



---

## Rapport TP3 - Refactoring

HAI913I - Évolution / Restructuration des logiciels

*Département informatique*

---

Jérémie BENTOLILA 21906253

Adam SAID 21905365

Arnaud COSSU

28 novembre 2023

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contenu du rapport . . . . .	1
1.2	Utilisation . . . . .	1
1.3	Exécution du TP . . . . .	1
<b>2</b>	<b>ExtractSuperclass (par Adam SAID)</b>	<b>1</b>
2.1	Le projet initial . . . . .	1
2.2	Le projet corrigé . . . . .	3
2.3	Avec ChatGPT . . . . .	5
<b>3</b>	<b>FieldEncapsulation (par Jérémie BENTOLILA)</b>	<b>8</b>
3.1	Le projet initial . . . . .	8
3.2	Le projet corrigé . . . . .	9
3.3	Avec ChatGPT . . . . .	11
<b>4</b>	<b>PushDown (par Arnaud COSSU)</b>	<b>13</b>
4.1	Le projet initial . . . . .	13
4.2	Le projet corrigé . . . . .	15
4.3	Avec ChatGPT . . . . .	18
<b>5</b>	<b>ExtractInterface</b>	<b>21</b>
5.1	Par Eclipse . . . . .	21
5.2	Par l’Homme . . . . .	24
5.3	Par l’analyse formelle . . . . .	25
5.4	Par ChatGPT . . . . .	26

# 1 Introduction

## 1.1 Contenu du rapport

Dans le cadre de ce TP, seront appliquées diverses techniques de refactoring, qui ont pour but d'améliorer la lisibilité et la qualité de projets que nous créerons, étant volontairement mal formés et/ou ne respectant pas les bonnes pratiques en programmation. Nous présenterons les étapes et les techniques utilisées pour apporter une solution quant à la transformation de ces projets. Nous expliquerons aussi les choix ayant été faits quant à ces opérations s'il est pertinent de le faire.

## 1.2 Utilisation

Si vous voulez exécuter / tester ce projet, vous pouvez retrouver le code sur la page [suiivante](#). Pour le lancer vous pouvez vous référer au README du projet ou bien à la section 1.3 de ce rapport.

## 1.3 Exécution du TP

Pour exécuter le TP :

1. Extraire l'archive dans un dossier
2. Ouvrir une console de commande à la racine du projet
3. Taper les commandes suivantes :

# 2 ExtractSuperclass (par Adam SAID)

Le premier type de refactoring mis en oeuvre ici est l'extraction de super-classe. Il est utilisé lorsque deux classes plutôt similaire (par leurs attributs) pourraient être regroupées en une super-classe contenant tous les attributs commun à ces deux classes.

## 2.1 Le projet initial

Voici le programme nécessitant une opération de refactoring ExtractSuperclass :

```
public class Car {  
    private String brand;  
    private int year;  
    private int numberOfDoors;
```

```

    public Car(String brand, int year, int numberOfDoors) {
        this.brand = brand;
        this.year = year;
        this.numberOfDoors = numberOfDoors;
    }

    // Getters et setters...
}

public class Motorcycle {
    private String brand;
    private int year;
    private boolean hasSideCar;

    public Motorcycle(String brand, int year, boolean hasSideCar) {
        this.brand = brand;
        this.year = year;
        this.hasSideCar = hasSideCar;
    }

    // Getters et setters...
}

public class Main {
    public static void main(String[] args) {
        // Dans ce programme les classes Car et Motorcycle ont des attributs communs
        // Il serait préférable d'avoir une super-classe pour regrouper ces 2 attributs

        // Utilisation de la classe Car
        Car car = new Car("Toyota", 2022, 4);
        System.out.println("Car Brand: " + car.getBrand());
        System.out.println("Car Year: " + car.getYear());
        System.out.println("Number of Doors: " + car.getNumberOfDoors());
        car.drive();
        System.out.println();

        // Utilisation de la classe Motorcycle
        Motorcycle motorcycle = new Motorcycle("Harley-Davidson", 2022, false);
        System.out.println("Motorcycle Brand: " + motorcycle.getBrand());
        System.out.println("Motorcycle Year: " + motorcycle.getYear());
        System.out.println("Has SideCar: " + motorcycle.isHasSideCar());
        motorcycle.ride();
    }
}

```

## 2.2 Le projet corrigé

Le programme possédait des classes ayant des attributs communs pouvant être rassemblés en une classe commune `Vehicle`.

Il existe dans Eclipse une fonctionnalité 'extract class', permettant à partir de deux classes ou plus d'extraire une classe de données. Cette fonctionnalité n'est pas adaptée et ne répond pas à notre problématique.

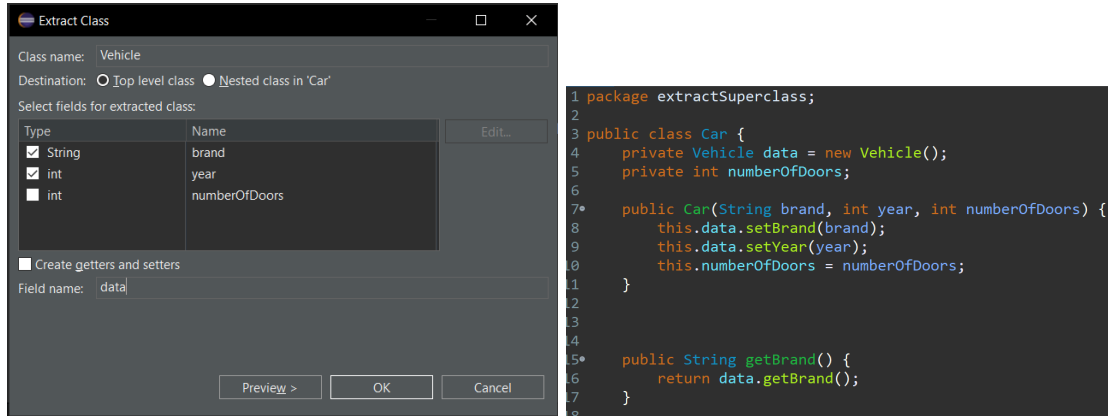


FIGURE 1 – Mauvaise solution

Il existe une autre fonctionnalité, permettant à partir d'une classe d'extraire une superclasse en choisissant les attributs / méthodes à extraire, et dans laquelle nous pouvons ajouter des classes en second pour lesquelles on souhaite extraire les attributs :

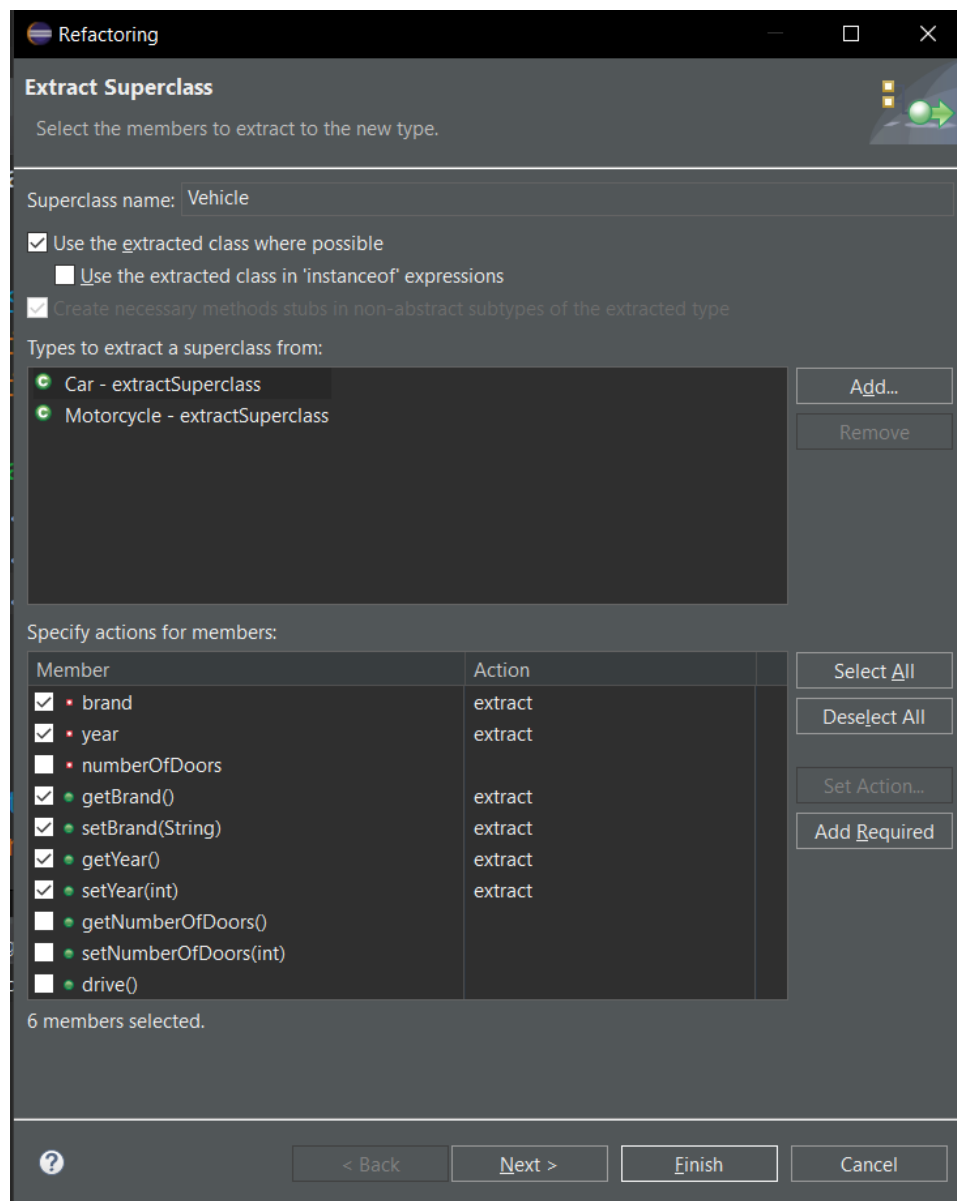


FIGURE 2 – Extraction d'une superclasse

Nous obtenons alors ici la classe Vehicle suivante :

```
public class Vehicle {
    protected String brand;
    protected int year;

    public Vehicle() {
        super();
    }
}
```

```

    }
    // getters, setters pour brand et year
}

```

Et la classe `Car` suivante :

```

public class Car extends Vehicle {
    private int numberOfDoors;

    public Car(String brand, int year, int numberOfDoors) {
        super();
        this.brand = brand;
        this.year = year;
        this.numberOfDoors = numberOfDoors;
    }
    // code inchangé par la suite
}

```

Nous remarquons ici est que les attributs / méthodes des classes ajoutées en second ne sont pas extraits (par exemple ici seuls les attributs / méthodes de `Car` ont été extraits, et l'attribut `hasSideCar` de `Motorcycle` n'apparaît pas). Le seul changement notable sur la classe `Motorcycle` est l'ajout de `extends Vehicle` sur sa déclaration.

Pour pallier le problème, nous pouvons simplement enlever dans la classe `Motorcycle` les attributs en doublon. Nous obtenons alors la classe `Motorcycle` suivante :

```

public class Motorcycle extends Vehicle {

    private boolean hasSideCar;

    public Motorcycle(String brand, int year, boolean hasSideCar) {
        super(brand, year);
        this.hasSideCar = hasSideCar;
    }
    // code inchangé
}

```

Pour ce refactoring, il n'est pas nécessaire de modifier le main, celui-ci fonctionne même après l'extraction de notre superclasse.

## 2.3 Avec ChatGPT

J'ai demandé à GPT de corriger mon code sans même lui préciser l'action à effectuer et voila le résultat :

PROMPT: Voici mon code pour un programme simple en Java. Je voudrais que si nécessaire tu appliques un refactoring dessus : "suivi du code du projet"

Réponse: Votre idée de regrouper les attributs communs des classes Car et Motorcycle dans une super-classe est bonne. Cela permettra de réduire la duplication de code. Je vais vous montrer comment vous pouvez créer une super-classe Vehicle pour regrouper ces attributs communs.

```
package extractSuperclass;

public class Main {
    public static void main(String[] args) {
        // Utilisation de la classe Car
        Car car = new Car("Toyota", 2022, 4);
        System.out.println("Car Brand: " + car.getBrand());
        System.out.println("Car Year: " + car.getYear());
        System.out.println("Number of Doors: " + car.getNumberOfDoors());
        car.drive();
        System.out.println();

        // Utilisation de la classe Motorcycle
        Motorcycle motorcycle = new Motorcycle("Harley-Davidson", 2022, false);
        System.out.println("Motorcycle Brand: " + motorcycle.getBrand());
        System.out.println("Motorcycle Year: " + motorcycle.getYear());
        System.out.println("Has SideCar: " + motorcycle.isHasSideCar());
        motorcycle.ride();
    }
}

package extractSuperclass;

public class Vehicle {
    private String brand;
    private int year;

    public Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }
}
```



```

    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }
}

package extractSuperclass;

public class Car extends Vehicle {
    private int numberOfDoors;

    public Car(String brand, int year, int numberOfDoors) {
        super(brand, year);
        this.numberOfDoors = numberOfDoors;
    }

    // Getter et setter...
}

package extractSuperclass;

public class Motorcycle extends Vehicle {
    private boolean hasSideCar;

    public Motorcycle(String brand, int year, boolean hasSideCar) {
        super(brand, year);
        this.hasSideCar = hasSideCar;
    }

    // Getter et setter...
}

```

J'ai créé une nouvelle classe `Vehicle` qui contient les attributs communs `brand` et `year`. Les classes `Car` et `Motorcycle` étendent maintenant cette classe, héritant ainsi des attributs communs. Cela réduit la duplication de code et permet d'ajouter d'autres fonctionnalités communes à l'avenir, si nécessaire.

On constate donc que GPT a compris de lui même ce qui n'allait pas et à appliquer un refactoring correct en créant une classe véhicule possédant les attributs commun à `Car` et `Motorcycle` et à même expliqué ses actions à la fin.

## 3 FieldEncapsulation (par Jérémie BENTOLILA)

Le premier type de refactoring mis en oeuvre ici est l'encapsulation de champs. Il est utilisé lorsque les champs sont définis en tant que `public`, alors que l'on souhaiterait plutôt les mettre en `private` et accéder à leur valeurs soit par le biais de **getters** (pour la lecture) ou bien des **setters** (pour l'écriture).

### 3.1 Le projet initial

Le programme suivant a été écrit afin d'illustrer ce refactoring :

```
public class Employee {
    public String name;
    public int age;
    public double salary;

    public Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public void displayEmployeeDetails() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
    }
}

public class Manager extends Employee {
    public double bonus;

    public Manager(String name, int age, double salary, double bonus) {
        super(name, age, salary);
        this.bonus = bonus;
    }

    public void displayManagerDetails() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Bonus: " + bonus);
    }
}

public class Main {
```

```

public static void main(String[] args) {
    Employee emp = new Employee("John Doe", 30, 50000.0);
    emp.displayEmployeeDetails();

    Manager manager = new Manager("Jane Smith", 35, 80000.0, 1000.);
    manager.displayManagerDetails();

    // Utilisation directe des champs publics d'un objet Employee
    emp.name = "Alice Johnson";
    emp.age = 25;
    emp.salary = 60000.0;
    emp.displayEmployeeDetails();

    // Utilisation directe des champs publics d'un objet Manager
    manager.name = "Bob Williams";
    manager.age = 56;
    manager.salary = 90000.;
    manager.bonus = 5000.0;
    manager.displayManagerDetails();
}
}

```

### 3.2 Le projet corrigé

Le programme présentait des attributs des classes en public, ce qui est contraire aux principes de la programmation objets car cela permettrait à n'importe qui de modifier ces attributs sans vérification ni contrôle.

Pour corriger ce problème j'ai passé les attributs public des classes "Employee" et "Manager" en `private` et j'ai généré les getters et setters automatiquement grâce à Eclipse :

```

package models;

public class Manager extends Employee {
    private double bonus;

    public Manager(String name, int age, double salary, double bonus) {
        super(name, age, salary);
        this.bonus = bonus;
    }
}

```

FIGURE 3 – Passage des attributs en private

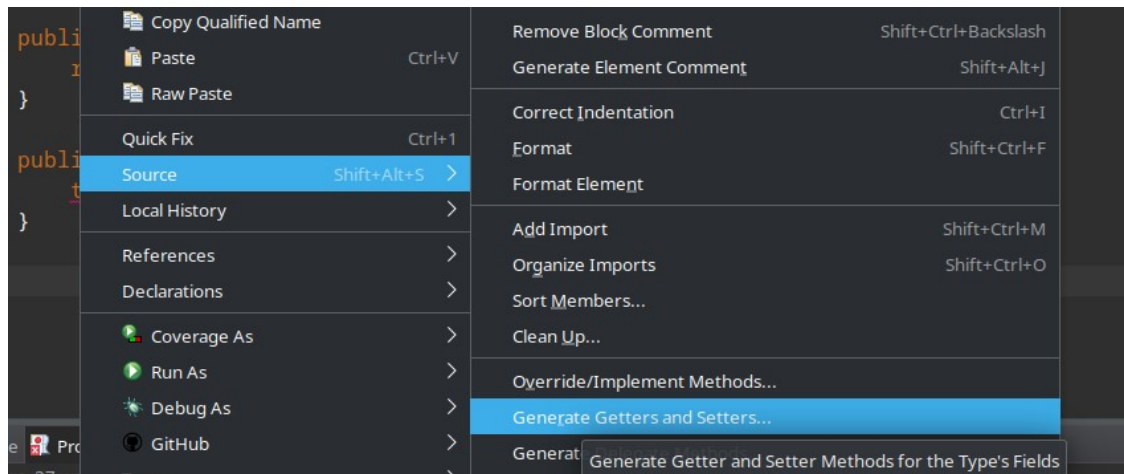


FIGURE 4 – Génération des getters et setters avec Eclipse

Grâce à Eclipse il est très facile de générer tout les accesseurs à nos attributs, cela permet de gagner du temps et de ne pas en oublier. Cependant il pourrait être intéressant d'avoir une fonction pour basculer tous ou certains de nos attributs à une certaine visibilité. Ce qui automatiserait encore plus le refactoring.

Il a par contre fallu modifier le main car l'accès aux attributs n'est plus possible en faisant un appel de message sur l'objet lui même, il a donc fallu changer le code pour passer par les accesseurs :

```
// Code précédent
// Utilisation directe des champs publics d'un objet Employee
// emp.name = "Alice Johnson";
// emp.age = 25;
// emp.salary = 60000.0;
// emp.displayEmployeeDetails();
// Utilisation directe des champs publics d'un objet Manager
// manager.name = "Bob Williams";
// manager.age = 56;
// manager.salary = 90000.;
// manager.bonus = 5000.0;
// manager.displayManagerDetails();

// Code actuel
// Updating via setters
emp.setName("Alice Johnson");
emp.setAge(25);
emp.setSalary(60000);
```

```

emp.displayEmployeeDetails();

manager.setName("Bob Williams");
manager.setAge(56);
manager.setSalary(90000);
manager.setBonus(5000);
manager.displayManagerDetails();

```

### 3.3 Avec ChatGPT

Voici ce que nous pouvons obtenir en demandant de l'aide à chatGPT :

PROMPT: Voici quelques classes d'un petit projet java, si tu vois un refactoring possible pour celui-ci applique-le : \*code décrit dans la section 3.1\*

Réponse: Dans le code fourni, les champs des classes Employee et Manager sont déclarés comme publics, ce qui permet un accès direct et non restreint à leurs valeurs. Cela viole le principe d'encapsulation et peut mener à des problèmes de contrôle et de maintenabilité.

Pour améliorer cela, je suggère d'appliquer le principe d'encapsulation en rendant les champs privés et en fournissant des méthodes publiques (getters et setters) pour accéder et modifier ces champs. De plus, je propose de réduire la redondance de code en évitant la répétition de la logique d'affichage des détails des employés et des managers.

Voici une version refactorisée du code en utilisant l'encapsulation et en réorganisant les méthodes d'affichage pour éviter la duplication :

```

class Employee {
    private String name;
    private int age;
    private double salary;

    public Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

```

```

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
    }
}

class Manager extends Employee {
    private double bonus;

    public Manager(String name, int age, double salary, double bonus) {
        super(name, age, salary);
        this.bonus = bonus;
    }

    public double getBonus() {
        return bonus;
    }

    public void setBonus(double bonus) {
        this.bonus = bonus;
    }

    @Override
    public void displayDetails() {
        super.displayDetails();
        System.out.println("Bonus: " + bonus);
    }
}

```

```

public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee("John Doe", 30, 50000.0);
        emp.displayDetails();

        Manager manager = new Manager("Jane Smith", 35, 80000.0, 1000.);
        manager.displayDetails();

        // Utilisation des setters pour modifier les valeurs
        emp.setName("Alice Johnson");
        emp.setAge(25);
        emp.setSalary(60000.0);
        emp.displayDetails();

        manager.setName("Bob Williams");
        manager.setAge(56);
        manager.setSalary(90000.);
        manager.setBonus(5000.0);
        manager.displayDetails();
    }
}

```

Dans cette version refactorée, les champs des classes `Employee` et `Manager` sont rendus privés et des méthodes getter et setter sont utilisées pour accéder et modifier ces champs. De plus, la méthode `displayDetails` est redéfinie dans la classe `Manager` pour afficher les détails spécifiques du manager, tout en appelant la méthode de la classe parente pour afficher les détails communs.

ChatGPT est en effet correct dans sa correction, et corrige bien notre problème d'encapsulation. Celui-ci va même au delà de nos attentes en nous suggérant de modifier notre méthode d'affichage sur `Manager` ; qui est en effet redondante et qui peut-être simplifiée à partir d'un appel à la méthode d'affichage de la classe mère (`Employee`).

## 4 PushDown (par Arnaud COSSU)

### 4.1 Le projet initial

Voici le programme nécessitant une opération de refactoring `ExtractSuperclass` :

```

package main;

public class Main {

```

```

        public static void main(String[] args) {
            System.out.println("Création d'une moto : ");
            System.out.println("-----");
            Moto moto = new Moto("Harley", 1235);
            System.out.println("Ma moto : " + moto.toString());

            System.out.println("Création d'une voiture : ");
            System.out.println("-----");
            Voiture voiture = new Voiture("Mazda RX-7", 321, 3, 23);
            System.out.println("Ma voiture : " + voiture.toString());
        }
    }

package main;
public class Moto extends Vehicule {
    public Moto() {
        super();
    }
    public Moto(String name, double horsePower) {
        super(name, horsePower);
    }
}

package main;
public class Vehicule {
    private String name;
    private double horsePower;
    private int doorNumber;
    private double trunkLiters;

    public Vehicule(String name, double horsePower, int doorNumber, double trunkLiters) {
        super();
        this.name = name;
        this.horsePower = horsePower;
        this.doorNumber = doorNumber;
        this.trunkLiters = trunkLiters;
    }

    public Vehicule(String name, double horsePower) {
        super();
        this.name = name;
        this.horsePower = horsePower;
    }

    public Vehicule() {}
}

```



```

@Override
public String toString() {
    return this.name + "\n"
        + "Chevaux : " + this.horsePower + "\n"
        + "Nombre de portes : " + this.doorNumber + "\n"
        + "Litrage du coffre : " + this.trunkLiters + "\n";
}

public String getName() {return name;}
public void setName(String name) {this.name = name;}
public double getHorsePower() {return horsePower;}
public void setHorsePower(double horsePower) {this.horsePower = horsePower;}
public int getDoorNumber() {return doorNumber;}
public void setDoorNumber(int doorNumber) {this.doorNumber = doorNumber;}
public double getTrunkLiters() {return trunkLiters;}
public void setTrunkLiters(double trunkLiters) {this.trunkLiters = trunkLiters;}
}

package main;
public class Voiture extends Vehicule {

    public Voiture() {
        super();
    }

    public Voiture(String name, double horsePower, int doorNumber, double trunkLiters) {
        super(name, horsePower, doorNumber, trunkLiters);
    }

    public Voiture(String name, double horsePower) {
        super(name, horsePower);
    }
}

```

## 4.2 Le projet corrigé

Les attributs `trunkLiters` et `doorNumber` étaient dans la classe mère `Vehicule` alors qu'ils ne sont utilisés que par la sous classe `Car` et pas `Moto`. Il est donc intéressant de passer ces deux attributs directement dans la classe fille pour garder une certaine cohérence et éviter d'accéder à des attributs non initialisés.

Pour corriger ce problème il a fallu supprimer les deux attributs et les mettre dans la classe fille manuellement puis générer les getters et setters associés. J'ai également mis à jour la méthode

toString de Car pour prendre en compte les nouveaux attributs :

```
public class Voiture extends Vehicule {
    private int doorNumber;
    private double trunkLiters;

    • public Voiture() {
        super();
    }

    • public Voiture(String name, double horsePower, int doorNumber, double trunkLiters) {
        super(name, horsePower);
        this.doorNumber = doorNumber;
        this.trunkLiters = trunkLiters;
    }

    • public Voiture(String name, double horsePower) {
        super(name, horsePower);
    }

    • public int getDoorNumber() {
        return doorNumber;
    }

    • public void setDoorNumber(int doorNumber) {
        this.doorNumber = doorNumber;
    }
}
```

FIGURE 5 – Ajouts des attributs et génération des accesseurs

```
@Override
public String toString() {
    return super.toString() + "doorNumber : " + doorNumber + "\ntrunkLiters : " + trunkLiters;
}
```

FIGURE 6 – Redéfinition de la méthode toString

Eclipse ne prend pas en compte le refactoring pushDown. Ce qui peut se comprendre car il peut être compliqué pour le programme d'analyser le code afin de voir les héritages et savoir qui utilise les attributs. A part donc la génération des accesseurs, le refactoring de ce programme a été fait complètement manuellement. Par contre le main n'a pas été modifié :

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Création d'une moto : ");
        System.out.println("-----");
        Moto moto = new Moto("Harley", 1235);
        System.out.println("Ma moto : " + moto.toString());
    }
}
```

```

        System.out.println("Création d'une voiture : ");
        System.out.println("-----");
        Voiture voiture = new Voiture("Mazda RX-7", 321, 3, 23);
        System.out.println("Ma voiture : " + voiture.toString());

    }

}

public class Vehicule {

    private String name;
    private double horsePower;

    public Vehicule(String name, double horsePower) {
        super();
        this.name = name;
        this.horsePower = horsePower;
    }

    public Vehicule() {}

    @Override
    public String toString() {
        return this.name + "\n"
            + "Chevaux : " + this.horsePower + "\n";
    }

    public String getName() {
        return name;
    }

    // Autres accesseurs
}

public class Voiture extends Vehicule {
    private int doorNumber;
    private double trunkLiters;

    public Voiture() {
        super();
    }

    public Voiture(String name, double horsePower, int doorNumber, double trunkLiters) {
        super(name, horsePower);
    }
}

```

```

        this.doorNumber = doorNumber;
        this.trunkLiters = trunkLiters;
    }

    public Voiture(String name, double horsePower) {
        super(name, horsePower);
    }

    public int getDoorNumber() {
        return doorNumber;
    }

    // Autres accesseurs

    @Override
    public String toString() {
        return super.toString() + "doorNumber : " + doorNumber + "\ntrunkLiters : "
            + trunkLiters;
    }
}

// La classe moto n'a pas été modifiée

```

### 4.3 Avec ChatGPT

Voici ce que nous pouvons obtenir en demandant de l'aide à chatGPT :

PROMPT: Voici quelques classes d'un petit projet java, si tu vois un refactoring possible pour celui-ci applique-le : \*code décrit dans la section (code initial fourni) 3.1\*

Réponse: La méthode de refactorisation "Push Down" consiste à déplacer des membres (méthodes ou champs) d'une classe parente vers une classe enfant. Dans votre cas, nous pouvons appliquer cette méthode en déplaçant les champs spécifiques à la classe Voiture dans cette classe plutôt que de les laisser dans la classe Vehicule. Cela simplifiera la conception et permettra une meilleure encapsulation des détails spécifiques à la voiture. Voici le code refactored en utilisant la méthode "Push Down" :

```

package main;

public class Main {

    public static void main(String[] args) {
        System.out.println("Création d'une moto : ");
        System.out.println("-----");
    }
}

```

```

        Moto moto = new Moto("Harley", 1235);
        System.out.println("Ma moto : " + moto.toString());

        System.out.println("Création d'une voiture : ");
        System.out.println("-----");
        Voiture voiture = new Voiture("Mazda RX-7", 321, 3, 23);
        System.out.println("Ma voiture : " + voiture.toString());
    }
}

package main;

public class Moto extends Vehicule {
    public Moto() {
        super();
    }

    public Moto(String name, double horsepower) {
        super(name, horsepower);
    }
}

package main;

public class Vehicule {
    private String name;
    private double horsepower;

    public Vehicule(String name, double horsepower) {
        this.name = name;
        this.horsePower = horsepower;
    }

    public Vehicule() {
    }

    @Override
    public String toString() {
        return this.name + "\n"
            + "Chevaux : " + this.horsePower + "\n";
    }

    public String getName() {
        return name;
    }
}

```

```

    public void setName(String name) {
        this.name = name;
    }

    public double getHorsePower() {
        return horsePower;
    }

    public void setHorsePower(double horsePower) {
        this.horsePower = horsePower;
    }
}

package main;

public class Voiture extends Vehicule {
    private int doorNumber;
    private double trunkLiters;

    public Voiture() {
        super();
    }

    public Voiture(String name, double horsePower, int doorNumber, double trunkLiters) {
        super(name, horsePower);
        this.doorNumber = doorNumber;
        this.trunkLiters = trunkLiters;
    }

    @Override
    public String toString() {
        return super.toString()
            + "Nombre de portes : " + this.doorNumber + "\n"
            + "Litrage du coffre : " + this.trunkLiters + "\n";
    }

    public int getDoorNumber() {
        return doorNumber;
    }

    public void setDoorNumber(int doorNumber) {
        this.doorNumber = doorNumber;
    }

    public double getTrunkLiters() {
        return trunkLiters;
    }
}

```

```

    }

    public void setTrunkLiters(double trunkLiters) {
        this.trunkLiters = trunkLiters;
    }
}

```

Cette refactorisation permet de déplacer les champs spécifiques à la classe Voiture dans cette classe, ce qui améliore la lisibilité et la maintenance du code.

J'ai réalisé un premier prompt qui était le suivant :

**Voici le code de mon projet, si tu vois un refactoring possible fais le et explique moi la méthode de refactoring utilisée**

Or, ce dernier m'a uniquement proposé des améliorations graphiques, comme l'ajout de séparateurs, ou de print dans le code. Voici un aperçu de sa première réponse :

Réponse Votre code est déjà assez propre, mais nous pouvons effectuer quelques refactoiings pour le rendre plus lisible et maintenable. Voici quelques suggestions : Utiliser des constantes pour les messages de sortie : Utilisez des constantes pour les messages répétés afin de faciliter la modification et la gestion des sorties. Utiliser des méthodes pour l'affichage : Créez des méthodes dédiées pour l'affichage de la moto et de la voiture. Supprimer les constructeurs vides inutiles : Les constructeurs vides dans Voiture et Moto sont redondants, car la classe parent Vehicule a déjà un constructeur vide. Ajouter des commentaires pour la documentation : Ajoutez des commentaires pour expliquer brièvement le but de chaque classe et méthode.

## 5 ExtractInterface

### 5.1 Par Eclipse

Le refactoring ExtractInterface vise à extraire à partir d'une classe une interface type contenant tout les attributs et méthodes de la classe et à changer cette dernière pour expliciter son implémentation de la classe.

Sur le code donné en sélectionnant toutes les classes en même temps la méthode d'Eclipse ne renvoi rien surement car les classes possèdent des différences :

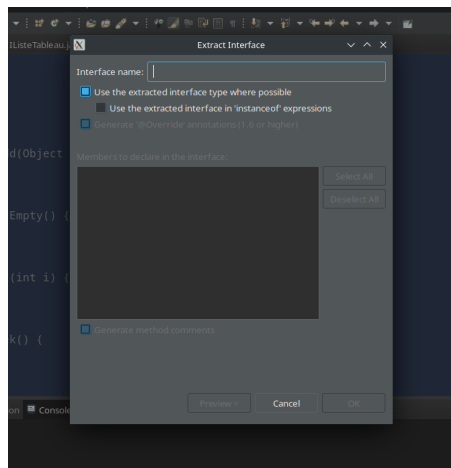


FIGURE 7 – On peut voir que Eclipse ne sait pas quoi mettre dans l'interface

A l'inverse en sélectionnant les classes une par une Eclipse arrive bien à extraire une interface :

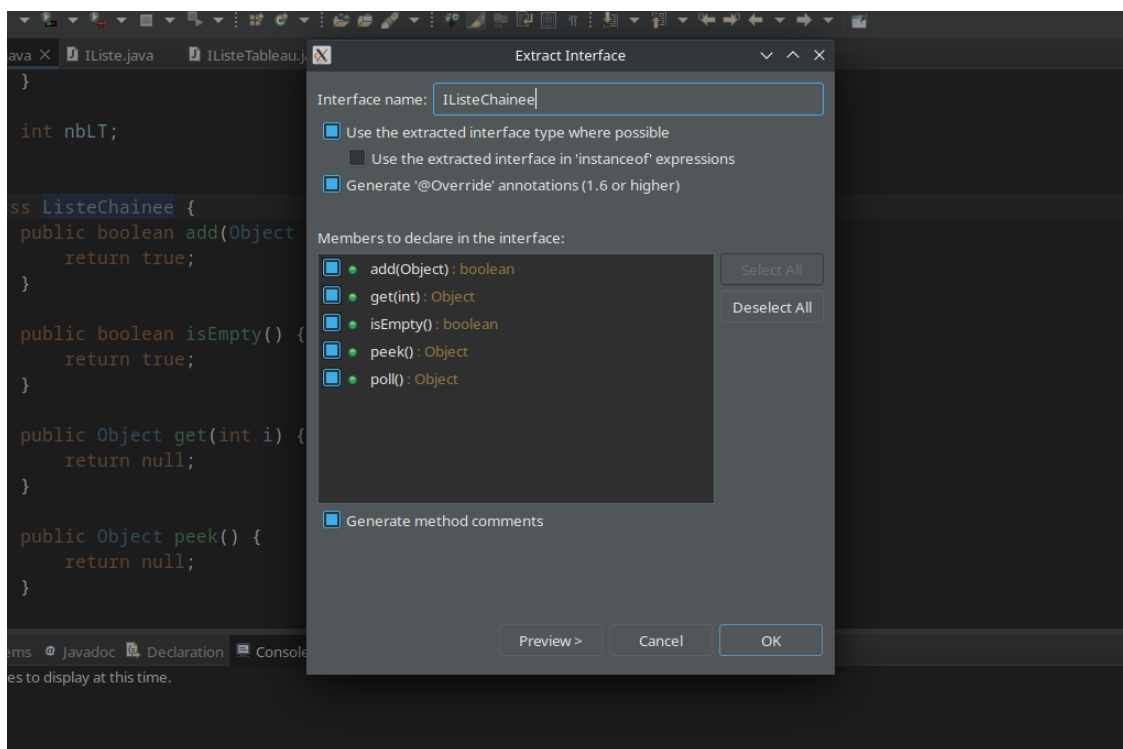


FIGURE 8 – Pour la classe ListeChaine Eclipse à trouver quoi extraire de la classe



Voilà l'interface extraire par Eclipse pour la classe ListeChaine :

```
interface IListeChaine {  
  
    boolean add(Object o);  
  
    boolean isEmpty();  
  
    Object get(int i);  
  
    Object peek();  
  
    Object poll();  
  
}
```

Voilà la réécriture de la classe après refactoring par Eclipse :

```
class ListeChaine implements IListeChaine {  
    @Override  
    public boolean add(Object o) {  
        return true;  
    }  
  
    @Override  
    public boolean isEmpty() {  
        return true;  
    }  
  
    @Override  
    public Object get(int i) {  
        return null;  
    }  
  
    @Override  
    public Object peek() {  
        return null;  
    }  
  
    @Override  
    public Object poll() {  
        return null;  
    }  
}
```

```

        private void secretLC() {
        }
    }
}

```

## 5.2 Par l'Homme

Le problème est qu'en ne sélectionnant les classes qu'une par une il n'y a pas de groupement des méthodes et on se retrouve encore avec nos duplicata de méthodes. Pour avoir une hiérarchie de classes sans duplication il faudrait avoir trois interfaces : une générale avec les méthodes présentes partout `add` et `isEmpty`, une qui étendrait cette première en ajoutant la méthode `get` et une autre qui étendrait la première interface en ajoutant les méthodes `peek` et `poll`. On pourrait en suite avoir nos classes qui implémenteraient ces interfaces, avec `ListeTableau` qui implémenterait la première et la seconde interface et qui ajouterait `secretLT()` et `staticLT()`. `ListeChaine` qui implémente les trois et ajoute `secretLC()`. La classe `QueueDoubleEntree` qui implémenterait les interfaces une et trois et ajouterait `secretQDE()`. `QueueAvecPriorite` qui implémente les interfaces une et trois et ajouterait `comparator()` et `secretQAP()`.

Voici une bonne hiérarchie qui éviterait les doublons en utilisant des interfaces :

```

// Interface commune pour les listes et les queues
interface IListQueue {
    boolean add(Object o);
    boolean isEmpty();
}

// Interface commune pour les listes
interface IList extends IListQueue {
    Object get(int i);
}

// Interface commune pour les queues
interface IQueue extends IListQueue {
    Object peek();
    Object poll();
}

class ListeTableau implements IList {
    private void secretLT() {
    }

    public static void staticLT() {
    }
}

```

```

class ListeChaine implements IList, IQueue {
    private void secretLC() {
    }
}

class QueueDoubleEntree implements IQueue {
    private void secretQDE() {
    }
}

class QueueAvecPriorite implements IQueue {
    public Object comparator() {
        return null;
    }

    private void secretQAP() {
    }
}

```

### 5.3 Par l'analyse formelle

En utilisant un algorithme de factorisation des attributs maximale et compacte on se retrouve avec le même résultat, ce qui nous évite de réfléchir et faire le code à la main.

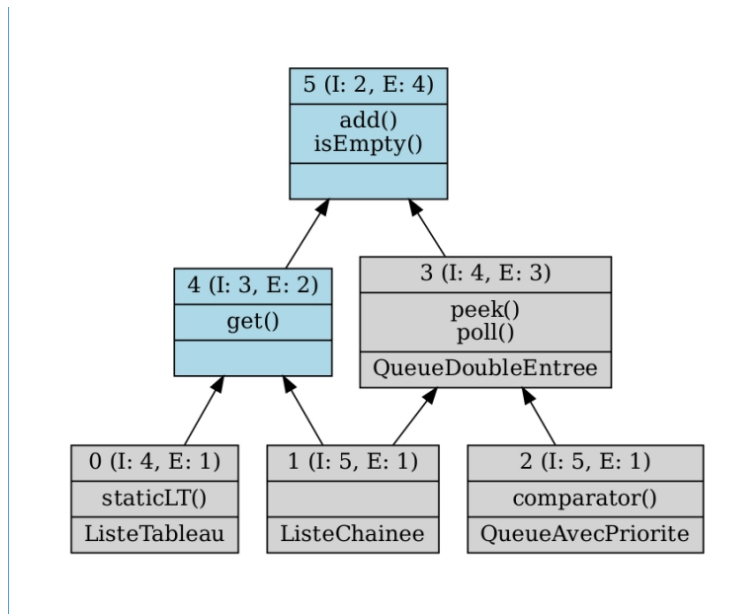


FIGURE 9 – Refactoring renvoyé par AOCposet

## 5.4 Par ChatGPT

On peut également demander à ChatGPT d'appliquer le refactoring sur notre code, voilà le résultat :

PROMPT: Refait le code suivant en créant des interfaces pour éviter les duplicata (code du projet).

REPONSE:

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

interface Queue {
    boolean add(Object o);
    boolean isEmpty();
    Object peek();
    Object poll();
}
```

```

interface PriorityQueue extends Queue {
    Object comparator();
}

```

```

class ArrayList implements List {
    @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() {
        // Implementation spécifique
    }

    public static void staticLT() {
        // Implementation spécifique
    }

    int nbLT;
}

```

```

class LinkedList implements List {
    @Override
    public boolean add(Object o) {
        return true;
    }

    @Override
    public boolean isEmpty() {
        return true;
    }

    @Override
    public Object get(int i) {

```

```

        return null;
    }

    public Object peek() {
        return null;
    }

    public Object poll() {
        return null;
    }

    private void secretLC() {
        // Implementation spécifique
    }
}

class DoubleEndedQueue implements Queue {
    @Override
    public boolean add(Object o) {
        return true;
    }

    @Override
    public boolean isEmpty() {
        return true;
    }

    @Override
    public Object peek() {
        return null;
    }

    @Override
    public Object poll() {
        return null;
    }

    private void secretQDE() {
        // Implementation spécifique
    }
}

class PriorityQueueImpl implements PriorityQueue {
    @Override
    public boolean add(Object o) {
        return true;
    }
}

```

```

    }

    @Override
    public boolean isEmpty() {
        return true;
    }

    @Override
    public Object peek() {
        return null;
    }

    @Override
    public Object poll() {
        return null;
    }

    @Override
    public Object comparator() {
        return null;
    }

    private void secretQAP() {
        // Implementation spécifique
    }
}

```

On peut avoir que premièrement il a traduit nos noms de classes sans raison. De plus il a fait une interface List qui pourrait être scindé en deux ce qui éviterait les duplicata avec Queue (on notera au passage qu'il a appelé l'interface List alors qu'en Java cette interface existe déjà). De plus il a créé une classe ArrayList qui existe déjà en Java. Outre ces erreurs la création d'interface n'est pas trop mal et les classes implémente bien ces interfaces pour éviter une partie des duplicata. En résumé ça n'est pas parfait mais avec quelques essais en plus on pourrait avoir un résultat convenable.