/Users/Nicolas/Documents/Etudes/Automne 2014/LOG2410/Travail personnel/PagePresentation/LogoPolytechnique.pdf

LOG3430

Méthodes de test et de validation du logiciel

Travail pratique 3

Section N°1

Nicolas Richard – 1681198

Adrien Budet – 1721823

David Kanaa – 1816481

Polytechnique Montréal

19 octobre 2015

Corrections et ajustements suite au dernier travail

Si certes, nous avions déjà, lors de la remise du précédent travail, effectué les changements au code source de manière à respecter la structure indiquée, nous avons toutefois quand même décidé d'apporter quelques changements supplémentaires à nos tests unitaires. En effet, tenant compte de la myriade de cas de tests générés (plus de 3000) dans le cadre du test *AC*, nous avons revu nos spécifications de tests de manière à réduire drastiquement le nombre de cas de test en ce qui a trait au constructeur de *SuiteChainee*. Ainsi donc, vous retrouverez ci-dessous la nouvelle spécification de test :

Tableau 1 - Spécification des tests du constructeur

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Paramètre | Catégorie | Choix | Propriété | Contrainte |
| Chemin | Existant | Nom d’un fichier | C |  |
| Inexistant | Chaîne vide |  |  |
| Opérateur | Valide | Addition | V |  |
| Soustraction | V |  |
| Multiplication | V |  |
| Division | V |  |
| Invalide | Chaîne vide | [Erreur] | |
| Valeurs | Identiques | {4, 4} |  | Si V |
| Différentes | {8, 3} |  | Si V |
| Taille | Hors bornes | 1 | [Erreur] | |
| 11 | [Erreur] | |
| Dans les bornes | 2 |  | Si V |
| {3-9} |  | Si V |
| 10 |  | Si V |
| EtatVide | Vide | true |  | Si V |
| Non-vide | false |  | Si V, C |

Aussi, nous avons ajouté des suites de tests JUnit réunissant les tests de type *boîte* noire selon leur critère respectif de manière à pouvoir obtenir un seul rapport *JaCoCo* par critère.

Analyse de la couverture des branches des tests type *boîte noire*

D’abord, il convient de mentionner que la couverture des tests de type *boîte noire* n’est pas complète et ce, autant pour ceux répondant au critère *EC* que ceux répondant au critère *AC*. En effet, nous n’avions pas testé l’ensemble des méthodes, car nous testions uniquement les spécifications de l’énoncé. Du coup, dans le cas de la classe *SuiteChainee*, seul le constructeur et les méthodes qu’il appel furent testées. Toutefois, en ce qui a trait aux classes implémentant l’interface *Calculator*, leur couverture fut complète, ce qui nous retire le fardeau de devoir écrire des tests de type *boîte blanche* supplémentaires en vue de les tester.

Par ailleurs, si l’on jette un coup d’œil attentif aux instructions n’ayant pas été couvertes, nous retrouvons parmi celles-ci de nombreuses instructions de type *finally* imbriquées dans des instructions *catch*. Ainsi, il est possible d’affirmer qu’une couverture à 100 % de notre module est impossible, car ces instructions ne sont appelées qu’en vertu de facteurs extérieurs au code de notre logiciel.

Conception des tests de type *boîte blanche*

Puisque l’énoncé spécifie que l’on doit couvrir l’ensemble des branches de nos méthodes, le critère de couverture des arrêtes et des conditions nous semble tout indiqué. Si certes, nous aurions pu opter pour une couverture intégrant seulement les arrêtes, nous nous sommes rendus compte, après avoir porter une plus minutieuse attention aux rapports générés par *JaCoCo* que la couverture des branches nécessitait la couverture des conditions.

Ainsi donc, à partir des rapports fournis par *JaCoCo,* nous avons identifiés les portions de code n’ayant pas été couvert par les tests actuels et avons tout simplement sélectionné des ensembles de données de tests permettant d’atteindre ces portions de code. Par la suite, nous avons écrit, pour chaque ensemble de données, une méthode de test spécifique dont le contenu est plutôt similaire aux tests de type *boîte noire* effectués plus tôt. À ce titre, il convient d’admettre que l’implémentation de ces tests diffère grandement de celle utilisée dans le précédent travail, puisque nous avions une seule méthode de test au sein de laquelle nous injections, depuis un mécanisme externe, les données de test. Ce choix de conception fut strictement pris de manière à respecter les exigences de l’énoncé.

Par ailleurs, puisque l’ensemble contenant les ensembles de données de test générés par le critère *EC* est un sous-ensemble de son équivalent généré par le critère *AC*, nous n’avons pas vu la pertinence de réaliser des tests de type *boîte blanche* spécifiques aux branches non-couvertes par le critère *AC*, puisqu’elles étaient déjà couvertes par les tests de type *boîte blanche* réalisés de manière à couvrir les lacunes des tests issus du critère *EC*. De même, en vue de minimiser le plus possible le jeu de test requis et sachant que la classe *SuiteChainee* utilise la classe *ListeChainee*, nous avons choisis nos cas de test de *SuiteChainee* de manière à couvrir le plus possible *ListeChainee*. Ainsi, nous évitons d’écrire des tests inutilement. Toutefois, dans certains cas, il était impossible de tester *ListeChainee* au travers de *SuiteChaine*; c’est pourquoi nous avons une classe de tests de type *boîte blanche* dédiée à *Liste Chainee*.

Énumération des tests de type *boîte blanche* effectués

Ci-dessous vous retrouverez une liste des divers cas de tests réalisés dans le cadre de ce travail de manière à rendre complète la couverture des branches. À ce titre, il est important de noter que nous n’avons pas réalisé de tests supplémentaires pour les portions de code étant déjà couvertes par les tests de type *boîte noire*.

**testGetAtHorsBornesAvant**

Permet de tester la branche de la méthode *getAt* lançant une exception.

**testGetAtHorsBornesApres**

Permet de tester la branche de la méthode *getAt* lançant une exception.

**testRemoveItemUnSeulElement**

Permet de tester la branche principale de la méthode *removeItem*.

**testRemoveItemPlusieurs**

Permet de tester les branches de la méthode *removeItem* réassignant les valeurs de tête et de queue de la liste.

**testRemoveAtPremier**

Permet de tester la boucle et les branches de la méthode *removeAt* réassignant les valeurs de tête de la liste.

**testRemoveAtDernier**

Permet de tester la boucle et les branches de la méthode *removeAt* réassignant les valeurs de queue de la liste.

**testSetAtHorsBornesAvant**

Permet de tester la branche de la méthode *setAt* lançant une exception.

**testSetAtHorsBornesApres**

Permet de tester la branche de la méthode *setAt* lançant une exception.

**testRemoveAtHorsBornesAvant**

Permet de tester la branche de la méthode *removeAt* lançant une exception.

**testRemoveAtHorsBornesApres**

Permet de tester la branche de la méthode *removeAt* lançant une exception.

**testRemoveAtBorneZero**

Permet de tester la branche de la méthode *removeAt* traitant les index ayant pour valeur zéro.

**testRemoveAtIntraBornes**

Permet de tester la branche principale de la méthode *removeAt*.

**testRemoveItemInexistant**

Permet de tester la branche principale de la méthode *removeAt* au sein de *SuiteChainee* et la boucle et la condition indiquant l’inexistance de la valeur au sein de la classe *ListeChainee.*

**testReset**

Permet de tester l’unique branche de la méthode.

**testSetAtIntraBornes**

Permet de tester la branche principale de la méthode.

**testSuiteConstructeurHorsBornes**

Permet de tester le cas où la limite est trop grande.

**testMainSuiteValide**

Permet de tester la branche principale de la méthode *main* ainsi que la branche de la méthode *createOperator* sélectionnant l’opérateur de multiplication.

**testSuiteIsValide**

Permet de tester le cas où la suite contient moins de deux éléments et que la méthode *isValid* est appelée.

Analyse de la couverture obtenue à l’aide des nouveaux tests

Avant toute chose, il convient d’admettre que nous ne sommes pas parvenus à obtenir une couverture de 100 %. Si certes, comme discuté plus haut, le code gérant les exceptions ne fut pas couvert, satisfaisant du coup nos attentes, notre échec fut aussi causé par un vice au sein même de *JaCoCo*. En effet, dans le cas des instructions *switch* faisant appel à une variable de condition de type *String*, *JaCoCo* est incapable de repérer les branches superflues créées par la machine virtuelle Java.[[1]](#footnote-1) Du coup, nous ne parvenons pas à obtenir un taux de couverture de 100 % pour la méthode privée *createOperator*, même si en réalité nous avons couvert toutes les branches.

Par ailleurs, notons que la couverture est identiques pour les deux critères.

Discussions à propos des différences entre les critères

Si l’on analyse de près les rapports intermédiaires, nous pouvons remarquer que le critère *AC* est légèrement plus performant en affichant un taux de couverture supérieur de 4 %, ce gain étant attribuable à sa plus grande couverture des méthodes sauvegardant le fichier et sélectionnant l’opérateur de la suite. À cet égard, il n’est guère étonnant que ce critère soit plus performant, car son jeu de données de test est bien plus grand. Toutefois, il conviendra de noter que la grandeur d’un jeu de données n’étant pas élaboré en fonction du code des méthodes ne semble pas linéairement relié au taux de couverture.

Conclusion

Au final, au cours de ce travail pratique, nous avons analyser la couverture donnée par les tests de type *boîte noire* réalisés lors du précédent travail. Par la suite, nous avons, avec de nouveaux tests de type *boîte blanche* étendue la couverture de notre code de manière à obtenir presque 100 % de couverture, tout cela en décelant de nombreuses branches inutiles au sein de notre code qui le complexifiait pour rien.

1. R. Hoffman, Marc. FilteringOptions. *GitHub*. [En ligne]. URL : <https://github.com/jacoco/jacoco/wiki/FilteringOptions>. [Consulté le 18 octobre 2015] [↑](#footnote-ref-1)