École Polytechnique de Montréal
Département de génie informatique et génie logiciel
LOG1000 – Ingénierie logicielle

Travail pratique #4

Hiver 2020

Introduction aux tests unitaires

1. Mise en contexte théorique

L'objectif des tests est de détecter des défauts, afin d'éviter que les client(e)s aient à subir des défaillances. Une défaillance étant une déviance de la fonctionnalité attendue, qui peut aller d'une faute de frappe dans le texte présenté par le logiciel, jusqu'à un « crash » qui rend le logiciel inutilisable.

Une manière de détecter les défauts est d'exécuter le logiciel, morceau par morceau, afin de voir si chacun de ces morceaux n'en contient pas un. Ce morceau est typiquement appelée une « unité » et peut se limiter à une méthode, une fonction, etc. C'est de là que la pratique prend son nom de « tests unitaires ».

2. Objectifs

Le but de ce travail pratique est de vous initier à l'utilisation d'un cadriciel (*framework*) de tests unitaires, soit CppUnit. Le cadriciel CppUnit était très populaire par le passé. Il a été depuis remplacé par des cadriciels plus faciles à utiliser, mais un peu comme le langage C++, si vous pouvez vous servir de CppUnit, vous pourrez vous servir des autres.

- Comprendre le fonctionnement d'un cadriciel de tests unitaires,
- Apprendre à choisir et rédiger des tests unitaires pertinents afin de répondre à un critère donné,
- Déterminer comment gérer les résultats des tests unitaires exécutés.

3. Mise en contexte pratique

On vous demande de tester un logiciel qui n'a pas de tests unitaires et a malheureusement connu quelques défaillances. Vous êtes toujours en contact avec l'équipe de développement originale qui peut corriger les défauts trouvés mais qui n'a pas le temps de tester le travail fait.
Le logiciel fait la lecture d'un fichier de client(e)s, reçoit une facture et retourne le rabais à appliquer sur la facture, sur la base d'une série de règles d'affaires complexes.

Tant qu'à simplement tester manuellement le logiciel, vous décidez d'en profiter pour ajouter des tests unitaires automatisés afin de pouvoir détecter les défauts plus facilement et plus rapidement. Cela devrait éviter que d'autres défaillances se produisent dans l'avenir.

3.1. Outils nécessaires

L'énoncé de ce TP assume que vous avez les outils nécessaires installés, et que vous utilisez une plateforme Linux. Ces outils sont installés sur la machine virtuelle fournie par le département. Il y a seulement un outil supplémentaire à installer, soient les fichiers d'entête (*header files*) de CppUnit. Sur la machine virtuelle, ces fichiers peuvent être installés avec la commande :

- sudo dnf install cppunit-devel

4. Travail à effectuer

4.1. Exécutez un test unitaire

Dans le code fourni « TP4.zip », il n'y a aucun test unitaire, ni projet pour exécuter les tests unitaires. Ajoutez un projet CppUnit et rédigez un test pour la fonction «Rabais::getRabais(Facture, int)». N'importe quel test peut faire l'affaire : l'objectif ici est d'avoir un projet CppUnit fonctionnel.

Les commandes pour tester doivent être ajoutées au Makefile existant.

Basez-vous sur les exemples et exercices faits en cours pour savoir comment monter un projet CppUnit.

4.2. Rédigez le graphe de flot de contrôle du code

Rédigez le graphe de flot de contrôle (CFG, *Control Flow Graph*) pour le code de la fonction «Rabais::getRabais(Facture, int)». Placez le graphe dans le rapport.

4.3. Rédigez les cas de tests pour la couverture des branches

Rédigez, en utilisant le graphe de flot de contrôle, conceptuellement les cas de tests pour le code de la fonction «Rabais::getRabais(Facture, int)». Les cas de tests rédigés doivent assurer la couverture des branches de la fonction. Voici un exemple de cas de test rédigé de manière conceptuelle :

- d1 = <{input_a=50, input_b=true}, {output_a=true, outputb=[5,8,10]}>

Vous avez cependant besoin des exigences afin de déterminer les valeurs attendues («oracle») de la fonction. Voici donc les exigences pour la fonction «Rabais::getRabais(Facture, int)» :

ID	Exigence
SRS01	Les code-clients qui sont plus grands que 25 000 correspondent à des codes d'employé(e)s.
SRS02	Les employé(e)s doivent avoir accès à un rabais d'employé(e)s de 15%.
SRS03	Les employé(e)s n'ont pas accès à aucun autre rabais que le rabais d'employé(e)s.
SRS04	Les personnes de plus de 55 ans doivent avoir un rabais additionnel de 10%.
SRS05	Un rabais additionnel doit être attribué pour les zones géographiques suivantes, basées sur les trois premiers symboles du code postal. ● Zone H1C : 4% additionnel, ● Zone H3P : 3% additionnel, ● Zone J4O : 2% additionnel.
SRS06	Un rabais additionnel doit être attribué par nombre d'année depuis la date d'adhésion de 2% par 3 années complètes.
SRS07	Le rabais additionnel basé sur la date d'adhésion ne peut pas dépasser 10%.
SRS08	Un rabais additionnel doit être attribué de 1% par tranche de 120$ sur la facture.
SRS09	Le rabais additionnel basé sur la facture ne peut pas dépasser 6%.

Les cas de tests rédigés de manière conceptuelle doivent se trouver dans le rapport.

4.4. Codez les cas de tests dans le projet CppUnit

Codez les cas de tests conceptuels décrits précédemment dans le projet CppUnit. Assurez-vous d'identifier les cas de tests codés avec le même identifiant que les cas conceptuels, afin de faciliter la traçabilité entre vos cas conceptuels et les cas de tests codés.

Notez que vos tests unitaires ne doivent pas re-construire l'ensemble du logiciel. Ils doivent construire le strict minimum pour exécuter la fonction à tester. Vous n'avez pas à rédiger de «stubs» : vous pouvez utiliser les classes existantes et supposer qu'elles ont déjà été testées et qu'elles ne contiennent pas de défauts.

4.5. Trouvez un défaut

Grâce aux tests unitaires que vous avez rédigés, trouvez un défaut dans la fonction testée. Ne corrigez pas le défaut, mais :

- Indiquez la présence du défaut dans le rapport,
- Indiquez dans le rapport quelle exigence n'est pas respectée,
- Suggérez dans le rapport une correction possible du code.

Votre travail se limite à tester. Laissez les développeurs corriger leur travail !

5. Rétroaction

Nous travaillons à l'amélioration continue des travaux pratiques de LOG1000. Ces questions peuvent être répondues très brièvement.

- Combien de temps avez-vous passé sur le travail pratique, en heures-personnes, en sachant que deux personnes travaillant pendant trois heures correspondent à six heures-personnes. Est-ce que l'effort demandé pour ce travail pratique est adéquat ? Répondez dans le rapport.
- Avez-vous des recommandations à faire afin d'améliorer ce travail pratique ? Répondez dans le rapport.

6. Livrable à remettre, procédure de remise et retard

Votre rapport devra être contenu dans un fichier nommé «rapport.pdf», qui doit être soumis à travers votre répertoire Git, dans un dossier appelé TP4 ainsi que votre code. N'oubliez pas de faire «git add», «git commit» et «git push» sur le rapport et votre code à la fin de votre TP pour vous assurer que le tout soit soumis !

Pour effectuer la remise sur git, veuillez indiquer un « tag » sur votre commit final, afin de bien identifier quel commit correspond à la « version finale » de votre travail. Votre tag doit être « TP4 ». N'oubliez pas de push votre tag avec **git push --follow-tags** .

Seule une remise électronique est exigée. **La remise doit se faire sur Git, avant le lundi 23 Mars, 12h00**. Si jamais il y avait des problèmes avec votre serveur Git, envoyez votre rapport et code par courriel à votre chargé(e) de travaux pratiques en expliquant la situation. Les courriels des chargé(e)s de travaux pratiques se trouvent sur le site Moodle du cours.

Une pénalité de retard de 10% sera appliquée pour tout rapport remis après la limite. Un 10% additionnel de pénalité sera appliqué par jour de retard additionnel.

7. Barème de correction

Voici le barème de correction utilisé pour évaluer votre TP4 :

Élément	**Pondération**
Projet CppUnit fonctionnel avec commandes ajoutées au Makefile existant.	20%
Graphe de flot de contrôle de la fonction à tester.	20%
Cas de tests rédigés de manière conceptuelle et assurant la couverture des branches.	20%
Cas de tests codés et correspondant aux cas de tests conceptuels.	30%
Au moins un défaut a été trouvé, est présenté dans le rapport et lié avec l'exigence non-respectée, et une correction est suggérée.	10%
Rétroaction (section 5 de l'énoncé).	<aucune>
Format de la remise (présence de tag, mauvais nom de fichier, etc.)	jusqu'à -10%
Total	**100%**