Master Informatique - Université Montpellier – 2018

Opérations de refactoring sous Eclipse

travail réalisé par :
Ali LACHGUER _ M2 AIGLE

Ce document contient les trois parties du TP à réaliser, création de programmes, application de refactoring, et réflexion libre.

1. Création de programmes nécessitant un refactoring

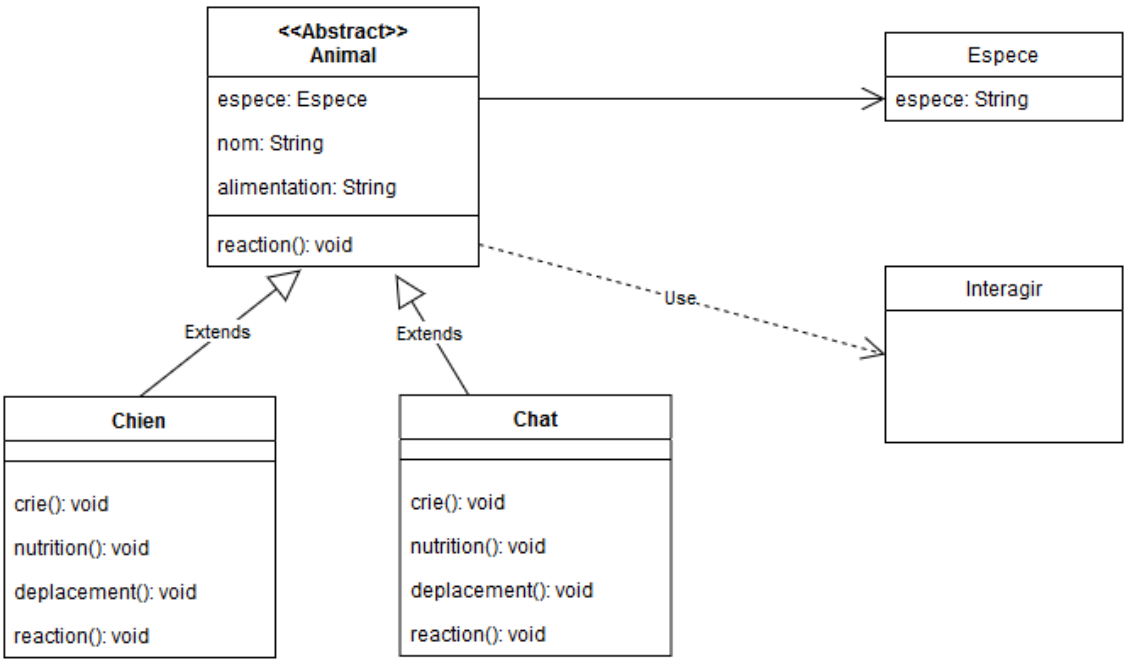
Pour cette première partie, j'ai réalisé un programme sur lequel il est possible de réaliser les deux refactoring qui me sont affectés (extract interface, et introduce indirection).

Le programme est une simple application console qui permet de créer des animaux, les nommer, les nourrir, et recevoir une réaction après une interaction avec eux.
Le programme contient une classe abstraite « Anima » qui permet de créer plusieurs type d'animaux grâce l'héritage de cette classe. Les attributs de cette classe sont :

- Espece : classe de type Espece qui renvoie l'espèce de l'animal en String.
- Nom : variable String contenant le nom de l'animal.
- Alimentation : variable String contenant ce que mange l'animal.

Cette classe abstraite contient aussi une méthode « reaction() » qui renvoie un String expliquant la réaction de l'animal lors d'une interaction avec lui. Cette méthode doit être redéfinie dans les sous-classes, afin d'avoir la réaction correspondante à chaque animal.

Pour l'exemple de ce programme, j'ai créé deux sous-classes (« Chien » et « Chat ») de la classe abstraite « Animal ». Ces classes contiennent toutes les deux des nouvelles méthodes avec la même signature, mais avec un contenu différent.
La classe « Interagir » permet depuis son constructeur de retourner la réaction de l'animal mis en paramètre.



Premier refactoring : Extract interface

Ce refactoring permet de définir des méthodes qui sont en commun entre plusieurs classes, dans une interface. Ceci va permettre aux classes de simplement implémenter l'interface, et redéfinir les méthodes de l'interface.

Dans notre exemple, la classe « Chien » et « Chat » contiennent tous les deux les méthodes : crie(), nutrition(), deplacement(), reaction(). La méthode reaction est un redéfinition de la méthode existante dans la classe « Animal ».

Ce qu'il faudra faire est que ces méthodes doivent être déclarées dans une interface, et que les classes « Chat » et « Chien » implémentent cette nouvelle interface. Pour cela, on doit sélectionner les fonctions en commun et sélectionner la fonctionnalité «**Refactor > Extract Interface ...** », et donner le nom de l'interface. Cela va créer une nouvelle interface avec le nom donné, il faudra après ajouter les déclarations des méthodes en commun, et indiquer aux classes concernées d'implémenter la nouvelle interface extraite.

Pour la méthode reaction(), il ne sera pas nécessaire de l'ajouter à l'interface, elle est déjà définie dans la classe abstraite « Animal », et sera utilisée dans la classe « Interagir » qui prend en paramètre un Animal, et donc le résultat de l'appel de la fonction reaction() dépendra de la sous-classe mise en paramètre. Une simple redéfinition de cette méthode est suffisante au niveau des sous-classes de « Animal », sinon la méthode reaction() définie dans la classe abstraite sera appelée par défaut.

Deuxième refactoring : Introduce indirection

Ce refactoring permet de rediriger l'appelant d'une méthode vers une nouvelle méthode, cette dernière appelle la méthode d'origine. Cette indirection est utile lorsqu'on a besoin d'ajouter de nouveaux comportements à une méthode, mais cette méthode ne peut pas être modifiée car elle appartient à une bibliothèque par exemple.

Dans notre application, on a la classe « interagir » qui appelle la méthode reaction() d'un animal mis en paramètre. Le refactoring qui permet l'indirection va essayer de créer une nouvelle méthode statique qui prend l'appelant en paramètre pour qu'elle appelle la méthode reaction().

Dans notre application nous avons :

```
public class Interagir {

        public Interagir(Animal animal){
                animal.reaction();
        }
}
```

Après avoir effectué le refactoring de l'indirection, on obtiendra:

```
public class Interagir {

        public Interagir(Animal animal){
                indirect(animal);
        }

        public static void indirect(Animal animal) {
                animal.reaction();
        }
}
```

Pour effectuer le refactoring de l'indirection sur une méthode, on sélectionne la méthode qu'on veut refactorer, puis invoquer « **Refactor > Introduce Indirection...** », qui demandera le nom de la nouvelle méthode, et le nom de la classe qui la déclare. Toutes les références de la méthode refactorée seront mis à jour avec l'appel de la nouvelle méthode.

2. Application de refactorings sur les programmes d'un autre développeur

Pour cette deuxième partie du TP, j'ai récupéré le programme de mon binôme *Salem Mohri*, le programme permet de créer des voitures, motos, ou avions, et leur définir le nom, la marque, le modèle, etc. Les deux refactorings seront appliqués sur le même code, et non pas sur deux programmes différents.

La classe abstraite « Moyendetransport » contient deux attribut : marque et catégorie. Ces attributs permettent de définir la catégorie et la marque d'un moyen de transport. Les sous-classes de cette classe abstraite sont les types de moyens de transport qu'on peut avoir, ex : (voiture, moto, vélo, etc.). Cette classe possède aussi une méthode qui permet de déterminer la date de début de l'assurance du moyen de transport et sa date de fin, la méthode prend deux paramètres de type java.util.Date.

Les classes « Voiture », « Moto », et « Avion » sont des sous-classes de la classe abstraite « MoyendeTransport », ils possèdent tous les trois un attribut « model » de type String, un getter et setter pour cet attribut, et une méthode vitesse() qui affiche la vitesse du moyen de transport.

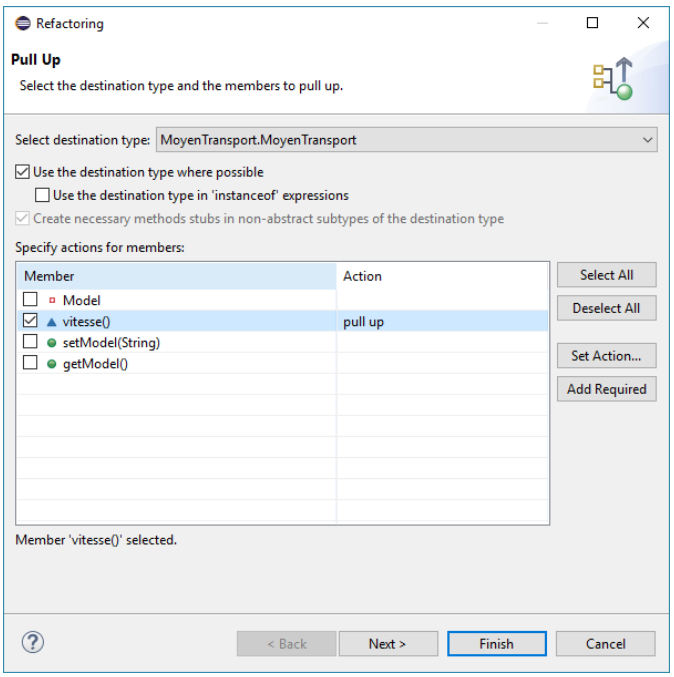
1. Le premier refactoring à appliquer sur le code de ce programme est **Pull UP :**

Ce refactoring permet de déplacer les méthodes identiques des sous-classes vers la classe mère, ceci va enlever la duplication de code, et de pouvoir modifier le code de ces méthodes depuis dans un seul endroit à la place d'aller chercher toutes les sous-classes qui redéfinissent cette méthode et le modifier.

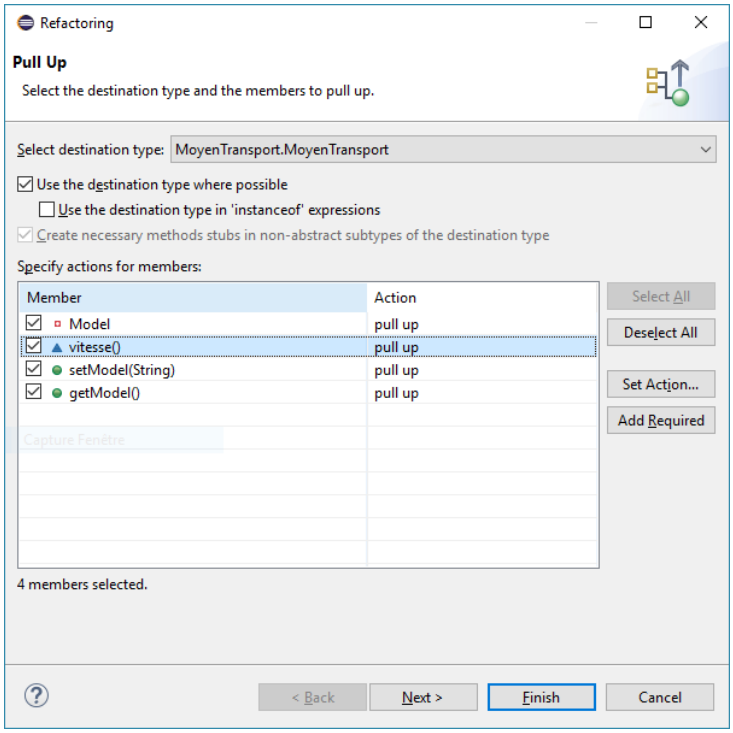
Dans le code de notre programme, les trois sous-classes (Voiture, Moto, Avion) de la classe abstraite « MoyenDeTrasport » possèdent les fonctions vitesse(), getModel(), et setModel(String Model), ainsi qu'un attribut « Model » de type String. Le refactoring va donc déplacer ces fonctions vers la classe abstraite mère, ainsi que l'attribut « Model » qui est utilisé dans les méthodes getModel() et setModel(String Model).

La méthode vitesse() affiche la vitesse en k/h de chaque type de véhicule, comme toutes les sous-classes doivent posséder cette méthode, il vaut mieux la déplacer vers la classe mère même si elle prend des fois une valeur différente. Ceci permet aux sous-classes qui n'ont pas besoin de redéfinir la méthode vitesse() d'avoir la valeur par défaut déclarée dans la méthode vitesse() de la classe mère.

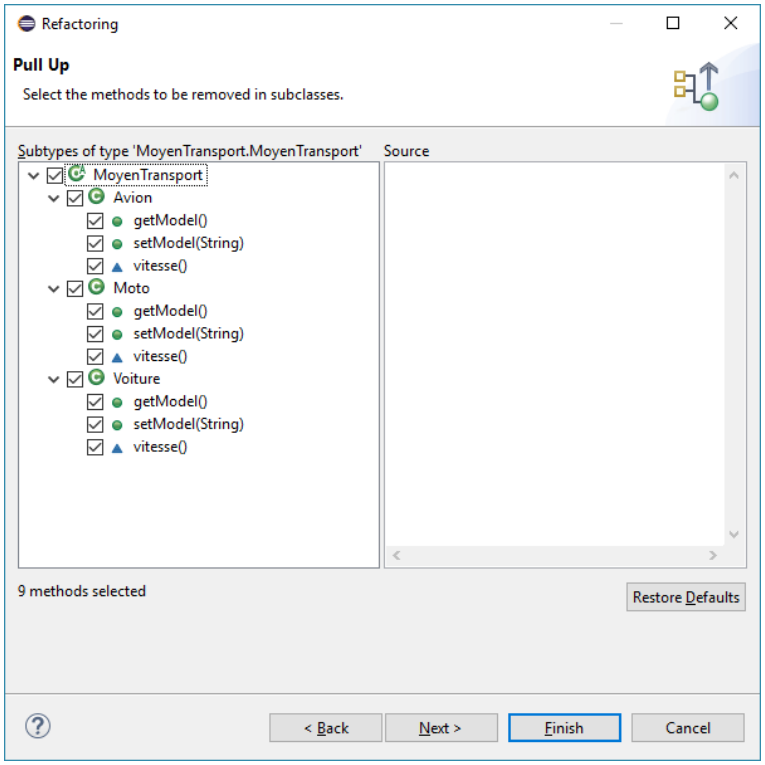
Pour effectuer le refactoring sous Eclipse, il faut sélectionner la méthode vitesse() depuis une sous-classe, puis invoquer « **Refactor > Pull Up...** » depuis le menu contextuel, pour avoir la fenêtre suivante :



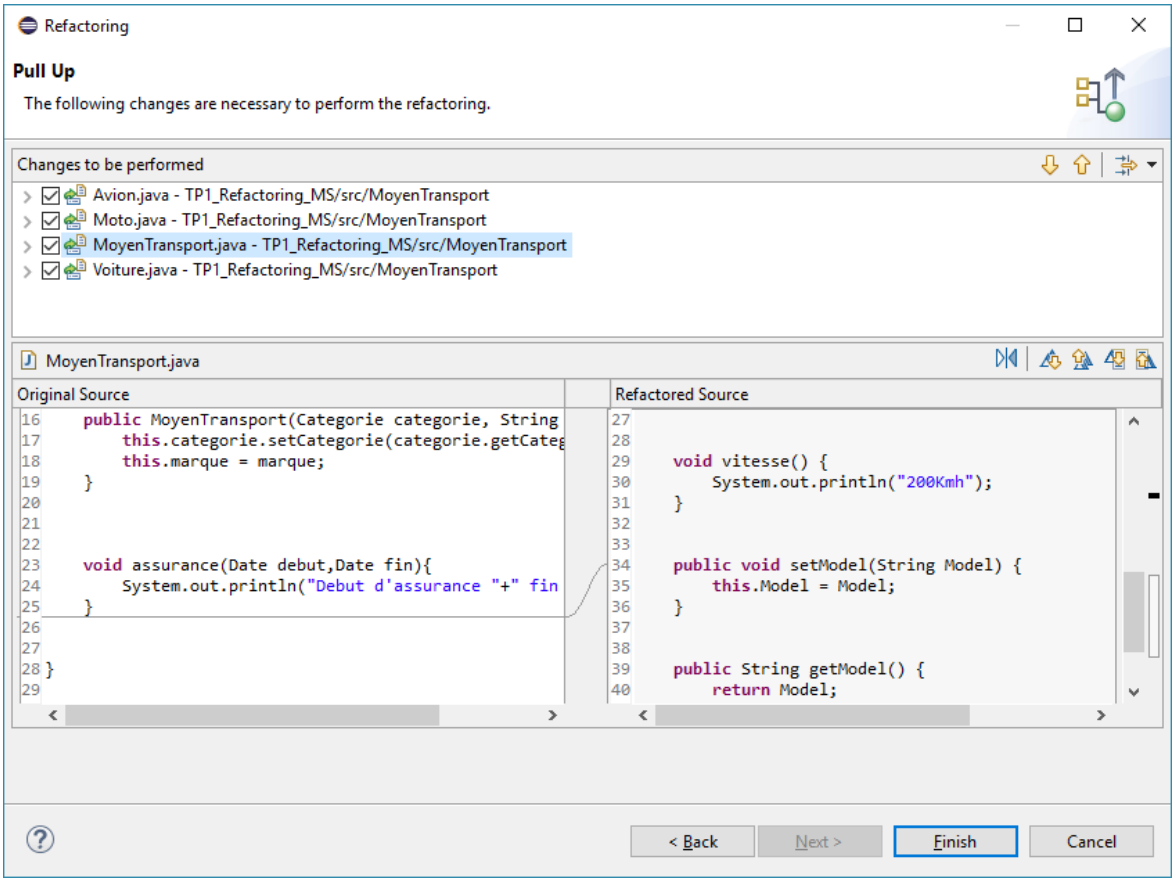
La nouvelle fenêtre nous suggère d'autres méthodes et attributs sur lesquelles ont peut effectuer un « Pull Up » à part la fonction sélectionnée (vitesse()) , on peut donc tous sélectionner pour avoir toutes les méthodes déplacées vers la classe mère en exécutant le Pull Up qu'une seule fois.



En cliquant sur « Next », on nous suggère les autres sous-classes de la classe « MoyenDeTransport » sur lesquelles on peut effectuer le Pull Up. On va alors sélectionner tous sélectionner.



On obtient une nouvelle fenêtre qui montre les changements qui seront effectués sur les classes sélectionnées.



Avant le refactoring pour la classe mère :

```
public abstract class MoyenTransport {
        Categorie categorie;
        String marque;

        public MoyenTransport() { }


        public MoyenTransport(Categorie categorie, String marque) {
                this.categorie.setCategorie(categorie.getCategorie());
                this.marque = marque;
        }

        void assurance(Date debut,Date fin){
                System.out.println("Debut d'assurance "+" fin d'assurance");
        }


}
```

Après le refactoring de la classe mère on obtient :

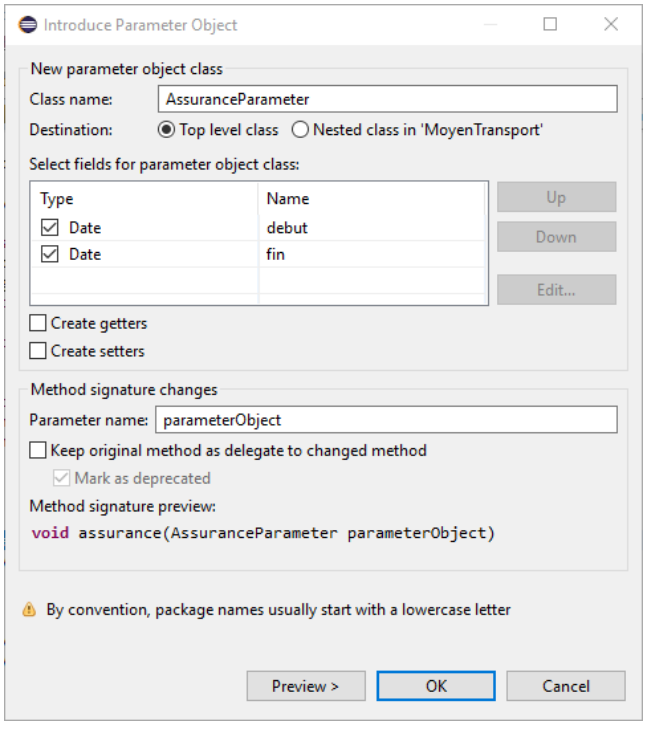
```java
public abstract class MoyenTransport {
	Categorie categorie;
	String marque;
	private String Model = "";

	public MoyenTransport() { }

	public MoyenTransport(Categorie categorie, String marque) {
		this.categorie.setCategorie(categorie.getCategorie());
		this.marque = marque;
	}

	void assurance(Date debut,Date fin){
		System.out.println("Debut d'assurance "+" fin d'assurance");
	}

	void vitesse() {
		System.out.println("200Kmh");
	}

	public void setModel(String Model) {
		this.Model = Model;
	}

	public String getModel() {
		return Model;
	}
}
```

On voit donc que les méthodes vitesse(), setModel(), getModel(), ainsi que l'attribut « Model » ont été déplacé dans la classe mère « MoyenDeTransport ». Les sous-classes ne comportent plus ces méthodes, on pourra redéfinir après la méthode vitesse() dans les sous-classes qui veulent changer la valeur de retour, sinon les sous-classes auront la valeur par défaut définie dans la méthode de la classe mère.

2. Le deuxième refactoring à appliquer est **Introduce Parameter Object**

Ce refactoring est utilisé lorsqu'on a un ensemble de paramètres qui se repétent dans plusieurs méthodes, il permet de remplacer ces paramètre par un objet, afin d'éviter une duplication de paramètres et de rendre le code plus lisible avec un seul objet à la place d'avoir une liste de paramètres.

Dans notre programme, ce refactoring va être utilisé sur la méthode assurance(Date debut, Date fin) de la classe abstraite « MoyenDeTransport ». Le refactoring va remplacer les deux paramètres de type Date par une classe qui contiendra ces deux arguments.
En sélectionnant la fonction « assurance », et en appelant le refactoring depuis « **Refactor > Introduce parameter object...** », on obtient la fenetre suivante.



Cette nouvelle fenêtre montre le nom de la nouvelle classe qui sera créée, les paramètres qui feront partie de cette nouvelle classe, et le nom de l'objet qui sera mis comme nouveau paramètre. On peut aussi voir un aperçu de la signature de la méthode.

Après avoir cliqué sur « OK », on voit que la nouvelle a bien été créée, ainsi que la signature de la méthode a changé qui utilise maintenant un objet de la nouvelle classe.

La nouvelle classe ressemble à cela :

```
    public class AssuranceParameter {
            public Date debut;
            public Date fin;

            public AssuranceParameter(Date debut, Date fin) {
                    this.debut = debut;
                    this.fin = fin;
            }
    }
```

la méthode « assurance » ressemble quant à elle à cela :

```
void assurance(AssuranceParameter parameterObject){
        System.out.println("Debut d'assurance "+" fin d'assurance");
}
```

On voit que la fonction a aussi changé lors de son appel au niveau de la fonction main() :

```
  public class Main {

          public static void main(String[] args) {
                  MoyenTransport moyenTransport = new Moto(new Categorie("Deux roue"), "BMW");

                  moyenTransport.assurance(new AssuranceParameter(new Date(12), new Date(12)));

          }
  }
```

3. Réflexion libre sur les refactorings d'Eclipse

On voit que les refactorings proposés sur le catalogue sont plus nombreux que ceux offerts par Eclipse, cela est peut être dû au fait qu'Eclipse offre une liste limitée de refactorings et cela pour chaque élément sélectionné, la liste offre différents refactorings d'un élément à l'autre, ces refactorings suggérés sont considérés comme les choix pouvant être utilisés par l'élément sélectionné.

Tous les refactorings offerts par Eclipse peuvent être effectués automatiquement par Eclipse, sans avoir avoir à les faire manuellement. Si le nombre de refactorings d'Eclipse est inférieur à celui dans le catalogue, c'est qu'il est possible qu'Eclipse ne peut pas encore offrir tous les refactorings du catalogue à être exécutés automatiquement sans modification manuelle.