Université Claude Bernard Lyon 1
Département Informatique - Polytech Lyon

4A Apprentis, 2018-2019

ISI3

TP5. Tests Unitaires et Mocks

Laëtitia Matignon

Vous trouverez dans le projet un dossier `lib` composé de :

- un fichier `persons.jar` pour le package `persons`
- un fichier `people.jar` pour le package `people`
- un fichier `mockito.jar` pour `Mockito`[1], le *framework* de *mock* à utiliser dans la seconde partie du TP

Le dossier `javadoc` contient la documentation des packages `persons` et `people`.

Mise en place Après avoir ajouté à votre *Build Path* les .jar et la javadoc, créer un `source folder` nommé `test` et un package `exercice1` dans ce source folder. Créer ensuite dans ce package une classe de test JUnit5[2] (*JUnit Test Case*) que vous nommerez `PersonTest` .

1 Tests Boite Noire

Au fil des questions suivantes, vous devez définir des cas de tests pour tester les méthodes de l'interface `IPerson`. Ces cas de tests doivent couvrir efficacement le domaine d'entrée de ces méthodes. Les cas de tests seront exécutées pour tester les différentes classes implémentant l'interface `IPerson`. **La classe `persons.Person` est normalement sans erreurs, les classes du package `people` ont toutes au moins une erreur.**

Question 1 *Lisez la documentation de l'interface `IPerson`. On souhaite tester les méthodes `wasBorn` et `getAge`. Pour chaque méthode, réalisez une* ***analyse partitionnelle*** *en proposant un partitionnement en classes d'équivalence pour chaque donnée d'entrée. Choisissez les données de tests et oracles correspondant pour obtenir les cas de test. Vous trouverez des informations sur le partitionnement en classes d'équivalence dans la partie Partie 3.1 « Méthodes de test fonctionnels » du cours `https://projet.liris.cnrs.fr/sycosma/wiki/lib/exe/fetch.php?media=wiki:cours3test.pdf`*

Question 2 *Pour tester la classe `Person` du package `persons`, implémentez les cas de tests définis à la question précédente pour les méthodes `wasBorn` et `getAge` dans*

1. `https://static.javadoc.io/org.mockito/mockito-core/1.10.19/org/mockito/Mockito.html`
2. `https://junit.org/junit5/`

la classe de test `PersonTest`*. Exécutez ces cas de tests. Tous les tests devraient être réussis (la classe* `Person` *est normalement sans erreurs).*

Question 3 *On souhaite maintenant faire passer ces tests aux classes du package* `people` *qui implémentent l'interface* `IPerson` *(donc* ***la classe*** `people.Personne` ***n'est pas à tester pour l'instant****). Pour cela, proposez une architecture pour faire passer les cas de tests existants à toutes ces classes sans réécrire de nouveaux tests.*

Question 4 *Pour chacune des classes du package* `people` *qui implémentent l'interface* `IPerson` *(donc la classe* `people.Personne` *n'est pas à tester ici), vous devez avoir au moins un test qui échoue sur chacune des classes. Si ce n'est pas le cas, votre analyse réalisée à la question 1 ne garantie pas une bonne couverture du domaine des données d'entrée.* ***Vous devez donc compléter vos cas de tests et refaire passer tous les tests à toutes les classes qui implémentent l'interface*** `IPerson` ***(y compris à*** `persons.Person`***).***

Question 5 *La classe* `Personne` *du package* `people` *n'implémente pas l'interface* `IPerson`*. Utiliser un patron de conception pour faire passer les tests précédents à la classe* `Personne`*.*

2 Tests en isolation et *Mock*

On souhaite maintenant implémenter et tester des requêtes sur des listes de personnes. On veillera à utiliser une interface générique pour représenter les personnes, afin de faciliter l'écriture des tests. On utilisera donc l'interface `IPerson` du package `persons`.

Question 6 *Créez une classe* `OutilsPerson` *dans laquelle vous devez implémenter une méthode qui prend en paramètre une liste de* `IPerson`*, une date au format* `GregorianCalendar`*, un âge (nombre d'années) minimal et un âge maximal (ce sont des entiers). Cette méthode doit retourner l'ensemble des personnes parmi la liste passée en paramètre dont l'âge à la date donnée est dans l'intervalle* $[ageminimal, agemaximal]$*. A titre d'exemple, la méthode pourra être utilisée pour retourner la liste des personnes qui auront entre 60 et 65 ans le 10 mai 2050.*

La méthode renverra une exception `IllegalArgumentException` *si l'age minimal est supérieur à l'age maximal.*

Question 7 *Ecrivez une méthode de recherche qui prend en paramètre une liste de* `IPerson` *et une date au format* `GregorianCalendar`*. Cette méthode retourne l'âge de la personne la plus âgée parmi la liste à la date donnée en paramètre. Si la liste en entrée est vide, la méthode retournera -1.*

Question 8 *On souhaite maintenant tester ces deux méthodes en utilisant des instances de* `IPerson`*. En quoi les objets mock peuvent être utiles dans notre cas ? Vous trouverez des informations sur les objets mock dans la partie Partie 4.2 « Doublure d'objets » du cours* `https://projet.liris.cnrs.fr/sycosma/wiki/lib/exe/fetch.`

php?media=wiki:cours3test.pdf.

Question 9 *Ecrivez des tests unitaires (comportement nominal et aux limites) pour les deux requêtes implémentées aux questions 6 et 7 en utilisant le framework* `Mockito`.

Question 10 *Trouvez un outil d'analyse de couverture de code utilisable avec votre IDE (par ex.* `http://www.eclemma.org/` *pour eclipse). Vérifiez avec cet outil que vos tests couvrent l'ensemble des lignes des requêtes.*

Question 11 *On souhaite vérifier par des tests que la méthode de la question 7 calcule le plus grand âge de façon " anonyme ", c'est-à-dire sans lire les noms et prénoms des personnes ; et en utilisant au moins une fois la méthode* `getAge`. *Vérifiez ces propriétés à l'aide des objets mock.*