



HAI913I

Module : Évolution et restructuration des logiciels

Rapport de TP 2 : Compréhension des programmes

Réalisé par :

M^{lle} MEKHNACHE Toudherth – GL M2

Référent :

M^d Marianne HUCHARD

Promotion 2023/2024

Table des matières

Introduction	1
1 Refactoring en Java	3
1.1 Refactoring avec Extract SuperClass pour la classe 'Voiture'	3
1.1.1 Programme avant le refactoring	3
1.1.2 Programme après le refactoring	5
1.2 Application des opérations de refactoring sur les programmes d'un autre groupe	9
1.2.1 Programme avant le refactoring	9
1.2.2 Programme après le refactoring	11
1.3 Testez ce que ChatGPT4 peut réaliser	15
1.3.1 Refactoring : Création de la Superclasse Animal	15
1.3.2 Analyse des Résultats du Refactoring	17
1.3.3 Conclusion	17
1.4 Refactoring with IntelliJ IDEA	18
1.4.1 Application de 'Extract Interface' dans IntelliJ IDEA	18
1.4.2 Analyse de l'Opération 'Extract Interface' dans IntelliJ IDEA . .	19
1.4.3 Hiérarchie d'interfaces sans duplication de signatures	19
1.4.4 Optimisation de la Hiérarchie d'Interfaces par l'Analyse Formelle de Concepts (AFC)	23
1.4.5 Utilisation de ChatGPT4 :	25
1.4.6 Observations sur les Résultats Fournis par ChatGPT4 sans Aide .	27
Conclusion	29

Introduction Générale

Le refactoring est une pratique essentielle dans le développement de logiciels qui consiste à restructurer le code existant sans en changer la fonctionnalité externe. Son objectif est d'améliorer la non-fonctionnalité du code, telle que sa lisibilité, sa réduction de complexité, ou encore sa facilité de maintenance et d'évolutivité. Dans le domaine de la programmation Java, le refactoring prend une importance particulière en raison de la nature fortement typée et de l'héritage de classe du langage, qui peuvent conduire à une duplication de code et à une complexité accrue.

Dans le cadre de ce projet, notre objectif était de mettre en œuvre deux techniques de refactoring courantes, 'Extract Superclass' et 'Extract Interface', à l'aide de l'environnement de développement intégré (IDE) IntelliJ IDEA. Ces méthodes sont particulièrement utiles pour gérer l'héritage et les relations entre classes et interfaces, en permettant une meilleure réutilisation du code et en renforçant les principes d'encapsulation et de polymorphisme. Notre démarche était guidée par la volonté de rendre le code plus modulaire et extensible, en réduisant la duplication et en clarifiant les responsabilités au sein de notre base de code.

Le refactoring en Java n'est pas seulement une question de modification de code ; c'est une démarche stratégique qui influence le cycle de vie du logiciel. En prenant en compte des aspects tels que la conception de l'architecture logicielle, les modèles de conception et les principes SOLID, le refactoring devient un puissant vecteur d'amélioration de la qualité et de la performance des applications Java.

Pour la réalisation de ce projet, nous avons utilisé ChatGPT4 afin de nous assister dans différentes tâches du processus de développement. Au-delà des tests exigés par le projet, ChatGPT4 a été particulièrement utile pour la rédaction de ce rapport, la proposition de solutions cohérentes adaptées à nos études, et d'autres applications pratiques. Cette intégration de l'intelligence artificielle a contribué à enrichir notre approche, en offrant une perspective innovante et un soutien précieux dans le cadre du développement de notre projet.

Chapitre 1

Refactoring en Java

1.1 Refactoring avec Extract SuperClass pour la classe 'Voiture'

1.1.1 Programme avant le refactoring

Le code suivant illustre l'implémentation initiale de la classe Voiture, avant l'application de toute méthode de refactoring. Cette version du code présente une structure de base sans optimisation via le refactoring. La figure 1.1 représente la classe Main utilisée pour tester la classe Voiture.

```
1 public class Voiture {
2
3     private String marque;
4     private String modele;
5     private int anneeFabrication;
6     private String couleur;
7     private float vitesse;
8     private int nombrePortes;
9
10    public Voiture(String marque, String modele, int
        ↪ anneeFabrication, String couleur, float vitesse, int
        ↪ nombrePortes) {
11        this.marque = marque;
12        this.modele = modele;
13        this.anneeFabrication = anneeFabrication;
14        this.couleur = couleur;
15        this.vitesse = vitesse;
16        this.nombrePortes = nombrePortes;
17    }
18    public String getMarque() {return marque;}
19    public void setMarque(String marque) {this.marque = marque; }
20    public String getModele() { return modele;}
21    public void setModele(String modele) {this.modele = modele;}
22    public int getAnneeFabrication() { return anneeFabrication;}
23    public void setAnneeFabrication(int anneeFabrication) { this.
```

```

24     ↪ anneeFabrication = anneeFabrication;}
25 public String getCouleur() {return couleur;}
26 public void setCouleur(String couleur) { this.couleur = couleur
27     ↪ ;}
28 public float getVitesse() {return vitesse;}
29 public int getNombrePortes() {return nombrePortes;}
30 public void setNombrePortes(int nombrePortes) {this.nombrePortes
31     ↪ = nombrePortes;}
32 public void setVitesse(float vitesse) { this.vitesse = vitesse;}
33 /** Methode declarer */
34 public float accelerer(float deltaVitesse) {this.vitesse +=
35     ↪ deltaVitesse; return vitesse;}
36 public float decelerer(float deltaVitesse) {this.vitesse = Math.
37     ↪ max(this.vitesse - deltaVitesse, 0f); return vitesse;}
38 }

```

```

public class MainWithRefactoring {

    public static void main(String [] args){
        Voiture maVoiture = new Voiture( marque: "Toyota", modele: "Camry", anneeFabrication: 2022,
        couleur: "Bleu", vitesse: 120.5f, nombrePortes: 4);

        float nouvelleVitesse = maVoiture.accelerer( deltaVitesse: 20.0f);
        float vitesseDecelerer = maVoiture.decelerer( deltaVitesse: 20.0f);
        // Utilisation de l'instance
        System.out.println("Voiture : " + maVoiture.getMarque() + " " + maVoiture.getModèle());
        System.out.println("Nombre de portes : " + maVoiture.getNombrePortes());
        System.out.println("Année de fabrication : " + maVoiture.getAnnéeFabrication());
        System.out.println("Vitesse : " + maVoiture.getVitesse());
        System.out.println("Vitesse après accélération : " + nouvelleVitesse);
        System.out.println("Vitesse après déccélération : " + vitesseDecelerer);
    }
}

```

FIGURE 1.1 – Class Main avant refactor

Défaut Identifié :

Le principal défaut de ce programme est la redondance du code. La classe Voiture contient des attributs et méthodes qui seraient communs à d'autres classes qui sont des classes de spécialisation de véhicules, c'est à dire que sont des attributs et methodes génériques aux Vehicules. Sans refactoring, ces attributs et méthodes devraient être répétés dans chaque nouvelle classe de véhicule, entraînant une duplication inutile du code. On va donc chercher à extraire ces attributs dans un concept plus abstrait.

1.1.2 Programme après le refactoring

1. **L'Opération de Refactoring Réalisée :** Nous avons réalisé l'opération de refactoring "Extract Superclass" sur la classe Voiture. Cette opération a consisté à extraire les attributs et méthodes communs de Voiture dans une nouvelle classe Vehicule.
2. **La Procédure Appliquée Sous IntelliJ IDEA :**
 - Nous avons commencé par ouvrir la classe Voiture dans IntelliJ IDEA, puis Nous avons fait un clic droit et sélectionné Refactor -> Extract Superclass.

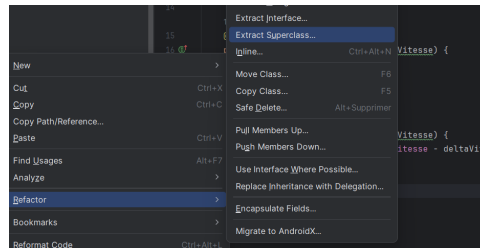


FIGURE 1.2 – Début du processus de refactoring dans IntelliJ IDEA

- Dans la fenêtre qui s'est ouverte, Nous avons nommé la nouvelle superclasse Vehicule et sélectionné les attributs et méthodes à y inclure, et aussi identifier les attributs ou méthodes qui pourraient être déclarés comme abstraits.

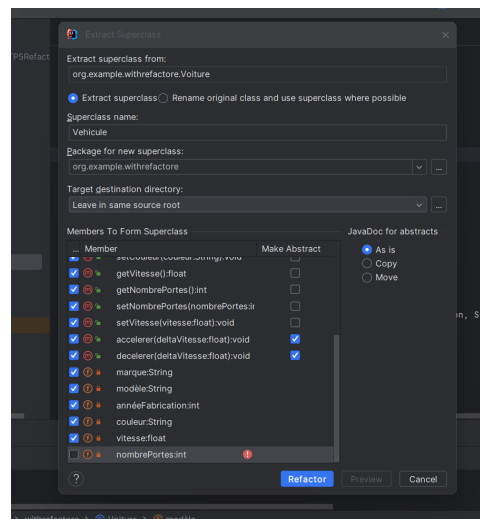


FIGURE 1.3 – Sélection des attributs et méthodes pour la SuperClass Vehicule

- Après avoir mis en évidence et corrigé les problèmes du programme précédent, nous utilisons les refactorings pour améliorer la structure du code.

La classe Véhicule apres le refactoring

Dans les classes suivnates, les attributs et méthodes communs de la classe Voiture sont sélectionnés pour former la base de la superclasse Vehicule.

```

1 public abstract class Vehicule {
2     protected String marque;
3     protected String modele;
4     protected int anneeFabrication;
5     protected String couleur;
6     protected float vitesse;
7
8     public Vehicule(String marque, String modele, int
9         ↪ anneeFabrication, String couleur, float vitesse) {
10         this.marque = marque;
11         this.modele = modele;
12         this.anneeFabrication = anneeFabrication;
13         this.couleur = couleur;
14         this.vitesse = vitesse;
15     }
16     public String getMarque() {return marque;}
17     public void setMarque(String marque) { this.marque = marque
18         ↪ ;}
19     public String getModele() { return modele;}
20     public void setModele(String modele) {this.modele = modele
21         ↪ ;}
22     public int getAnneeFabrication() {return anneeFabrication;}
23     public void setAnneeFabrication(int anneeFabrication) {this
24         ↪ .anneeFabrication = anneeFabrication;}
25     public String getCouleur() {return couleur;}
26     public void setCouleur(String couleur) { this.couleur =
27         ↪ couleur;}
28     public float getVitesse() {return vitesse; }
29     public void setVitesse(float vitesse) {this.vitesse =
30         ↪ vitesse;}
31     public abstract float accelerer(float deltaVitesse);
32     public abstract float decelerer(float deltaVitesse);
33 }

```

```

1 public class Voiture extends Vehicule {
2     private int nombrePortes;
3
4     public Voiture(String marque, String modele,
5         int anneeFabrication, String couleur, float vitesse,
6         int nombrePortes) {
7         super(marque, modele, anneeFabrication, couleur,
8             ↪ vitesse);
9         this.nombrePortes = nombrePortes;
10    }
11    public int getNombrePortes() {return nombrePortes;}
12    public void setNombrePortes(int nombrePortes) {this.
13        ↪ nombrePortes = nombrePortes;}
14
15    @Override
16    public float accelerer(float deltaVitesse) {
17        this.vitesse += deltaVitesse ;
18        return this.vitesse;
19    }
20    @Override
21    public float decelerer(float deltaVitesse) {
22        this.vitesse = Math.max(this.vitesse - deltaVitesse, 0f
23        ↪ );
24        return vitesse;
25    }
26 }

```

La création de la classe Vehicule à travers le processus de refactoring apporte plusieurs avantages significatifs à la structure globale du code. Tout d'abord, elle permet d'établir une hiérarchie claire et logique dans le code, alignant ainsi la structure du programme avec les concepts de la programmation orientée objet. En extrayant les attributs et méthodes communs à une superclasse, nous avons réduit la duplication de code et augmenté la clarté et la cohérence du code.

La spécificité de la classe Voiture est préservée, notamment avec l'attribut nombrePortes et ses méthodes associées (getters et setters). Cela permet à la classe Voiture de maintenir ses caractéristiques uniques tout en bénéficiant de l'héritage des propriétés communes de la superclasse Vehicule. Cette structure facilite la maintenance et l'ajout de nouvelles fonctionnalités à la classe Voiture sans affecter ni dupliquer le code commun.

3. L'Intérêt du Refactoring :

Ce refactoring a permis de réduire la duplication de code et d'améliorer la structure du code pour une meilleure maintenance. Il facilite également l'extension future du code, par exemple, pour l'ajout de nouvelles classes de véhicules.

4. Évaluation de la Mise en Œuvre Sous IntelliJ IDEA :

Le processus de refactoring sous IntelliJ IDEA s'est avéré intuitif et efficace. La visualisation graphique des classes et la facilité d'utilisation des outils ont simplifié

la tâche. Cependant, la sélection manuelle des éléments à extraire peut être un peu fastidieuse pour des classes plus grandes.

5. Résultats du Test et Modifications du Main :

Les tests avec la méthode main ont confirmé que le programme fonctionnait correctement après le refactoring. Aucune modification n'a été nécessaire dans la méthode main, ce qui démontre que le refactoring n'a pas perturbé la logique existante du programme.

```
public class MainWithRefactoring {  
  
    public static void main(String [] args){  
        Voiture maVoiture = new Voiture( marque: "Toyota", _modèle: "Camry", _annéeFabrication: 2022,  
        couleur: "Bleu", vitesse: 120.5f, nombrePortes: 4);  
  
        float nouvelleVitesse = maVoiture.accelerer( deltaVitesse: 20.0f);  
        float vitesseDecelerer = maVoiture.decelerer( deltaVitesse: 20.0f);  
        // Utilisation de l'instance  
        System.out.println("Voiture : " + maVoiture.getMarque() + " " + maVoiture.getModèle());  
        System.out.println("Nombre de portes : " + maVoiture.getNombrePortes());  
        System.out.println("Année de fabrication : " + maVoiture.getAnnéeFabrication());  
        System.out.println("Vitesse : " + maVoiture.getVitesse());  
        System.out.println("Vitesse après accélération : " + nouvelleVitesse);  
        System.out.println("Vitesse après déccélération : " + vitesseDecelerer);  
    }  
}
```

FIGURE 1.4 – Classe Main utilisant Voiture après refactoring

6. Conclusion :

La mise en œuvre du refactoring pour créer la classe Vehicule a eu un impact positif non seulement sur la classe Voiture, mais aussi sur l'ensemble du projet. Elle offre une base solide pour l'ajout de nouvelles classes de véhicules dans le futur, facilitant ainsi l'extension du code. La classe Vehicule agit comme un fondement sur lequel d'autres types de véhicules peuvent être construits, partageant des attributs et des méthodes communs tout en permettant à chaque sous-classe de définir ses spécificités.

Ce refactoring n'a pas affecté l'accès aux objets existants, comme le démontre l'absence de modification nécessaire dans la méthode main. Cette stabilité du code existant, tout en introduisant une structure améliorée, est un indicateur clair de l'efficacité du refactoring effectué.

1.2 Application des opérations de refactoring sur les programmes d'un autre groupe

Cet exemple de classes a été développé par mes collègues **El Hadji Ahmadou GUEYE** et **Nathan HENRY**. Afin de réaliser une implémentation efficace et d'effectuer le refactoring nécessaire, j'ai sollicité auprès de ce groupe le code des deux classes Cat et Dog. Le code source qu'ils m'ont fourni, et sur lequel j'ai travaillé pour appliquer le refactoring, est présenté ci-dessous :

1.2.1 Programme avant le refactoring

Le code suivant illustre l'implémentation initiale des classes Cat et Dog avant l'application de toute méthode de refactoring. Cette version du code présente une structure de base sans optimisation via le refactoring. La classe Main utilisée pour tester les classes Cat et Dog.

```
1 // Default : la methode makeSound est repetee dans les classes Dog et
  ↪ Cat
2 // Operation de Refactoring Visee : "Extract Superclass"
3 public class Cat {
4     private String name;
5     private String color;
6
7     public void makeSound() { System.out.println("Miaou"); }
8     public void chaseMouse() { System.out.println("Le chat chasse
  ↪ une souris."); }
9     public String getName() { return name; }
10    public void setName(String name) { this.name = name; }
11    public String getColor() { return color; }
12    public void setColor(String color) { this.color = color; }
13 }
```

```
1 // Default : la methode makeSound est repetee dans les classes Dog et
  ↪ Cat
2 // Operation de Refactoring Visee : "Extract Superclass"
3 public class Dog {
4     private String name;
5     private String color;
6
7     public void makeSound() { System.out.println("Wouaf"); }
8     public void fetchBall() { System.out.println("Le chien rapporte
  ↪ la balle."); }
9     public String getName() { return name; }
10    public void setName(String name) { this.name = name; }
11    public String getColor() { return color; }
12    public void setColor(String color) { this.color = color; }
13 }
```

```
1
2 public class Main {
3     public static void main(String[] args) {
4         Dog dog = new Dog();
5         Cat cat = new Cat();
6
7         dog.makeSound();
8         cat.makeSound();
9         dog.fetchBall();
10        cat.chaseMouse();
11    }
12 }
```

Les défauts identifier :

Les classes Cat et Dog présentent une redondance notable, particulièrement dans la méthode `makeSound`, répétée dans chaque classe. Cette duplication de code réduit la modularité et complique la maintenance. De plus, l'absence de structure héritée limite l'extensibilité, rendant difficile l'ajout de nouvelles classes d'animaux avec des comportements similaires. Ces problèmes mettent en évidence la nécessité d'un refactoring pour optimiser la structure du code.

1.2.2 Programme après le refactoring

Après avoir obtenu le code source des classes Cat et Dog de mes collègues, Nous avons commencé le processus de refactoring pour extraire une superclasse commune Animal. Les étapes suivantes illustrent ce processus :

1. **L'Opération de Refactoring Réalisée :** Nous avons effectué l'opération "Extract Superclass" pour regrouper les attributs et méthodes communs des classes Cat et Dog dans une nouvelle classe Animal.
2. **La Procédure Appliquée Sous IntelliJ IDEA :** Les étapes suivies étaient :
Sélection de la classe Cat, clic droit, puis choix de Refactor -> Extract Superclass.

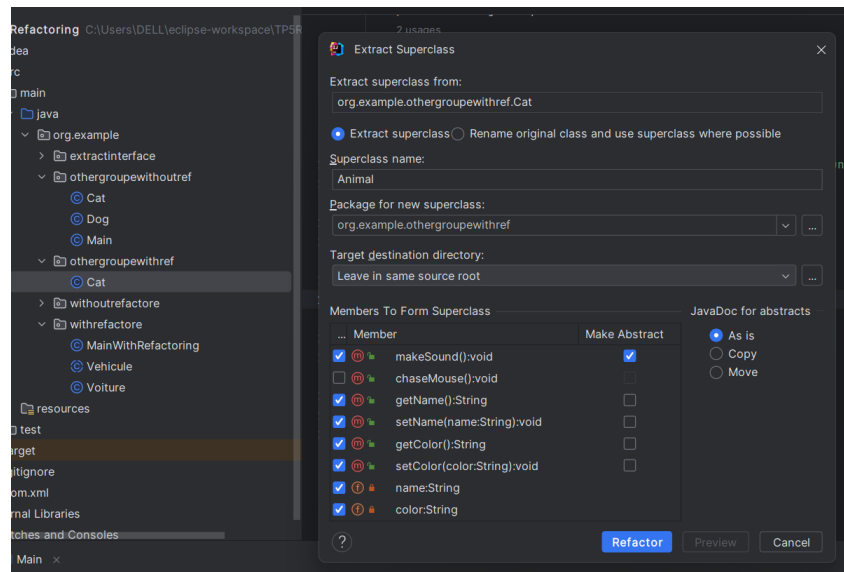


FIGURE 1.5 – Sélection des attributs et méthodes pour la SuperClass Animal a partir de la classe Cat

Lors de l'opération de refactoring avec l'IDE IntelliJ IDEA pour la classe Dog, nous avons constaté que le refactoring utilisant l'option 'Extract SuperClass' présente certaines limitations lorsqu'il est appliqué à plusieurs classes de spécialisation. Les figures 1.6, 1.7 suivantes illustrent ces limitations dans le processus de refactoring.

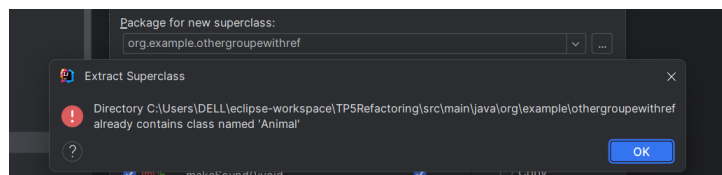


FIGURE 1.6 – Le refactoring n'est pas possible avec la classe Dog utilisant la même superclasse Animal

La Figure 1.6 illustre une tentative infructueuse de refactoring avec la classe Dog, mettant en évidence les limitations ou les difficultés rencontrées lors de cette opération.

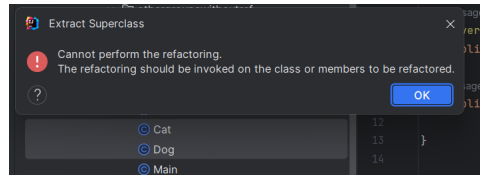


FIGURE 1.7 – Tentative de refactoring simultané des classes Cat et Dog pour générer la superclasse Animal

La Figure 1.7 démontre une tentative de réaliser le refactoring en utilisant conjointement les classes Cat et Dog dans le but de créer la superclasse Animal. Cette approche a également révélé des défis spécifiques liés à l'intégration de ces deux classes dans une même superclasse Animal.

La classe Animal apres le refactoring

Dans les classes suivnates, les attributs et méthodes communs de la classe Cat sont sélectionnés pour former la base de la superclasse Animal.

```

1 public abstract class Animal {
2     private String name;
3     private String color;
4     public abstract void makeSound();
5     public String getName() {return name;}
6     public void setName(String name) {this.name = name;}
7     public String getColor() {return color;}
8     public void setColor(String color) {this.color = color;}
9 }

```

```

1 public class Cat extends Animal {
2
3     @Override
4     public void makeSound() { System.out.println("Miaou"); }
5     public void chaseMouse() { System.out.println("Le chat
6         ↳ chasse une souris."); }
7 }
8
9 public class Dog extends Animal{
10
11     @Override
12     public void makeSound() { System.out.println("Wouaf");}
13     public void fetchBall() { System.out.println("Le chien
14         ↳ rapporte la balle."); }
15 }

```

La classe Animal, après le refactoring, intègre désormais les attributs et méthodes communs. Ce processus a permis aux classes Cat et Dog de conserver leurs méthodes spécifiques. En particulier, les méthodes fetchBall() et chaseMouse() restent exclusives respectivement à Dog et Cat. Il est important de noter que la

classe Dog a été intégrée manuellement en tant que sous-classe de Animal, démontrant ainsi un effort supplémentaire pour maintenir la cohérence et l'efficacité du refactoring. Cette approche permet une meilleure organisation du code, tout en préservant les fonctionnalités uniques de chaque classe.

3. **L'Intérêt du Refactoring** : Le refactoring a permis de réduire la redondance du code et d'améliorer la modularité, facilitant ainsi les futures extensions et la maintenance du code. Cependant, il est important de noter que le refactoring présente certaines limitations. Par exemple, dans notre cas, l'utilisation de la superclasse Animal a nécessité des ajustements manuels, notamment pour intégrer correctement la classe Dog.
4. **Évaluation de la Mise en Œuvre Sous IntelliJ IDEA** : L'outil de refactoring d'IntelliJ IDEA est intuitif et efficace pour la plupart des opérations. Cependant, la nécessité de sélectionner manuellement les éléments à extraire peut s'avérer laborieuse, particulièrement pour les classes de grande taille ou complexes. Cette limite peut augmenter le risque d'erreurs lors du refactoring de structures de code plus élaborées.
5. **Résultats du Test et Modifications du Main** : Les tests réalisés avec la classe Main ont validé le bon fonctionnement du programme après le refactoring. Aucune modification n'a été requise dans la méthode main, même après l'intégration manuelle de la classe Dog en tant que sous-classe de Animal. Cette intégration réussie démontre que le refactoring, bien que partiellement manuel, a été effectué avec succès, permettant à la fois aux classes Cat et Dog d'hériter des propriétés et méthodes de la superclasse Animal. Le fait que les tests dans Main fonctionnent sans modification supplémentaire témoigne de l'efficacité de l'approche adoptée et de la compatibilité de la nouvelle structure avec le code existant.

```
1
2 public class Main {
3     public static void main(String[] args) {
4         Dog dog = new Dog();
5         Cat cat = new Cat();
6
7         dog.makeSound();
8         cat.makeSound();
9         dog.fetchBall();
10        cat.chaseMouse();
11    }
12 }
```

6. Conclusion :

Cette étude de cas de refactoring illustre l'importance cruciale d'une analyse préalable et d'une planification minutieuse dans le processus de refactoring. Bien que le refactoring puisse conduire à une structure de code plus propre et plus facile à maintenir, il peut également révéler des défis, comme illustré par les difficultés rencontrées lors du refactoring de la classe Dog. Cette expérience met en lumière

la nécessité d'une compréhension approfondie des relations entre les classes et des fonctionnalités communes avant d'initier un refactoring.

Face à ces défis, en particulier avec la classe Dog, nous avons opté pour une solution manuelle. Au lieu de s'appuyer uniquement sur les outils automatisés de refactoring d'IntelliJ IDEA, qui ont présenté certaines limitations, nous avons manuellement implémenté la classe Dog en tant que sous-classe de Animal. Cette approche manuelle nous a permis de surmonter les obstacles du refactoring automatique et d'assurer que la classe Dog hérite correctement des attributs et méthodes pertinents de la nouvelle superclasse Animal. Cette démarche souligne l'importance d'une certaine flexibilité et d'une intervention manuelle dans les cas où les outils automatisés ne répondent pas pleinement aux besoins spécifiques du projet.

1.3 Testez ce que ChatGPT4 peut réaliser

Dans cette section, nous avons sollicité l'aide de ChatGPT4 pour effectuer le refactoring des classes 'Cat' et 'Dog'. Voici les résultats obtenus :

1.3.1 Refactoring : Création de la Superclasse Animal

Pour effectuer le refactoring de ce code en utilisant l'opération "Extract Superclass", nous allons créer une classe parente qui contiendra la méthode commune makeSound(). Les classes Dog et Cat hériteront ensuite de cette superclasse. Voici comment le code pourrait être restructuré :

Création de la Superclasse Animal :

La nouvelle superclasse Animal contiendra les attributs communs name et color, ainsi que la méthode makeSound(). Cette méthode sera abstraite car le son spécifique dépend de l'animal (chat ou chien).

```
1 public abstract class Animal {
2     protected String name;
3     protected String color;
4
5     public Animal(String name, String color) {
6         this.name = name;
7         this.color = color;
8     }
9
10    public abstract void makeSound();
11
12    public String getName() {
13        return name;
14    }
15
16    public void setName(String name) {
17        this.name = name;
18    }
19
20    public String getColor() {
21        return color;
22    }
23
24    public void setColor(String color) {
25        this.color = color;
26    }
27 }
```


Modifier les Classes Cat et Dog :

Les classes Cat et Dog hériteront de la classe Animal. Chaque classe fournira sa propre implémentation de makeSound().

```
1 public class Cat extends Animal {
2     public Cat(String name, String color) {
3         super(name, color);
4     }
5
6     @Override
7     public void makeSound() {
8         System.out.println("Miaou");
9     }
10
11     public void chaseMouse() {
12         System.out.println("Le chat chasse une souris.");
13     }
14 }
15
16 public class Dog extends Animal {
17     public Dog(String name, String color) {
18         super(name, color);
19     }
20
21     @Override
22     public void makeSound() {
23         System.out.println("Wouaf");
24     }
25
26     public void fetchBall() {
27         System.out.println("Le chien rapporte la balle.");
28     }
29 }
```

Adapter la Classe Main pour Utiliser les Nouvelles Classes :

La classe Main reste inchangée, car les modifications de structure n'affectent pas la manière dont les objets Dog et Cat sont utilisés.

```
1 public class Main {
2     public static void main(String[] args) {
3         Dog dog = new Dog("Buddy", "Brown");
4         Cat cat = new Cat("Whiskers", "Black");
5
6         dog.makeSound();
7         cat.makeSound();
8         dog.fetchBall();
9         cat.chaseMouse();
10    }
11 }
```

Ce refactoring améliore la structure du code en éliminant la redondance et en favorisant la réutilisation du code. Les modifications apportées rendent également le système plus facile à maintenir et à étendre.

1.3.2 Analyse des Résultats du Refactoring

Lors de l'interaction avec ChatGPT4 pour le refactoring des classes 'Cat' et 'Dog', nous avons observé plusieurs aspects clés. ChatGPT4 a proposé de créer une superclasse commune 'Animal' pour encapsuler les attributs et méthodes partagés par les deux classes. Cette approche visait à réduire la redondance et à améliorer la structure du code. Toutefois, certaines limitations sont apparues lors de cette démarche.

Premièrement, bien que ChatGPT4 ait fourni une structure de base solide pour le refactoring, il était nécessaire de peaufiner manuellement le code pour s'assurer qu'il corresponde exactement aux besoins spécifiques du projet. De plus, ChatGPT4 s'est concentré sur les aspects structurels du refactoring, sans nécessairement prendre en compte les subtilités de la logique métier ou les interactions complexes entre les classes.

1.3.3 Conclusion

Ce refactoring démontre l'efficacité de l'extraction d'une superclasse pour réduire la redondance du code et améliorer l'organisation et la maintenance du code source. En identifiant et en regroupant les fonctionnalités communes, on parvient à un code plus propre et plus modulaire, tout en conservant les spécificités de chaque classe.

1.4 Refactoring with IntelliJ IDEA

Nous avons suivi la démarche indiquée dans le sujet de TP pour extraire une interface d'un corpus de classes ou d'une seule classe. Nous présentons nos observations dans la liste suivante :

1.4.1 Application de 'Extract Interface' dans IntelliJ IDEA

Pour étudier l'opération de refactoring 'Extract Interface' sur l'exemple de 4 classes fourni, nous avons procédé comme le montre la figure 1.8 :

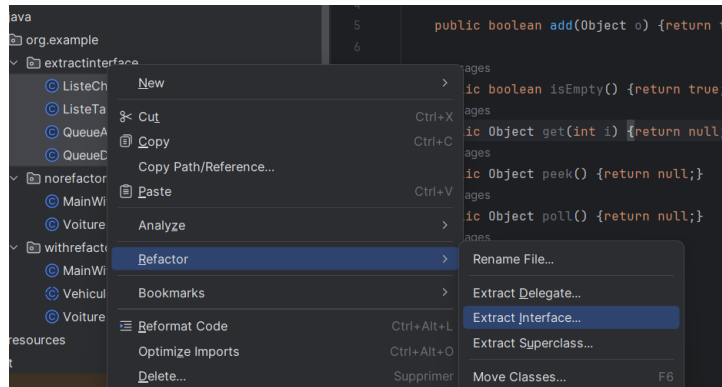


FIGURE 1.8 – Extract interface avec IntelliJ IDEA

Sélection de Toutes les Classes Simultanément

Nous avons tenté d'appliquer l'opération 'Extract Interface' en sélectionnant simultanément les quatre classes (ListeTableau, ListeChaine, QueueDoubleEntree, QueueAvecPriorite) dans IntelliJ IDEA.

Cependant, nous avons constaté que l'extraction d'interface sur l'ensemble des classes en même temps n'est pas possible dans IntelliJ IDEA, comme illustré dans la figure 1.9 suivante.

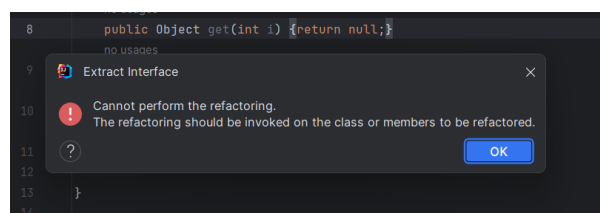


FIGURE 1.9 – Extract interface n'est pas possible avec plusieurs classes

Sélectionner chaque classe individuellement

Lorsque nous avons appliqué l'opération 'Extract Interface' individuellement sur chaque classe dans IntelliJ IDEA, nous avons observé que l'extraction d'une seule classe à la fois

permet une sélection précise des éléments à extraire dans l'interface. Cette approche est illustrée dans la figure 1.10.

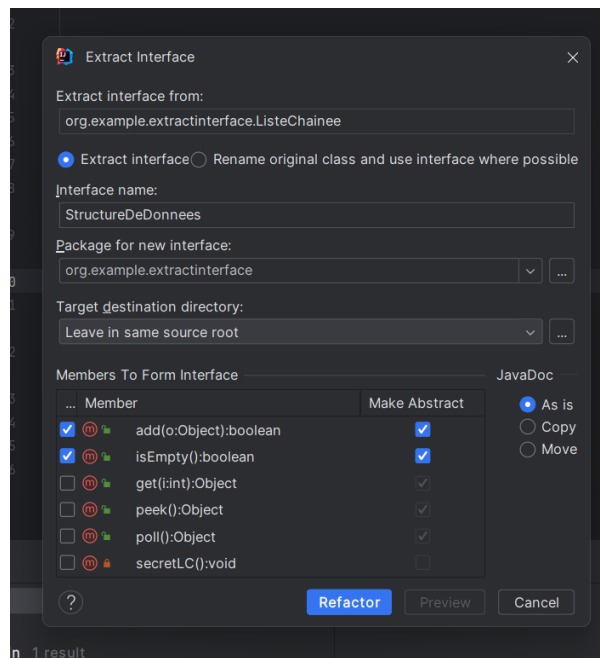


FIGURE 1.10 – Extract interface d'une seule classe

Nous avons constaté que cette méthode fonctionne parfaitement pour la première classe. Cependant, lorsque nous avons tenté d'appliquer la même interface à une deuxième classe, nous nous sommes heurtés à une limitation : le nom de l'interface existait déjà. Cette contrainte souligne une limitation importante de l'opération 'Extract Interface' dans le contexte de l'IDE IntelliJ IDEA, où la duplication de noms d'interface entre différentes classes n'est pas prise en charge. Nous avons rajouté les autres classes manuellement pour implémenter l'interface StructureDeDonnees.

1.4.2 Analyse de l'Opération 'Extract Interface' dans IntelliJ IDEA

L'opération de refactoring 'Extract Interface' dans IntelliJ est conçue pour extraire les méthodes communes de plusieurs classes et les regrouper dans une interface unique. Cette méthode est bénéfique pour diminuer la redondance du code et améliorer la modularité.

Toutefois, lors de la mise en œuvre de l'opération 'Extract Interface' sur des classes individuellement, il a été constaté que l'outil ne factorise pas les méthodes communes dans une interface commune. Chaque classe se voit attribuer une interface distincte, menant potentiellement à une duplication des signatures de méthodes. Cette limitation indique que l'opération ne parvient pas à identifier et à regrouper automatiquement les méthodes similaires de différentes classes au sein d'une même interface.

1.4.3 Hiérarchie d'interfaces sans duplication de signatures

L'objectif de cette section est de démontrer la création d'une hiérarchie d'interfaces optimisée, éliminant la duplication de signatures tout en assurant une spécialisation ap-

propriété des comportements pour chaque type de collection. Nous avons procédé en suivant les principes de conception pour améliorer l’extensibilité et la maintenance du code.

- Notre analyse initiale a identifié les méthodes communes entre les classes, et celles qui étaient spécifiques à chacune, afin de déterminer les interfaces nécessaires.
- Nous avons élaboré une structure d’interfaces illustrée dans le diagramme de classe UML suivant, qui démontre les relations d’héritage et les extensions entre les interfaces.
- Au final, Nous définissons les interfaces qui sont en relation directe avec les classes avec des méthodes précises à chaque interface (si ça existe) puis nous définissons les classes qui implémentent chaque interface.

Le diagramme (Voir Figure 1.11) révèle un système d’interfaces qui s’étendent les unes des autres, définissant des méthodes spécifiques pour chaque type de collection tout en évitant la duplication.

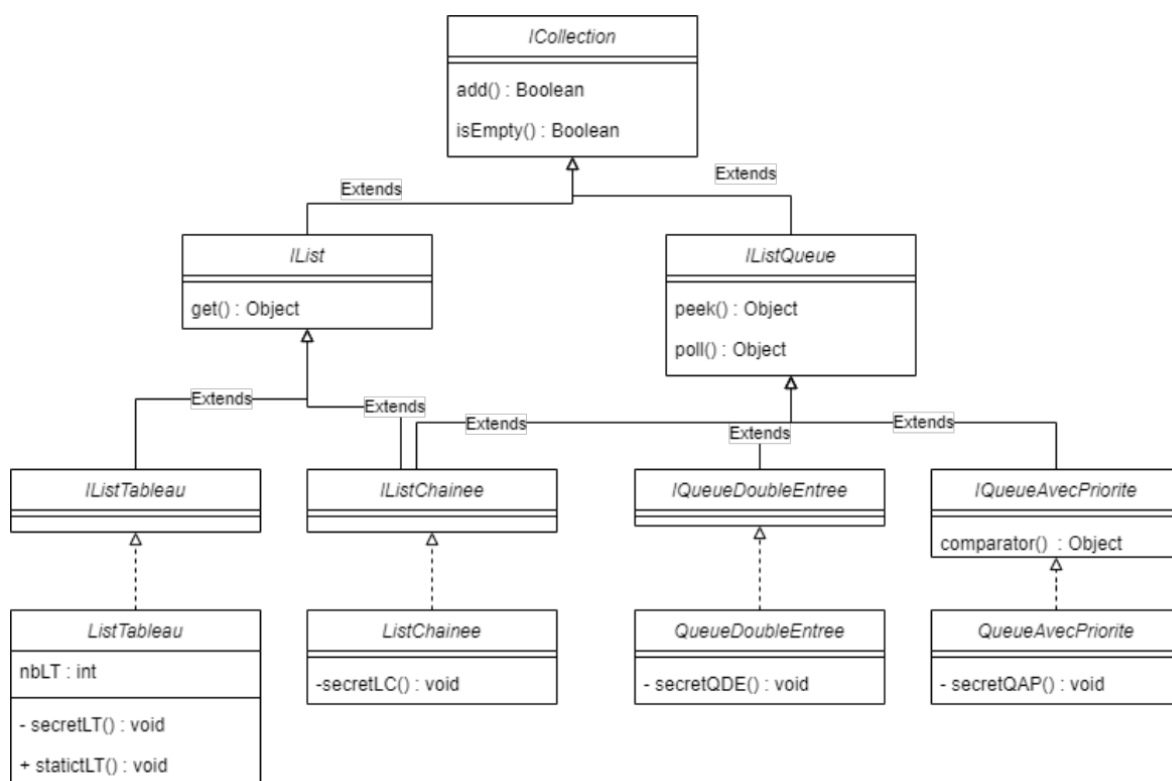


FIGURE 1.11 – Diagramme de classe des interfaces refactorisées manuellement

Nous avons ensuite manuellement implémenté les interfaces et classes comme détaillé ci-dessous :

Les interfaces

Nous définissons d'abord les interfaces de base pour les collections, les listes et les files d'attente, chacune ajoutant ses propres méthodes spécifiques.

```
1 public interface ICollection {
2     boolean add(Object o);
3     boolean isEmpty();
4 }
```

```
1 public interface IList extends ICollection{
2     public Object get(int i);
3 }
```

```
1 public interface IListQueue extends ICollection {
2     public Object peek();
3     public Object poll();
4 }
```

```
1 public interface IListTableau extends IList{
2 }
```

```
1 public interface IListeChaine extends IList, IListQueue{
2 }
```

```
1 public interface IQueueAvecPriorite extends IListQueue{
2     public Object comparator() ;
3 }
```

```
1 public interface IQueueDoubleEntree extends IListQueue{
2 }
```

Les classes

Chaque classe implémente les méthodes spécifiées par ses interfaces. Les méthodes spécifiques comme `secretLC()` dans `ListeChaine` ou `secretQAP()` dans `QueueAvecPriorite` restent dans les classes, car elles sont uniques à ces types.

```
1 public class ListeChaine implements IListeChaine {
2     @Override
3     public boolean add(Object o) {return true;}
4     @Override
5     public boolean isEmpty() {return true;}
6     @Override
7     public Object get(int i) {return null;}
```

```

8      @Override
9      public Object peek() {return null; }
10     @Override
11     public Object poll() {return null;}
12     private void secretLC(){}
13 }

```

```

1 public class ListeTableau implements IListTableau {
2     @Override
3     public boolean add(Object o) {return true;}
4     @Override
5     public boolean isEmpty() {return true;}
6     @Override
7     public Object get(int i) {return null;}
8     private void secretLT(){}
9     public static void staticLT() {}
10    int nbLT;
11 }

```

```

1 public class QueueAvecPriorite implements IQueueAvecPriorite {
2     @Override
3     public boolean add(Object o) {return true;}
4     @Override
5     public boolean isEmpty() {return true;}
6     @Override
7     public Object peek() {return null;}
8     @Override
9     public Object poll() {return null;}
10    @Override
11    public Object comparator() {return null;}
12    private void secretQAP(){}
13 }

```

```

1 public class QueueDoubleEntree implements IQueueDoubleEntree {
2     @Override
3     public boolean add(Object o) {return true;}
4     @Override
5     public boolean isEmpty() {return true;}
6     @Override
7     public Object peek() {return null;}
8     @Override
9     public Object poll() {return null;}
10    private void secretQDE(){}
11 }

```

Chaque classe concrète implémente l'interface correspondante, comme démontré dans les extraits de code suivants. Les méthodes sont implémentées pour retourner des valeurs par défaut, et des méthodes privées sont ajoutées pour illustrer la capacité de chaque classe à conserver ses propres comportements internes.

Cette approche assure que les classes ‘ListeChaine’, ‘ListeTableau’, ‘QueueAvecPriorite’, et ‘QueueDoubleEntree’ implémentent toutes les méthodes requises par leurs interfaces respectives sans redondance, adhérant ainsi aux principes de conception orientée objet et facilitant la maintenance et l’extension du code.

1.4.4 Optimisation de la Hiérarchie d’Interfaces par l’Analyse Formelle de Concepts (AFC)

Dans cette partie de projet, nous allons produire un fichier .csv, dont les lignes (objets) sont les noms des classes et les colonnes (attributs) sont les signatures des méthodes publiques. Puis nous allons lui appliquer un algorithme qui calcule une factorisation des attributs qui est à la fois maximale et compacte, comme le montre la figure 1.12.

name	;add	;isEmpty	;get	;peek	;poll	;secretLT	;staticLT	;secretLC	;secretQDE	;secretQAP	;comparator
ListTableau	; 1 ;	1 ;	1 ;	0 ;	0 ;	1 ;	1 ;	0 ;	0 ;	0 ;	0 ;
ListChaine	; 1 ;	1 ;	1 ;	1 ;	1 ;	0 ;	0 ;	1 ;	0 ;	0 ;	0 ;
QueueDoubleEntree	; 1 ;	1 ;	0 ;	1 ;	1 ;	0 ;	0 ;	0 ;	1 ;	0 ;	0 ;
QueueAvecPriorite	; 1 ;	1 ;	0 ;	1 ;	1 ;	0 ;	0 ;	0 ;	0 ;	1 ;	1 ;

FIGURE 1.12 – CSV des classes et de leur méthodes publiques

En appliquant l’analyse formelle de concepts, comme illustrée dans la figure 1.13, nous pouvons voir comment un algorithme pourrait regrouper les méthodes en interfaces plus efficacement. Cela peut montrer des opportunités pour une factorisation plus poussée ou révéler des redondances qui n’ont pas été adressées manuellement.

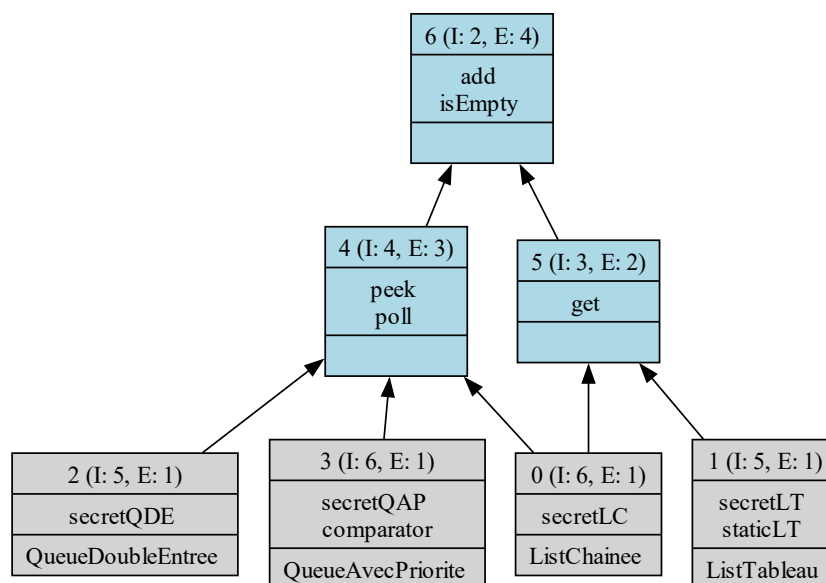


FIGURE 1.13 – Treillis des interfaces des classes avec RCA

Comparaison des Résultats :

Dans cette section, nous comparons les interfaces dérivées manuellement avec celles suggérées par le treillis de l'Analyse Formelle de Concepts (AFC).

- Le treillis peut identifier des opportunités pour regrouper davantage de méthodes communes dans des interfaces uniques, réduisant ainsi la redondance qui pourrait subsister dans l'implémentation manuelle.
- La structure suggérée par l'AFC peut indiquer une hiérarchisation différente, voire plus optimale, comparée à notre approche manuelle. Cette structure propose une organisation des interfaces qui pourrait améliorer l'extensibilité et la maintenance du code.
- Les interfaces `ICollection`, `IList`, et `IListQueue`, identifiées tant manuellement qu'à travers l'AFC, présentent une cohérence dans la définition des méthodes. Ces interfaces constituent la fondation sur laquelle les autres interfaces et classes se construisent.
- Les interfaces `IListTableau`, `IListChaine`, `IQueueDoubleEntree`, et `IQueueAvecPriorite`, bien que non présentes dans le treillis de l'AFC, sont directement implémentées par les classes dans notre approche manuelle. Cela souligne un écart entre la factorisation manuelle et celle obtenue par l'AFC, qui privilégie une approche plus directe dans l'assignation des responsabilités aux classes.
- Notamment, la méthode `Comparator()` reste exclusive à la classe `QueueAvecPriorite` dans notre implémentation manuelle mais plutôt elle est associée à l'interface dont `QueueAvecPriorite` hérite, alors que le treillis de l'AFC ne l'associe à aucune interface spécifique, mais directement à la classe qui la définit. Cela met en lumière la singularité de cette méthode et sa spécificité à la classe en question, indiquant que certaines fonctionnalités peuvent rester encapsulées au niveau de la classe sans nécessiter une abstraction supplémentaire.
- En effet, nous avons pensé à des factorisations supplémentaires en ajoutant des interfaces qui pourraient être éventuellement implémentées par de nouvelles classes non fournies dans le contexte. Par exemple, l'interface `IListTableau` pourrait être implémentée par une classe supplémentaire `ListTableauDynamique`, `ListTableauArbres`, etc.
- Pour finir, l'utilisation du résultat produit par l'analyse formelle de concepts pourrait être très utile pour effectuer un refactoring `Extract Interface`, de qualité en un temps record, en particulier sur des programmes complexes.

1.4.5 Utilisation de ChatGPT4 :

Dans cette section nous allons vous présenter les résultats retournés par ChatGPT4 en lui donnant le sujet, sans l'aider à lui bien expliquer ce qu'il faut faire, ou même lui corriger les choses.

ChatGPT4 :

Je vais vous aider à extraire les interfaces de vos classes en Java en suivant le principe du refactoring 'Extract Interface'. Commençons par identifier les méthodes communes entre les quatre classes (ListeTableau, ListeChaine, QueueDoubleEntree, QueueAvecPriorite) que vous avez mentionnées.

1. **Analyse des Méthodes Communes :** Les méthodes communes entre ces classes sont :

```
1 public boolean add(Object o)
2 public boolean isEmpty()
3 public Object peek()
4 (présente dans ListeChaine, QueueDoubleEntree, QueueAvecPriorite)
5 public Object poll()
6 (présente dans ListeChaine, QueueDoubleEntree, QueueAvecPriorite)
```

Notez que la méthode `public Object get(int i)` n'est présente que dans ListeTableau et ListeChaine. Nous devons prendre une décision sur la manière de la gérer dans notre interface.

2. Proposition d'Extraction des Interfaces

- (a) **Interface ICommonListOperations** Cette interface contiendra les méthodes communes à ListeTableau et ListeChaine.

```
1 public interface ICommonListOperations {
2     boolean add(Object o);
3     boolean isEmpty();
4     Object get(int i); // Spécifique à ListeTableau et
5                         ↪ ListeChaine
6 }
```

- (b) **Interface ICommonQueueOperations**

Cette interface contiendra les méthodes communes à ListeChaine, QueueDoubleEntree, et QueueAvecPriorite.

```
1 public interface ICommonQueueOperations {
2     boolean add(Object o);
3     boolean isEmpty();
4     Object peek();
5     Object poll();
6 }
```

3. Modification des Classes

(a) Classe ListeTableau

```
1 class ListeTableau implements ICommonListOperations {
2     public boolean add(Object o) { /* Implementation */ }
3     public boolean isEmpty() { /* Implementation */ }
4     public Object get(int i) { /* Implementation */ }
5     // Autres methodes et attributs
6 }
```

(b) Classe ListeChaine

```
1 class ListeChaine implements ICommonListOperations,
2 ICommonQueueOperations {
3     public boolean add(Object o) { /* Implementation */ }
4     public boolean isEmpty() { /* Implementation */ }
5     public Object get(int i) { /* Implementation */ }
6     public Object peek() { /* Implementation */ }
7     public Object poll() { /* Implementation */ }
8     // Autres methodes et attributs
9 }
```

(c) Classe QueueDoubleEntree

```
1 class QueueDoubleEntree implements ICommonQueueOperations {
2     public boolean add(Object o) { /* Implementation */ }
3     public boolean isEmpty() { /* Implementation */ }
4     public Object peek() { /* Implementation */ }
5     public Object poll() { /* Implementation */ }
6     // Autres methodes et attributs
7 }
```

(d) Classe QueueAvecPriorite

```
1 class QueueAvecPriorite implements ICommonQueueOperations {
2     public boolean add(Object o) { /* Implementation */ }
3     public boolean isEmpty() { /* Implementation */ }
4     public Object peek() { /* Implementation */ }
5     public Object poll() { /* Implementation */ }
6     public Object comparator() { /* Implementation
7         ↪ spécifique */ }
8     // Autres methodes et attributs
9 }
```

4. Conclusion

Cette méthode d'extraction des interfaces permet de réduire la duplication de code en regroupant les méthodes communes. Elle facilite également la maintenance et l'extension du code en clarifiant les relations entre les classes. N'hésitez pas à me faire part de vos commentaires ou ajustements souhaités !

1.4.6 Observations sur les Résultats Fournis par ChatGPT4 sans Aide

Lors de l'examen des résultats fournis par ChatGPT4 pour l'implémentation de refactoring, certaines observations notables ont été faites concernant la définition des interfaces *ICollectionOperations* et *IQueueOperations* :

- Les deux interfaces implémentent des méthodes communes, notamment *IsEmpty()* et *add()*. Cette approche peut prêter à confusion car elle ne suit pas les pratiques habituelles de conception d'interfaces, où chaque interface devrait idéalement représenter un ensemble distinct de fonctionnalités.
- Dans la classe *ListeChaine*, il est observé que cette classe implémente les deux interfaces *ICollectionOperations* et *IQueueOperations*. Une ambiguïté survient avec les méthodes *add()* et *IsEmpty()* qui sont redéfinies (*overridden*). Il est difficile de déterminer l'interface d'origine de ces méthodes dans cette configuration, ce qui pourrait conduire à des problèmes de clarté et de maintenance du code.

Limitations de Refactoring dans un IDE Comme IntelliJ IDEA

— **Complexité des Structures de Code :**

Dans des projets plus complexes, avec des structures de code imbriquées et des dépendances, l'outil de refactoring intégré à IntelliJ IDEA peut avoir des difficultés à identifier tous les éléments à refactoriser de manière exhaustive.

— **Refactoring Manuels :**

Certaines opérations de refactoring, en particulier celles impliquant des logiques métier complexes ou des cas particuliers, peuvent nécessiter une intervention manuelle, ce qui n'est pas toujours intuitif ou immédiatement évident dans un IDE.

— **Risques de Rupture de Fonctionnalités :**

Bien que les IDE comme IntelliJ IDEA offrent généralement des aperçus des changements de refactoring, il existe toujours un risque que des modifications non anticipées entraînent des ruptures ou des bugs dans le code existant.

— **Tests et Validation :**

Après un refactoring, particulièrement un refactoring majeur, il est crucial de réaliser des tests complets pour s'assurer que toutes les fonctionnalités du programme restent intactes. Cela peut nécessiter des tests unitaires et d'intégration supplémentaires, ce qui n'est pas directement géré par l'outil de refactoring.

Conclusion Générale

Le parcours entrepris dans ce projet de refactoring Java a été une occasion d'apprécier l'impact significatif de techniques bien choisies sur la qualité et la pérennité d'une base de code. En nous concentrant sur les opérations 'Extract Superclass' et 'Extract Interface' au sein de l'IDE IntelliJ IDEA, nous avons renforcé l'architecture de notre application, éliminé la duplication de code et amélioré la cohérence de notre modèle d'objets.

L'expérience a souligné la nécessité d'une compréhension profonde de la structure du code et de la logique métier pour réaliser un refactoring efficace. Les outils fournis par IntelliJ IDEA ont grandement facilité le processus, mais nous avons également appris l'importance de l'analyse et de la conception manuelles, en particulier lorsqu'il s'agit de refactorings complexes qui dépassent les capacités de détection et de transformation automatisées.

En définitive, ce projet a confirmé que le refactoring est une activité essentielle pour maintenir et améliorer une base de code dans un environnement de développement agile. Les outils IDE, tout en étant puissants, doivent être utilisés en conjonction avec le jugement et l'expertise des développeurs pour atteindre un équilibre optimal entre automatisation et contrôle manuel. La perspective et les leçons acquises ici serviront de référence pour les futures initiatives de développement et de refactoring.