1. Rappels généraux

Chaque semaine vous allez devoir rendre une partie de votre travail sur Moodle. Ça sera généralement la partie centrale d'un projet. Ces documents seront évalués dans le cadre de la note de TP. Ainsi, il n'y aura pas un TP noté final, mais de multiples travaux à faire dans le cadre d'un contrôle continu. Les TP ne doivent pas être fait à la maison, seulement pendant les séances, en présence d'un enseignant.

Ce système de notation repose sur l'honnêteté collective. Il vous est interdit de copier des sources entre vous. Si des tricheries sont constatées, on sera obligé de revenir à des devoirs sur table et des contrôles de TP sur table également. À vous de voir ce que vous préférez.

Cette semaine, on va travailler sur une étude de cas.

2. Étude de cas

Le but de cette étude est d'écrire les tests permettant de valider une bibliothèque *jar*, fournie sans ses sources. On ne dispose que de son cahier des charges.

Dans un premier temps, on construit le cadre de travail sur Eclipse, puis on écrit les tests nécessaires.

2.1. Cadre de travail

La première étape consiste à « installer » une bibliothèque de fonctions fournie sous la forme d'un fichier jar accompagné de son pom. En fait, cette bibliothèque est en deux exemplaires, l'une sans bug et l'autre avec des bugs que vos tests devront mettre en évidence.

1. Téléchargez (clic droit sur chaque lien + *Enregistrer la cible du lien sous...*) evalexpr-1.0-ok.jar et evalexpr-1.0-ok.pom dans le dossier Téléchargements.
2. Téléchargez (idem) evalexpr-1.0-pb.jar et evalexpr-1.0-pb.pom dans le dossier Téléchargements.
3. Ouvrez un terminal à cet endroit et exécutez ces deux commandes :

```
mvn install:install-file -Dfile=evalexpr-1.0-ok.jar -DpomFile=evalexpr-1.0-ok.pom
mvn install:install-file -Dfile=evalexpr-1.0-pb.jar -DpomFile=evalexpr-1.0-pb.pom
```

Ces commandes « installent » ces fichiers dans votre « dépôt Maven local ». Il s'agit simplement du dossier `~/.m2` dans votre compte, et y installer un *jar* consiste à le mettre dans le bon sous-dossier, en fonction de son *package* et sa version.

Ensuite, on crée un projet Maven qui utilise cette bibliothèque.

4. Créez un nouveau projet Maven dans le *workspace* Eclipse :
   - GroupID : `fr.iutlan.ql`
   - ArtifactID : `tp3`
5. Remplacez le fichier `pom.xml` par ce fichier pom.xml (enregistrer la cible sous... et changez le nom).
6. Mettez à jour le projet Maven.

Regardez le fichier `pom.xml`. Il contient ces lignes :

```
<!-- fichier jar fourni sans ses sources -->
<dependency>
  <groupId>fr.iutlan.ql</groupId>
  <artifactId>evalexpr</artifactId>
  <version>1.0-ok</version>  <!-- mettre 1.0-ok ou 1.0-pb -->
</dependency>
```

C'est simplement la dépendance vers l'un ou l'autre des deux *jar* installés dans l'étape 3. Avec ces lignes, vous pouvez importer leurs classes dans vos tests.

2.2. Cahier des charges à tester

La dépendance `evalexpr-1.0` (version ok ou pb) contient une seule classe, `EvalExpr`. Cette classe possède deux méthodes statiques, `EvalExpr.toAFile(String nomfichier)` et `EvalExpr.toAString()`. Ces deux méthodes sont des fabriques qui retournent une instance de `EvalExpr`. Une instance de `EvalExpr` est appelée « évaluateur ». La méthode `toAFile` retourne une exception si le nom du fichier n'est pas acceptable pour le système d'exploitation.

Un évaluateur ne possède qu'une seule méthode, `eval`. Elle demande un paramètre, une chaîne contenant une expression arithmétique toute simple, de la forme $n1$ *op* $n2$, avec $n1$ et $n2$ étant deux entiers, et *op* étant un caractère parmi `+`, `-`, `*`, `/` (division entière) et `%` (reste de la division entière). Ce dernier est calculé à l'aide de la méthode `remainder` de Java, voir la documentation et non pas la méthode `mod`, voir la documentation .

Note technique : les entiers sont représentés par des `BigInteger`. Cette classe donne une précision quasiment infinie aux entiers. On peut calculer avec des entiers qui ont 50 chiffres tous significatifs.

Il peut y avoir des espaces avant, après et entre tous ces éléments, ou aucun espace. Les entiers peuvent avoir un signe `+` ou `-`.

La méthode `eval` calcule la valeur de l'expression, qui est également un entier.

- Si l'évaluateur a été créé par `EvalExpr.toAFile(String nomfichier)`, alors la méthode `eval` enregistre l'expression fournie, suivie du signe `=`, suivi du résultat dans le fichier indiqué. La méthode `eval` retourne le nom du fichier. Le fichier est créé s'il n'existe pas, et il est complété s'il existait déjà.

Par exemple :

```
EvalExpr evaluator = EvalExpr.toAFile("sommes.txt");
evaluator.eval("2+3");
String nf = evaluator.eval("-7*-3");
System.out.println("Les calculs sont dans "+nf);
```

crée le fichier `sommes.txt` et y écrit deux lignes « `2+3=5` » suivie de « `-7*-3=21` ».

- Si l'évaluateur a été créé par `EvalExpr.toAString()`, alors la méthode `eval` retourne l'expression fournie, suivie du signe `=`, suivi du résultat.

Par exemple :

```
EvalExpr evaluator = EvalExpr.toAString();
String v1 = evaluator.eval("2+-3");
System.out.println(v1);
```

affiche « `2+-3=-1` ».

Dans le cas d'une erreur, par exemple mauvaise écriture d'un nombre, de l'expression, ou opérateur inconnu, la méthode `eval` déclenche une exception et ne modifie pas le fichier.

2.3. Travail à faire

Votre travail consiste à vérifier le cahier des charges à l'aide de tests, et éventuellement, trouver des bugs, des oublis ou des limites bizarres, pas mentionnées dans ce cahier des charges.

1. Créez le *package* `fr.iutlan.ql.tp3`, du côté des tests (`src/test/java`). Il n'y aura rien côté `src/main/java`.
2. Créez la classe `TestsEvalExpr` :

```java
package fr.iutlan.ql.tp3;

import static org.assertj.core.api.Assertions.*;
import static org.junit.jupiter.api.Assertions.*;
import java.io.File;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.util.stream.Stream;

import org.assertj.core.util.Files;
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.*;

import fr.iutlan.ql.evalexpr.EvalExpr;

@SuppressWarnings("unused")
public class TestsEvalExpr {

    // fichier créé par les tests, nom et encodage
    private static final String NomFichier = "result.txt";
    private static final Charset UTF_8 = StandardCharsets.UTF_8;

    @BeforeEach
    @AfterEach
    void removeFichier() {
        // TODO supprimer le fichier, qu'il existe ou non
    }


    @Test
```

```
    void testAdditionPosPosFichierValide() {
        try {
            // ARRANGE
            EvalExpr evaluator = EvalExpr.toAFile(NomFichier);
            final String expr = "3+2";
            final String valeur = "5";

            // ACT
            String res = evaluator.eval(expr);

            // ASSERT
            assertThat(res).isEqualTo(NomFichier);
            File file = new File(NomFichier);
            assertThat(file).exists();
            assertThat(file).content(UTF_8).isEqualTo(expr+"="+valeur);
        } catch (Exception e) {
            // il ne devrait pas y avoir d'exception dans ce test
            fail();
        }
    }

    @Test
    void testExpressionInvalideGenereException() {
        try {
            // ARRANGE
            EvalExpr evaluator = EvalExpr.toAFile(NomFichier);
            final String expr = "bla@blabla";

            // ASSERT
            assertThrows(Exception.class, () -> {
                // ACT
                evaluator.eval(expr);
            });

            // ASSERT
            // TODO à vous de compléter, vis à vis du cahier des charges
        } catch (Exception e) {
            // il ne doit pas y avoir d'autre exception que celle prévue
            fail();
        }
    }
}
```

3. Lancez la classe par `Run As/JUnit Test`. Il ne doit y avoir aucune erreur avec des tests *corrects*. Notez comment des exceptions qui arriveraient de manière imprévue dans le test sont interceptées et transformées en échec. Et voyez comment une expression incorrecte qui doit déclencher une exception est vérifiée et comment une exception qui arriverait ailleurs est transformée en échec également.

4. Que dit le cahier des charges concernant la création du fichier quand l'expression est incorrecte ? Complétez la méthode `testExpressionInvalideGenereException`.
5. Modifiez la version de evalexpr en `1.0-pb` dans le fichier `pom.xml`. Avec ceci, quand vous aurez programmé d'autres tests, il devra y avoir des erreurs.

Le but du TP est de mettre en évidence les bugs de la version `1.0-pb` à l'aide de tests unitaires. **Important:** les mêmes tests qui échouent sur la version `1.0-pb` doivent **tous réussir** sur la version `1.0-ok`. Il est stupide de programmer des tests qui échouent sur les deux versions – sauf si vous avez trouvé un vrai bug dans la version correcte. Mais faites très attention à coller au cahier des charges. Si vous appelez la méthode `eval` avec une expression qui n'est pas construite comme spécifié dans le cahier des charges, alors ne vous étonnez pas que ça plante.

Il est très important de bien nettoyer le contexte d'exécution des tests, en particulier supprimer les fichiers créés. C'est le rôle de la méthode `removeFichier`. Elle est annotée par `@AfterEach` et `@BeforeEach` pour supprimer le fichier avant et après chaque test.

5. Complétez la méthode `removeFichier`. Relisez le TP2 pour savoir comment supprimer un fichier.
6. Programmez de nouveaux tests pour valider, ou non, `EvalExpr.toAFile.eval` par rapport à son cahier des charges.

Comme il y a énormément de possibilités, vous vous limiterez à par exemple :

- Tester l'addition avec des zéros, des nombres positifs, négatifs, mélangés, avec des espaces avant, après. Si tous ces tests passent, alors on peut considérer que l'analyse des nombres est correcte.
- Vérifier que les 5 opérations fonctionnent. Il suffit de tester sur quelques valeurs, par exemple avec des signes différents.
- Fournissez des paramètres incorrects pour différentes raisons, et vérifier que ça plante comme il se doit. Attention, la notion de paramètres incorrects est relative au cahier des charges. N'inventez rien.
- Occupez-vous d'abord entièrement de `EvalExpr.toAFile` avant de passer à `EvalExpr.toAString`.

Il est recommandé d'utiliser des tests paramétrés, voir ci-dessous.

Vous devez écrire des tests qui vérifient le cahier des charges. La priorité est que vos tests réussissent tous avec la version correcte de `EvalExpr`. Ensuite, au moins un de vos tests doit échouer avec la version incorrecte de `EvalExpr`. Normalement, plusieurs tests doivent échouer car la version incorrecte compte plusieurs bugs. Lors de la notation du TP, d'autres versions incorrectes seront soumises à vos tests pour voir s'ils détectent bien des anomalies par rapport au cahier des charges. Donc vos tests doivent signaler par un échec que le cahier des charges n'est pas respecté.

Attention, si vos tests échouent avec des *errors*, c'est que vous avez fait une erreur de programmation dans vos tests, et ça vous coûtera de nombreux points (note largement sous la moyenne). Le non-respect du cahier des charges se traduit par des *failures*, pas par des *errors*. Le pire pour vous sont des tests qui ne se compilent pas.

3. Partie théorique

Voici quelques rappels et des nouveautés concernant les tests unitaires. Il faut relire le sujet du TP2 qui concerne la présentation de AssertJ.

3.1. Test d'exceptions

Certains tests unitaires consistent à vérifier qu'une méthode émet une exception spécifique dans certaines circonstances, par exemple quand les paramètres sont incorrects. On a vu dans le TP1 que cela s'écrit ainsi en JUnit5 :

```
@Test
void method2WithNullMustThrowNullPointerException() {
    // ARRANGE
    CLASSE obj = new CLASSE();

    // ASSERT
    assertThrows(NullPointerException.class, () -> {

        // ACT
        obj.method2(null);
    });
}
```

L'action qui doit déclencher l'exception est à placer dans la fonction lambda. Le test réussit s'il y a l'exception attendue, ici une `NullPointerException`. Le test échoue si l'exception ne se produit pas.

3.2. Tests paramétrés

Quand on a plusieurs tests à faire sur la même méthode, avec des valeurs différentes pour ses paramètres et son résultat, il est intéressant de *paramétrer* la méthode de test elle-même. C'est à dire qu'on appelle la même méthode de test avec des valeurs différentes, et donc ce sont autant de tests différents qui sont effectués.

Par exemple, au lieu d'écrire ceci :

```
@Test
void absPositiveMustBeSame() {
    // ARRANGE
    double number = 10.8797;

    // ACT
    double result = Math.abs(number);

    // ASSERT
    assertThat(result).isCloseTo(number, within(1e-10));
}

@Test
void absNegativeMustBeOpposite() {
    // ARRANGE
    double number = -23.4654;
```

```
    // ACT
    double result = Math.abs(number);

    // ASSERT
    assertThat(result).isCloseTo(-number, within(1e-10));
}
```

On voudrait n'écrire que ceci, qui serait en plus beaucoup plus polyvalent (sauf que ça ne marche pas comme ça) :

```
@Test
void checkAbsFunction(double number, double expectedResult) {
    // ARRANGE : le paramètre number

    // ACT
    double result = Math.abs(number);

    // ASSERT
    assertThat(result).isCloseTo(expectedResult, within(1e-10));
}

// ici il faut écrire de quoi injecter différents couples
// de nombres dans checkAbsFunction
```

Pour commencer, il faut rajouter deux annotations sur la méthode de test. D'abord `@ParameterizedTest` qui indique qu'il y a des paramètres. Ensuite, il faut une annotation pour dire d'où viennent les valeurs qui seront fournies à la méthode de test, les arguments du test. Il y a plusieurs annotations possibles, comme `@ValueSource`, mais la plus générale est `@MethodSource`.

Voici ce que ça donne (exemple à adapter pour le TP) :

```
@ParameterizedTest
@MethodSource
void checkAbsFunction(double number, double expectedResult) {
    // ARRANGE : le paramètre number

    // ACT
    double result = Math.abs(number);

    // ASSERT
    assertThat(result).isCloseTo(expectedResult, within(1e-10));
}
```

Ensuite, il faut programmer la méthode qui génère les arguments. Alors, c'est étonnant, mais c'est une deuxième méthode qui a exactement le même nom, `checkAbsFunction`, mais d'une part, elle est `static` (méthode de classe) et d'autre part, elle n'a ni paramètre ni annotation, et elle retourne une sorte de liste des valeurs à injecter dans la méthode de test :

```
// méthode qui génère les valeurs des paramètres de checkAbsFunction
private static Stream<Arguments> checkAbsFunction() {
    return Stream.of(
        Arguments.of( 10.8797, 10.8797),
        Arguments.of(-23.4654, 23.4654),
        Arguments.of(  0.0000,  0.0000)
    );
}
```

- Le type générique `Stream` représente une suite de données. C'est comme une liste, mais avec des méthodes spécifiques. Ici, on génère une suite d'instances de `Arguments`.
- La classe `Arguments` est fournie par JUnit5 (en fait, c'est une interface). La méthode de fabrique `Arguments.of` regroupe les valeurs dans une sorte de tableau (en fait, c'est une lambda qui retourne le tableau).

4. Rendu du TP

Ouvrez un terminal bash dans le dossier de votre TP, là où il y a le fichier pom.xml, puis tapez ceci :

```
mvn clean
cd ..
tar cfvz tp3.tgz tp3
```

Puis déposez le fichier `tp3.tgz` dans la zone de dépôt du TP3 sur Moodle R4.02 Qualité de développement .

ATTENTION votre travail est personnel. Si vous copiez pour quelque raison que ce soit le travail d'un autre, vous serez tous les deux pénalisés par un zéro.

En cas de souci quelconque, envoyez un mail à `pierre.nerzic@univ-rennes1.fr` pour expliquer le problème.