TD5 – Oracle et Diagnostic *(Jean-Marie Mottu)*

Faites un fork du projet suivant et récupérez le code dans IntelliJ :
https://univ-nantes.io/iut.info1.qd1.automatisationtests/butinfo1-qd1-td5

Partie 1 – Diagnostic

Considérons la classe ListeNotes qui manipule des notes d'étudiants.

Exercice 5.1. Diagnostic de ListeNotes – méthode `moyenne()`

Question 5.1.1. Implémentez ces cas de test (il vaut mieux ne pas utiliser de test paramétrique à cause d'un problème avec IntelliJ gênant les questions suivantes)

CT1(DT1({}), 0)
CT2(DT2({0}), 0)
CT3(DT3{0,0,0}), 0)
CT4(DT4{5}),5)
CT5(DT5{5,5},5)
CT6(DT6{5,10},7.5)

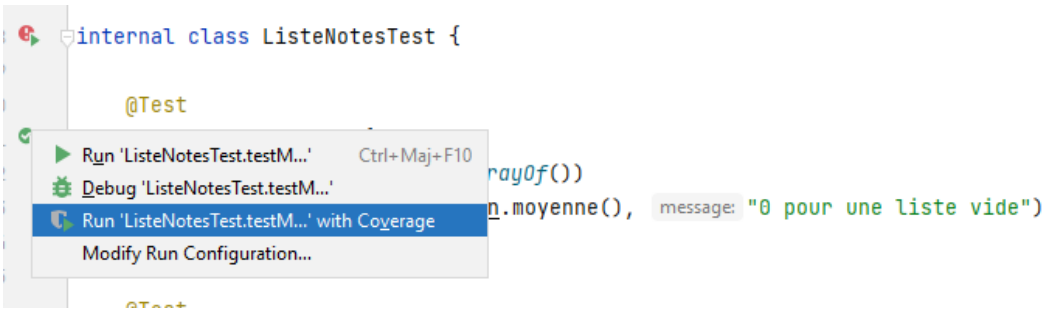
Question 5.1.2. Appliquez manuellement l'algorithme de Jones sur la méthode `moyenne`

Si l'algorithme ne permet pas d'ordonner sans égalité les lignes de code, on peut ajouter de nouveaux cas de test.

Reprenons cela avec l'aide d'IntelliJ.

Question 5.1.3. Exécutez les cas de test un à un en affichant la couverture de code dynamiquement

Pour cela, cliquez sur le « play » dans la marge au niveau de la signature de la méthode de test et faites « run … with Coverage »



Revenez dans la classe sous test et observez dans la marge de la méthode sous test : dans la marge (ou le code selon votre configuration) du vert et du rouge indiquent quelles instructions ont été ou n'ont pas été exécutées.

Question 5.1.4. Vérifiez dans vos matrices de diagnostic si les couvertures correspondent à ce que vous aviez fait manuellement.

Une fois que l'algorithme de diagnostic a classé les instructions dans l'ordre de leur potentialité d'être fausses ou faussement exécutées à cause d'un faux prédicat, il faut comprendre et corriger la faute.

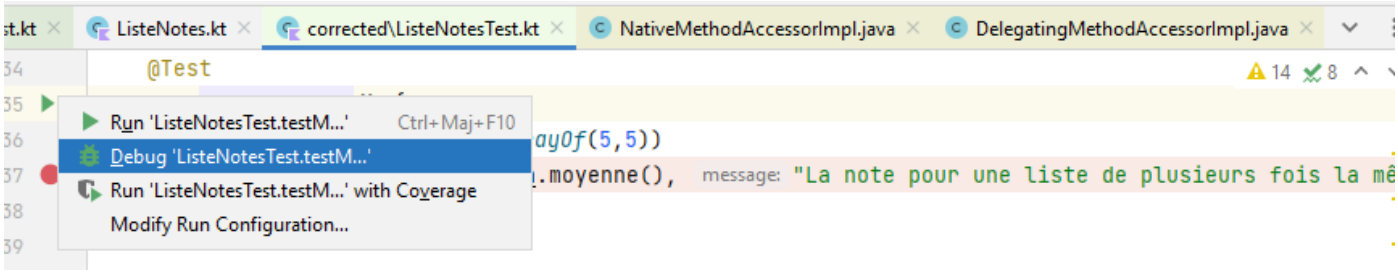
Pour cela, le mode debug de l'IDE est indispensable.
On évitera de décorer le code : le testeur n'a pas le droit de faire du temporaire au risque qu'il soit définitif (ou masque des fautes, ce qui est improbable avec des println, mais ils perturbent quand même les performances) :
https://www.francetvinfo.fr/decouverte/bizarre/un-radar-disait-fuck-you-aux-automobilistes-qui-roulaient-trop-vite_84555.html

Il s'agit d'une forme de traçabilité dynamique, pas à pas.

Question 5.1.5.

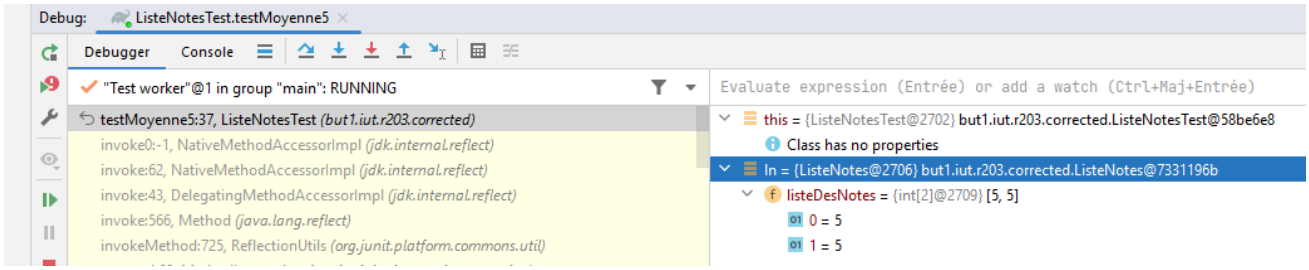
Commencez par ajouter un point d'arrêt dans le cas de test 3 de la fonction moyenne, à la ligne de l'appel à la méthode sous test :

Ce point d'arrêt est nécessaire pour dire où commencer le diagnostic.
Un clic dans la marge ajoute ce point rouge, puis lancez le test en « debug » :



Dans un onglet « debugger », vous obtenez la pile d'appel : tous les appels de méthodes qui sont imbriqués depuis le début de l'exécution du programme (un `main` qu'on n'a même pas lancé nous-même mais qui l'a été par le framework de test) jusqu'à la méthode de test où l'on a mis le point d'arrêt.

Dans l'onglet « evaluate expression », vous obtenez des informations sur l'état du système :



La classe n'a pas d'attribut (properties) mais la liste est bien créée et remplie.

Question 5.1.6.

Avancez pas à pas.

On peut avancer par instruction « step over », ou par appel « step into ».
Progressez jusqu'à trouver l'instruction fausse et constatez le problème au deuxième tour de boucle.
La lecture est difficile, un IDE comme IntelliJ montre même ses limites : une seule variable somme est listée alors que le bug est d'en affecter une nouvelle à chaque tour de boucle au lieu de mettre à jour celle de la fonction.

Question 5.1.7. Plutôt que du pas-à-pas, exploitez le résultat de l'algorithme de SBFL en plaçant des points d'arrêt aux instructions de la fonction moyenne les mieux classées (donc ayant plus de chances d'être fausses) :

Il faut en placer plusieurs :

- Un à l'instruction désignée première, et potentiellement fausse
- D'autres aux prédicats y menant et qui potentiellement sont faux en menant à cette instruction.

Question 5.1.8. Avancez pas à pas.

Question 5.1.9. Corrigez le code de manière à faire passer les tests.

Exercice 5.2. Diagnostic de ListeNotes – méthode `nombreOccurence()`

Question 5.2.1. Répétez les questions de l'Exercice 5.1 pour diagnostiquer la méthode `nombreOccurence ()` (sauf la Question 5.1.2)

Pour cela, implémentez ces cas de test :
CT1(DT1({},-5), IllegalArgumentException)
CT2(DT2({},50), IllegalArgumentException)
CT3(DT3{},5), 0})
CT4(DT4{5},5), 1)
CT5(DT5{5,5},5), 2)
CT6(DT6{5,0,5},5), 2)
CT7(DT7{0,5,0,5},5), 2)
CT8(DT8{0,5,0,5},6), 0)

Exercice 5.3. Diagnostic de Factorielle

Vous retrouvez dans ce code une classe `OperationsUnaires` avec la méthode que vous avez déjà testée : `factorielle`

Recréez une classe de test et mettez-y les tests que vous récupérerez de votre projet du TD2.
Si vous n'en avez pas, implémentez :

```
CT1(-1, IllegalArgumentException)
CT2(13, ArithmeticException)
CT3(0, 1)
CT4(3, 6)
```

Question 5.3.1. Répétez les questions de l'Exercice 5.1 pour diagnostiquer la méthode factorielle () (sauf la Question 5.1.2)

Deuxième partie –Oracle (subsidiaire)

Exercice 5.4. Oracle heuristique

Pour le moment vous avez conçu vos cas de test avec des oracles discrets que vous avez implémentés en contrôlant les valeurs exactes dans des assertions.

Question 5.4.1. Considérons les fonctions moyenne, nombreOccurence, pour chacune d'entre elle concevez un oracle heuristique.

Par exemple pour la fonction `factorielle`, un oracle heuristique serait factorielle(x) >=0 en exprimant une propriété sur n'importe quelle sortie. Ainsi pour n'importe quelle donnée de test, on peut utiliser cet oracle.

On peut aussi faire des oracles heuristiques qui lient entrée et sortie : une propriété sur une entrée implique une propriété sur la sortie. Dans le CM, nous avions : pour x entre 0 et PI, alors 0<=sin(x)<=1.

Question 5.4.2. Considérons factorielle, identifiez une relation métamorphique

Troisième partie – Développement et test collaboratif (subsidiaire)

Mettez-vous en binôme (un seul trinôme par groupe de TD)

Exercice 5.5. Exercice 5 – Développement

Question 5.5.1. Chaque binôme implémente sur son ordi, sans concertation une de ses deux méthodes :

- Un binôme : Quelle est la note maximale et combien d'élèves l'ont
- L'autre binôme : Quelle est la note minimale et combien d'élèves l'ont

Exercice 5.6. Exercice 6 – Tests fonctionnels de Min/Max

Les binômes développeurs prennent le rôle de testeur pour tester le code de l'autre.

Question 5.6.1. Chacun conçoit des tests fonctionnels pour la fonction qu'il teste :

Il faut un minimum de spécification pour tester ces méthodes :
Les notes sont des entiers. La note maximale étant 20 et la minimale 0.
La liste de notes n'est pas ordonnées, elle contient une liste de notes qu'on imagine entrées par le professeur au fur et à mesure des copies corrigées. Chaque note est donc la note d'un élève.

Question 5.6.2. Implémentez ces cas de tests.

Question 5.6.3. Exécutez ces cas de tests.

Si vous avez tous vos tests qui passent du premier coup, bravo.

Exercice 5.7. Exercice 7 – Diagnostic de Min et Max

Question 5.7.1. Effectuez le diagnostic et la correction de vos méthodes grâce à l'algorithme de Jones avec l'aide de l'IDE pour la couverture et le mode débug.

Question 5.7.2. Echangez vos codes et faites une revue de code statique du code de votre binôme. Vous vérifierez par exemple ces recommandations :

http://w4.uqo.ca/iglewski/ens/inf1583/Inspection/insp_code1.html

https://gitlab.univ-nantes.fr/naomod/testing/labs/-/blob/master/revue-code/regles-codage-chess.adoc

https://kotlinlang.org/docs/coding-conventions.html#naming-rules