LOG3430 – Méthodes de test et de validation du logiciel

Laboratoire 1

Tests unitaires

Département de génie informatique et de génie logiciel

École Polytechnique de Montréal

Soumis par

Roman Zhornytskiy (1899786)

Hakim Payman (1938609)

Gabriel Tagliabracci (1935775)

Groupe : 02

Soumis à Noureddine Kerzazi

Hiver 2020

1. Mise en contexte théorique

L'objectif des tests unitaires est de valider que chaque unité d'un logiciel sous test fonctionne comme prévu. Par unité on désigne la plus petite composante testable d'un logiciel. Généralement, cette composante possède une ou plusieurs entrées et une seule sortie.

En programmation procédurale, une unité peut être un petit programme, une fonction, une procédure, etc. En programmation orientée objet, la plus petite unité est une méthode qui peut appartenir à une classe de base ou à une classe dérivée. Les tests unitaires ne doivent jamais interagir avec des éléments d'une base de données ou des objets spécifiques à certains environnements, ou bien à des systèmes externes. Si un test échoue sur une machine parce qu'il requiert une configuration appropriée, alors il ne s'agit pas d'un test unitaire.

Comme présenté dans les notes de cours, les Frameworks des tests unitaires, les pilotes "Drivers", les Stubs, Spies et les faux objets "Mocks" sont utilisés pour faciliter les tests unitaires.

Il existe plusieurs Frameworks de tests unitaires spécifiques à chaque langage. Parmi les Frameworks les plus populaires on trouve JUnit pour Java, Unittest et Pytest pour Python, RSpec pour Ruby, Karma, Jasmine, Mocha, Chai et sinon pour JavaScript, etc.

Vous pouvez consulter ces liens pour plus de détails :

- **https://docs.python.org/fr/3.8/library/unittest.html**
- **https://docs.python.org/3/library/unittest.mock.html**
- **https://docs.python.org/3/library/unittest.html#unittest.TestCase.debug**

2. Objectifs

- **Mettre en pratique les connaissances théoriques acquises sur les tests unitaires.**
- **Se familiariser avec les tests unitaires impliquant un accès aux bases de données.**
- **Être en mesure d'analyser des graphes de flot de contrôle**
- **Comprendre les critères de couverture (de conditions, chemin, flot de données).**

3. Mise en contexte pratique

Le code à tester consiste en une application de gestion des contacts. Elle permet de consulter, ajouter, mettre à jour et supprimer des contacts qui se caractérisent par : Id, Nom, Prénom, Téléphone, Email. L'Id est un identifiant unique qui est géré par la base de données SQLite (un entier qui s'auto-incrémente commençant par 1). De plus, la logique d'application considère le couple d'attributs (nom et prénom) comme identifiant unique du contact. De ce fait, on ne peut pas avoir deux contacts avec les mêmes noms et prénoms. En outre, l'application garde un attribut booléen indiquant si le contact est à jour ou pas ainsi qu'un attribut date contenant la date de la dernière mise à jour. Ces deux informations sont utiles pour désactiver les contacts non mis à jour pendant un certain temps (dans notre cas : 3 ans) et le notifier à l'utilisateur.

Structure du code

Dans ce laboratoire, le code est modulaire pour faciliter le développement des tests unitaires pour chaque module séparément.

Contact représente notre modèle pour un contact, avec les attributs suivants (id, first_name, last_name, phone, mail, updated, updated_date). Vous la trouvez dans le fichier *models.py*.

ContactDAO représente l'objet d'accès aux données qui va permettre de se connecter à la base de données, créer la table Contact, exécuter des opérations élémentaires (lire, ajouter, mettre à jour, supprimer une ou plusieurs lignes dans la table des données). Vous la trouvez dans le fichier *DAOs.py*.

ContactService représente la classe service qui rassemble les fonctions métiers qui peuvent être utilisées directement par une interface graphique (par exemple). Elle dépend de la classe ContactDAO pour sauvegarder les traitements dans la base des données. Vous la trouvez dans le fichier *services.py*.

Tests

TestContactDAO est la classe de test qui rassemble les tests unitaires pour les méthodes de la classe ContactDAO[1]. Le premier test est fourni ainsi que les deux méthodes *setUp* et *tearDown* qui s'exécutent respectivement avant et après l'exécution de chaque test unitaire. La méthode *setUp* assure, en premier lieu, la création d'une nouvelle base de données vide et initialise la

[1] Cette classe est dans le fichier DAOs.py

table de données. En deuxième lieu, la méthode *tearDown* supprime la base de données après l'exécution du test. Ces deux opérations vont vous permettre de créer un contexte initial contrôle pour bien écrire indépendamment les tests unitaires. Vous la trouvez dans le fichier *testDAOs.py*. Pour lancer les tests, il vous suffit d'exécuter ce fichier avec .c :> python -m *unittest testDAOs.py*.

TestContactService[2] est la classe de test qui contient les tests unitaires pour les méthodes de la classe ContactService[3]. Le premier test est fourni ainsi que la méthode *setUp* qui va permettre de créer un mock pour l'objet ContactDAO et le connecter à la classe ContactService sous test pour pouvoir isoler la logique de la classe de ses dépendances afin de les tester correctement. Vous la trouvez dans le fichier *testServices.py*. Pour lancer les tests, il vous suffit d'exécuter ce fichier avec python :
c :>*python -m unittest testServices.py*.

4. Travail à effectuer

4.1. **Complétez** la classe *TestContactDAO*. **5 points**

Y a-t-il un test qui a échoué, si oui proposez une correction du code ?

Oui, le test de la méthode delete_by_names échouait tout le temps : elle me donnait des faux positives (le test passait alors que la méthode ne faisait pas son travail). Cela était dû au fait que, dans le code source, la requête SQL qui devait effectuer la suppression était suivi de *connection.close()* au lieu de *connection.commit()*. Il suffit de remplacer le *connection.close()* par *connection.commit()* pour corriger l'erreur.
Voir les fichiers test_DAOs.py et DAOs.py.

4.2. Complétez la classe TestContactService. **5 points**

En suivant la même logique, proposez un ou plusieurs tests unitaires pour la méthode *verify_contacts_status* en l'ajoutant à la classe *TestContactService*.

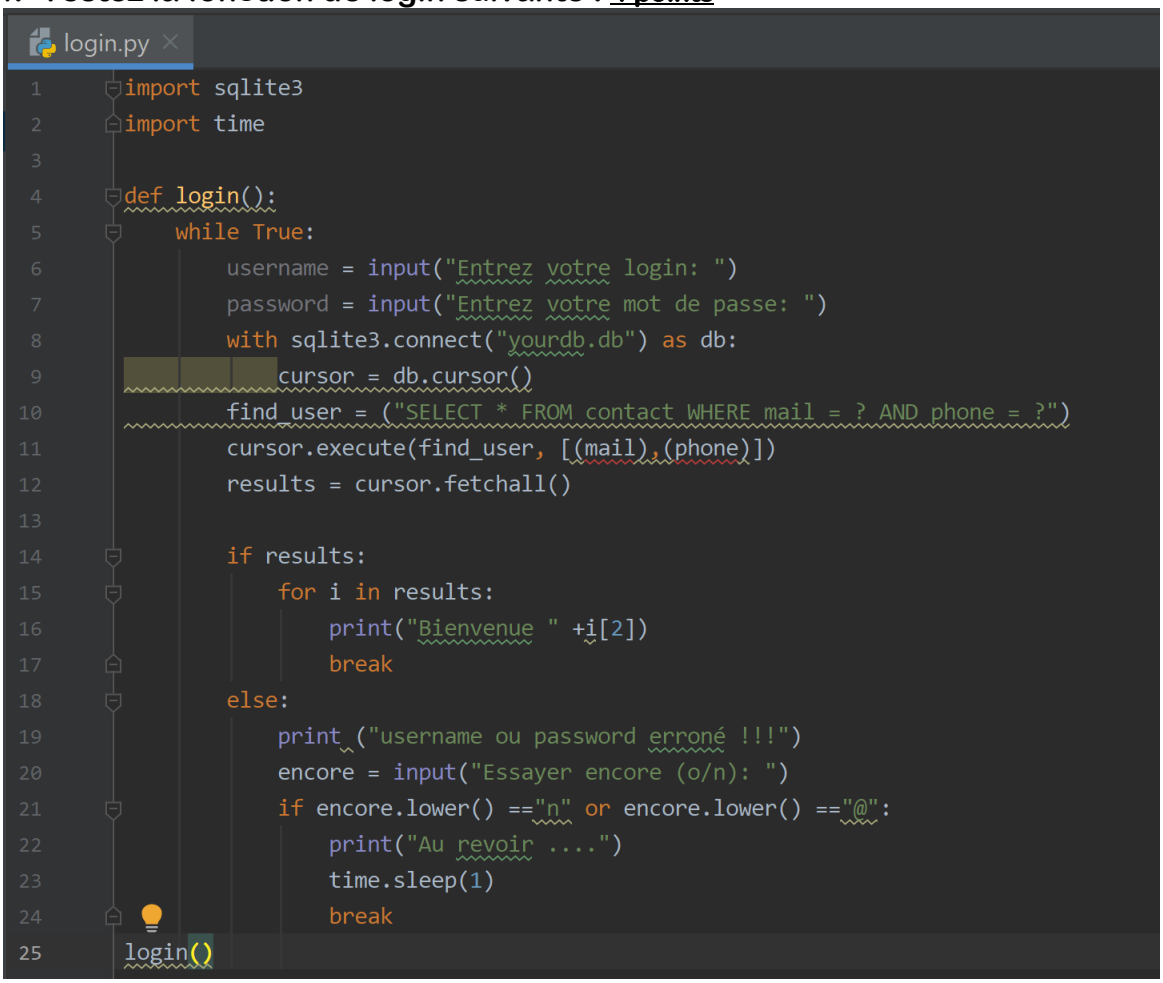
Voir le fichier test_services.py.

4.3. Complétez les deux fonctions de la classe *TestContactService* : *check_phone* et *check_mail*. Puis proposez les tests unitaires adéquats pour les deux méthodes développées. **2 points**

Voir les fichier test_services.py et services.py.

[2] Qui se trouve dans le fichier test_Services.py
[3] Qui se trouve dans la classe services.py

4.4. Testez la fonction de login suivante : **4 points**



```python
import sqlite3
import time

def login():
    while True:
        username = input("Entrez votre login: ")
        password = input("Entrez votre mot de passe: ")
        with sqlite3.connect("yourdb.db") as db:
            cursor = db.cursor()
        find_user = ("SELECT * FROM contact WHERE mail = ? AND phone = ?")
        cursor.execute(find_user, [(mail),(phone)])
        results = cursor.fetchall()

        if results:
            for i in results:
                print("Bienvenue " +i[2])
                break
        else:
            print ("username ou password erroné !!!")
            encore = input("Essayer encore (o/n): ")
            if encore.lower() =="n" or encore.lower() =="@":
                print("Au revoir ....")
                time.sleep(1)
                break
login()
```

Voir les fichiers login.py et test_login.py.

4.5. L'équipe souhaite adopter une approche de développement piloté par les tests. Votre collègue a créé les tests unitaires suivants et vous demande d'ajouter le code nécessaire pour satisfaire ces tests. 4 points

```python
import unittest
from contact import Contact

class TestContact(unittest.TestCase):
    def test_email(self):
        self.cont_1 = Contact('Noureddine', 'Kerzazi', 50000)
        self.cont_2 = Contact('Bram', 'Adam', 60000)

        self.assertEqual(self.cont_1.email, 'Noureddine.Kerzazi@polymtl.ca')
        self.assertEqual(self.cont_2.email, 'Bram.Adam@polymtl.ca')

        self.cont_1.first = 'Noureddine2'
        self.cont_2.first = 'Bram2'
        self.assertEqual(self.cont_1.email, 'Noureddine2.Kerzazi@polymtl.ca')
        self.assertEqual(self.cont_2.email, 'Bram2.Adam@polymtl.ca')

    def test_fullname(self):
        self.cont_1 = Contact('Noureddine', 'Kerzazi', 50000)
        self.cont_2 = Contact('Bram', 'Adam', 60000)

        self.assertEqual(self.cont_1.fullname, 'Noureddine Kerzazi')
        self.assertEqual(self.cont_2.fullname, 'Bram Adam')
        self.cont_1.first = 'Noureddine2'
        self.cont_2.first = 'Bram2'
        self.assertEqual(self.cont_1.fullname, 'Noureddine2 Kerzazi')
        self.assertEqual(self.cont_2.fullname, 'Bram2 Adam')
    def test_apply_raise(self):
        self.cont_1 = Contact('Noureddine', 'Kerzazi', 50000)
        self.cont_2 = Contact('Bram', 'Adam', 60000)
        self.cont_1.apply_raise()
        self.cont_2.apply_raise()
        self.assertEqual(self.cont_1.pay, 52500)
        self.assertEqual(self.cont_2.pay, 63000)

if __name__ == '__main__':
    unittest.main()
```

Voir le fichier models.py.

5. Livrables attendus

Les livrables suivants sont attendus :

- **Un rapport pour le laboratoire avec des réponses sur les questions.**
- **Le dossier COMPLET contenant le projet.**

Le tout à remettre dans une seule archive zip avec pour nom matricule1_matricule2_lab1.zip à téléverser sur Moodle.

Le rapport doit contenir le titre et numéro du laboratoire, les noms et matricules des coéquipiers ainsi que le numéro du groupe.

Consultez le site Moodle du cours pour la date et l'heure limite de remise des fichiers. Un retard de]0,24h] sera pénalisé de 10%, de]24h, 48h] de 20% et de plus de 48h de 50%.