INF3430 – Méthodes de test et de validation du logiciel

TP No. 3

Groupe 1 :

1437737 – Maxime Jacob

1539437 – Dan Vatnik

1776903 – Cédrick Busque

Présenté à :

Mlouki, Ons

Polytechnique Montréal

27 février 2017

Retour sur TP1

Durant le TP1, il fallait implémenter un module qui était en mesure de faire une addition, soustraction, division et une multiplication sans utiliser les opérateurs usuels +, -, * et /. Ce module devait contenir entre 2 et 5 classes en plus des interfaces pour les classes.

Il fallait aussi fournir un module de gestion de pile qui permettait d'empiler et de dépiler un élément à celle-ci. En plus, nous devions être capable de vérifier si la pile était vide, quelle était la valeur à sa tête et sa longueur. Finalement, nous avions ajouté une fonction qui nous permettait de vider la pile.[1]

Retour sur TP2

Dans le TP2, nous avons construit des classes d'équivalence valides pour toutes nos variables de test. Nous avons établi quatre variables de test : équation, nPair, nImpair et longueur. Pour la variable *équation*, nous avons établi 4 classes d'équivalence : *add*, *soust*, *mult* et *div.* Elles correspondent aux opérations que nous pouvons effectuer. Pour la variable *nPair*, nous avons établi 3 classes d'équivalence : en entier pair entre la valeur minimal d'un entier (int) et 0 exclusivement, 0 et un entier pair entre 0 exclusivement et la valeur maximale d'un entier (int). Pour la variable *nImpair*, nous avons établi 2 classes d'équivalence : un entier impair entre la valeur minimale d'un entier (int) et 0 exclusivement, 0 et un entier impair entre 0 exclusivement et la valeur maximale d'un entier (int). Finalement, nous avons établi une seul classe d'équivalence pour la variable longueur qui est un entier entre 2 et 10 inclusivement.

À partir de ces classes d'équivalence, nous avons établi deux suites de tests suivant la partition EC(Each Choice) et AC(All Combinations). La première partition a généré 4 cas de test et la deuxième partition a généré 24 cas de test.

Rapport TP3

Tests EC

La partition de tests EC conçue durant le TP2 a généré 4 cas de tests. Cette suite de tests a permis une couverture de code de 75.4% de la classe SuitePImpl et de 73.0% de la classe CalculatorImpl.

Pour arriver à une couverture de la plus complète possible il fallait ajouter des cas de test en utilisant la technique de tests en boite blanche. Pour arriver à une couverture complète, il fallait s'assurer que les tests parcourent toutes les branches possibles des méthodes des classes. Jacoco nous aidait à y arriver en nous permettant de voir quelles lignes n'étaient pas parcourues par nos tests. Les tests qui nous ont permis d'atteindre la couverture optimale sont les suivants :

public void testSuitePEC5() -> PileImpl pile = (PileImpl)suite.build("div", 14, 7, 11);

Ce test couvre la partie du code qui vérifie que la longueur est inférieure ou égale à 10.

public void testSuitePEC6() -> PileImpl pile = (PileImpl)suite.build("div", 13, 7, 3);

[1]Information pris directement sur https://moodle.polymtl.ca/pluginfile.php/372649/mod_resource/content/2/TP1_log3430%20%281%29.pdf

Ce test couvre la partie du code qui vérifie que le deuxième paramètre de build() est pair.

public void testSuitePEC7() -> PileImpl pile = (PileImpl)suite.build("div", 14, 6, 3);

Ce test couvre la partie du code qui vérifie que le troisième paramètre de build() est impair.

public void testSuitePEC8() -> suite.printPile(pile);

Ce test couvre la partie du code qui vérifie que le print() dépile la pile.

public void testSuitePEC9() –> PileImpl pile = (PileImpl)suite.build("div", 14, 13, 6);

Ce test couvre la partie du code qui vérifie la division par 0.

public void testSuitePEC10() -> PileImpl pile = (PileImpl)suite.build("Syskey", 14, 13, 6);

Ce test couvre la partie du code qui vérifie que l'opération est valide.

public void testSuitePEC11() -> PileImpl pile = (PileImpl)suite.build("div", 14, 13, 0);

Ce test couvre la partie du code qui vérifie que la longueur est supérieure ou égale à 2.

public void testSuitePEC12() –> PileImpl pile = (PileImpl)suite.build("mult", 14, -1, 3);

Ce test couvre la partie du code qui multiplie un nombre positif par un nombre négatif. Ce test ne passe pas. Il y a une erreur dans le code.

public void testSuitePEC13() -> PileImpl pile = (PileImpl)suite.build("div", -14, -1, 3);

Ce test couvre la partie du code qui divise un nombre négatif par un nombre négatif.

public void testSuitePEC14() -> PileImpl pile = (PileImpl)suite.build("div", 14, -1, 3);

Ce test couvre la partie du code qui divise un nombre positif par un nombre négatif.

public void testSuitePEC15() –> PileImpl pile = (PileImpl)suite.build("div", 2, 3, 3);

Ce test couvre la partie du code cryptique qui aurait requis un commentaire dans le code (CalculatorImpl.java , ligne 54-55).

public void testSuitePEC16() -> PileImpl pile = (PileImpl)suite.build("add",6, -3, 5);

Ce test couvre la partie du code qui additionne un nombre négatif à un nombre positif.

public void testSuitePEC17() -> PileImpl pile = (PileImpl)suite.build("soust",-6, -3, 5);

Ce test couvre la partie du code qui soustrait un nombre négatif à un nombre négatif.

En ajoutant ces tests au EC déjà présent, nous avons pu aller chercher une couverture de 98.4% pour CalculatorImpl et 98.4%(même pourcentage) pour SuitePImp. Nous n'avons pas réussi à aller chercher le 100% pour SuitePImpl car il y a un switch case qui ne nous permet pas de cascader à travers celui-ci. Pour ce qui est de CalculatorImpl, c'est à cause d'un booléen qui ne peut être que faux à la line 41.

Test AC

La partition de tests AC conçue durant le TP2 a généré 24 cas de tests. Cette suite de tests a permis une couverture de code de 77.8% de la classe SuitePImpl et de 98.4% de la classe CalculatorImpl.

Pour arriver à une couverture de la plus complète possible il fallait ajouter des cas de test en utilisant la technique de tests en boite blanche. Pour arriver à une couverture complète, il fallait s'assurer que les tests parcourent toutes les branches possibles des méthodes des classes. Jacoco nous aidait à y arriver en nous permettant de voir quelles lignes n'étaient pas parcourues par nos tests. Les tests qui nous ont permis d'atteindre la couverture optimale sont les suivants :

public void testSuitePAC25() -> PileImpl pile = (PileImpl)suite.build("div", 14, 7, 11);

Ce test couvre la partie du code qui vérifie que la longueur est inférieure ou égale à 10.

public void testSuitePAC26() -> PileImpl pile = (PileImpl)suite.build("div", 13, 7, 3);

Ce test couvre la partie du code qui vérifie que le deuxième paramètre de build() est pair.

public void testSuitePAC27() -> PileImpl pile = (PileImpl)suite.build("div", 14, 6, 3);

Ce test couvre la partie du code qui vérifie que le troisième paramètre de build() est impair.

public void testSuitePAC28() -> suite.printPile(pile);

Ce test couvre la partie du code qui vérifie que le print() dépile la pile.

public void testSuitePAC29() -> PileImpl pile = (PileImpl)suite.build("Syskey", 14, 13, 6);

Ce test couvre la partie du code qui vérifie que l'opération est valide.

public void testSuitePAC30() -> PileImpl pile = (PileImpl)suite.build("div", 14, 13, 0);

Ce test couvre la partie du code qui vérifie que la longueur est supérieure ou égale à 2.

En ajoutant ces tests au AC déjà présent, nous avons pu aller chercher une couverture de 98.4% pour CalculatorImpl et 98.4%(même pourcentage) pour SuitePImp. Nous n'avons pas réussi à aller chercher le 100% pour SuitePImpl car il y a un switch case qui ne nous permet pas de cascader à travers celui-ci. Pour ce qui est de CalculatorImpl, c'est à cause d'un booléen qui ne peut être que faux à la line 41. Comme on peut remarquer, il a fallu beaucoup moins de tests supplémentaires pour arriver à la même

couverture que pour le EC. Cependant, le nombre de tests total est plus grand pour le AC car plusieurs tests se recoupent.