TDD - architecture dirigée par les tests

Les tests demandés sont des tests unitaires **en isolation**, sauf si précisé.

Exercice 1 : Sessions et SRP

Un entreprise commercialise un service quelconque, pour lequel l'utilisateur peut créer autant de sessions qu'il le veut. Le gestionnaire de sessions gère les sessions courantes. On peut lui ajouter autant de sessions qu'on le souhaite. Il permet de toutes les tuer d'un coup, qu'elles soient actives ou pas. On s'en fiche. On fait quelque chose de très simple : on peut ajouter des sessions, et toutes les tuer. (D'après un article de blog de Grzegorz Galezowski)

Q 1. 1 : Écrire en TDD tests et implémentation de `SessionManager` (le gestionnaire de sessions), pour la fonctionnalité « ajouter des sessions ». □

Supposons qu'un de vos collègues ait codé ceci :

```
def add(self, session):
    new_session = Session()
    self.les_sessions.append(new_session)
```

Q 1. 2 : Que pensez-vous de ce code ? Pourquoi ? □

Q 1. 3 : Écrire en TDD tests et implémentation de `SessionManager` (le gestionnaire de sessions), pour la fonctionnalité « tuer toutes les sessions ». □

L'entreprise change de propriétaire et de politique. Le service est maintenant par défaut gratuit, et dans ce cas une seule session est possible à la fois. Le service permettra à nouveau d'exécuter autant de sessions que voulu si le client paye d'une manière ou d'une autre. Pour faire simple, on gère ça par un simple booléen qui indique si l'ajout de sessions est contraint ou non (on se fiche de comment est géré l'abonnement dans cet exo).

Q 1. 4 : En toute bonne logique, vous devez récrire le code et les tests pour « ajouter des sessions ». Est-ce que les tests pour « tuer toutes les sessions « doivent être maintenus ? □

Le nouveau patron trouve que les ventes d'abonnement ne décollent pas vite. Il veut maintenant que le service par défaut permette d'exécuter 4 sessions courantes, pour bien montrer au client que les performances sont excellentes.

Q 1.5 : Quel impact sur vos tests ? Quel diagnostic faites-vous ? Proposer une nouvelle architecture qui permet de faire varier le nombre de sessions acceptées sans pour autant obliger à maintenir les tests qui ne devraient pas être concernés. □

Exercice 2 : Les emprunts dans une bibliothèque

On s'intéresse aux emprunts et retour de livres dans une bibliothèque. Certains livres ne sont que consultables.

Lors de l'emprunt les membres de la bibliothèque s'identifient. Chaque membre peut emprunter au maximum 20 livres empruntables pour une durée de 30 jours. L'« emprunt » créé est stocké dans une base de données persistante. Lors de l'emprunt les caractéristiques de l'emprunt sont affichées à l'écran.

Q 2. 1 : On souhaite tester la classe qui gère le service des emprunts, pour la fonctionnalité « emprunter un livre ». Quelles responsabilités identifiez-vous ? Quelles sont celles qui sont du ressort du service des emprunts ? Quels cas proposez-vous de tester ? □

Q 2.2 : Tester et écrire en TDD la fonction `emprunter` en traitant uniquement la création de l'emprunt. Quel pattern avez-vous utilisé ? □

Lors du retour d'un livre le membre de la bibliothèque n'a pas à s'identifier. L'emprunt correspondant au livre est clôturé. Si le membre a dépassé la durée de 30 jours alors le retard est signalé au service litiges.

Q 2.3 : On souhaite tester la fonctionnalité « rendre un livre ». Quels cas proposez-vous de tester ? □

Q 2.4 : Tester et écrire en TDD la fonction `rendre`, en déléguant à l'emprunt la responsabilité du calcul « en retard ou pas ». □

Q 2.5 : On souhaite tester et écrire en TDD la classe représentant un emprunt. Quelles fonctionnalités, quels comportements ? □

Pour mémoriser la date de l'emprunt et pour vérifier son éventuel retard lors du rendu du livre, on a besoin de manipuler des dates (notamment la date du jour). Regardez la doc de la classe `date` du module `datetime` : méthode `today`. Regarder aussi la doc de la classe `timedelta` du module `datetime`. On a par ex :

```
>>> auj = date.today()
>>> auj
datetime.date(2017, 2, 12)
>>> auj + timedelta(days=7)
datetime.date(2017, 2, 19)
```

Q 2.6 : Imaginons qu'en oubliant momentanément de développer en TDD, on écrive ce constructeur pour la classe `Emprunt` :

```
def __init__(self, livre, membre):
        ...
        self.date = date.today()
        ...
```

Cette conception produit-elle un code testable ? Pourquoi ?
NB : pour les adeptes du contournement, `mockito` ne permet pas de mocker les méthodes `builtin` et donc n'accepte pas un `when(datetime.date).today().thenReturn(...)` □

Q 2.7 : Que proposez-vous ? Si vous n'avez pas d'idée il est possible d'ajouter dans votre module une simple méthode `aujourdhui` qui retourne la date du jour et de la mocker :
`when(monmodule).aujourdhui().thenReturn(...)` □

Q 2.8 : Tester et écrire en TDD cette classe. □

Q 2.9 : Écrire un test intégrant le service des emprunts et la classe représentant un emprunt pour tester le cas d'un livre emprunté puis rendu en retard (les autres dépendances restent à l'état de mocks). □