LOG1000 – Ingénierie Logicielle – TP4
Tests Unitaires

Objectifs:

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

Les outils :

Vous allez utiliser l'outil de tests « **CppUnit** » pour tester les fichiers sources fournis avec le TP.

Durant la première partie du TP, vous serez ramené à tester les fonctionnalités de la classe « Vector » et de la classe « Stack», qui sont définies dans les fichiers « Vector.h », « Stack.h » et implémentées dans « Vector.cpp », « Stack.cpp ». Le fichier « main.cpp » présente un exemple simple d'utilisation du Vector et du Stack.

En principe, Vector et Stack représentent des structures qui permettent de stocker des éléments d'un type donné. Dans le cadre de ce laboratoire, on considère juste les chaines de caractères (string) comme vecteurs.

Veuillez utiliser le fichier **« Makefile »** pour compiler le code fourni, comme vous l'avez vu dans le TP1.

Enoncé :

Dans le cadre de ce TP, on vous demande de faire des **tests unitaires** pour les fonctions principales de la classe « Vector » et la classe «Stack» 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).**

E1) Diagramme de flot de contrôle [/50]

1. À partir du code source (voir "Vector.cpp" et "Stack.cpp"), dessinez les diagrammes de flot de contrôle pour les 3 méthodes « insert », « get » et « set » de la classe Vector, et pour les 3 méthodes «pop», « top » et «push» de la classe Stack. [/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 conditions. [/15]

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 contient pas d'ambiguïtés.

E2) Cas de tests et jeu de données [/10]

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 :

```
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 [/40]

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 le squelette fourni dans le dossier « tests ». Il contient un fichier source par méthode testée, par exemple le fichier « getTest.h » est utilisé pour tester la méthode « get » de la classe « Vector ». 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, en ajoutant une méthode par chemin dans le fichier source approprié. Par exemple, il faut ajouter la méthode « test3 » dans le fichier « getTest.h » pour tester le 3ème chemin que vous avez identifié dans E1 et E2 pour la méthode « remove » de la classe « Vector ». Il est permis d'ajouter des nouvelles méthodes, mais il ne faut pas changer la signature des classes fournies (leurs noms). Attention, votre code source sera évalué dans cette question, assurez-vous d'écrire un code de bon qualité et bien documenté. [/30]

2. Pour exécuter vos tests, il faut exécuter la commande « make » et ensuite lancer l'exécutable « testsVector ». Exécutez les tests unitaires et copiez les résultats affichés dans votre rapport. [/10]

E5) Contribution au projet Ring [/10]

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 conditions. [/4]

Une fois vous avez fini les questions précédentes, veuillez-vous 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]

ATTENTION

Légende utilisée au cours de ce laboratoire :

- **Le texte en gras** représente des éléments de cours qui doivent être compris pour répondre aux exercices
- Les lignes précédées d'une lettre minuscule (1.) représentent la progression conseillée dans l'exercice
- Le texte en vert représente les questions auxquelles vous devrez répondre textuellement dans vos rapports
- Le texte en rouge représente des consignes à suivre pour assurer le bon fonctionnement des exercices. **Un non-respect de ces consignes entraînera des pertes de points sévères**.

- ❖ Le texte bleu souligné représente des liens vers les ressources disponibles sur Moodle. Il suffit de Ctrl+clic sur ce texte pour y accéder.

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.

DATE DE REMISE :

Groupe 2 : 1 Avril 2018 à 23h55

Groupe 1 : 8 Avril 2018 à 23h55