Master Informatique GL et Ico - Université Montpellier - 2021

Opérations de refactoring sous eclipse

Travaux pratiques évalués (travaux sur 2 semaines)

Dans ces travaux, nous étudions les opérations de refactoring d'eclipse. Celles-ci sont documentées, par exemple ici (mais cherchez la documentation pour votre version d'eclipse si besoin) :
`https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Fconcepts%2Fconcept-refactoring.htm`
Un site plaisant pour comprendre les opérations de refactoring est également accessible ici :
`https://refactoring.guru/fr`

Vous devriez pouvoir également utiliser intelliJ, vous y retrouverez plus ou moins les mêmes fonctionnalités (non testé récemment). Le travail sera réalisé par binôme (*personne*1, *personne*2). Mettez vos deux noms dans tous les fichiers produits.

1 Création de programmes nécessitant un refactoring

Dans le menu 'Refactoring' d'eclipse, choisissez deux operations de refactoring, à l'exception de 'rename' : une opération pour la *personne*1, une autre opération pour la *personne*2 du binôme. Pour cette étape, *personnei*, $i = 1, 2$, doit inventer un programme pour l'opération de refactoring qui lui est affectée et sur lequel il sera pertinent de l'appliquer. Vous aurez donc **deux** programmes à réaliser par binôme, des programmes qui ne sont pas très bien faits et présentent des défauts à corriger grâce à une des opérations de refactoring choisie. Chaque programme doit contenir une méthode `main` qui teste le code assez exhaustivement.
Résultat attendu : Pour chacune des deux opérations de refactoring, le programme Java nécessitant son application et sa fonction de test `main` ; les programmes doivent fonctionner correctement ; ils doivent également contenir un commentaire expliquant l'opération de refactoring et ce que vous en attendez (quel problème pertinent vous espérez qu'il va résoudre).

2 Application des opérations de refactoring sur les programmes de l'autre personne (du binôme)

*personne*1 va appliquer le (ou les) refactoring(s) attendu(s) sur le programme de *personne*2, et inversement.

Résultat attendu : Pour chacun des deux programmes à 'refactorer', le programme Java obtenu après application de l'opération de refactoring identifiée sous eclipse ; le main doit encore fonctionner ; Un document expliquant (1) l'opération de refactoring réalisé, (2) la procédure appliquée sous eclipse (vous ferez des copies d'écran des différentes étapes), (3) quel intérêt vous lui voyez, (4) si vous trouvez que la mise en œuvre sous eclipse est bien réalisée ou non, et pourquoi, enfin (5) si le test s'est bien déroulé et si vous avez eu besoin de modifier le `main`.

3 Comparaison de 2 catalogues de refactorings

Résultat attendu : Trouvez un refactoring proposé dans https ://refactoring.com/catalog/ que vous ne trouvez pas dans eclipse, et inversement un refactoring d'eclipse que vous ne trouvez pas dans https ://refactoring.com/catalog/. Expliquez-les en quelques lignes.

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

Nous étudions plus en détail cette opération avec l'exemple suivant qui introduit 4 classes fictives qui partagent des signatures de méthodes publiques (par signature on entend ici : type de retour, nom, liste de types de paramètres, par exemple `public boolean add(Object o)`, qui par ailleurs est dupliquée 4 fois puisqu'elle apparaît dans toutes les classes). Ne prêtez pas attention au code actuel des méthodes, qui pourrait être différent.

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

class ListeChainee{
public boolean add(Object o) {return true;}
public boolean isEmpty() {return true;}
public Object get(int i) {return null;}
public Object peek() {return null;}
public Object poll() {return null;}
private void secretLC(){}
}

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

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

Démarche à suivre :

- Essayez d'appliquer l'opération de refactoring 'Extract interface' d'eclipse en sélectionnant toutes ces classes à la fois, puis chacune de ces classes individuellement. Expliquer ce que vous obtenez.
- L'opération de refactoring 'Extract interface' d'eclipse a-t-elle factorisé les signatures de méthodes communes, ou bien restent-elles dupliquées entre interfaces ?
- Selon vous, quel serait le résultat à atteindre pour obtenir une hiérarchie d'interfaces sans duplication de signatures : répondez par le code Java qui vous semblerait pertinent à produire contenant l'ensemble des interfaces et l'ensemble des classes avec leurs entêtes montrant les interfaces implémentées. Vous observez qu'il faut réaliser une opération de refactoring à l'échelle de l'ensemble des classes initiales. C'est un exemple de ce que M. Fowler appelle un 'big refactoring'.
- Si l'ensemble de classes initial était plus grand avec des duplications encore plus enchevêtrées entre les classes, vous auriez des difficultés à le faire 'à la main'. Une technique va consister à utiliser l'analyse formelle de concepts pour structurer le travail. Dans un premier temps, vous allez produire un fichier csv, dont les lignes (objets) sont les noms des classes et les colonnes (attributs) sont les signatures des méthodes publiques. Puis vous allez lui appliquer un algorithme qui calcule une factorisation des attributs qui est à la fois maximale et compacte. Pour cela suivez ce qui est indiqué en annexe.
- Analysez le résultat produit et comparez avec ce que vous aviez fait à la main : avez-vous appliqué la

même factorisation ; commentez la manière dont vous pourriez utiliser le résultat produit par l'analyse formelle de concepts.

Résultat attendu : Les codes et les fichiers de données produits, ainsi qu'un document expliquant votre démarche et votre compréhension.

5 Annexe : application de l'analyse formelle de concepts

1. Créez un fichier listes.csv (voir format ci-dessous avec un petit exemple associant des animaux à des caractéristiques qu'ils peuvent avoir, par exemple 'ladybird' a pour caractéristique 'flies' et n'a pas 'nocturnal').
2. Récupérez le fichier fca4j-cli-0.4.jar à l'adresse `https://www.lirmm.fr/fca4j/`
3. Placez fca4j-cli-0.4.jar dans le répertoire './'
4. Créez le répertoire ./Listes (commande `mkdir ./Listes`)
5. Placez listes.csv dans le répertoire './Listes/'
6. Créez le répertoire ./Listes/AOCposet (commande `mkdir ./Listes/AOCposet`)
7. Lancez fca4j depuis './'
   commande `java -jar fca4j-cli-0.4.jar AOCPOSET Listes/listes.csv -i CSV -s SEMICOLON -g Listes/AOCposet/listesaocposet.dot`
8. Produisez une vue pdf de l'AOCposet
   commande `dot -Tpdf Listes/AOCposet/listesaocposet.dot -o Listes/AOCposet/AOCposet.pdf`

Format du fichier csv attendu (1 indique que l'animal a la caractéristique, sinon il y a 0) :

```
;flies;nocturnal;feathered;migratory;red-bill;elytra;sea-habitat;wood-habitat;six-legged;eats-fish;water-habitat
ladybird;1;0;0;0;0;1;0;0;1;0;0
bat;1;1;0;0;0;0;0;0;0;0;0
ostrich;0;0;1;0;0;0;0;0;0;0;0
greater-flamingo;1;0;1;1;0;0;1;0;0;1;1
silver-gull;1;0;1;0;1;0;1;0;0;1;1
little-tern;1;0;1;1;0;0;1;0;0;1;1
great-auk;1;0;1;0;0;0;1;0;0;1;1
wood-pecker;1;0;1;0;0;0;0;1;0;0;0
giant-otter;0;0;0;0;0;0;0;0;0;1;1
arctic-tern;1;0;1;1;1;0;1;0;0;1;1
```