MGL7460 : Laboratoire #5
Évaluation de la qualité de code Java avec `SonarQube`

10 novembre 2016

Le but du présent laboratoire est de vous familiariser avec l'utilisation d'un outil d'analyse de code tel que `SonarQube`. Du code Java vous est fourni et vous devez l'analyser avec `SonarQube` puis, si possible, corriger certains des problèmes identifiés par `SonarQube`.

Pour une description du code fourni et des classes et méthodes, consultez l'annexe A.

---

1 Pour obtenir le code à analyser et modifier

Pour obtenir les fichiers pour ce laboratoire, téléchargez l'archive suivante sur le Bureau puis désarchivez-la :

`http://www.labunix.uqam.ca/~tremblay/MGL7460/Labos/Arbre7460.zip`

2 Comment utiliser SonarQube

2.1 Utilisation sur un poste Windows du labo

Voici les étapes à suivre pour utiliser SonarQube sur un des postes Windows du PK-4665 :

a. Lancez SonarQube par l'intermédiaire de «Rechercher les programmes et fichiers» :

```
C:\sonarqube\bin\windows-x86-64\StartSonar.bat
```

b. Lancez un *shell* de commandes par l'intermédiaire de «Rechercher les programmes. . . » :

```
cmd
```

c. Par l'intermédiaire du *shell* de commandes créé, allez dans le répertoire contenant les fichiers téléchargés puis lancez l'analyse :

```
\D:\Utilisateurs\votreid> cd Desktop\Arbre7460
\D:\Utilisateurs\votreid\Desktop\Arbre7460> C:\sonar-scanner\bin\sonar-runner.bat
```

Source de ces informations :
http://docs.sonarqube.org/display/SONAR/Get+Started+in+Two+Minutes

2.2 Utilisation sur votre machine personnelle

Si vous désirez installer et exécuter SonarQube sur votre machine personnelle, voir informations à la page suivante :
http://docs.sonarqube.org/display/SONAR/Get+Started+in+Two+Minutes

Dans ce cas, procédez à l'installation requise avant le laboratoire!

3 Comment procéder avec le code fourni

Après avoir lancé `SonarQube` tel que décrit plus haut, allez à l'URL suivant avec un fureteur : `http://localhost:9000`

3.1 Analyse des *Bugs & Vulnerabilities*

Examinez les *Bugs* identifiés par `SonarQube`. De quels niveaux sont ces *bugs*? Voyez-vous comment, sans trop de difficulté, corriger certains d'entre eux? Si oui, faites les corrections appropriées puis relancez l'analyse du code.

3.2 Analyse des *Issues*

Examinez les divers *Issues* identifiés comme étant des *Code smells*, notamment dans les fichiers suivants :

- `DiversesMethodes.java`
- `Feuille.java`
- `Noeud.java`

Pouvez-vous corriger certains de ces problèmes? Si oui, faites les corrections appropriées puis relancez l'analyse du code.

Note : Vous pouvez ignorer les *issues* qui traitent des aspects suivants :

- *Move this file to a named package*
- *Document this public class* o *Document this public class*

3.3 Analyse de la couverture des tests

Allez dans la partie intitulée «*Coverage*», cliquez sur la ligne *Coverage* en dessous du pourcentage indiqué puis examinez les divers fichiers (qui ne sont pas des programmes de tests) quant à leur couverture — par exemple, `Feuille.java`, notamment la méthode `equals`. Que constatez-vous?

Remarque : L'analyse de la couverture des tests a été faite **au préalable** avec l'outil `JaCoCo`.[1] Les résultats, bruts, de cette analyse ont été conservés dans le fichier `jacoco.exec` (fichier par défaut). Ces résultats sont mis en forme et **liés au code source** par `SonarQube` par l'intermédiaire du fichier de configuration (fichier à la racine du répertoire fourni) :

```
$ cat sonar-project.properties
...
sonar.jacoco.reportPath=./jacoco.exec
```

[1] `http://www.eclemma.org/jacoco`

A Ce que fait le code fourni

Le code fourni définit différentes méthodes sur des classes pour des arbres n-aires — donc des arbres avec un ou plusieurs enfants. Ces arbres sont définis dans les classes `Arbre` (superclasse abstraite), `Noeud` (interne) et `Feuille` (terminal). Donc, les méthodes pour les cas de base (noeuds terminaux) sont définies dans la classe `Feuille`, alors que les méthodes pour les arbres avec enfants sont définies dans la classe `Noeud`.

Les arbres *ne sont pas des arbres génériques* – donc le champ `valeur` de chaque noeud (interne ou feuille) est toujours **un entier**.

La figure 1 présente un diagramme de classes UML décrivant l'organisation de ces classes et méthodes — `equals`, `similaire` et `toString` sont omises sur ce diagramme.

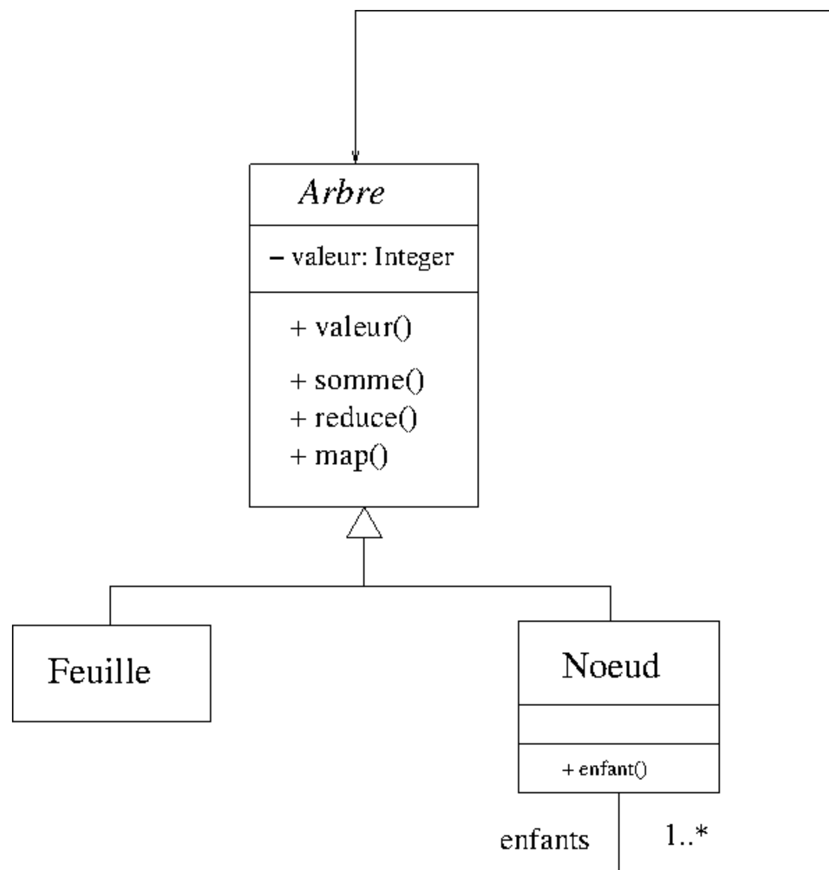

Figure 1: Diagramme de classes UML pour les arbres n-aires.

Des fichiers de tests unitaires (JUnit) sont aussi fournis — `Tester{Somme,Reduce,Map}` — de même qu'un fichier avec `DiversesMethodes` auxiliaires, méthodes utilisées dans les tests unitaires et contenant certains problèmes ou *code smells*.

Voici une brève description des différentes méthodes, toutes mises en oeuvre de façon parallèle en utilisant la bibliothèque `java.util.concurrent` :

a. `somme()` : Fait la somme des champs `valeur` des différents noeuds de l'arbre :

```
Arbre a0 = new Noeud( 12, new Feuille(10), new Feuille(20) );
Arbre a  = new Noeud( 100, a0, a0 );

assertEquals( 100 + (12 + 10 + 20) + (12 + 10 + 20),
              a.somme() );
```

b. `reduce()` : Fait une **réduction** par un opérateur binaire des champs `valeur` :

```
Arbre a0 = new Noeud( 12, new Feuille(10), new Feuille(20) );
Arbre a  = new Noeud( 100, a0, a0 );

assertEquals( 100 * (12 * 10 * 20) * (12 * 10 * 20),
              a.reduce( (x, y) -> x * y )
```

c. `map()` : Applique une lambda-expression sur chacun des champs `valeur` d'un arbre pour produire un nouvel arbre **avec la même structure** — voir la méthode `similaire` — mais avec des champs valeurs obtenus par l'application de la fonction :

```
Arbre a0 = new Noeud( 12, new Feuille(10), new Feuille(20) );
Arbre a  = new Noeud( 100, a0, a0 );

Arbre r0 = new Noeud( 120, new Feuille(100), new Feuille(200) );
Arbre r  = new Noeud( 1000, r0, r0 );

assertTrue( r.equals( a.map( (x) -> x * 10 ) ) );
```

Note : La mise en oeuvre parallèle est tout à fait secondaire, donc ce n'est pas important si vous ne comprenez pas tous les détails de création et manipulation des *threads*.