LOG1000: Ingénierie logicielle Processus de développement d'un projet logiciel open source

Travail pratique # 4

Automne 2018

DATE DE REMISE :
Groupe B1 : 23 Novembre 2018 à 23h55
Groupe B2 : 30 Novembre 2018 à 23h55

Objectifs:

- Faire des tests unitaires et du débogage afin de garantir la qualité de votre logiciel.
- S'initier aux outils de tests unitaires et de débogage.

Les outils :

La classe "HashMap.cpp" représente une table de hachage, c.-à-d. une structure qui peut stocker des mappings d'une clef à une valeur. Par exemple, un HashMap avec les entrées (1,"abc") et (2,"def") retournera le string "def" si on donne la clef 2, tandis que rien ne sera retourné lorsque l'on donne la clef 3 (car il n'y a pas d'entrée pour cette clef).

Avant de commencer le TP4, on vous recommande fortement de consulter la page Wikipedia au sujet des tables de hachage: https://fr.wikipedia.org/wiki/Table_de_hachage. Cela explique des notions cruciales pour comprendre le code source fourni.

Enoncé :

Dans le cadre de ce TP, on vous demande de faire des **tests unitaires** pour les fonctions principales de la classe « HashMap » dans le but d'assurer leur bon fonctionnement. Afin de réaliser des bons tests unitaires, il faut créer un test convenable pour chaque chemin indépendant et possible d'une fonction à tester, ce qui revient à élaborer en première partie un diagramme de flot de contrôle (**Control Flow Graph). Ensuite vous allez déboguer** « HashMap » s'il y a des tests échoués.

E1) Diagramme de flot de contrôle [/50]

1. À partir du code source (voir "HashMap"), dessinez les diagrammes de flot de contrôle pour les 3 méthodes « get », « insert » et « delete » de la classe HashMap [/30]
2. Pour chaque diagramme, calculez la complexité cyclomatique. N'oubliez pas de calculer la complexité avec les deux approches vues en classe pour contrôler vos résultats. [/5]
3. Sur chacun des diagrammes, définissez les chemins nécessaires à parcourir pour couvrir toutes les instructions. [/15]

NOTE:

> Un chemin d'exécution est chaque parcours possible. La couverture des chemins implique à la fois la couverture des instructions et la couverture des points de tests.

Vous pouvez faire des dessins avec Powerpoint ou un autre logiciel (par exemple UMLet), puis inclure les diagrammes résultants dans **votre rapport**. Alternativement, vous pourriez aussi utiliser un format textuel pour les chemins. Le plus important est que votre réponse ne contienne pas d'ambiguïtés.

E2) Cas de tests et jeu de données [/20]

Afin d'appliquer vos tests unitaires, vous devez d'abord établir les cas de tests et bâtir un jeu de données. Cette phase n'est que de la conception, aucune implémentation ne doit être faite avec CppUnit à ce point (ça viendra dans E3).

Dans le cadre des méthodes à tester, veuillez définir pour chaque chemin des entrées (des valeurs pour les paramètres de la méthode à tester ou d'autres objets ou variables qui peuvent être manipulés) et la sortie attendue. Par exemple, si on veut tester la méthode suivante en satisfaisant la couverture des branches :

```
int foo (int x ) {
  if (x > 10) {
    return 1;
  } else {
    return 0;
  }
}
```

Des entrées et des sorties attendues seraient par exemple :

Entrées	Sorties
20	1
5	0

Utilisez une notation similaire pour spécifier les jeux de tests correspondant aux chemins identifiés dans E1. Si un certain chemin n'est pas faisable en pratique, mentionnez-le.

E3) Implémentation et exécution des tests unitaires [/30]

Dans cette section, il faut implémenter les tests unitaires qui correspondent aux chemins que vous avez identifiés dans « E1 », et les jeux de tests et les résultats attendus que vous avez énumérés dans « E2 ». Pour cela, il faut compléter les squelettes fournis dans le dossier « tests ». Il contient un fichier source par méthode testée, par exemple le fichier « putTest.h » est utilisé pour tester la méthode « put » de la classe « HashMap ». Pour plus d'information sur CppUnit, on vous réfère vers http://www.yolinux.com/TUTORIALS/CppUnit.html.

1. Implémentez vos cas de tests avec CppUnit. **Votre code source sera vérifié dans cette question, il nous faut voir tous les cas de tests et une implémentation conforme à CppUnit**. Utilisez les 3 fichiers de test, un par fonction à tester, c.-à- d. un pour « get », « insert » et « delete ». [/10]

2. Exécutez les tests unitaires et copiez les résultats qui s'affichent dans l'onglet "output" au rapport. Voici un exemple de ce qui est affiché: [/10]

   Test:: foo *: OK OK (1)*

3. Quels sont les tests qui font défauts? Pour chacun des tests qui font défauts, faites une capture d'écran et identifiez les conditions (chemins) qui causent les défauts (captures d'écran). [/10]

E4) Débogage [/30]

1. Pour chaque bogue, expliquez votre méthodologie, c.-à-d. la séquence d'hypothèses avec discussion comment vous avez essayé de refuter l'hypothèse. Finalement, quel est la cause de chaque bogue? [/10]

2. Réglez chaque bogue en modifiant le code source, ce dernier sera évalué en fonction de sa pertinence pour résoudre les bogues ciblés. [/10]

3. Exécutez les tests unitaires de nouveau, copiez les résultats s'affichant dans l'onglet "Output" au rapport et faites une capture d'écran pour montrer que les défauts ont été résolus. [/10]

E5) Contribution au projet Ring [/20]

Dans la deuxième partie du TP, vous allez contribuer à l'implémentation des tests unitaires du projet open source « Ring ».
Veuillez choisir une méthode du tableau 1 et chercher son implémentation sur le lien suivant: https://github.com/savoirfairelinux/ring-daemon/tree/master/src

Tableau 1 : méthode à tester

Fichier	Méthodes
account_factory.cpp	std::Shared_ptr<Account> createAccount(const char* const accountType, const std::string& id)
account_factory.cpp	void removeAccount(Account& account)
account_factory.cpp	void removeAccount(const std::string& id)
Base64.cpp	std::string encode(const std::vector<uint8_t>& dat)
Base64.cpp	std::vector<uint8_t> decode(const std::string& str)
Smartools.cpp	setFrameRate(const std::string& id, const std::string& fps)
Smartools.cpp	setResolution(const std::string& id, int width, int height)
Smartools.cpp	setRemoteAudioCodec(const std::string& remoteAudioCodec)
Smartools.cpp	setLocalAudioCodec(const std::string& localAudioCodec)
Smartools.cpp	setLocalVideoCodec(const std::string& localVideoCodec)
Smartools.cpp	setRemoteVideoCodec(const std::string& remoteVideoCodec, const std::string& callID)
utf8_utils.cpp	bool utf8_validate(const std::string & str)
utf8_utils.cpp	std::string utf8_make_valid(const std::string & name)

Ensuite répondez aux questions suivantes:

1. À partir du code source, dessinez les diagrammes de flot de contrôle pour la méthode choisie. [/2]

2. Calculez la complexité cyclomatique avec les deux approches vues en classe pour contrôler vos résultats. [/2]

3. Tracez les chemins nécessaires à parcourir pour couvrir toutes les instructions. [/4]

Une fois vous avez fini les questions précédentes, veuillez consulter le lien suivant https://github.com/savoirfairelinux/ring-daemon/tree/99fa0a67ee7e6536df6d04184f67e5c7701a35c8/test/unitTest et examinez les tests qui étaient faits pour cette méthode.

4. Est ce que les tests qui étaient conçues couvrent tous les chemins que vous avez trouvés? Sinon, quels sont-ils les cas de tests manquants? [/2]

Rédaction du rapport :

- Votre rapport sera un **DOCUMENT PDF** contenant les captures d'écran et les réponses aux questions demandées.
- Le rapport sera remis dans un dossier nommé TP4 dans votre répertoire Git. N'oubliez pas de vérifier après la remise si votre rapport est bel et bien visible sur le serveur et pas seulement sur votre copie locale (git ls-files).
- Jusqu'à **10%** de la note peut être enlevé pour la qualité du français, et la présentation du rapport.