

Méthodes et Outils pour la Conception Avancée

Castel Antonin, Reboul Paul, Vandendorpe Thomas

April 13, 2018

Contents

1	Introduction	2
2	Modularité, maintenabilité, réutilisabilité	3
2.1	Titre	3
2.2	Titre	3
3	Qualité des tests	4
3.1	Les tests unitaires	4
3.1.1	Exemple de test	4
3.1.2	Couverture	6
3.1.3	Exemple de couverture	6

Introduction

Ce dossier rassemble les travaux effectués lors du cours de “Méthodes et Outils pour la Conception Avancée”. L’objectif ici, est de montrer les bonnes pratiques à avoir lors de la conception d’un programme. Nous illustrerons ces bonnes pratiques à partir de l’exemple d’un programme de puissance 4 (en langage C), initialement codé de façon peu rigoureuse, que nous tenterons d’améliorer tout au long de ce dossier. Par améliorer, on entend la maintenabilité, la réutilisabilité et la documentation du code, la qualité et la couverture des tests, la détection de défauts et d’erreurs, l’analyse et l’amélioration des performances, ainsi que l’analyse de la vulnérabilité à certaines attaques. Nous expliquerons les différentes méthodes et outils permettant ces améliorations...

Modularité, maintenabilité, réutilisabilité

Titre

...

Titre

...

Qualité des tests

On ne peut pas parler de qualité du logiciel sans parler des tests. Bien que ne prouvant pas l'absence de bug dans un code, les tests exécutés sur un programme sont de bons indicateurs sur sa qualité. Il est nécessaire de “penser tests” tôt dans la création d'un logiciel, car les tests sont écrits au fur et à mesure de la conception et sont destinés à être exécutés plusieurs fois, ceci afin de vérifier que les modifications et ajouts dans le code permettent toujours de valider les tests précédents. Et ceci est d'autant plus vrai lors d'un projet en équipe où les modifications des uns peuvent affecter les modifications des autres. Nous allons dans cette partie présenter deux méthodes permettant d'assurer une certaine rigueur dans les tests: l'une au sujet des tests unitaires et la seconde concernant l'analyse de couverture.

Les tests unitaires

Les tests unitaires sont des tests s'appliquant à une partie d'un programme et permettant de vérifier le programme par bloc. Un bloc de programme est en faite une fonction, et l'idée est d'écrire un test qui appelle une certaine fonction avec certains paramètres (judicieusement choisis si possible, c'est à dire couvrant le plus de branches possibles) et de vérifier que le résultat renvoyé par la fonction est bien le résultat attendu. Il est possible d'écrire ces tests en suivant un processus de développement dirigé par les tests, qui consiste à écrire les tests avant d'écrire la fonction correspondante. Mais on peut également les écrire si on dispose déjà de la fonction à tester en utilisant sa spécification (qui devrait être facilement accessible si le code a été documenté avec un outil adéquat), ce qui arrive si l'on est amené à retoucher un code existant.

TODO: Nécessiter de prendre en compte/modifier des fonctions pour permettre le test unitaire (valeur retour etc...) + découpage code en fonction

Exemple de test

Illustrons ce principe de tests unitaire à l'aide d'un exemple tiré d'un programme de puissance 4 (en langage C). Nous souhaitons écrire un test pour la fonction suivante:

```

int checkfull(Board board);

/*
 * Verifie si une grille (Board) est pleine
 *
 * Parameters
 *   board: une table de jeu
 *
 * Returns
 *   0 si la grille est pleine, 1 sinon
 */

```

Il existe de nombreux frameworks permettant l'automatisation des tests unitaires, dans cet exemple, on utilise CuTest. Il peut être intéressant de remarquer (en gras) l'assignation des variable `actual` (valeur renvoyé par la fonction testée), de `expected` (valeur attendue) et leur comparaison (qui fonctionne un peu comme une assertion).

```

void Test_checkfull(CuTest *tc){
    int i,j,expected,actual;
    Board b=boardVide(15,22);
    for (i=0;i<b.height;i++){
        for (j=1;j<=b.width;j++){
            actual = checkfull(b);
            expected = 1;
            CuAssertIntEquals(tc, expected, actual);
            board_put(&b,j);
        }
    }
    actual = checkfull(b);
    expected = 0;
    CuAssertIntEquals(tc, expected, actual);
}

```

Les tests écrits de cet façon pourrons ensuite tous être exécuter en même temps et des informations sur les échecs des tests seront disponibles:

```

.....F.....

There was 1 failure:
1) Test_checkfull: src/CuTestTest.c:221: expected <0> but was <1>

!!!FAILURES!!!
Runs: 17 Passes: 16 Fails: 1

```

```

.....

OK (17 tests)

```






Comme les tests sont ensuite facile à relancer, il est important de les réexécuter à chaque changement important dans le code afin d'évité la régression.

Couverture

La couverture est un indicateur permettant de savoir quelles parties du code ont été couvertes par les tests. Utiliser la couverture seule n'est pas forcément intéressant, puisqu'elle n'indique pas si les tests ont été validés ou non. En revanche, en combinaison des tests unitaire vu précédemment, on a un indicateur de la qualité des tests très puissant puisqu'on a l'information du pourcentage du code tester et du nombre de tests validés.

Exemple de couverture

Nous avons utilisé l'outil gcov afin d'observer la couverture de notre puissance 4. Comme ce dernier ne donne qu'un résultat textuel de la couverture, nous avons complété son utilisation avec l'outil lcov, qui utilise ce que gcov a produit mais rajoute une interface html afin que la couverture soit plus facile à lire. On peut ainsi observer le pourcentage de lignes/fonctions visitées par les tests:

Filename	Line Coverage ↕	Functions ↕
actions.c	 100.0 % 39 / 39	100.0 % 2 / 2
appli.c	 93.8 % 91 / 97	100.0 % 1 / 1
board.c	 90.4 % 206 / 228	100.0 % 15 / 15
ia.c	 93.9 % 77 / 82	100.0 % 4 / 4
score.c	 100.0 % 45 / 45	100.0 % 5 / 5

En plus des pourcentage de couverture, on peut également directement voir sur le code ce qui a été couvert(en bleu) et ce qui ne l'a pas été(en rouge). Cela est pratique lorsqu'on écrit des tests en essayant de passer par le plus de branches possibles (conditions,boucles,...) du programme.

```
20 :         fseek(file,start,SEEK_SET);
20 :         highscoresTag='0';
20 :         highScores=0;
780 :         while(fscanf(file,"%c", &c7)!=EOF){
760 :             fscanf(file,"< Highscores >%d < / Highscores %c", &highScores, &highscoresTag);
760 :             if( (highscoresTag=='>') && (highScores>=4) && ftell(file)<End ) {highscoresflag=1; break;}
20 :         }
20 :         if(width==0 || widthflag==0){
0 :             width=7;
0 :             printf("Incorrect file format, default value of width (7) is loaded\n");
20 :         }
20 :         if(height==0 || heightflag==0){
0 :             height=6;
0 :             printf("Incorrect file format, default value of height (6) is loaded\n");
20 :         }
20 :         if(highScores==0 || highscoresflag==0){
0 :             highScores=5;
0 :             printf("Incorrect file format, default value of highscore list (5) is loaded\n");
20 :         }
20 :         fclose(file);
20 :         (*w) = width;
20 :         (*h) = height;
20 :         (*hs) = highScores;
20 :     }
```