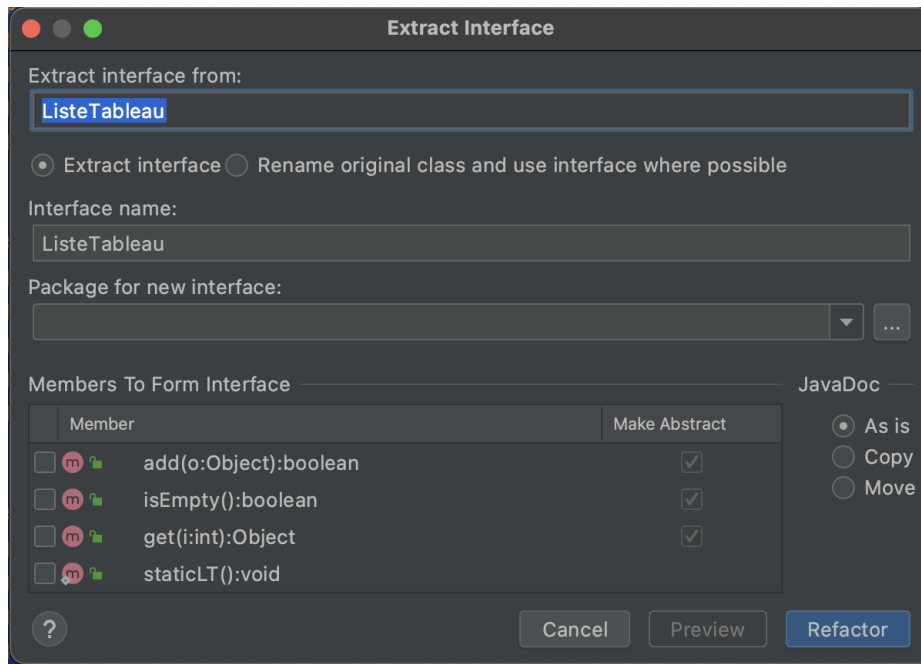


Figure 6 - Interface "Extract Interface" en sélectionnant toutes les classes

```
public interface IAListeTableau {  
    boolean add(Object o);  
    boolean isEmpty();  
    Object get(int i);  
}
```

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

## Exercice 5

En analysant le résultat produit, on remarque que pour chaque ensemble de signatures de méthodes communes, une interface est créée.

Il est différent de notre version car nous avons fait le choix de dupliquer la méthode `Object get(int i)` dans les interfaces `ListeTableau` et `ListeChaine` afin d'éviter de créer une interface pour une seule méthode, car à ce stade nous ne savions pas ce qui était le mieux entre dupliquer une seule méthode ou créer une interface pour une seule méthode.

À l'aide de l'AOCposet on a pu se rendre compte que notre version devrait utiliser une interface contenant la méthode `Object get(int i)` au lieu de dupliquer la méthode.

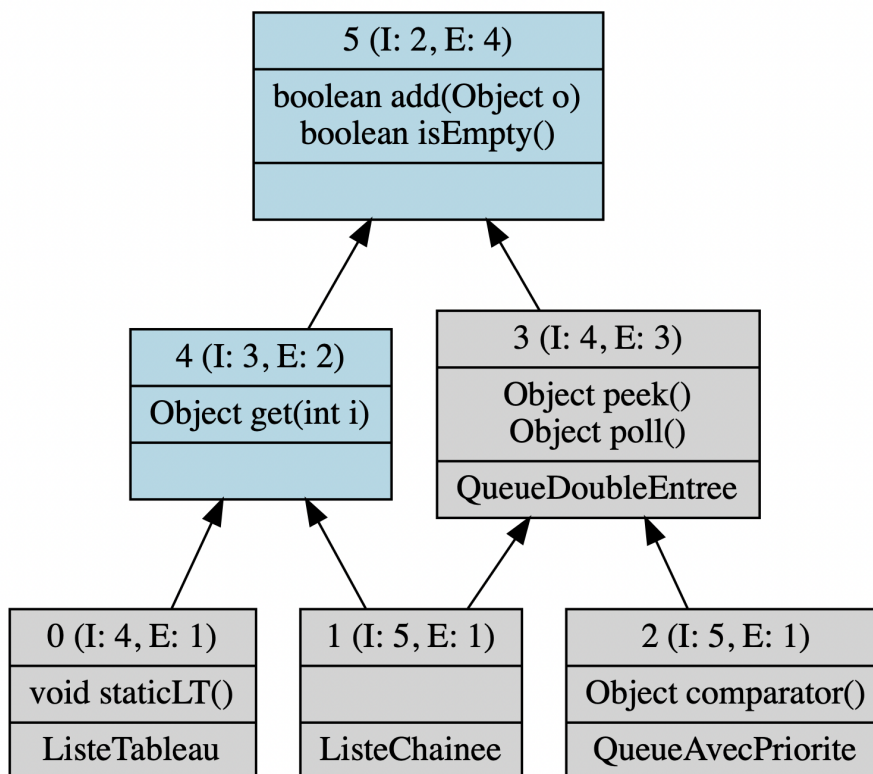


Figure 7 - AOCposet généré