Abdelli Farès, L3 Maths-Info

Génie Logiciel

2eme partie du projet : amélioration du code

Pour rappel, nous avons choisi avec mon binome de travailler sur la librairie Eclipse Collections. Voici le répertoire git sur lequel nous avons chacun fait nos modifications.

Dans ce rapport je vais détailler les modifications que j'ai pu apporter au code, expliquer pourquoi selon moi elles sont pertinentes et leur lien avec le cours. J'aborderai également les difficultés que j'ai rencontrées lors de la mise en place de certaines améliorations.

Petites modifications :

- Renommage d'expression dans le fichier eclipse-collections/src/main/java/org/eclipse/collections/impl/block/factory/StringFunctions.java:

Ici j'ai simplement simplifié l'expression booléenne `object.length() < 1` par l'expression plus claire et concise `object.isEmpty()`. Cette "amélioration" est minime mais permet de ne pas avoir à se questionner sur la signfication du nombre magique 1.

- Suppression d'une ligne inutile dans le fichier eclipse-collections/src/main/java/org/eclipse/collections/impl/set/mutable/UnifiedSet.java.

Comme l'indique le titre du commit, la réduction de lignes dans une méthode est considérée comme modification moyenne mais j'estime que dans ce cas précis c'est une (très) petite modification. La ligne `this.loadFactor = loadFactor;` est simplement inutile car on affecte la valeur d'un attribut à lui même. Cet attribut n'est même pas utilisé dnas la méthode.

- Remplacement d'un calcul constant par sa valeur dans le fichier eclipse-collections/src/main/java/org/eclipse/collections/impl/set/mutable/UnifiedSet.java. Dans le code

```
case 3 :
   if (this.three instanceof ChainedBucket)
       {
           this.removeLongChain(this, i - 3);
           this.removeLongChain(this, 0);
           return;
       }
```

on peut se passer du calcul de i-3 car i est constant et égal à 3. Remplacer i-3 par 0 nous évite donc un calcul non nécessaire et réduit la complexité asymptotique de la méthode remove. Biensur le gain de performance est négligeable, mais existe quand même théoriquement.

Modification moyenne :

- Ma modification moyenne a consisté à supprimer de la duplication de code entre 2 méthodes dans le fichier eclipse-collections/src/main/java/org/eclipse/collections/impl/map/mutable/ConcurrentHashMap.java. Je précise que pour cette amélioration plus ambitieuse que les précedentes, j'ai testé le code pour m'assurer que tout fonctionnait, et il apparait que sur les 169 000 tests du projet, cette factorisation de code n'a posé de problème à aucun d'entre eux.

Les méthodes `transfer` parcourt les éléments d'un tableau (src) en paramètre et les copie vers un autre conteneur (dest). `reverseTransfer` effectue la même opération en parcourant le tableau dans l'autre sens. Ces méthodes ont donc le même code à quelques exceptions près comme le sens de la boucle `for`. D'après le cours, une telle répétition de code est un "Code Smell" qu'il faudrait refactorer.

J'ai donc utilisé le pattern de refactoring "Extract Method" (également vu en cours) afin d'extraire les parties communes de code pour les mettre dans une nouvelle méthode `processTransfer` qui sera appelée par `transfer` et `reverseTransfer`. Le principe de cette méthode est que le sens de boucle (incrémentation ou décrémentation de l'indice) est choisi à l'aide d'un paramètre `isReverse` qui permet de choisir le "bon chemin". La complexité cyclomatique de `processTransfer` est légerement plus élevée que celle de transfer ou reverseTransfer prises individuellements car on doit parfois faire des disjonctions de cas en fonction de `isReverse`, ce qui rajoute des conditions en `if`. Cependant si on prend en compte la complexité cyclomatique totale de la classe ConcurrentHashMap.java, alors cette dernière est moins élevée qu'avant les modifications car on a rassemblé 2 longues méthodes en une.

On peut également noter qu'un autre avantage de cette factorisation de code, est qu'elle ne nécéssite pas de grand changements dans le reste du code car les méthodes `transfer` et `reverseTransfer` existent toujours et sont dans le même fichier. Ainsi par exemple, lorsque la méthode `resize` de `ConcurrentHashMap.java` atteint l'instruction `this.transfer(oldTable, resizeContainer)`, tout fonctionne de la même manière qu'avant les modifications. C'est notamment grâce à cela que les tests sont tous passés simplement.

Grande modification :

Lors de le 1ère partie du projet, nous avions compté au moins 5 God Class, réparties dans les 3 packages auxquels nous nous étions interéssés. J'ai choisi de d'essayer de décomposer l'une d'elles : la classe `ByteHashSet.java` (eclipse-collections/src/main/java/org/eclipse/collections/impl/set/mutable/primitive/ByteHashSet.java).

Initialement, cette classe contenait 1735 lignes de code, la moyenne du package `set` étant de 370. Toujours d'après le cours sur les Code Smells, on peut reconnaitre une God Class au fait qu'elle effectue beaucoup de taches différentes, ce qui était le cas ici. J'ai donc essayé de "ranger" les méthodes par groupes, qui deviendront par la suite des classes à qui ByteHashSet "déleguera" les taches. Voici le brouillon de la décomposition.

L'idée est d'instancier chaque rôle comme un attribut de la God Class, et appeller ces rôles quand on en aura besoin. Les classes correspondants aux type de méthodes ont comme attribut la God Class, afin d'y récupérer les informations nécéssaires au bon fonctionnement des méthodes qu'elles implémentent. Exemple

avec les fonctions prédicats :

Dans ByteHashSet.java :

`protected ByteHashSetPredicates bhsetPredicate = new ByteHashSetPredicates(this);`

Et dans ByteHashSetPredicates.java : On retouve l'attribut `protected ByteHashSet bhset;` et son constructeur :

`public ByteHashSetPredicates(ByteHashSet set){this.bhset = set;}` ,

ainsi que l'implémentation des méthodes prédicats.

La méthode contains de ByteHashSet.java suivante :

```
public boolean contains(byte value)
    {
        if (value <= MAX_BYTE_GROUP_1)
        {
            return ((this.bitGroup1 >>> (byte) ((value + 1) * -1)) & 1L) != 0;
        }
        if (value <= MAX_BYTE_GROUP_2)
        {
            return ((this.bitGroup2 >>> (byte) ((value + 1) * -1)) & 1L) != 0;
        }
        if (value <= MAX_BYTE_GROUP_3)
        {
            return ((this.bitGroup3 >>> value) & 1L) != 0;
        }

        return ((this.bitGroup4 >>> value) & 1L) != 0;
    }
```

devient :

```
public boolean contains(byte value)
    {
        return this.bhsetPredicate.contains(value);
    }
```

De cette manière la classe est plus lisible, et les méthodes traitées par des classes bien définies en fonction du type de méhode. On augmente donc la lisibililté du code, tout en réduisant le nombre de lignes et la complexité cyclomatique de la classe. On a diminué de 400 le nombre de ligne de code. Ces modifications nous ont permis de nous rapprocher d'un "Clean Code", dont l'une des caractériqtique précisée par le cours est de garder les méthodes courtes. Pour toutes ces raisons, j'estime que cette décomposition est bénéfique au projet.

Cependant, je me dois de nuancer certains points. D'abord, des modifications aussi importantes necessitent biensur d'être testées. J'ai éxécuté les 169000 tests, qui utilisent pour beaucoup des méthodes de `ByteHashSet.java` qui est une classe importante de la librairie, et il en résulte que la grande majorité des tests passent, à l'exception de 9 d'entres eux. Je n'ai malheureusement pas pu comprendre d'où venait le problème. Enfin, malgrès une réduction des lignes de codes, je note toujours des répétitions de code entre méthodes, comme pour les differentes méthodes `equals` par exemple. Ces répétitions sont dûes aux 3 classes privées à l'interieur de ByteHashSet.java. J'ai d'abord pensé à les rendre publiques, et effectuer le même travail que pour la classe principale, mais le fait que le créateur ait décidé de mettre ces classes en `private` m'a fait douter de cette méthode car je redoutais qu'on puisse appeller leurs méthodes depuis l'exterieur de la classe alors qu'elles n'étaient pas faite pour cela. N'étant pa habitué à rencontrer plusieurs classes au sein d'une même classe, j'avoue avoir eu du mal à continuer le refactoring de la God Class.

Pour conclure sur cette grande modification, même si elle n'est pas parfaite et aurait certainement pu être poussée plus loin, je pense que l'idée de la décomposer comme je l'ai fait a malgrès tout permis quelques améliorations.

Listes des commits des classes de "roles" :

- ByteHashStringMethods.java
- ByteHashSetForEachMethods.java et ByteHashSetMathsMethods.java
- ByteHashSetModifiers.java