TD 5 - Tests et gestion de version pour vos développements Java

Qualité de développement

BUT 1 - IUT

T. Dufaud (thomas.dufaud@uvsq.fr), I. Robba

Ce TD a pour but de vous faire appliquer la gestion de version ainsi que l'activité de tests dans le cadre de vos développements en Java. Nous travaillons sous Windows, avec IntelliJ IDEA, JUnit et git bash.
À rendre : un lien vers votre dépôt git sur bitbucket.

---

1 Environnement de travail et exécution automatique avec JUnit

1.1 JUnit

Nous avons choisi des unités à tester et défini l'ensemble des données d'entrée et de sortie puis nous avons préparé les tests et défini des données de test. En suivant le processus de test illustré Figure 1.1, on s'intéresse maintenant à l'exécution du test et à la production des résultats des tests.

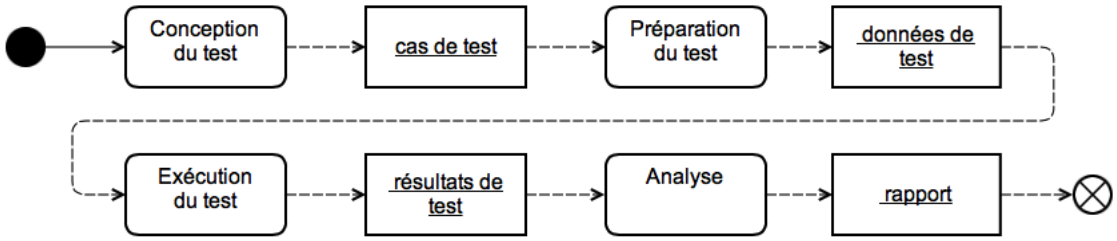

Figure 1 – Diagramme d'activité du processus de test

Afin d'améliorer la procédure d'exécution des tests, on peut utiliser des outils logiciels dédiés. Nous nous intéressons ici à JUnit.

JUnit est un framework permettant d'écrire des suites de tests unitaires en Java de façon simple et systématique. Il permet également de piloter la mise en oeuvre des tests.

Il repose sur les principes suivants :
— Une classe de tests unitaires est associée à une classe à tester.
— Une ou plusieurs méthodes de tests sont associées aux méthodes de la classe à tester.
— Une classe de tests unitaires hérite de la classe junit.framework.TestCase

Nous utilisons la version 5 de JUnit intégrée à IntelliJ. Les méthodes et les instructions de tests sont identifiées par des annotations Java tout comme pour la javadoc (Voir table 1). Ces annotations sont placées avant les méthodes de tests.

Annotation	**Description**
@Test	méthode de test (méthode effectuant le test de l'unité)
@Before	méthode exécutée avant chaque test
@After	méthode exécutée après chaque test
@Ignore	méthode qui ne sera pas lancée comme test

Table 1 – Exemple d'annotations pour JUnit

Une méthode de test peut avoir un nom quelconque mais doit :
— avoir une visibilité publique
— avoir un type de retour void
— n'avoir aucun paramètre
— être annotée @Test
— utilisée des instructions de test (Voir Table 2)

Annotation	**Description**
fail(String)	fait échouer la méthode de test et renvoie un message
assertTrue(boolean condition)	vérifie que la condition booléenne est vraie
assertsEquals(expected, actual)	teste si les valeurs sont les mêmes
assertsEquals(expected, actual, tolerance)	teste la proximité des valeurs à une tolérance donnée
assertNull(object)	vérifie si l'objet est null
assertNotNull(object)	vérifie si l'objet n'est pas null
assertSame(expected, actual)	vérifie si les variables référencent le même objet
assertNotSame(expected, actual)	vérifie si les variables ne référencent pas le même objet

Table 2 – Instructions de test pour JUnit

1.2 Dépôt git distant

▸ **Exercice 1.** *[Création d'un dépôt git]*

1. *Créez un dépôt git privé sur BitBucket pour versionner vos sources de TP Java sur le* ***développement du Calendrier****.*
2. *Partagez ce dépôt avec votre enseignant.*

2 Développement de tests unitaires avec JUnit

Dans cette section, vous développerez les tests unitaires de quelques méthodes de la classe *Date*.

▸ **Exercice 2.** *En appliquant la méthode de partition d'équivalence, concevez les tests pour la méthode dernierJourDuMois() de la classe Date. Quelle méthode serait-il alors logique de tester auparavant ?*

.

▸ **Exercice 3.** *On met en place l'environnement de test JUnit dans IntelliJ. Pour cela on suit la documentation proposée par IntelliJ au lien suivant :* `https://www.jetbrains.com/help/idea/junit.html`, *notamment la section Create tests*

1. *Choisissez l'unité à tester*
2. *Écrivez le test pour chaque classe de la partition d'équivalence avec JUnit*
3. *Exécutez le test et lire le rapport de test*

▸ **Exercice 4.** *Versionnez votre travail en accompagnant chaque commit d'un message précis.*

▸ **Exercice 5.** *En versionnant au fur et à mesure, refaites les exercices 2, 3 et 4 sur les méthodes :*

1. *Date() le constructeur sans paramètre qui retourne la date courante*
2. *dateDuLendemain()*
3. *dateDeLaVeille()*
4. *compareTo()*
5. *Placer une instruction System.out.println("Test méthode ...") en début de chaque méthode de test, puis exécutez à nouevau les tests. Qu'observez-vous ?*

▸ **Exercice 6.** *Dans l'environnement de test JUnit, il est possible d'ordonner les tests. Dans notre cas, par exemple, si le test de la méthode dernierJourDuMois() échoue, l'erreur peut venir soit de la méthode elle-même soit de la méthode estBissextile(), il serait donc pertinent d'ordonner ces 2 tests. Les tests peuvent être ordonnés de manière ascendante (bottom-up), c'est-à-dire de la méthode de plus bas niveau vers celle de plus haut niveau ; ou à l'inverse de manière descendante (top-down), de la méthode de plus haut niveau vers celle de plus bas niveau.*

Pour ordonner les tests, il faut modifier l'en-tête de la classe de la façon suivante :

@TestMethodOrder(MethodOrderer.OrderAnnotation.class) public class DateTest {...

Puis après chaque annotation @Test, on ajoute l'ordre d'exécution avec l'annotation @Order (i), i étant le numéro d'ordre du test

1. *De quelle manière (ascendante ou descendante) préférez-vous effectuer les tests ? Pour quelle raison ?*
2. *Ordonnez vos méthodes de test, puis re-exécutez vos tests*
3. *Plusieurs tests peuvent-il porter le même numéro d'ordre ?*