Rapport - IFT3913

TP4

André Meneses Paul Godbert

8 décembre 2023

1 Introduction

L'objectif de ce travail est de comprendre et utiliser les méthodes de test logiciel pour tester le projet "Currency Converter", créé par Jérémy Vinceno. Nous nous intéresseront particulièrement aux méthodes suivantes :

— currencyConverter.MainWindow.convert(String, String, ArrayList<Currency>, Double)
— currencyConverter.Currency.convert(Double, Double)

Nous avons répartis nos tests en deux grandes catégories : les tests boite noire et les tests boite blanche. Ce rapport est divisé en deux sections qui couvriront chacune une de ces deux méthodes de test. Nous y discuterons les approches utilisées ainsi que les réflexions liés à chacun des tests utilisés.

2 Tests Boite Noire

Une des choses à faire lorsque l'on teste une ou plusieurs méthode est de vérifier que cette dernière remplis bien toutes les spécifications. Pour cela, on utilise les tests boite noire. Le principe ici est d'écrire des tests permettant de vérifier les spécifications sans connaître la structure du code. on choisis donc des jeux de test pour qu'ils soient les plus représentatifs et efficaces possibles pour détecter toutes erreurs et/ou oublis dans les spécification.

2.1 Currency.convert()

Cette fonction étant la fonction réalisant le calcul de conversion, nous testerons la spécification relative au montant maximum et minimum.

la spécification stipule que cette fonction doit accepter seulement des montants entre 0 et 1 000 000. Nous pouvons donc émettre l'hypothèse que, si cette fonction s'attifait la spécification, elle devrait réaliser correctement la conversion pour des montant dans l'intervalle et lever une exception si les valeurs n'appartiennent pas à l'intervalle.

Pour tester cette hypothèse nous avons créer un jeu de test basé sur l'intervalle [0,1 000 000]. Ce jeu de test contient des valeurs valides et invalides ainsi que des valeurs aux frontières, ce qui nous donne le jeu de test T suivant :

$$T=\{-9000.0, -1.0, 0.0, 500000.0, 999999.0, 1000001.0, 1500000.0\}$$

nous utilisons ainsi deux subsets de ce jeu de test (valeur valides et invalides) pour élaborer deux fonctions de test qui nous permettrons de vérifier que cette fonction correspond bien à la spécification. Dans le premier test, réalisé à partir de valeurs correctes, nous vérifions simplement si la conversion est bien faite. Dans le deuxième, nous vérifions que la fonction lève bien une

exception lorsque les valeurs passées en tant que montant ne sont pas valides.

FIGURE 1 – Résultat des Test Boite Noire

Après avoir exécuter nos test sur le code, on peut se rendre compte (voir ci-dessus) que le test utilisant les fausses valeurs ne passe pas. En effet, la fonction ne renvoie pas d'exception puisqu'elle effectue tout de même le calcul de conversion, même avec des valeurs sensée être source d'erreur. On observe donc ici clairement un non respect de la spécification.

2.2 MainWindow.convert()

Cette fonction requiers plus de tests que la fonction précédente : Cette fois-ci nous devons également tester si les bonnes devises sont prises en compte. La spécification stipule que cette fonction que la fonction doit accepter les devises suivantes : USD, CAD, GBP, EUR, CHF, AUD. Nous pouvons émettre l'hypothèse que, si cette fonction satisfait la spécification, elle ne devrait pas réaliser la conversion si les devises choisies ne font pas partie de l'ensemble de devise valide. On émet donc l'hypothèse que la fonction retourne 0 si la conversion n'est pas possible.

Pour tester cette hypothèse on définis le jeu de test suivant : AUD, EUR, USD, BRL la raison pour laquelle nous avons choisis les devises AUD et BRL est que ce sont des devises moins reconnues mondialement que l'euro ou le dollar américain mais que l'une d'entre elle (AUD) fait partie de la spécification et l'autre (BRL) n'en fait pas partie.

Notre méthodologie de test est le suivante : nous testerons les 4 cas possibles :

— devises correctes et montants corrects
— devises correctes et montants incorrects
— devises incorrectes et montants corrects
— devises incorrectes et montants incorrects

Après avoir exécuter nos test sur le code, on peut se rendre compte (voir ci-dessus) que les tests vérifiant la présence des Dollars Australiens et ceux avec des montants incorrects ne passent pas. On peut en déduire que les spécifications ne sont pas respectées.

3 Tests Boite Blanche

Les tests boite blanche sont un peu différents de leur prédécesseurs : Ils servent à tester la robustesse et la structure du programme. Avec cette méthodologie de test, on choisis les jeux

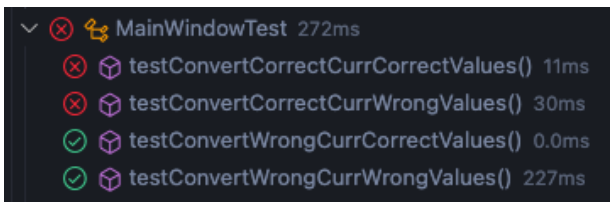


FIGURE 2 – Enter Caption

de tests de manière à tester les plus précisément possible les limites de notre code. Cela nous permet de détecter les éventuelles erreurs commises.

3.1 Currency.convert()

Le code de la fonction `Currency.convert()` ne comporte ni boucles ni conditions, se limitant à des opérations mathématiques simples. De ce fait, les tests de boîte blanche ne sont pas jugés nécessaires dans ce cas.

3.2 MainWindow.convert()

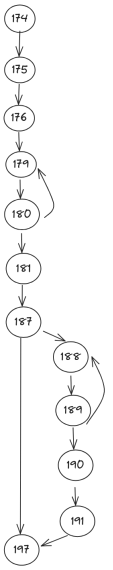
3.2.1 Couverture des instructions

Il est nécessaire de sélectionner un ensemble de tests de manière à ce que chaque instruction soit exécutée au moins une fois. On détermine un point de test d = (String `currency1`, String `currency2`, ArrayList <Currency>`currencies`, Double `amount`). Dans ce contexte, il suffit de disposer d'une liste `currencies` contenant au moins un élément, et que `currency1` appartienne à `currencies`, de même pour `currency2`. De cette manière, le code à l'intérieur des boucles sera activé et les conditions seront satisfaites.

Ainsi, nous avons procédé à la conversion de 100 unités monétaires entre chaque paire de devises présentes dans la liste `currencies`. Pour vérifier la conversion, nous avons contrôlé si la valeur retournée par la fonction `convert()` était différente de zéro. Dans ce cadre, l'utilisation d'un ensemble de données qui couvre toutes les instructions équivaut à tester si des conversions sont calculées pour les devises figurant dans la liste.

3.2.2 Couverture des arcs

Il est d'abord nécessaire de générer le graphe de flux de contrôle, comme indiqué ci-dessous :



Ensuite, il convient de définir un ensemble de données couvrant tous les arcs dans le graphe de flux de contrôle. Cela implique que nous avons besoin de données couvrant toutes les branches possibles des instructions *if*. Les chemins que nous n'avons pas encore explorés sont ceux où la devise passée en argument n'est pas présente dans la liste de devises et le cas où `currency2=Null`. Nous avons spécifiquement utilisé le 'Real brésilien' comme devise incorrecte. Nous avons testé chaque combinaison possible de paires de devises, correctes et incorrectes, pour vérifier le comportement du code. Ainsi, nous avons examiné si le code génère conversion égal à 0.0 quand une

devise incorrecte est passée dans *if* 180, ensuite dans le *if* 189, et dans les deux en même temps. Finalment, on a testé le cas où `currency2=Null` (*if* 187).

3.2.3 Couverture des chemins indépendantes

En examinant le graphe de flux de contrôle, on constate qu'il comprend quatre régions distinctes, ce qui reflète la complexité cyclomatique du graphe. De ce fait, il est nécessaire d'identifier quatre chemins indépendants pour une couverture complète des tests. Cependant, nous avons déjà couvert tous ces chemins dans nos tests précédents, ce qui signifie qu'aucun test supplémentaire n'est nécessaire dans ce contexte.

3.2.4 Couverture des décisions

Étant donné l'absence de conditions composées dans notre cas, l'exécution de tests spécifiques à ce type de structure n'est pas nécessaire.

3.2.5 Couverture des i-chemins

Dans ce scénario, les i-chemins se rapportent au nombre de recherches nécessaires dans la liste des devises avant de trouver la devise cible. Comme la liste inclut cinq types de devises différents, pour couvrir tous les i-chemins, on a changé la devise recherchée à chaque test, en respectant l'ordre de la liste. Cette méthode a été appliquée pour les deux boucles. Bien qu'en ayant testé toutes les combinaisons possibles de paires de devises, nous ayons théoriquement couvert tous les i-chemins, nous avons quand même réalisé des tests spécifiques pour ce cas, par souci de rigueur. En plus, pour chaque test, nous avons vérifié si le résultat de la conversion était différent de zéro, afin de confirmer que la conversion avait effectivement été réalisée

3.2.6 Résultats

On a définit 7 tests Junit, selon ce qui a été décrit ci-dessus :

1. TestInstructionsCoverage()
2. TestBranchCoverageIf180()
3. TestBranchCoverageIf189()
4. TestBranchCoverageBothIfs()
5. TestBranchCoverageIf187()
6. TestPathsLoop1()
7. TestPathsLoop2()

Tous les tests ont réussi, comme le montre l'image ci-dessous :

```
-------------------------------------------------------------------------------
Test set: currencyConverter.MainWindowTest
-------------------------------------------------------------------------------
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.046 s - in currencyConverter.MainWindowTest
```

4 Conclusion

En conclusion, les tests boite noire ont révélés que ces deux fonctions ne respectent pas les spécifications (les devises attendues ne sont pas toutes présentes, et le montants ne sont absolument pas bornés). D'un autre côté, les tests boite blanche montrent que le code ne présente pas de problème de structure.