Genie Logiciel (GEN) / Labo 2 - Tests unitaires - JUnit

Travail par groupe de 2-3 étudiants
Le travail sera évalué et valorisé pour 5% de la note de labo
Délai de remise : mercredi 17 mars à minuit par email. Démonstration en séance le jeudi 18 mars.
Lorsque vous avez terminé, envoyez-moi un email précisant les noms des membres de votre groupe et l'adresse de votre repository GitHub. Votre repo doit être privé et vous devez m'en donner l'accès (mon user github : patricklac).

Introduction

Un test unitaire est une méthode d'une classe dédiée aux tests qui permet de s'assurer du fonctionnement correct d'une fonctionnalité ou d'une partie d'une fonctionnalité d'une application. Il crée des objets à partir de classes de cette application, appelle des méthodes sur ceux-ci et vérifie les résultats obtenus.

L'ensemble des tests unitaires (une suite de tests) d'une application est conservé et ré-exécuté à chaque modification de celle-ci de manière à vérifier que cette modification n'amène pas de bugs dans les fonctionnalités qui tournaient auparavant (tests de non-régression).

Le framework JUnit permet d'effectuer ces tests dans l'environnement de programmation Java sous formes de librairies à importer dans le projet. L'import de librairies en java est plus facile à gérer avec un outil d'aide au développement de projet comme Maven qui sera utilisé dans ce laboratoire. Maven et JUnit sont pris en compte par IntelliJ qui facilite leur utilisation.

Mise en place de projet avec modules Maven sous IntelliJ

Dans ce labo, vous aurez à rendre deux exercices sur un seul repo privé github : deuxentiers et diary. Le plus simple est d'en faire 2 modules IntelliJ du même projet labo2 que vous associerez à votre repo. Pour cela créez d'abord un projet labo2 vide (New Project -> Empty Project) dans lequel vous pouvez déjà mettre un fichier .gitignore .

Maven est un outil de construction de projet de la fondation Apache qui permet entre autres :

- L'automatisation de taches (cibles) : compilation, test, déploiement, …
  Par exemple, dans le dossier du projet : mvn test
- La gestion des dépendances : les bibliothèques de classe externes seront automatiquement importées depuis internet à partir d'une déclaration en xml.

Pour cela Maven s'appuie sur :

- Un fichier de configuration : pom.xml (Project Object Model) à la racine du module
- Une arborescence de dossier standardisée :
  - src : sources, à écrire par le développeur
    - main : l'application

- java: classes organisées en packages
- resources: fichiers de configurations, de données, ...

▪ test : avec aussi des dossiers java et resources mais pour les tests

○ target: exécutables et fichiers pour l'application et les tests, produits par maven.

Créez un module Maven (new module -> Maven sous intellij), vous devez définir les paramètres suivants qui seront enregistrés dans le fichier pom.xml :

- Archétype : c'est le choix d'un type de projet. On utilisera dans ce labo le type de base par défaut (pas d'archétype)
- GroupId: nom de votre structure qui s'apparente à un nom internet écrit de gauche à droite (par exemple org.apache.maven pour le projet maven). A titre d'exemple, j'utilise le groupe ch.heigvd.pl .
- ArtifactId: nom du projet Maven, correspondant au module IntellIJ. Avec le GroupID, il sert à identifier les ressources Maven (archétypes, librairies, ...) publiées sur Internet, en vue d'être réutilisées dans de développement d'autres projets utilisant Maven. Choisissez deuxentiers pour le premier exercice.
- Version : par défaut, à chaque fois que l'on régénère son projet, le no de version est incrémenté automatiquement d'une unité. En phase de développement, on préfère évidemment que le no de version soit stable et ne soit pas incrémenté : il suffit de donner à cette version la valeur "1.0-SNAPSHOT".

Note : Le "GroupID/ArtifactId" n'est pas utilisé dans le module lui-même. Il ne serait utilisé par exemple que si vous publiez vous-même des libraires issues de ce projet dans un dépôt Maven, ce que l'on ne fera pas dans ce labo. Cependant, il est d'usage d'organiser ensuite vos packages sources avec une racine correspondant à ce "GroupID/ArtifactId". Par exemple pour le groupe ch.heigvd.pl et le projet deuxentiers, on choisirait le package racine ch.heigvd.pl.deuxentiers.

A la création du projet, Maven se connecte au dépôt central http://repo.maven.apache.org/ et télécharge un grand nombre de fichiers POM (Project Object Model) et JAR (librairies de classes Java) qui lui seront utiles de manière générale. Il est conseillé de laisser Maven importer tous les fichiers dont il aura besoin sans intervention en validant la proposition d'IntelliJ "Maven -> enable auto-import" qui vous est proposée à ce moment-là.

Par défaut, tous ces fichiers sont stockés dans votre dossier utilisateur dans un sous-dossier nommé ".m2". Pour les prochaines utilisations, Maven n'aura plus besoin de télécharger ces fichiers.

Premier test, classe DeuxEntiersTest

Saisissez le code suivant dans le package correspondant du dossier test/java, en l'ajustant à votre nom de package :

```
package ch.heigvd.pl.deuxentiers;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class DeuxEntiersTest {
    @Test
    void divise() {
        DeuxEntiers septTrois = new DeuxEntiers(7,3);
```

```
            assertEquals(2,septTrois.divise());
        }
    }
```

Il vous fera apparaître des erreurs car les librairies JUnit ne sont encore pas importées et la classe DeuxEntiers n'existe pas. Pour les résoudre, utilisez les suggestions d'IntelliJ (Lampe rouge) :

- "Add JUnit5.2 to ClassPath" (ou la dernière version JUnit disponible)
- "Create Class DeuxEntiers" en changeant la "Target destination directory" à "src/**main**/java/..." puis "create constructor" et "create method divise in DeuxEntiers" dans laquelle on ajoutera un "return 0" pour qu'elle compile.

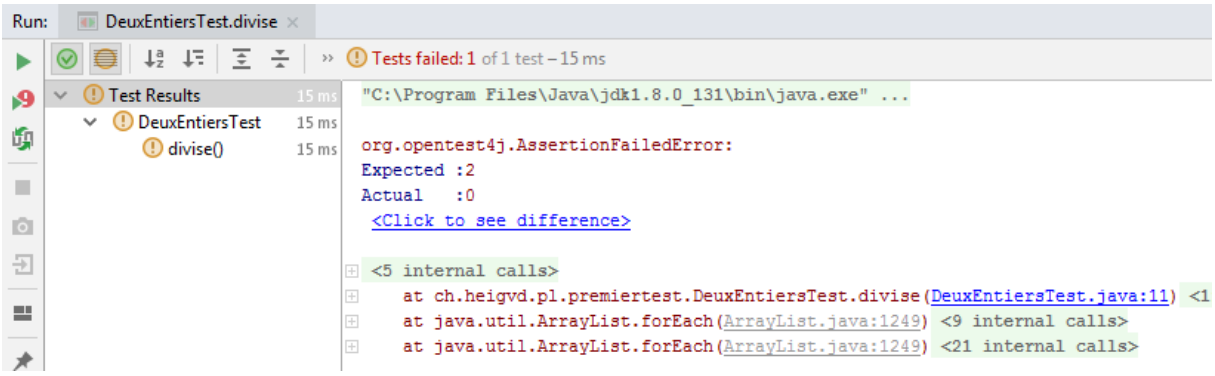
La première suggestion aura eu pour effet :

- D'ajouter la dépendance à JUnit dans votre pom.xml :

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>RELEASE</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

- D'importer les librairies correspondantes dans votre dossier ".m2" si cela n'avait jamais été fait.
- De faire apparaître ces librairies dans IntelliJ lorsque vous dépliez "External Librairies" de votre onglet projet.

Votre test est maintenant exécutable. Il produit le résultat suivant :



Il échoue car votre instruction assertEquals a comparé le résultat logique attendu "2" avec celui obtenu "0" : le test apparaît avec un icone d'avertissement marron. C'est normal, la classe DeuxEntiers n'est pas finie.

Complétez la méthode divise de DeuxEntiers et ré-exécutez le test : il réussit et il est maintenant marqué par IntelliJ avec une coche de validation verte.

Deuxième test, ajout d'un setUp

Créez une nouvelle méthode de test **modulo()** dans la classe DeuxEntiersTest. Avec Intellij, vous pouvez générer son entête (Alt-Insert Test Method). Cette méthode est comparable à divise() sauf

que vous devez espérer que l'appel de reste revoie 1 (7 % 3). Générez la méthode modulo() dans la classe DeuxEntiers en la laissant renvoyer 0 pour l'instant.

Le fait d'avoir à répéter la création de l'objet septTrois peut sembler lourde et l'on serait tenté de le faire dans un constructeur. Mais JUnit a pour principe d'exécuter chaque méthode de test dans un environnement indépendant. Dans une méthode de test, on ne doit pas tenir compte de ce qui a été fait dans une autre. Cela permet à JUnit de les exécuter dans n'importe quel ordre.

De plus, chaque test doit pouvoir être ré-exécuté à volonté pendant le développement. Cela impose de créer l'environnement complet du test (toutes les données dont il a besoin) au début de celui-ci. On ne doit pas non plus compter sur des données qui auraient été créées avec d'autres fonctionnalités de l'application. Cela peut amener une initialisation assez conséquente.

Si cette initialisation est commune à plusieurs méthodes de test, on peut en partager le code en définissant une méthode annotée @BeforeEach (alt-insert SetUp method) dans laquelle on factorisera la création de setTrois qui devra alors être déclaré comme variable d'instance. Ce code sera néanmoins ré-exécuté avant chaque activation de méthode de test, ce qui serait nécessaire si l'objet septTrois devait être modifié. Ce n'est pas le cas ici et on pourrait donc demander sa création unique avec une autre annotation JUnit5, @BeforeAll, et déclarer setTrois comme variable statique car JUnit5 crée un objet DeuxEntiersTest par méthode de test. JUnit5 définit aussi des annotations correspondantes pour la terminaison des méthodes de test (@AfterEach, @AfterAll).

Ne complétez pas encore la méthode modulo de DeuxEntiers et exécutez la classe de test complète (Run test sur l'entête de classe). Vous pourrez constater que les 2 tests sont exécutés, que divise est marqué avec une coche verte et modulo avec un avertissement marron. La classe de test est elle-même marquée avec un avertissement marron ainsi que "Test Results" (l'ensemble de tous les tests). Un seul test suffit à faire échouer une suite de test. Cela permet de savoir d'un seul coup d'œil si tout est en ordre ou non.

TDD, CI et Maven

Vous avez remarqué que l'on a pu écrire et exécuter la classe de test avant d'écrire la classe à tester, en ne générant que le minimum de celle-ci. Cette façon de faire est conforme à une approche appelée **TDD** pour "Test Driven Design" (**développement piloté pas les tests**), très utilisée dans les méthodes agiles qui seront étudiées dans le cours.

Une autre possibilité est de faire tourner les tests sous le contrôle de Maven et non plus d'IntelliJ. Cela ouvre la possibilité de les faire exécuter de façon automatique en arrière-plan et surtout sur un serveur partagé en liaison avec un serveur de versions de type GIT. On pourra ainsi demander l'exécution des tests à chaque commit. C'est une des étapes d'une pratique également fréquemment mise en œuvre dans les méthodes agiles appelée **CI** pour "Continuous Integration" (**intégration continue**).

Cependant, avec la version 5 de JUnit, la configuration de Maven pour l'exécution des tests est devenue un peu plus compliquée. Il faut inclure la dépendance à un moteur (engine) JUnit et non pas seulement à une api. De plus, bien que la documentation JUnit5 semble indiquer que cela devrait être implicite, il faut demander explicitement l'inclusion du moteur sous la forme d''une extension (plugin) pour le moteur surfire (maven-surfire-plugin).

Dans l'état des versions à ce jour, cela demande la configuration suivante dans le pom.xml :

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.4.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
<build>
    <finalName>maven-unit-test</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M3</version>
        </plugin>
    </plugins>
</build>
```

Vous pouvez alors lancer les tests avec la commande "mvn test" dans le dossier du projet (par exemple dans l'onglet "Terminal" d'IntelliJ) ou en ouvrant l'onglet "Maven Project" et en activant "test" sous "Lifecycle". La sortie produite à la console se présente différemment mais elle est équivalente à celle d'IntellJ.

L'exécution des tests produit également un état sous la forme d'un fichier (en version texte et xml) que vous trouverez dans le dossier surefire-reports du dossier target. Il contient un récapitulatif du nombre de tests exécutés, du nombre d'échec (Failures) ainsi que les traces d'échec.

```
-------------------------------------------------------------------------------
Test set: ch.heigvd.pl.deuxentiers.DeuxEntiersTest
-------------------------------------------------------------------------------
Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.016 s <<<
FAILURE! - in ch.heigvd.pl.deuxentiers.DeuxEntiersTest
divise  Time elapsed: 0.016 s  <<< FAILURE!
org.opentest4j.AssertionFailedError: expected: <2> but was: <0>
   at ch.heigvd.pl.deuxentiers.DeuxEntiersTest.divise(DeuxEntiersTest.java:21)
```

Un dernier test, les assertions et les exceptions

Il existe de nombreuses méthodes statiques dans la classe Assertions de JUnit pour tester ou comparer différents types de données :

http://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html

En voici quelques-unes parmi les plus usitées :

- assertEquals : teste si deux types primitifs sont égaux ou si 2 objets sont égaux au sens de la méthode equals.
- assertSame : teste si 2 variables ont la même référence sur un objet
- assertFalse et assertTrue : pour une condition booléeene
- assertThrows : pour savoir si une méthode a lancé une exception donnée.

- fail : permet de faire échouer sans conditions. A utiliser dans le cas d'une logique de comparaison plus complexe que celles offertes dans Assertions., comme par exemple rechercher la présence d'une valeur dans une collection. Si elle n'y est pas : fail.

Un test peut être **positif ou négatif**. Dans les 2 cas, il peut échouer ou réussir :

- un test positif vérifie une fonctionnalité en lui fournissant des données valides
- un test négatif vérifie que lorsqu'on lui fournit des données invalides, l'application réagit correctement : renvoi d'un code d'erreur ou d'une exception, ne fait pas le travail attendu, n'écrase pas de données ...

Dans la pratique, les tests négatifs sont plus sensibles que les tests positifs. Ils correspondent en général à des logiques de code sur lesquelles on passe plus rarement en exploitation. C'est la responsabilité du testeur d'anticiper les problèmes qu'ils risquent de poser un jour.

Créez une méthode de test diviseParZéro qui crée un objet septZéro avec 2 entiers 7 et 0. Le résultat est indéterminé mais supposons que l'on s'attende à obtenir 0 et qu'on le vérifie par une assertion.

En fait, on n'atteint pas l'assertion en exécution car la division par zéro lance une exception. La méthode de test est marquée avec un tiret rouge : c'est une erreur (error) et non pas une failure, mais c'est au moins aussi grave. Elles sont également clairement distinguées dans les rapports de test (les dernières lignes dans le rapport suivant):

```
-------------------------------------------------------------------------------
Test set: ch.heigvd.pl.deuxentiers.DeuxEntiersTest
-------------------------------------------------------------------------------
Tests run: 3, Failures: 1, Errors: 1, Skipped: 0, Time elapsed: 0.022 s <<<
FAILURE! - in ch.heigvd.pl.deuxentiers.DeuxEntiersTest
modulo  Time elapsed: 0.007 s  <<< FAILURE!
org.opentest4j.AssertionFailedError: expected: <1> but was: <0>
        at ch.heigvd.pl.deuxentiers.DeuxEntiersTest.modulo(DeuxEntiersTest.java:26)

diviseParZéro  Time elapsed: 0.001 s  <<< ERROR!
java.lang.ArithmeticException: / by zero
        at
ch.heigvd.pl.deuxentiers.DeuxEntiersTest.diviseParZéro(DeuxEntiersTest.java:32)
```

Dans ce cas, on doit vérifier que la méthode lance effectivement cette exception. Cela peut être fait avec JUnit5 par une assertion assertThrows. Le premier paramètre est le nom de la classe de l'exception attendue (ArtihmeticException.class). Le deuxième paramètre est prévu pour que vous puissiez appeler la méthode à tester dans un lambda mais il faudra ajuster la version du compilateur à java 8 en définissant explicitement « maven-compiler-plugin » dans le pom.xml : laissez faire IntelliJ lorsqu'il vous le suggère.

Vérifiez que le test passe avec cette assertion mais que si vous remettez "return 0" dans la méthode divise de DeuxEntiers, ce test échoue avec une failure car il attendait une exception.

Exercice Diary

Créez un 2ème module Maven « diary » dans votre projet IntelliJ labo2. Importez-y les classes de l'archive diary.zip fournie dans votre package sous main/java.

Ce projet est adapté d'un projet des compléments du livre Conception objet en Java avec BlueJ - une approche interactive Quatrième édition David J. Barnes et Michael Kolling Pearson Education, 2008.

Il simule un agenda avec des rendez-vous (Appointment) à poser dans une journée (Day). Pour utiliser ce projet, créez un objet Day, un ou deux objets Appointment. Utilisez la méthode makeAppointment pour poser les rendez-vous dans la journée Visualisez les rendez-vous de la journée avec la méthode showAppointments

Créez une classe de test pour la classe Day dans lesquels vous créerez des méthodes de test pour vérifier les fonctionnements décrits ci-après. Dans les tests, on vérifiera automatiquement les valeurs de retour des poses de rendez-vous ou de recherche de place.

1) Tests avec des rendez-vous d'une heure :
   - Pose de rendez-vous à 9h, 13h et 17h
   - Tentative de pose de rendez-vous à 8h et à 18h
   - Recherche d'une place libre (findSpace) pour un rendez-vous dans une journée qui contient un rendez-vous à 9h
   - Recherche d'une place libre dans une journée pleine
2) Tests avec des rendez-vous de 2 heures Concevez et implémentez des tests qui mettent en évidence le mauvais fonctionnement du projet avec des rendez-vous de 2 heures. Un des tests devra échouer avec une exception (error) et un autre avec une vérification incorrecte (failure). Il n'est pas demandé de corriger l'application.