Devoir de spécification et validation du logiciel

Université Lille1 — Master 1 informatique, S2
UFR IEEA — partie Test SVL -1h30- 2015-2016 — FIL

Durée totale : **1h30**. Tous documents autorisés. Nombre de pages : **1 + 2 annexes**

Toutes les classes de test seront écrites en Python3 (inutile de donner les clauses `import`). Tous les tests seront unitaires et en isolation. Les mocks nécessaires seront décrits avec mockito. Si le cas d'étude vous semble sous-spécifié (ce qui fait partie de l'exercice), c'est à vous de faire un choix et de l'indiquer dans les tests. Si vous utilisez une classe fournie par Python et que vous avez oublié le nom exact des méthodes fournies par cette classe, vous pouvez utiliser un nom de votre cru, en précisant en une phrase le comportement de cette méthode. Les tests et le code doivent être soignés et **très clairs** (y compris l'écriture).

Vous devez répondre aux questions ci-dessous en simulant sur le papier une démarche type Test Driven Development : votre réponse doit donc être incrémentale. Pour chaque incrément sur votre copie, vous devez fournir :

1. un cas de test
2. le code le plus simple qui permet de faire passer ce test, en maintenant la couverture de code à 100%.

À chaque nouveau cas de test, votre code doit faire passer le nouveau cas de test, mais aussi tous les cas de tests précédents.

1. Vous devez ré-écrire à chaque fois l'intégralité du code, c'est à dire l'intégralité de la classe testée (inutile de donner les clauses `import`). Si une partie du code bien identifiée ne change pas d'un test à l'autre, par exemple le constructeur, vous pouvez vous contenter d'indiquer **clairement** que cette partie est présente sans changement.
2. Vous devez préciser en français si vous jugez nécessaire de modifier les tests précédents, et si c'est le cas indiquer en quelques mots les modifications à apporter. Il n'est pas nécessaire de ré-écrire les tests, ni de les annoter, une phrase suffit. En fin de copie vous pouvez indiquer par une simple phrase un refactoring qu'il vous semblerait judicieux d'effectuer.

Vous pouvez introduire dans la classe testée les accesseurs et constructeurs qui vous semblent nécessaires à l'écriture des tests, mais il n'est pas demandé de les tester.

Exercice 1 : Caisse d'équipements pour chantier

Lors de la préparation d'un chantier, un chef de chantier d'une entreprise du bâtiment planifie les équipements nécessaires dans la « caisse de chantier » le jour où ce chantier aura lieu. La caisse de chantier mémorise les équipements qui seront utilisés en leur associant la quantité nécessaire (on aura par exemple une caisse contenant 2 nacelles et 1 échafaudage). En plus de sa date et de sa caisse de chantier, un chantier est défini par l'entrepôt dans lequel seront pris les équipements. Lors de l'ajout d'une quantité donnée x d'un certain équipement, la caisse associée au chantier interroge l'entrepôt associé au chantier pour savoir si l'équipement en question est bien disponible à la réservation le jour du chantier dans la quantité x. Si tel est le cas, l'ajout est effectif : la caisse contient maintenant x instance en plus pour cet équipement. De plus la caisse avise l'entrepôt d'une demande de réservation pour ces x équipements, à la bonne date. Si tel n'est pas le cas, l'ajout échoue.

Q 1. 1 : En utilisant le langage Guerkin associé à la démarche du Behavioral Driven Development (cf annexe A), donner les scénarios qui vous semblent judicieux pour tester/documenter la fonctionnalité d'ajout d'un équipement à une caisse d'équipements. □

Q 1.2 : Tester et coder la fonctionnalité d'ajout d'un équipement à une caisse d'équipements, de la manière qui vous semble la plus judicieuse, en respectant les scénarios de la question précédente et les instructions données en début de sujet. Les `doctest` ne sont pas demandées. Au cas où, l'annexe B rappelle l'interface d'une table associative Python. □

A Scénarios à la Guerkin

```
Feature: ...
   ...
   As a ...
   I want ...
   So that ...

   Scenario: ...
      Given ...
      And ...
      When ...
      Then ...
      And...
   ...
```

Le `Given` et les `And` étant optionnels.

B La table associative de Python3 : `dict`

Un dictionnaire est une collection d'associations (clé, valeur) comme les `HashMap` de Java. Les clés doivent être d'un type hachable.
NB : les mocks de mockito sont hachables.

```
>>> d = dict()
>>> d
{}
>>> d["cle1"] = 2
>>> "cle1" in d
True
>>> d["cle1"]
2
>>> d
{'cle1': 2}
>>> "cle2" in d
False
>>> "cle2" not in d
True
>>> d["cle2"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'cle2'
>>> d["cle1"] = 50
>>> d
{'cle1': 50}
```