Prof. Burkhart Wolff
wolff@lri.fr

T. Balabonski, D. Gallois-Wong, H. Brabra
blsk@lri.fr, diane.gallois-wong@lri.fr,
hayet.brabra@telecom-sudparis.eu

TD 5 - Test boîte noire

Semaine du 14 octobre 2017

On reprend l'exercice TD 3.2. Cette fois-ci, par contre, on va faire le développement des tests de manière formelle.

Exercice 1 (Test fonctionnel)
Rappel :

Une société vend deux produits A et B au prix unitaire de 5 € pour A et de 10 € pour B. Une commande comprend une certaine quantité du produit A et une certaine quantité du produit B. Le coût d'une commande est la somme totale des prix unitaires des produits commandés, à laquelle on applique une réduction selon les règles suivantes :

- Si la somme totale est supérieure ou égale à 200 €, on applique une réduction de 5%, si elle est supérieure ou égale à 1000 €, la réduction est de 20%. Ces deux réductions ne sont pas cumulables et portent sur la somme totale.
- La société souhaitant encourager la vente de A, on applique, sur le prix obtenu grâce à la règle précédente, une réduction supplémentaire de 10% si la commande comprend au moins 45 produits A.

1. Donner une specification formelle de l'opération `calculate_price(A,B)` en Moal. L'opération n'est pas censé d'avoir un effet de bord ou d'utiliser l'état.
2. Generer les tests abstraits via la "méthode DNF" :
   - developper a partir de `calculate_price(A,B) = r` la formule réprésentant la relation entre entrées `A` et `B` et sortie `r`. (Expliciter le *test - goal*).
   - "purifier" le *test - goal*, alors eliminer `let`'s et implications.
   - Calculer via DNF les cas abstrait de test
   - Raffiner ces cas par borderline analysis.

HINT : Utiliser les règles algébriques listées dans le cours. Utiliser la règle $F(\mathsf{if}\ c\ \mathsf{then}\ C\ \mathsf{else} D) = \mathsf{if}\ c\ \mathsf{then}\ F(C)\ \mathsf{else} F(D)$ si C, D ne sont pas de type $Bool$ et $(\mathsf{if}\ c\ \mathsf{then}\ C\ \mathsf{else} D) = ((c \wedge C) \vee (\neg c \wedge D))$ alternativement. Utiliser des abréviations si c'est plus pratique.

Dans l'exercice suivant, on s'intéresse à la conception d'une bibliothèque `Set<E>` pour représenter des ensembles d'objets de type `E`, en supposant que les objets de type `E` sont comparables deux à deux.

Exercice 2 (Interface)

Un objet de type `Set<E>` désigne un ensemble d'éléments de `E`, qui ne doit pas contenir `null`. La classe `Set<E>` doit fournir trois opérations publiques

```
isPresent(e: E): Boolean
add(e: E)
remove(e: E)
```

Toutes trois lèvent une exception `InvalidArgument` si le paramètre `e` fourni est égal à `null`. Sinon, l'opération `isPresent` renvoie `true` si et seulement si l'élément `e` est présent et `false` sinon, l'opération `add` modifie l'ensemble en y ajoutant l'élément `e`, ou ne fait rien s'il est déjà présent, et l'opération `remove` modifie l'ensemble en en retirant l'élément `e`, ou ne fait rien s'il est absent.

1. Écrire une formule représentant l'invariant imposé aux objets de la classe `Set<E>` et donner des spécification formelles aux trois opérations.
2. Mettre sous forme disjonctive la spécification de l'opération `Set.isPresent` et en déduire des cas de test.
3. Donner des tests concrets pour les trois opérations.