Tests Logiciels

M2 Informatique CD - Université d'Angers

TP1. Programmation par Contrats et JUnit

Exercice 1: Prise en Main JUnit 5 sous Eclipse

Créer un projet Java sous Eclipse 2022 IDE et donner la dépendance à JUnit5 dans le BuildPath (clic droit sur le projet, puis add libraries).
Les fonctions ci-dessous seront codées dans un classe Calculator d'un package du repertoire src, les tests dans une classe CalculatorTest d'un package du repertoire tests. Dans cet exercice, seule l'annotation JUnit @Test est utile.

1. Définir la fonction suivante et la tester en utilisant jUnit

```
1     public int add(int x, int y) {
2         return x + y;
3     }
```

2. Définir la fonction suivante et la tester (en testant aussi le cas de la division par zéro).

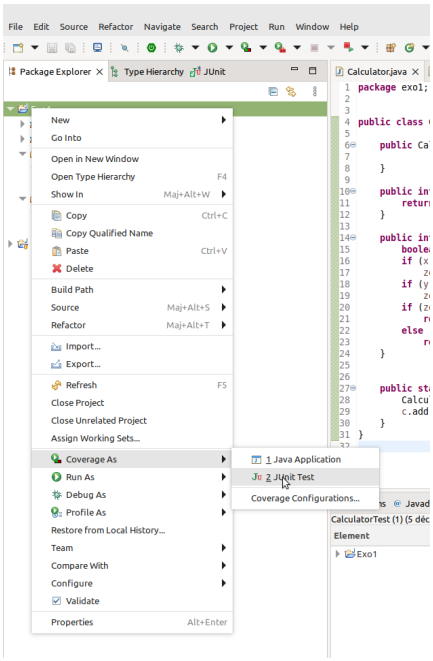
```
4     public int div(int x, int y) {
5         return x / y;
6     }
```

3. Modifier le code de la fonction div de manière à rendre le cas de la division par zero explicite dans le code et tester à nouveau.

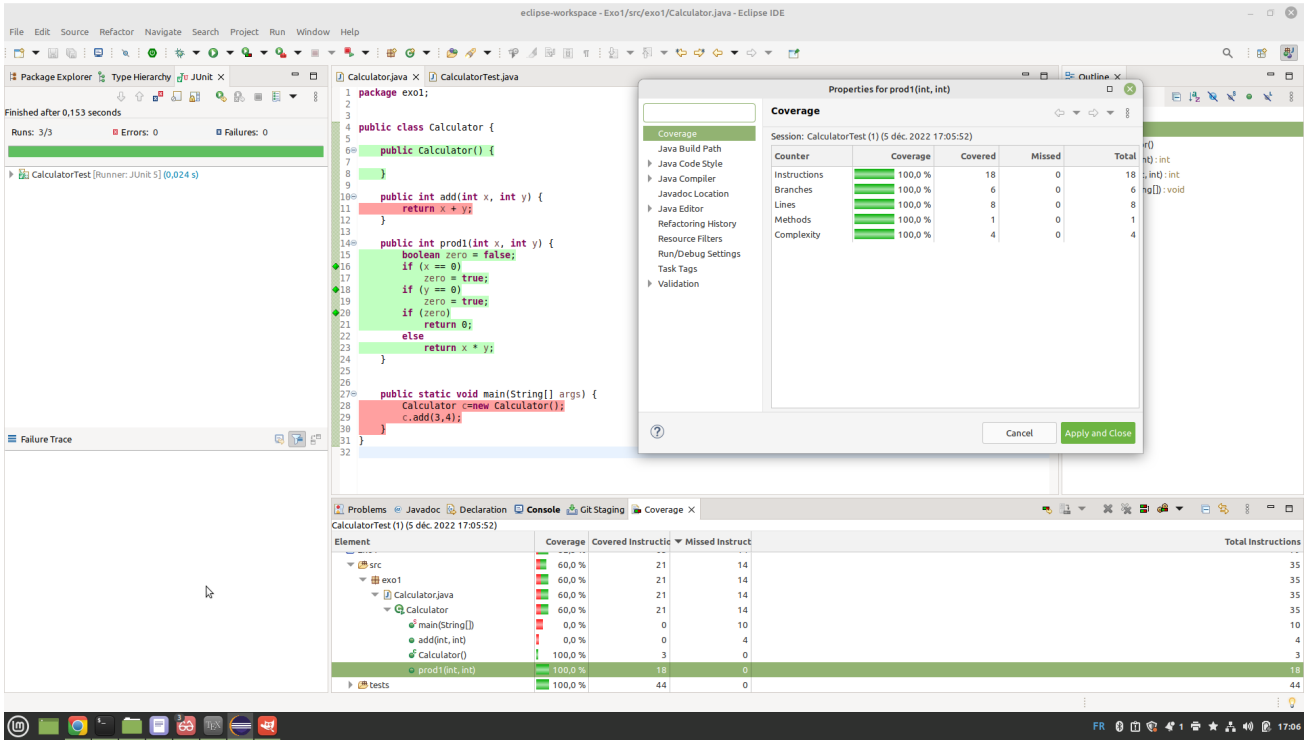
4. Définir la fonction suivante et la tester :

```
7       public int prod(int x, int y) {
8           boolean zero = false;
9           if (x == 0)
10              zero = true;
11          if (y == 0)
12              zero = true;
13          if (zero)
14              return 0;
15          else
16          return x * y;
17      }
```

5. Observer la couverture de tests via Coverage As:



Cela permet d'obtenir les statistiques de couverture (en bas), ainsi qu'une coloration des instructions couvertes et non couvertes. En cliquant droit sur une méthode de la liste des couvertures en bas, et en sélectionnant properties, on peut afficher le détail des statistiques de couverture sur cette méthode (boîte de dialogue affichée ci-dessous):



6. Faire en sorte d'avoir plus de 90% de couverture sur chaque méthode.

Exercice 2: Tests paramétrisés

Depuis la version 5 de jUnit, il est possible de regrouper des cas de test en utilisant les annotations montrées sur l'exemple suivant :

```
18      @ParameterizedTest
19      @CsvSource({"0, 1, 0", "1, 0, 0", "1, 1, 1"})
20      void testProd(int x, int y, int expected) {
21          int actual = calc.prod(x, y);
22          assertEquals(expected, actual);
23      }
```

La syntaxe utilisée est le format CSV (Comma Separated Values), avec une chaîne de caractères par cas de test. Le nombre et l'ordre des arguments donnés doit correspondre à ceux de la fonction de test. Des heuristiques sont utilisées pour reconnaître les valeurs des arguments et les convertir vers le type attendu. Les détails sont disponibles dans la documentation de JUnit 5 : `https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests`

- Reprendre les tests de l'exercice précédant, en regroupant les différentes données de test de chaque fonction en utilisant le format CSV.

Il est aussi possible de passer par un fichier CSV externe (l'option numLinesToSkip = 1 permet d'ignorer la ligne d'entête du fichier CSV):

```
24      @ParameterizedTest
25      @CsvFileSource(resources = "/testProd2.csv", numLinesToSkip =
            1)
26      void testProd2(int x, int y, int expected) {
27          int actual = calc.prod(x, y);
28          assertEquals(expected, actual);
29      }
```

- Créer un nouveau fichier "testProd2.csv" dans le répertoire test, et le remplir avec quelques données de test au format CSV. Tester le code de test ci-dessus utilisant ce fichier.

- Écrire une fonction shortest qui prend en entrée trois chaînes de caractères et renvoie la plus courte. Tester cette fonction en regroupant les données de test dans un fichier CSV externe. On fera attention aux taux de couverture de ces tests.

Il est aussi possible générer les cas de test, et de les stocker dans une collection, ou sous forme de Stream.

```
30      static IntStream range() {
31          return IntStream.range(0, 20).skip(10);
32      }
33      @ParameterizedTest
34      @MethodSource("range")
35      void testWithRangeMethodSource(int x) {
36          int actual = calc.add(x, x);
37          int expected = 2 * x;
38          assertEquals(expected, actual);
39      }
```

- Utiliser ce mécanisme pour tester plus efficacement la fonction shortest.

Exercice 3: Programmation par contrat (design by contract)

Le contrat d'une fonction est donné par un couple de propriétés pré-condition/post-condition (pre/post, requires/ensures, assume/assert, ...). Ces contrats sont éventuellement complétés par des invariants qui mentionnent des propriétés qui doivent être préservés par les fonctions.
Une syntaxe précise a été définie pour les contrats ou des invariants (JML: Java Modeling Language), qui doit être utilisée dans le source Java sous forme de commentaires :

- @requires: Defines a pre-condition on the method that follows.
- @ensures: Defines a post-condition on the method that follows.
- @invariant: Defines an invariant property of the class.
- \result : A special identifier for the return value of the method that follows.
- \old(variable) : A modifier to refer to the value of the variable at the time of entry into a method.

Par exemple, le contrat de add correspondant au test précédent pourrait s'écrire :

```
40      // @requires (0 <= x && x < 100);
41      // @ensures (\result == 2 * x);
```

Test d'une implantation des piles

La classe BoundedStack donnée ci-dessous implémente les primitives habituelles push, pop, peek, isEmpty et isFull pour des piles de char de taille bornée (par capacity), dont les éléments sont stockés dans le tableau elems et où top correspond à la première case libre du tableau. Remarquez que push, pop, et peek ne vérifient pas si la pile est pleine ou vide. Par conséquent il est nécessaire d'appeler isEmpty ou isFull avant d'ajouter, de supprimer ou de consulter le sommet de la pile.
Les exigences qui correspondent à ce code sont les suivantes :

- push: Place a new item on top of the stack (does not check if the the stack is full).
- pop: Remove and return the top stack item (does not check if the stack is empty).
- peek: Return top stack item without removing it (does not check if the stack is empty).
- isEmpty: Returns true if the stack is empty, otherwise returns false.
- isFull: Returns true if the stack is full, otherwise returns false.

```
42  class BoundedStack {
43      char[] elems;
44      int top;
45      // @invariant (top >= 0 && top <= elems.length);
46      // @requires (capacity > 0);
47      // @ensures (top == 0 && elems.length == capacity);
48      BoundedStack(int capacity) {
49          top = 0;
50          elems = new char[capacity];
51      }
52      // @requires (size >= 0) && (size < array.length);
53      // @ensures (top == size && elems.length == array.length)
            ;
54      BoundedStack(char[] array, int size) {
55          top = size;
56          elems = array;
57      }
58      // @requires (top > 0);
59      // @ensures \result == elems[top - 1];
60      char peek() {
61          return elems[top - 1];
62      }
63      // @requires (top < elems.length);
64      // @ensures (top == \old (top) + 1) && elems[\old (top)]
           == item;
65      void push(char item) {
66          elems[top] = item;
67          top = top + 1;
68      }
69      // @requires (top > 0);
70      // @ensures (top == \old (top) - 1) && \result == elems[
           top];
71      char pop() {
72          top = top - 1;
73          return elems[top];
74      }
75      // @ensures (\result == (top == elems.length));
```

```
76          boolean isFull() {
77              if (top == elems.length)
78                  return true;
79              else
80                  return false;
81          }
82          // @ensures (\result == (top == 0));
83          boolean isEmpty() {
84              if (top == 0)
85                  return true;
86              else
87                  return false;
88          }
89      }
```

1. Self Testing: Implémenter les invariants et les contrats de la classe BoundedStack en insérant des assertions Java4 (en utilisant le mot-clé assert) dans le corps des méthodes. Il pourra être nécesssaire d'introduire des variables auxiliaires pour garder les anciennes valeurs (old) de certaines variables.

2. Unit Testing: Implémenter les invariants et les contracts de la classe BoundedStack sous forme de cas de test JUnit 5. Vérifiez que vous obtenez bien 100% de couverture des décisions (branch coverage).

Test d'une implantation des files d'attente

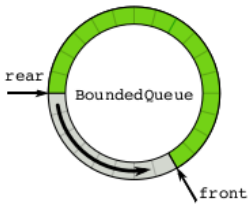
La classe BoundedQueue ci-dessous implémente les primitives habituelles (enqueue, dequeue, isFull, isEmpty) pour des files d'attente de char de taille bornée (par capacity), dont les éléments sont des entiers stockés dans le tableau buffer et où size correspond au nombre d'éléments dans la file d'attente, et front et rear correspondent respectivement à la tête et à la fin de la file.

```
90      class BoundedQueue {
91          int front;
92          int rear;
93          int size;
94          char[] buffer;
95          public Queue(int capacity) {
96              front = 0;
97              rear = 0;
98              size = 0;
99              buffer = new char[capacity];
100         }
101         public Queue(char[] array, int last) {
102             front = 0;
103             rear = last;
104             size = last;
```

```
105             buffer = array;
106         }
107         public boolean isEmpty() {
108             return (size == 0);
109         }
110         public boolean isFull() {
111             return (size == buffer.length);
112         }
113         public char head() {
114             return buffer[front];
115         }
116         public void enqueue(char c) {
117             size++;
118             buffer[rear] = c;
119             rear = (rear + 1) % buffer.length;
120         }
121         public char dequeue() {
122             char result = buffer[front];
123             size--;
124             front = (front + 1) % buffer.length;
125             return result;
126         }
127     }
```

Les exigences qui correspondent à ce code sont les suivantes :

- une file d'attente est correctement construite : buffer est initialisé de taille non nulle, size et rear sont égaux (et valent 0 pour le premier constructeur), et front vaut 0.
- si on ajoute (enqueue) un élément dans une file d'attente supposée non pleine, la file contient alors un élément de plus et son dernier élément est l'élément ajouté.
- si on enlève (dequeue) un élément d'une file d'attente supposée non vide, la file contient alors un élément de moins et on retourne l'élément en tête de file.
- isFull détermine bien si la file d'attente est pleine.
- isEmpty détermine bien si la file d'attente est vide.



1. Contrats: Traduire ces exigences en contrats et invariants JML.
2. Self Testing: Implémenter les invariants et les contracts de la classe BoundedQueue en insérant des assertions Java (en utilisant le mot-clé assert) dans le corps des methodes. Il pourra être nécesssaire d'introduire des variables auxiliaires pour garder les anciennes valeurs (old) de certaines variables.

3. Unit Testing: Implémenter les invariants et les contracts de la classe BoundedQueue sous forme de cas de test JUnit 5. Vérifiez que vous obtenez bien 100% de couverture des décisions (branch coverage).

Exercice 4: Test d'Application

Dans cet exercice nous allons chercher à tester une application de Recherche Documentaire, possédant des inter-dépendances entre les objets qu'elle manipule. On cherchera d'abord à réaliser des tests unitaires en isolant chaque composant, puis nous inspecterons la bonne communication entre composants via la mise en place de tests d'intégration.
Nous utiliserons pour cela le code donné sur moodle à l'adresse: `https//xx.zip`. Cette application possède les dépendances suivantes:

- Le module core définit la structure de données Document qui sera centrale dans l'application;
- Le module indexation s'occupe d'indexer la collection de documents pour préparer les recherches futures. La méthode index de la classe Index crée deux index: un index donnant la liste des nombres d'occurrences des mots apparaissant dans chaque document, et un index inversé possédant le même contenu mais où les entrées correspondent aux mots du vocabulaire (chaque entrée contient la liste de documents où le mot correspondant apparaît, associé à son nombre d'occurrences dans ce document). Pour faire cela la méthode s'appuie sur un Parser du fichier contenant la collection, ainsi que d'un TextRepresenter qui normalise les différentes variantes d'un mot (via stemming);
- Le module models contient les différents modèles de RI permettant de créer une liste ordonnée de tous les documents en fonction d'une requête utilisateur. Chaque modèle hérite de la classe abstraite IRModel, possédant une méthode getScores qui retourne les scores de tous les documents qui ont au moins un mot en commun avec la requête. Pour cela les modèles s'appuient sur un Weighter, en charge de créer les vecteurs de poids des différents mots des documents, selon un schéma défini, à partir de l'index. Les modèles doivent avoir été préparés pour pouvoir être appliqués.
- Le module evaluation est en charge de l'évaluation des différents modèles sur un corpus donné. En particulier, la classe EvalIRModel définit une méthode statique eval, qui produit un fichier de résultats pour les différents modèles passés en paramètres, selon les mesures d'évaluation passés en paramètres, pour un ensemble de requêtes définies dans un objet QueryParser (en charge de lire de fichier de requêtes et les jugements de pertinence des différents documents pour ces requêtes).

Tests unitaires

Réaliser à minima les tests suivants :

1. Tester la méthode ParserCISI.getDocument(String str), en charge de retournant l'objet Document correspondant à la chaîne passée en paramètres (supposée suivre le format spécifié en tête de classe, voir le fichier de données CISI dans Data pour des exemples).

2. Tester la méthode ParserCISI.nextDocument(), dans le cas où le parser a été initialisé et dans le cas où il ne l'a pas été. Tester aussi le cas où le format du fichier n'est pas correct.

3. Tester la méthode WeighterTF.getDocWeightsForStem(String stem). On souhaite le faire en isolation du fonctionnement de l'index dont la méthode dépend. Utiliser Mockito pour tester cette méthode (en vérifiant le nombre d'appels aux fonctions de l'Index).

4. Tester la méthode getScores(HashMap<String,Integer> query), dans une version avec normalized = False et une version où normalized = True. Dans le premier cas la méthode utilise pour classer les documents le produit scalaire des poids des documents avec ceux de la requête passée en paramètres, dans le second car c'est un cosinus (produit scalaire des vecteurs normalisés). Cette méthode doit être testée en isolation des autres classes, en utilisant Mockito.

5. Toujours en utilisant Mockito, tester la méthode AP.eval(Hyp hyp), qui étant donné un classement de documents fourni par le modèle Hyp et la liste des documents pertinents calcule la précision moyenne de ce classement (voir `https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)#Average_precision`).
Pour tous ces tests, chercher à atteindre un taux de couverture maximal sur les méthodes testées.

Tests d'integration

En utilisant Mockito pour ne tester que les composants concernés par le test défini, appliquer une approche bootom-up de tests d'integration, où l'on testera d'abord des couples de composants, puis des les composants qui en dépendent, et ainsi de suite.

Présentation des Tests

Experimenter l'utilisation de @Tag, @DisplayName et @Nested, pour améliorer la lisibilité des rapports de tests.