/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 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 - 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 | B | Si V, B |
| {3-9} | B | Si V, B |
| 10 | B | Si V, B |
| EtatVide | Vide | true |  | Si V, B |
| Non-vide | false |  | Si V, C, B |

Choix de conception des tests de type *boîte noire*

Classes implémentant l’interface *Calculator*

Si certes il existe de nombreuses similitudes entre les différentes implémentations de l’interface *Calculator* compte tenu de la nature identique des paramètres d’entrée, nous avons tout de même établi les catégories de tests de manière indépendante pour chacune des classes. Au final, nous avons obtenu 6 classes pour chaque test ; il s’agit des mêmes dans chaque cas. À ce titre, vous trouverez ci-dessous les différentes trames de tests :

Tableau 1 - Trames de test des classes *Addition*, *Soustraction* et *Multiplication*

|  |  |  |  |
| --- | --- | --- | --- |
| Paramètre | Catégories | Choix | Propriétés |
| Val1 | Valeur négative | -2147483648 |  |
| {-2147483647, -2} |  |
| -1 |  |
| Valeur nulle | 0 |  |
| Valeur positive | 1 |  |
| {2, 2147483646} |  |
| 2147483647 |  |
| Val2 | Valeur négative | -2147483648 |  |
| {-2147483647, -2} |  |
| -1 |  |
| Valeur nulle | 0 |  |
| Valeur positive | 1 |  |
| {2, 2147483646} |  |
| 2147483647 |  |

Tableau 2 - Trame de test la classe *Division*

|  |  |  |  |
| --- | --- | --- | --- |
| Paramètre | Catégories | Choix | Propriétés |
| Val1 | Valeur négative | -2147483648 |  |
| {-2147483647, -2} |  |
| -1 |  |
| Valeur nulle | 0 |  |
| Valeur positive | 1 |  |
| {2, 2147483646} |  |
| 2147483647 |  |
| Val2 | Valeur négative | -2147483648 |  |
| {-2147483647, -2} |  |
| -1 |  |
| Valeur nulle | 0 | [Erreur] |
| Valeur positive | 1 |  |
| {2, 2147483646} |  |
| 2147483647 |  |

À l’inverse de la catégorie *Valeur nulle* qui ne contient logiquement qu’un choix, nous avons appliqué le principe des valeurs limites dans la sélection des divers choix des catégories *Valeur négative* et *Valeur positive.* S’il est vrai en théorie que les entiers n’ont pas de limite mathématique, cela n’est toutefois pas des entiers de type *Integer* en *Java* qui ont une capacité limite allant de -2147483648 à 2147483647. Si la valeur d’un argument est à l’extérieur de cet intervalle, sa valeur se verra automatique convertie en une valeur dans l’intervalle, d’où le choix des valeurs définissant la capacité limite du type *Integer* à titre de valeurs limites des choix.

Une fois les trames de test réalisées, nous avons écrit les différents cas de tests correspondants à l’aide du cadre d’application JUnit. Pour ce faire, nous avons eu recours aux fonctionnalités de *tests paramétrisés*, permettant d’exécuter le même test plusieurs fois et ce, avec des paramètres d’entrée différents à chaque fois. Ainsi, nous évitons la duplication inutile du code, car nous n’avons qu’à écrire une fois la méthode de test au lieu d’écrire une méthode de test par paire de données de test, ce qui s’avéra immensément utile dans le cas des tests AC.

Constructeur de la classe principale

En ce qui a trait au constructeur de la classe *SuiteChainee*, seule méthode de cette classe à tester dans le cadre de ce travail, nous avons suivis les mêmes techniques que celles appliquées dans les tests de type *boîte noire* des classes implémentant l’interface *Calculator*. Ainsi, nous avons encore une fois eu recours aux *tests paramétrisés* pour implémenter nos cas de tests et ce, pour les mêmes raisons que celles citées plus haut. Évidemment, nous aurions ne pas utiliser cette technique et insérer plusieurs assertions au sein d’une même méthode de test de manière à simuler la technique utilisée. Toutefois, cette approche à comme désavantage de ne pas laisser les autres cas de test être exécutés dans le cas où l’un d’entre eux échoue.

Tableau 3 - Trame du constructeur de la classe *SuiteChainee*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Paramètre | Catégorie | Choix | Propriété | Contrainte |
| Chemin | Existant | Nom d’un fichier | C | Si V, B |
| Inexistant | Chaîne vide | [Erreur] | |
| f4f4 | C | Si V,B |
| Opérateur | Valide | Addition | V, O | Si B, C |
| Soustraction | V, O | Si B, C |
| Multiplication | V, O | Si B, C |
| Division | V | Si B, C |
| Invalide | dsfsf | [Erreur] | |
| Val1 | Valeur négative | -2147483648 |  | Si V, B, C |
| {-2147483647, -2} |  | Si V, B, C |
| -1 |  | Si V, B, C |
| Valeur nulle | 0 |  | Si V, B, C |
| Valeur positive | 1 |  | Si V, B, C |
| {2, 2147483646} |  | Si V, B, C |
| 2147483647 |  | Si V, B, C |
| Val2 | Valeur négative | -2147483648 |  | Si V, B, C |
| {-2147483647, -2} |  | Si V, B, C |
| -1 |  | Si V, B, C |
| Valeur nulle | 0 |  | Si O, B, C |
| Valeur positive | 1 |  | Si V, B, C |
| {2, 2147483646} |  | Si V, B, C |
| 2147483647 |  | Si V, B, C |
| Taille | Hors bornes | 1 | [Erreur] | |
| 11 | [Erreur] | |
| Dans les bornes | 2 | B | Si V, C |
| {3-9} | B | Si V, C |
| 10 | B | Si V, C |
| EtatVide | Vide | true |  | Si V, B, C |
| Non-vide | false |  | Si V, B, C |

Analyse des tests effectués

Le nombre de cas de tests par la méthode Each Choice (EC) est considérablement plus faible que par la méthode All Choices (AC). Par exemple, le test d’un opérateur par la méthode EC demande une dizaine de tests tandis que la méthode AC en demande une cinquantaine. Dès que le nombre de paramètres à tester s’agrandit, la méthode AC peut devenir très gourmande : le test du constructeur demande moins de 10 cas si l’on utilise EC, tandis qu’il demande plus de 4000 tests si l’on utilise AC ! La méthode AC est certes bien plus rigoureuse puisqu’elle teste toutes les combinaisons de paramètres. Mais il convient de s’interroger sur le rapport entre le temps passé à effectuer les tests et le nombre d’anomalies détectées. La méthode EC peut alors fournir un ratio bien plus élevé, tout en permettant de tester convenablement un programme.

Dans un autre ordre d’idées, l’exécution des tests de type *boîte noire* nous a permis de constater une faille dans les spécifications du travail. En effet, si, pour une suite ayant comme opération la division, les deux valeurs initiales sont identiques, la génération de la suite finira par générer une valeur 0 qui sera utilisé comme diviseur, ce qui occasionnera inévitablement une erreur. Or, rien dans la spécification n’indique ce que l’on doit faire dans un tel cas.

Conclusion

Ce TP fût intéressant puisqu’il nous a permis d’appliquer concrètement des principes de tests vus en cours et de nous interroger sur les enjeux du choix d’une méthode de test en particulier. Il nous a également permis de parfaire notre connaissance de la librairie JUnit et ainsi de nous sensibiliser aux problématiques d’automatisation des tests.