

Développement informatique avancé orienté application - TP

Spécifications, tests unitaires et débbuging

Virginie Van den Schrieck

Ce TP se base sur les classes réalisées pendant les TPs précédents. Si vous n'avez pas fini ce dernier, mettez-vous en ordre au plus vite, et soyez sûrs de rattraper votre retard avant la prochaine séance !

1 Spécifications et tests unitaires

Pour produire un logiciel de bonne qualité, il est indispensable de tester le code produit. Les tests s'effectuent à différents niveaux : Niveau composant, niveau système, niveau utilisateur, ...(voir par exemple [http://fr.wikipedia.org/wiki/Test_\(informatique\)](http://fr.wikipedia.org/wiki/Test_(informatique))). Il existe de nombreux types de tests, et de nombreuses classifications. Nous allons ici nous intéresser aux tests unitaires (http://fr.wikipedia.org/wiki/Test_unitaire) : Des tests portants sur des petites portions de code, c'est-à-dire, dans le cas de Java, sur des méthodes.

Les tests unitaires sont notamment préconisés dans la méthode TDD : Test Driven Development (http://fr.wikipedia.org/wiki/Test_Driven_Development). Cette pratique consiste à écrire les tests d'une méthode AVANT d'en écrire le code. Concrètement, cela nécessite cinq étapes :

1. Ecrire un premier test
2. Vérifier qu'il échoue (car le code à tester n'existe pas encore) afin de s'assurer que le test est valide
3. Ecrire juste le code suffisant pour passer le test
4. Vérifier que le test passe
5. Refactoriser le code, à savoir l'améliorer en gardant les mêmes fonctionnalités. Il faut évidemment s'assurer que le test passe toujours.

Pour pouvoir écrire un test avant d'avoir écrit la méthode, il est important d'avoir bien documenté la méthode :

- Quels sont les paramètres et les conditions sur ces derniers ?
- Quel est le résultat attendu ?

Ces informations doivent figurer dans la javadoc, qui est en fait la première chose à produire lorsqu'on conçoit une méthode.

Pour les curieux : Ces principes sont repris de la programmation par contrat, que vous pouvez découvrir ici : http://fr.wikipedia.org/wiki/Programmation_par_contrat

1.1 Exercices

Pour écrire les spécifications javadoc d'une méthode, on doit donc décrire ce que sont les paramètres à transmettre en input à la méthode. Par exemple, pour une méthode qui calcule le carré d'un nombre, nous avons besoin d'un paramètre. Nous allons par facilité nous restreindre aux nombres entiers. On peut calculer le carré de n'importe quel nombre entier, positif, négatif ou nul. Il n'y a donc pas de restriction sur la valeur de ce paramètre. Au niveau de la valeur de retour, la méthode renvoie donc le carré d'un nombre, c'est-à-dire le résultat de sa multiplication par lui-même. Le résultat est toujours positif. La spécification Javadoc résultant de cette réflexion est la suivante :

```
/**
 * Cette méthode calcule le carré d'un nombre.
 * @param n : Un nombre entier quelconque
 * @return Un nombre >= 0, résultat de la multiplication de n par lui-même
 */
public int carre(int n){
    //TODO : A implementer
    return 0;
}
```

Sur papier, écrivez la spécification (préconditions, postconditions + signature) des méthodes suivantes :

- Vérifier si un nombre est pair
- Calculer le périmètre d'un carré sur base de la longueur de son côté
- Vérifier si un caractère donné apparaît dans une chaîne de caractères

1.2 Valeurs de test

Toujours sur papier, pour chacune des méthodes de l'exercice précédent, proposez un jeu de valeurs de tests qui permet de tester si la méthode respecte sa spécification, ainsi que les valeurs de retour attendue dans chaque cas. Les valeurs doivent respecter les conditions posées sur les paramètres dans la javadoc.

Par exemple, pour la méthode `carré`, une série de tests pourrait être la suivante :

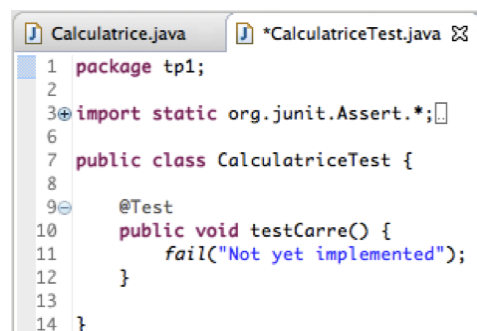
Valeurs des inputs	Output attendu
0	0
1	1
-1	1
-2	4
2	4

Remarquez que nous testons le cas limite 0, et que nous prenons des valeurs de test dans les nombres positifs et dans les nombres négatifs, afin de couvrir un maximum de cas possibles.

1.3 Découvrir JUnit

Une fois les valeurs de tests définies, il est temps de passer à l'implémentation des tests, PUIS de la méthode. Prenons la méthode qui calcule le carré d'un nombre comme exemple. Nous avons créé la méthode dans une classe appelée `Calculatrice`, en spécifiant la signature, la Javadoc, et en renvoyant une valeur arbitraire pour que la méthode puisse être exécutée même si elle n'est pas encore implémentée.

Pour pouvoir la tester avec les valeurs indiquées à l'exercice précédent, nous allons utiliser l'outil JUnit, qui permet de tester facilement des méthodes Java. Eclipse va encore une fois nous faciliter la vie pour cela. Cliquez sur la classe `Calculatrice` dans le Package Explorer, et choisissez `New / JUnit Test Case`. Vérifiez qu'Eclipse propose bien de tester la classe `Calculatrice`, et laissez les autres options aux valeurs par défaut. Cliquez sur `Next`, et observez que vous pouvez sélectionner les méthodes à tester. Appuyez ensuite sur `Finish`. Vous obtenez alors la classe `CalculatriceTest` telle que ci-dessous :

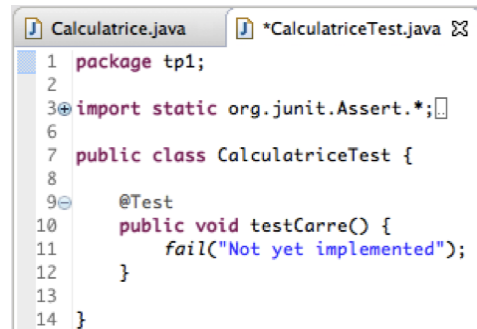


```
1 package tp1;
2
3 import static org.junit.Assert.*;
4
5
6
7 public class CalculatriceTest {
8
9     @Test
10     public void testCarre() {
11         fail("Not yet implemented");
12     }
13
14 }
```

FIGURE 1 – Création d'un test unitaire

Il faut à présent implémenter le test, avec les valeurs identifiées plus haut. On va utiliser pour cela la méthode `assertEquals`, qui prend deux

paramètres : Le résultat attendu, puis le résultat calculé (attention à ne pas inverser les deux). Cela donne dans notre cas :



```
1 package tp1;
2
3 import static org.junit.Assert.*;
4
5
6
7 public class CalculatriceTest {
8
9     @Test
10    public void testCarre() {
11        fail("Not yet implemented");
12    }
13
14 }
```

FIGURE 2 – Implémentation des tests unitaires

Pour les curieux : JUnit offre d'autres méthodes de vérification qu'assertEquals. Vous pouvez les trouver ici : <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

Ensuite, nous allons exécuter le test, en indiquant à Eclipse qu'il s'agit d'un test unitaire :

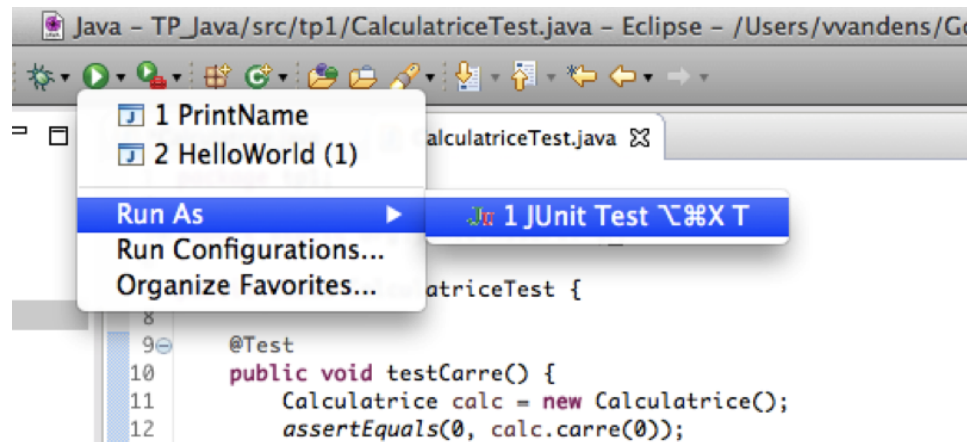


FIGURE 3 – Exécution des tests unitaires

La View JUnit s'ouvre, indiquant l'échec du test. Il nous indique qu'il a obtenu la valeur 0 en lieu et place de la valeur 1. C'est normal, vu que nous n'avons pas encore implémenté la logique de la méthode carré (voir figure).

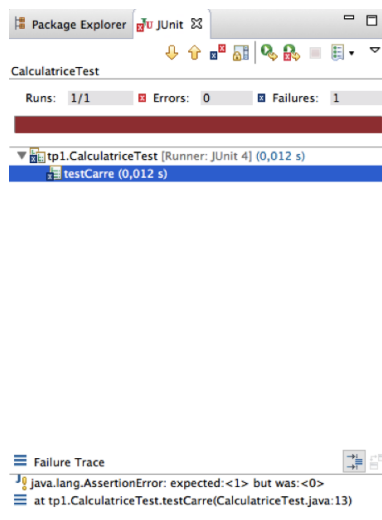


FIGURE 4 – Echec initial du test non implémenté

Nous implémentons à présent la méthode :

```
/**
 * Cette méthode calcule le carré d'un nombre.
 * @param n : Un nombre entier quelconque
 * @return Un nombre  $\geq 0$ , résultat de la multiplication de n par lui-même
 */
public int carre(int n){
    return n*n;
}
```

Et cette fois, lorsque nous exécutons la classe de test, nous obtenons :

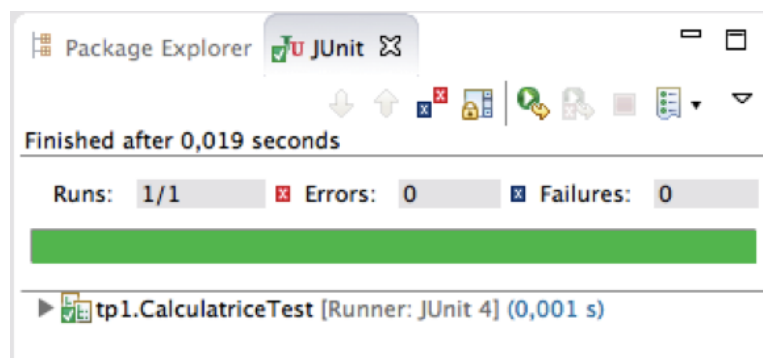


FIGURE 5 – Réussite du test

1.4 Ecrire des tests

Vous allez à présent reprendre les spécifications des méthodes des deux premiers exercices (`estPair`, `perimetreCarre` et `contientCaractere`), et les implémenter suivant la méthode présentée. Vous allez donc :

1. Créer la classe dans laquelle sera utilisée la méthode (on peut ici exceptionnellement se simplifier la vie et créer une classe `ExerciceTDD` dans laquelle seront implémentées les trois méthodes. Cela n'a rien d'orienté-objet, mais ce n'est pas le but de l'exercice.)
2. Ecrire la signature et la javadoc de chaque méthode, SANS écrire les instructions qui permettront de l'implémenter.
3. Créer une classe `JUnit ExerciceTDDTest`, qui contiendra les tests des trois méthodes
4. Implémenter les tests et vérifier qu'ils échouent bien
5. Implémenter les trois méthodes et les valider.

Remarque : A partir de maintenant, il vous est demandé, pour toutes les méthodes non-triviales que vous écrivez (c'est-à-dire celles avec un minimum de logique algorithmique), de systématiquement utiliser cette démarche et donc de produire la spécification javadoc et les tests JUnit correspondants.

2 Debugging Eclipse

Eclipse nous offre un autre outil bien utile : Le debugger. Un debugger est un environnement d'exécution dans lequel il est possible d'interrompre l'exécution à n'importe quel endroit pour visualiser l'état du programme et les valeurs des variables à cet instant précis (voir <http://fr.wikipedia.org/wiki/Debogueur>) Les outils de débogage d'Eclipse sont regroupés dans la perspective Debug.

Au centre, nous avons la View affichant le code source, et la View outline affichant la structure de nos classes. En bas, nous avons bien sûr la View console. Les nouvelles View qui vous nous intéresser principalement sont celles du haut : la View Debug et la zone avec les deux Views Variables et Breakpoints. La View Debug montre à quel endroit l'exécution est suspendue (numéro de ligne, thread, ...). La View Variables permet d'inspecter les valeurs courantes en mémoire, et la View Breakpoints liste les points d'arrêts qui ont été configuré. Un point d'arrêt se configure simplement en faisant un clic droit sur le numéro de la ligne sur laquelle on veut interrompre l'exécution, et en sélectionnant `Toggle Breakpoint`. Un breakpoint se supprime de la même manière. On peut également activer/désactiver temporairement les breakpoints depuis la View Breakpoints.

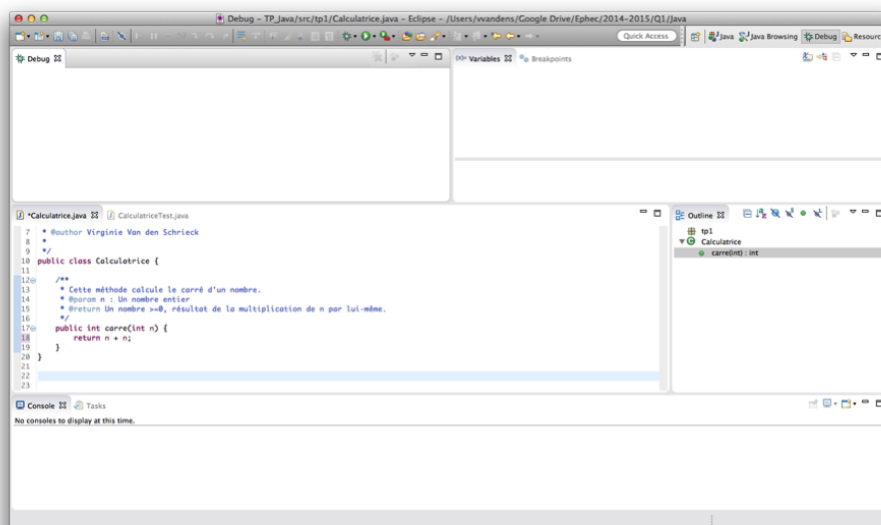


FIGURE 6 – Perspective Debug dans Eclipse



FIGURE 7 – Breakpoint Eclipse

Au niveau de l'exécution du programme, tant qu'aucun breakpoint n'est défini, le déroulement est le même que dans l'environnement normal. Par contre, si on définit des breakpoints et qu'on exécute le programme en utilisant le bouton « Debug as », le programme va s'exécuter uniquement jusqu'au premier breakpoint. On peut ensuite continuer l'exécution « pas à pas » en utilisant les commandes appropriées.

Enfin, pour inspecter la valeur des variables à un moment précis de l'exécution, on utilise la View Variables. Combinée avec l'exécution pas-à-pas,

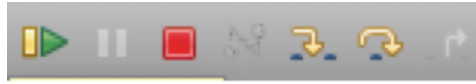


FIGURE 8 – Boutons de commande de l'exécution en mode debug (aller au prochain breakpoint, pause, stop, exécuter la ligne courante, terminer l'exécution de la méthode courante)

c'est un outil très puissant pour suivre l'exécution d'une portion de code.

2.1 Débugger une méthode

Pour tester cet outil de débugging, nous allons utiliser une classe StatToolBox, qui est disponible sur le Campus Virtuel. Prenez le temps d'en lire la documentation (en lisant les commentaires Javadoc ou en générant la documentation HMTL). Nous nous intéresserons à la méthode `moyenne()`.

1. Proposez une série de tests unitaires pour tester la méthode `moyenne()`
2. Implémentez ces tests en JUnit. Que constatez-vous ?
3. Utilisez l'outil de débugging pour exécuter la méthode pas à pas. Observez à chaque étape les valeurs des variables utilisées. Essayez de retrouver l'erreur en utilisant cette technique d'examen systématique de l'évolution des valeurs des variables.

3 Exercice supplémentaire

Créez une classe `Etudiant`. Chaque étudiant a une date de naissance (créez une classe pour les dates), un nom, un prénom, un numéro de matricule et un tableau avec les résultats à 5 examens. Spécifiez les constructeurs, la méthode `toString()` et une méthode `moyenne` qui renvoie la moyenne des résultats. Créez ensuite la classe de tests unitaires permettant de tester les méthodes spécifiées. Lorsque les tests échouent adéquatement, implémentez les méthodes conformément aux spécifications.