amU Faculté des sciences Aix Marseille Université

Année 2025-26

Domaine Sciences et Technologies
Master informatique
Génie logiciel : TP 1
Codes UE : SINA09EL
Tests unitaires en boite blanche

1 Tests unitaires sous JUnit

Cette première partie du TP constitue un rappel de l'utilisation de l'environnement JUnit de tests unitaires. JUnit est un environnement de tests unitaires pour Java et nous allons l'utiliser au sein de l'IDE IntelliJ [1] via le moteur de production Gradle https://gradle.org/.

La version JUnit actuelle est la version 5. La version 4 est de même philosophie, s'appuyant également sur les annotations introduites depuis Java 5. La version 4 reste très utilisée, mais elle n'est pas pleinement compatible avec la version 5 [2].

La version 5 est néanmoins une évolution majeure qui offre de nouvelles constructions. Le site pour JUnit 5 se trouve à https://junit.org/junit5 [3].

1.1 Écriture de test

Comme évoqué nous allons utiliser IntelliJ et JUnit principalement dans le cadre d'automatisation de production logicielle via l'outil Gradle. Il est bien évidemment possible d'utiliser un autre outil de ce type, tel `maven`, ou bien simplement JUnit sous IntelliJ et en mode natif.

Avec JUnit, on peut créer une classe de test. Les cas de test s'y retrouvent sous la forme de méthodes identifiées (précédées) par l'annotation JUnit `@Test` [4]. Un cas de test pourra utiliser des assertions de la classe `Assertions` de JUnit ou comme nos exemples, celles de `AssertJ`. Ces assertions qui seront utilisées pour vérifier notamment des égalités (de valeurs, objets, ...) et feront échouer le test si elles ne sont pas vraies.

Voici un exemple de (méthode de) test :

```
@Test
void testSumReal() {
  Complex z1 = new Complex(1.0F, 2.0F);
  Complex z2 = new Complex(3.0F, 4.0F);

  float expected = 1.0F + 3.0F;

  Complex z = z1.sum(z2);
  assertThat(z.getRealPart()).as("problem with Real part of Sum")
          .isCloseTo(expected, within(EPSILON));
```

[1] https://www.jetbrains.com/fr-fr/idea/

[2] Les versions 3 et antérieures de JUnit utilisaient d'autres techniques pour mettre en place les tests, ces versions ne sont plus vraiment utilisées

[3] Un site qui pourrait également vous être utile https://www.jmdoudoux.fr/java/dej/chap-junit5.htm

[4] Lors de l'apparition de cette annotation, IntelliJ vous proposera notamment si nécessaire d'importer les classes JUnit nécessaires aux tests. Attention à bien choisir la version 5 de JUnit dans ce cas.

```
}
```

D'autres annotations JUnit permettent de gérer les initialisations et cas particuliers. Comme toutes les annotations Java, elles sont de la forme `@mot-clé`.

- `@BeforeAll` et `@AfterAll` permettent d'exécuter des instructions avant et après l'exécution de la suite de tests [5].
- `@BeforeEach` permet de définir des initialisations à faire avant chaque test (typiquement définir un objet qui sera utilisé par tous les tests) et `@AfterEach` est similaire mais est effectué après chaque test [6].
- `@Disabled` permet de ne pas effectuer le test qui suit.

Les annotations `@...` et assertions `assert...` sont à importer si nécessaire (voir les fichiers exemples et la documentation).

1.2 Utilisation de Junit

On suppose dorénavant que vous disposez de votre propre projet sur le serveur `etulab` avec une version locale de celui-ci sur votre machine (voir "Rendu des TPs : spécifications").

Cette première partie va vous faire voir ou revoir un usage basique de JUnit.

1. Vous disposez dans les répertoires `src/main/java/complex` et `src/test/java/complex` les fichiers `Complex.java` et `ComplexTest.java` respectivement. Le premier contient une implémentation des nombres complexes et le second un exemple de fichiers pour tester cette implémentation.
2. Sous IntelliJ, vous avez la possibilité de lancer soit l'ensemble des tests de la classe `ComplexTest`, soit individuellement chaque méthode de test (via le triangle vert d'exécution dans la colonne à gauche de la fenêtre du code source).
   - Lancez la méthode de test `testGetterImaginary`. Que constatez-vous ?
   - Lancez l'ensemble des tests de la classe `ComplexTest`. Que constatez-vous ?
3. Modifiez à minima la classe de test (au regard de ce qui a été implémenté ou non) afin que la suite des tests réussisse.
4. En rajoutant des instructions d'impression, vérifier que `@BeforeAll` et `@AfterAll` sont effectuées avant et après chaque exécution de l'ensemble des tests. Idem pour `@BeforeEach` et @`AfterEach` avant chaque méthode de test.
5. Modifiez le code de `@BeforeEach` afin de mutualiser le code de création des objets `Complex` sur lequel va se faire chacun des tests.

Nous allons maintenant utiliser `Gradle` pour continuer notre campagne de test ; le résultat des tests sera matérialisé comme un rapport au format `html`. Vous pouvez utiliser gradle en ligne de commande pour lancer les tests avec `gradle test` (gradle prend en premier argument la tâche à exécuter ici test pour lancer des tests). Sous IntelliJ, vous pouvez ouvrir la fenêtre de l'outil `Gradle` (`View` > `Tool Windows` > `Gradle`). Vous pouvez

[5] Cela constitue ce qu'on appelle le "test fixture".

[6] Ceci constitue les préambules et postambules des cas de tests. A noter qu'ils seront exécutés avant et après chaque (méthode de) test mais que le code est identique pour tous.

alors depuis la fenêtre ouverte lancer les tests (`Tasks` > `verification` > `test`). Vous retrouverez le résultat de vos tests dans votre projet comme un artefact html dans `build/reports/tests/test/html`.

6. Relancez les tests via gradle et vérifier le contenu du fichier de rapport des tests. Combien de tests ont été lancés ? Combien de tests sont passés avec succès ? Quel est le temps d'exécution de la suite de tests ?
7. Ajouter trois méthodes pour tester la méthode `inverse` de la classe `Complex` : la première et la seconde devront tester que les parties réelle et imaginaire du résultat sont correctes dans le cas d'un complexe non nul tandis que la troisième devra tester la levée d'une exception dans le cas contraire.
8. Complétez la méthode `product` de la classe `Complex` et tester votre implémentation avec les tests fournis.
9. Complétez le code de la méthode static `infinite` afin que l'exécution de cette méthode ne termine pas puis activer le test sur `infinite`. Que se passe-t-il ? Modifier le test pour qu'il échoue si `infinite` ne termine pas en 100ms (utiliser l'assertion `assertTimeoutPreemptively` (voir la partie `Timeout` de la documentation). Vérifier que l'exécution des tests termine.

Avertissement : JUnit 5 permet de programmer une gestion de test très évoluée (voir la documentation) ce qui signifie qu'il est possible d'introduire des erreurs dans les suites de test.

2 Couverture de tests avec JaCoCo

Lorsqu'on souhaite tester une implémentation, on va définir un ensemble de tests (également appelé *suite de tests*). La question qui se pose alors est de savoir si cette suite est "suffisante" [7], à savoir qu'elle permet de tester les différents cas d'utilisation de cette implémentation. On peut alors parler de qualité d'une suite de tests.

Une idée simple pour mesurer cette qualité est de considérer sa couverture ; chaque test va, lors de son exécution, exécuter un certain nombre d'instructions de l'implémentation à tester. Cette instruction ou ce bloc d'instructions sera alors dit couvert par ce test. Intuitivement, si une instruction ou un bloc d'instructions n'est couvert pour aucun test de la suite de tests alors nous n'avons aucune garantie sur ce que fait ce morceau de code.

Ce tte seconde partie du TP introduit à l'utilisation de l'outil JaCoCo (Java Code Coverage) (https://www.jacoco.org/jacoco/trunk/index.html). JaCoCo est en fait une bibliothèque de (mesure de) couverture de code Java utilisable depuis un IDE ou au sein d'outils d'automatisation de production logicielle comme Gradle ou Maven. JaCoCo s'utilise également en complément d'un environnement de tests tel JUnit (celui que nous allons considérer) ou TestNG.

2.1 Premiers pas avec JaCoCo

Nous allons utiliser JaCoCo sous IntelliJ avec l'outil gradle [8] Pour cela, il vous faudra ouvrir la fenêtre gradle et lancer `Tasks > verification > test` ou `jacocoTestReport`).

[7] Formellement, elle ne le sera jamais...

[8] Il est également possible de l'utiliser directement sous IntelliJ. Par ailleurs, IntelliJ possède son propre outil de calcul de couverture que nous n'allons pas utiliser dans ce TP mais libre à vous de l'essayer.

Cette action va produire un fichier `index.html` de rapport que vous trouverez dans `tp2/build/reports/jacoco/test/html`. Vous y découvrirez un tableau indexé en ligne par les différents packages présents dans le projet et en colonne des informations quantitatives quant-à la couverture du code. On peut alors naviguer au sein du package, par exemple en cliquant sur `palindrome` et voir apparaitre un autre tableau qui concerne alors les classes de ce package. En cliquant sur une classe, on a de nouvelles informations concernant les méthodes de cette classe. Enfin en cliquant sur une méthode, on voit apparaitre le code de celle-ci où les instructions apparaissent comme colorées.

2.1.1 Calcul de la couverture par JaCoCo

Nous allons maintenant aller plus avant sur ce rapport fourni par JaCoCo.

2.1.1.1 Classe `Palindrome`

1. Comprendre le sens des couleurs (vert, rouge, jaune).
2. Comprendre le sens des taux de couverture pour une classe, les méthodes, les lignes et les branchements
3. Comprendre les couleurs données aux instructions de la classe de test.
4. Pourquoi certaines conditions booléennes sont en jaune ?
5. Est-ce qu'il est possible de couvrir toutes les instructions de la classe `Palindrome` ?
6. Enlever certains tests (par `@Disabled`) et voir comment la couverture évolue.

2.1.1.2 Classe `PartialCoverage` Effectuer les mêmes opérations avec la classe `PartialCoverage.java` en ajoutant les cas de test nécessaires. Que conclure ?

2.2 Etude de Couverture de code

2.2.1 Couverture des classes de l'application `Complex`

Reprendre la classe `Complex` (comme complétée précédemment). Vérifier la couverture de vos tests sur les méthodes de cette classe.

1. Analyser les résultats pour la partie compte-rendu.
2. Ajouter des tests pour aller jusqu'à un taux de couverture de 100% si cela est possible. Pour chaque test ajouté, vous signalerez en commentaire quelle instruction supplémentaire il couvre.

2.2.2 Files à double extrémité

Le but de cette partie sera de coder des files à double extrémité ou *deque* (abréviation de l'anglais *double-ended queue*). Une file à double extrémité est un type abstrait permettant d'ajouter et de supprimer des données à la fin (*queue*) ou au début (*tête*), réunissant ainsi les avantages des files classique et des piles. Vous allez coder et tester une classe pour les files à double extrémité nommée `ArrayDoubleEndedQueue` qui implémentera l'interface `DoubleEndedQueue` qui définit les méthodes suivantes :

— `void addFirst(E e)` qui ajoute un élément en début de file ;
— `void addLast(E e)` qui ajoute un élément en fin de file ;

— `E removeFirst()` qui renvoie et supprime l'élément en début de file ;
— `E removeLast()` qui renvoie et supprime l'élément en fin de file ;
— `E getFirst()` qui renvoie l'élément en début de file ;
— `E getLast()` qui renvoie l'élément en fin de file ;
— `int size()` qui renvoie le nombre d'éléments de la file ;
— `boolean contains(Object o)` qui renvoie `true` si la file contient un élément égal (par `equals`) à `o` et `false` sinon.

La classe `ArrayDoubleEndedQueue` utilisera un tableau d'`Object` pour stocker les éléments de la file (il faudra donc faire un *cast* pour obtenir le bon type d'objet). La taille du tableau sera égale à la capacité (*capacity*) de la file (déterminée à la construction de la file) et ne changera pas par la suite. La file se comptera comme un buffer circulaire avec comme attribut l'indice du premier élément et le nombre d'éléments de la file. Les éléments de la file seront donc compris entre `indexFirst` inclus et `(indexFirst + size) % capacity` exclus (voir figure ci-dessous). Toutes les cases ne correspondant pas à des éléments de la file devront être à `null`. Lorsqu'une opération est impossible (accès à un élément quand il n'y a pas d'élément dans la file ou ajout d'un élément quand la file a atteint sa capacité maximale), une exception devra être levée conformément à la Javadoc de `DoubleEndedQueue`.

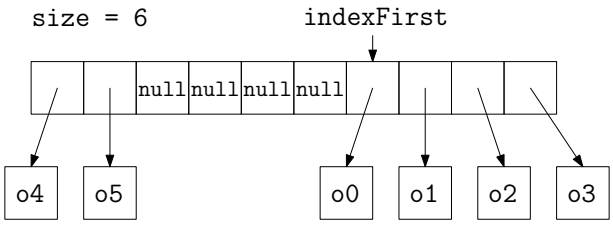

FIGURE 1 – image

1. Créez une classe `ArrayDoubleEndedQueue` implémentant l'interface `DoubleEndedQueue`. Pour le moment, contentez-vous de définir des méthodes par défaut (ne retournant `null`, `0` ou `false` et sans autre code). Vous pouvez réaliser cela rapidement sous IntelliJ en faisant clic droit puis *generate* puis *implements method* un fois que vous avez défini que la classe implémente l'interface.
2. Créez et codez la classe `ArrayDoubleEndedQueueTest` testant que la classe `ArrayDoubleEndedQueue` respecte les contraintes décrites dans la documentation de `DoubleEndedQueue`. Pour le moment, ne modifiez pas la classe `ArrayDoubleEndedQueue`.
3. Complétez la classe `ArrayDoubleEndedQueue` afin qu'elle passe les tests de `ArrayDoubleEndedQueueTest`. Quel est le taux de couverture de votre suite de test ?
4. Si besoin, complétez les tests de `ArrayDoubleEndedQueueTest` afin d'obtenir une couverture de tous les arcs de `DoubleEndedQueue`.

3 Rendu demandé

Il vous faudra rendre la suite de test que vous avez écrite pour `ArrayDoubleEndedQueue` ainsi qu'un compte-rendu présentant cette suite. Il faudra préciser pour chaque cas de test son objectif ainsi que les données de test ainsi que le résultat attendu.