Contrats et tests unitaires

Note : cas pratique = écrire un test sur travis pour les QCM du cours

Introduction aux tests

Pourquoi tester ?

Quand on créé un programme, on écrit un code pour réaliser une tache particulière. En faut il que le code fasse ce que l'on attend de lui (fiabilité) partout (portabilité) et de façon efficace (performances... qui sont plus des benchmarks). Ce sont des objectif de qualité logiciel.

Types de tests

Un programme peut être vu à plusieurs échelles, allant de la simple instruction qui fait une seule chose à un programme complexe et complet, avec de nombreuses interactions (système, user, fichier, réseau, etc).

instruction → fonction → modules → programme

Il existe en conséquence plusieurs niveaux de tests, selon ce que l'on va tester. On peut tester le comportement d'une simple instruction (pour vérifier par exemple qu'un compilateur suit correctement la norme C++ ou le comportement particulier d'un système), d'une fonction, d'un ensemble de fonctions, d'une classe, d'un module, d'un programme complet. (Ce chapitre s'aux tests des fonctions uniques).

La complexité de création des tests va dépendre la complexité de ce que

l'on veut tester, en particulier le nombre d'interactions de ce que l'on veut tester avec d'autres éléments que l'on ne teste pas (ou tout au moins, pas testé directement). Le test le plus simple (qui ne teste qu'une seule chose) sera appelé test unitaire. (Mais on va voir que ce n'est pas si simple).

Par exemple, si on veut tester la fonction std::sort, on peut créer une collection de valeurs non triées, appeler std::sort dessus et vérifier ensuite que le résultat est correctement trié. On ne teste qu'une seule chose : une seule fonction. (En pratique, ce n'est pas totalement vrai... on va tester aussi, indirectement, la collection, la valeur, le système, etc. Mais on va considérer que c'est unitaire... tant que l'on a pas montré que ce n'est pas unitaire)

Au contraire, pour tester une requête à une base de données en ligne par exemple, il ne sera pas possible de tester cela sans tester aussi le réseau, la connexion à la base de données, etc.

- test unitaire : test une chose (vague...)
- test fonctionnel
- test intégration
- test package : teste la création de paquet (msi, deb, etc.)
- test déploiement : si l'installation sur système vierge permet d'avoir l'application fonctionelle
- etc.

unitaire ? Par exemple tester addition sur une classe que l'on créé BigInt. BigInt a, b, c = a + b; Mais en fait plusieurs choses testé : la construction, l'initialisation, l'addition, l'égalité. Donc pas unitaire. Mais... Si on écrit plusieurs tests (1 pour la construction, 1 pour l'initialisation, 1 pour l'égalité, etc), chaque nouveau test va tester plusieurs choses, mais une seule chose qui n'a jamais été testé avant. Donc on peut considéré comme unitaire (sauf on trouve ensuite que l'on a oublié un test intermédiaire...)

Reproductibilité des tests

Par exemple, fonction qui mélange une liste de valeurs (1, 2, 3, 4, 5). Comment tester cela (ie comment écrire une fonction “is_randomised” ?)

Par exemple, passer une liste triée et vérifier que liste n'est pas triée ensuite. Mais... un liste aléatoire peut prendre n'importe quel ordre... dont la même liste. Si on écrit ce type de test et que l'on répète les tests, le test échouera de temps en temps. Si on veut pouvoir vérifier les échecs, il faut pouvoir reproduire exactement le test qui échoue.

Règle : toujours afficher les valeurs utilisées quand les tests échouent, pour reproduire. Et plus généralement, toutes infos utilent pour reproduire (fichiers, web, etc)

Note pas si simple en pratique. Par exemple, un test utilisant un fichier peut échouer parce que quelqu'un a modifié le système (droit d'accès au fichier, fichier supprimé, etc), un test accédant à une ressource en ligne peut échouer à cause d'un problème réseau, du site temporairement indispo, etc. Si un test échoue, on peut perdre beaucoup de temps à chercher une erreur dans le code alors que l'erreur peut venir d'autre chose.

Pour simplifier le travail de correction des bugs :

- un max d'infos : afficher les infos, voire créer un fichier de log très détaillés, avec tous les messages du système, du réseau, etc. Cf système spécifique de log.
- travailler dans un environnement contrôlé. Utiliser le même système pour faire plusieurs tests, faire plusieurs fois la même série de tests, ou simplement utiliser l'ordi de dev pour faire les tests : on n'est pas sur qu'il n'y a pas d'effet secondaire. Pour corriger : système dédié (machine virtuelle, ordi dedié) et “propre” (réinstalle le système à partir d'une image a chaque tests, docker).

Idem pour les problèmes de version des logiciels/libs, et les libs installées ou non.

Tests automatiques et manuels

Idéalement, tout tester, pour garantir que tout le code est correcte. Mais il faut penser à tous les cas possibles... ce qui n'est pas possible. On ne peut tester que ce a quoi on pense. Et améliorer les tests avec le retours des users.

Donc, même si c'est pas possible de tout tester, on teste beaucoup de choses. Test automatique = vont être réalisé automatiquement et produire un rapport de tests. On pourra se focaliser uniquement sur les tests qui échouent.

Tests manuels = écrire une procédure à réaliser par une personne, qui va devoir reproduire le test manuellement et constater le succès ou non du tests.

Par exemple, le test précédent pour std::sort peut être automatisé. Mais c'est plus difficile de tester un UI automatiquement (on peut difficilement écrire un test qui va dire si la couleur choisit est harmonieuses... si ca fait joli)

Méthodologie de développement

Ecrire des tests n'est pas un but, c'est un moyen d'attendre un objectif : la qualité logiciel. Mais pas suffisant : les tests peuvent passer et que la qualité ne sont pas correcte. Par exemple, on peut oublier des cas à tester (et donc laisser des erreurs).

La démarche qualité ne consiste donc pas à écrire d'un côté du code et de l'autre des tests, mais à penser l'ensemble du processus en un tout cohérent. Il existe plusieurs méthodologies, avec leurs avantages et défauts, le but de ce cours n'est pas de toutes les voir, on va se focaliser sur une approche : l'intégration continue.

Agile

Rappel historique ?

Pour résumé, processus itératif (en général court), incrémentale et adaptative.

- itératif = travail organisé en cycle qui vont se reproduire, à plusieurs échelles
- incrémental = chaque itération ajoute au résultat final
- adaptatif = on corrige à chaque itération

chaque cycle suit une démarche similaire, que l'on va reproduire à chaque fois. Pour respecter l'adaptatif, il faut au moins :

- définition des objectifs au début du cycle (voir évaluation des cycles précédents)
- réalisation
- vérification que les objectifs sont atteins, voir ce qui a empêcher le succès, les points forts à améliorer, les points faibles à corriger, etc.

Plusieurs échelles de cycles :

- cycle des versions de logicielles (majeur ou mineur). Sur plusieurs mois ou années, avec nouvelles grosses features du soft
- cycles interne des versions (alpha, beta, RC, etc) : mois
- sprints : quelques semaines

écrire des issues de plus en plus détaillées, jusqu'à arriver à des issues correspondant à des taches unitaires pour le dev.

Intégration continue

A chaque modification du code :

- vérifier que respect les nouveaux tests
- anciens tests n'échouent pas

Vérifier le plus souvent possibles, pour éviter que chaque modification n'apporte pas une régression.

- compiler : vérifier que le code est correcte (au niveau syntaxe). Activer les warnings. Voir analyse statique du code.
- lancer les tests. Vérifier que les tests passent. Les nouveaux tests, pour vérifier que les modifications font ce que l'on attend d'elles, les anciens tests pour vérifier qu'il n'y a pas de régression
- partager le code

Mais compiler et tester tout prend du temps, surtout avec le problème d'environnement clean cité avant. Pas possible de faire un test complet à chaque ligne de code modifié. Plusieurs niveaux de vérifications :

- IDE : les IDE modernes vérifient en cours de frappe les erreurs de syntaxes. Premier niveau d'aide et correction.
- local : sur son propre ordi, lancer la compilation (simple build, pas rebuild all) et les tests (au moins les tests correspondant à la fonctionnalité sur laquelle on travaille)
- sanity build. Quand on a fait écrit le code correspondant à une issue :
  - créer une branche
  - commit ses modifs sur cette branche (gestion de versions de code partagé, github)
  - envoyé au serveur
  - lancer un build simple
  - lancer les tests simples (par exemple que les unitaires)
  - merger si ok
- nightly build : plus complet et plus régulier, la nuit en général, quand plus personne ne commit
  - clean system
  - rebuild complet

    - tous les tests, en particulier unitaie, fonctionnels, analyse static et dynamique, benchmars, packetage, deploiement, etc
- build + tests encore plus poussé avant de sortir une version de dev (alpha, beta, RC) ou public (version majeur ou mineure)

Existe des serveurs de CI en ligne, en particulier pour les projets open source : travis, ci-appvery...

Développement dirigé par les tests

TDD

Les tests expriment ce le comportement que l'on veut obtenir. On décrit ces comportements (SP), avec le client si nécessaire, on implémente les tests, puis on écrit le code qui réalise ce comportement.

1. permet de ne pas oublier de faire des tests (les devs n'aiment pas écrire des tests en général, donc on commence par le plus casse pied)

2. c'est tout :) Ou pas...

Programmation par contrats

Fonction = prend des arguments en entrée, réaliser une tache et retourne un résultat. Contrat entre celui qui écrit une fonction et celui qui l'utilise : “si tu me donnes telles valeurs en entrée, je garantie que ma fonction fait telle chose et retourne telle valeur”.

- pré-conditions : critères de validité des valeurs en entrée
- post-conditions : critères de validité des valeurs en sortie
- (invariant : cas spécifique des classes, vu plus tard)

Pureté fonctionnelle : fonction n'interagit avec l'extérieur que via ses entrées et sortie. Pas de variable externe (global, static) ou d'indirection. Plus facile à tester, mais souvent pas le cas (cas particulier : std::cout, std::cin, etc)

post-condition

Test plus simple à écrire, puisque on peut tester directement ces conditions

Dans un TU, on appelles la fonction avec des valeurs valides et on regarde le résultat

quelles valeurs ? quels cas particuliers tester ?

pré-condition

critère sur les valeurs entrées, vérifier que le code utilisateur (dans le programme) respecte bien ces conditions : assert en début

assert vs exception. Contrat fort vs faible (value, at et [] de QVector)

Egalement possible d'écrire des TU pour vérifier que les valeurs sont bien vérifiées (par exemple, si une fonction prend une valeur < 10 en entrée, appeler la fonction avec 11 pour vérifier si la fonction testes bien ses préconditions. Si ce n'est pas le cas, la fonction pourrait être mal utilisée sans qu'on le sache).

testabilité

facilité à écrire un test, à être testé. Plus un comportement sera bien spécifié, bien isolé et que les résultats attendus en fonction des entrées données sont bien connus, plus il sera facile d'écrire des tests.

`void sort_or_suffle(collection, bool)` ⇒ nécessite de tester 2 comportements 2 fonctions (sort + suffle) ⇒ 1 comportement à tester à chaque fois, plus simple

Couverture des tests

= vérifie tout le code est testé

mock/stub

==⇒ POO