TP2 : Tests unitaires

Exercice 1 : Modélisation du Receiver

À partir du travail effectué au TD 4, réalisez le diagramme de classe (opérations comprises) correspondant à la partie « receiver » du mini-éditeur.

Créez un projet Java `fr.istic.m1.aco.miniediteur.v1` dans Eclipse. Dans ce projet, créez un package `receiver`. À partir du diagramme de classe réalisé, générez et corrigez si nécessaire le code correspondant dans votre package.

Vous devez obtenir un squelette de chacune des classes de votre diagramme de classe contenant leurs attributs et les squelettes de leurs méthodes (contenant uniquement le lancement d'une `UnsupportedOperationException`), le tout commenté à l'aide des tags Javadoc appropriés.

Une fois que vous avez ces squelettes, vous pouvez passer à l'exercice suivant.

Exercice 2 : Tests unitaires

Créez dans votre projet un nouveau répertoire source `test` (clic droit → New → Source Folder). Il sera utilisé pour stocker tous les tests unitaires créés au cours du développement du mini-éditeur. Créez un package `receiver` et dans ce package une classe de test JUnit par classe générée (clic droit → New → JUnit Test Case). Nommez-la `TestNomDeLaClasseATester`, puis cliquez sur `Browse` en bas à droite de la fenêtre pour sélectionner la classe à tester (`Class under test`), une fenêtre de recherche s'ouvre dans laquelle vous pouvez entrer le nom de la classe à tester. Choisissez ensuite `Next` pour sélectionner les méthodes à tester, puis cliquez sur `Finish`.

Vous obtenez des classes JUnit contenant des squelettes de méthodes de test (portant le tag `@Test`) pour chacune des méthodes des classes à tester.

Remplissez la classe et les squelettes des méthodes de test. Lancez ensuite vos classes de test (Run as → JUnit test). Les résultats des tests s'affichent dans la fenêtre JUnit. Tous les tests doivent réussir. Au fur et à mesure de l'implémentation de vos méthodes, les tests unitaires devraient toujours réussir. Vous pouvez ainsi vous assurer que vos méthodes réalisent la bonne fonction et que vos modifications ne font pas régresser votre application.

Vous pouvez utiliser les méthodes JUnit suivantes dans vos méthodes de test :

`fail(String)` : Le test échoue en affichant le message passé en paramètre.

`assertTrue(true)` : Le test réussit.

`assertsEquals([String message], expected, actual)` : Vérifie si les valeurs passées en paramètres sont les mêmes. Si ce n'est pas le cas, échoue en affichant le message.

`assertsEquals([String message], expected, actual, tolerance)` : Idem que `assertsEquals` pour les double et les float, la valeur passée pour le paramètre pour `tolerance` indique le nombre de décimales qui doivent être égales.

`assertNull([message], object)` : Vérifie que l'objet est null.

`assertNotNull([message], object)` : Vérifie que l'objet n'est pas null.

`assertSame([String], expected, actual)` : Vérifie que les deux variables référencent le même objet.

`assertNotSame([String], expected, actual)` : Vérifie que les deux variables ne référencent pas le même objet.

`assertTrue([message], boolean condition)` : Vérifie que `condition` vaut `true`.

Pour tester qu'une méthode lance bien une exception, vous pouvez ajouter entre parenthèse une annotation derrière le tag `@Test` de la méthode de test concernée, de cette manière :

```
@Test(expected=ExceptionAttendue.class)
```

De cette manière, si la méthode testée ne lance pas l'exception attendue, le test échouera.

Si l'exception attendue n'est pas une sous-classe de RuntimeException, il faudra également ajouter à la méthode de test une clause `throws ExceptionAttendue` de cette manière :

```
@Test(expected=ExceptionAttendue.class)
public void testMethodeTestee() throws ExceptionAttendue {
        …
}
```