Test unitaires et Test Driven Development

1 Testons les intervalles

Au cours de ce TP, nous allons recréer et tester la classe Intervalle que vous avez écrite en TP auparavant. Nous allons réécrire la classe au fur et à mesure, en même temps qu'on en écrit les tests.

1.1 Mise en place du projet

1. Dans votre IDE, créez une classe Intervalle avec deux attributs `double min,max`. Générez les getters de ces attributs ("source > generate getters and setters")
2. En cliquant droit sur votre classe intervalle dans l'explorateur de paquets, choisissez "new > Junit test Case". Eclipse devrait vous proposer de selectionner des méthodes à tester, choisissez simplement pour le moment la méthode héritée `equals`. Intégrez Junit5 au projet et au build path comme suggéré.
   Si vous avez un module (vous devriez), vous avez encore des erreurs car il manque le *require* dans le *module.info* et que la bibliothèque *Jupiter* a été intégrée au *classpath* plutôt qu'au *modulepath*. En quelques clics, corrigez ces deux erreurs (premier clic sur le *import*, intégrer org.junit... au *module.info*, puis clic dans le *module.info*, déplacer l'entrée du *classpath* au *modulepath*).
3. Vous pouvez maintenant exécuter les tests, le test actuel échoue, on le corrigera plus tard. En attendant, commentez juste l'instruction `fail` pour que vos tests passent.

1.2 Premiers tests

4. La classe Intervalle disposera d'un constructeur par défaut, sans paramètre, qui génère l'intervalle $[0,0]$. Écrivez un premier test `testConstructeur` qui vérifie que les bornes min et max d'un intervalle construit selon ce constructeur sont égales à 0. Vous utiliserez la méthode `assertEquals` qui prend en premier argument la valeur cible et en second la valeur à tester. Sans constructeur, le constructeur par défaut de `Intervalle` valide le test!
5. Nous allons ajouter la méthode `setMinMax`, commençons par le test `testSetMinMax`. Il doit créer un intervalle puis appeler la méthode (inexistante) `setMinMax` avec un premier paramètre plus petit que le second et vérifier la bonne affectation des attributs `min` et `max`, puis avec un premier paramètre plus grand et vérifier à nouveau. Vous pouvez constater que même si Eclipse détecte l'erreur, le code s'exécute et le test échoue, sans casser les autres tests.
6. Écrivez maintenant la méthode `setMinMax` dans la classe intervalle. Lancez les tests.
7. Nous allons maintenant ajouter un constructeur avec deux arguments, pour les bornes de l'intervalle. Commencez par créer le test du constructeur en testant de la même façon des ordres différents sur les paramètres. Cela peut sembler inutile en anticipant l'utilisation de la fonction `setMinMax` dans le constructeur, mais nous ne pouvons être sur que des modifications ultérieures du code ne risquent pas de faire un choix différent. Donc on fait ce test `testConstructeurParametre`.
8. Utilisez la suggestion de correction pour intégrer le bon constructeur. Votre test devrait passer maintenant, mais vous noterez que ce sont `testConstructeur` et `testSetMinMax` qui ne passent plus! On appelle ça une *régression*, une modification du code qui casse une fonctionnalité opérationnelle auparavant. Corrigez cette régression en intégrant un constructeur sans arguments.

1.3 Plus de tests

9. Ajoutez un test pour la méthode `getLongueur` qui doit retourner la longueur de l'intervalle, et intégrez la méthode en question.
10. Ajoutez le test pour la méthode `estIncluse` qui prend un double en paramètre et retourne si oui ou non la valeur est incluse dans l'intervalle. Vous pourrez utiliser les méthodes `assertTrue` et `assertFalse`.
11. Générez les méthodes `equals` et `hashcode` [1] de `Intervalle` en vous appuyant sur les deux attributs `min` et `max` ("source>generate hashcode and equals"), puis complétez `testEquals`.
12. Définissez des tests pour la méthode `contient(Intervalle autre)` qui est censé retourner si l'intervalle `autre` est contenu dans l'intervalle courant. Vous devriez envisager 6 scénarios de tests, bien qu'une implémentation simple existe... [2]
13. Même question pour la méthode `intersection(Intervalle autre)` qui retourne l'intersection entre deux intervalles. Utilisez les mêmes scénarios. Si l'intersection est vide, on renverra `null` (et on le documentera !).

1.4 Avec des exceptions

Pour permettre de manipuler les exceptions, nous allons intégrer des intervalles dont les bornes sont infinies. L'infini existe en java comme constante dans la classe `Double`, on y trouve les constantes `Double.POSITIVE_INFINITY` pour $+\infty$ et `Double.NEGATIVE_INFINITY` pour $-\infty$.

14. Ajouter deux tests où vous créez un intervalle dont une borne est infinie, l'un avec $+\infty$ et l'autre avec $-\infty$. Vous pouvez tester la longueur des intervalles et une valeur contenue.
15. Pour lever une exception, nous allons vérifier la longueur d'un intervalle dont les deux bornes sont à $+\infty$. Évidemment, une telle longueur ne peut pas être bien définie. Nous allons donc faire en sorte de lever une exception `ArithmeticException` lors de l'appel de la méthode `getLongueur` sur un intervalle dont la borne min est $+\infty$ ou la borne max est $-\infty$. Écrivez le test qui vérifie la levée d'exception (avec `assertThrows`, vous pouvez vous aider du cours) puis modifiez le code de production pour qu'il passe le test.
16. Nous avons changé d'avis : plutôt que de lever une simple exception `ArithmeticException`, nous allons utiliser notre propre exception `IntervalleException`. Deux stratégies s'offrent à nous, vous allez tester les deux.
    — La première consiste à faire hériter `IntervalleException` de `ArithmeticException`. C'est plutôt une bonne stratégie et vous noterez que cela ne pose pas de problème particulier.
    — La seconde consiste à faire hériter `IntervalleException` de `Exception` seulement. Dans ce cas, le compilateur ne considère pas qu'il y a des **`throws`** explicites dans les entêtes de méthodes et de test, il est donc nécessaire de les ajouter à la main ou d'encapsuler tous nos appels de **`try ... catch`**. Il serait absurde d'attraper ces exceptions que nous ne savons pas gérer, la bonne solution consiste donc à ajouter aux entêtes de fonctions les **`throws`** `IntervalleException`.

2 Renforcement : le conteneur

Si vous avez fini, vous devriez reprendre le sujet de TD et implémenter avec la procédure du *Test Driven Development* que nous venons de faire les conteneurs.

1. `https://fr.wikipedia.org/wiki/Java_hashCode()`
2. Idéalement, chaque scénario est l'objet d'un test individuel avec un nom explicite.