Test et Validation du Logiciel
(GLG101)

Programmation par contrat

1 Introduction

Il est possible générer des cas de test, et de les produire sous forme de `Stream`. Ces données de test peuvent même être aléatoires comme, par exemple, 20 nombres entiers aléatoires n tels que $0 \leqslant n < 100$:

```
static IntStream random() {
   return new Random().ints(0, 100).limit(20);
}

@ParameterizedTest
@MethodSource("random")
void testadd3(int x) {
   int actual = calc.add(x, x);
   int expected = 2 * x;
   assertEquals(expected, actual);
}
```

Question

1. Tester l'exemple ci-dessus, puis modifier l'annotation `@ParameterizedTest` de manière à pour afficher les valeurs aléatoires qui sont testées :

   ```
   @ParameterizedTest(name = "Trying␣{0}")
   ```

Remarques. Ce type de tests aléatoires apporte toutefois de nouvelles difficultés :

– quel résultat attendre pour un test donné ?

– comment générer des données aléatoires pertinentes pour des types complexes ?

La réponse à la première question dépend des exigences. On se contente généralement de vérifier une propriété donnée du résultat (la *post-condition* du test). Ces propriétés peuvent caractériser la fonction testée complètement (par exemple, pour une fonction qui trie un tableau de valeurs, il suffit de vérifier que le résultat contient bien les mêmes valeurs dans l'ordre), ou seulement partiellement (par exemple, la même taille). On parle alors de *Property Based Testing*[1].

La réponse à la seconde question nécéssite d'une part de fournir des générateurs aléatoires pour des types de données structurés, et d'autre part de filtrer les entrées pertinentes des tests à l'aide d'une *pré-condition*.

2 Programmation par contrat (*design by contract*)

Le contrat d'une fonction est donné par le couple de propriétés *pré-condition*/*post-condition*. (*pre*/*post*, *requires*/*ensures*, *assume*/*assert*...). Ces contrats sont éventuellement complétés par des *invariants* qui mentionnent des propriétés qui doivent être préservés par les fonctions.

1. La première implantation de *Property Based Testing*, appelée QuickCheck, a été développée pour Haskell, mais des implantations récentes existent aussi pour Java comme jqwick (https://jqwik.net) ou ScalaCheck (https://www.scalacheck.org).

Pour pouvoir être utilisés pour les tests, ces contrats doivent être executables. On parle alors de « programmation par contrat » (*design by contract*)[2].

Une syntaxe précise a été définie pour les contrats ou des invariants (*Java Modeling Language*[3]), qui doit être utilisée dans le source Java sous forme de commentaires :

`@requires`.
: *Defines a pre-condition on the method that follows.*

`@ensures`.
: *Defines a post-condition on the method that follows.*

`@invariant`.
: *Defines an invariant property of the class.*

`\result`.
: *A special identifier for the return value of the method that follows.*

`\old (variable)`.
: *A modifier to refer to the value of the* `variable` *at the* ***time of entry*** *into a method.*

Par exemple, le contrat de `add` correspondant au test précédent pourrait s'écrire :

```
// @requires (0 <= x && x < 100);
// @ensures (\result == 2 * x);
```

2.1 Test d'une implantation des piles

La classe `BoundedStack` donnée ci-dessous implémente les primitives habituelles `push`, `pop`, `peek`, `isEmpty` et `isFull` pour des piles de `char` de taille bornée (par `capacity`), dont les éléments sont stockés dans le tableau `elems` et où `top` correspond à la première case libre du tableau.

Remarquez que `push`, `pop`, et `peek` ne vérifient pas si la pile est pleine ou vide. Par conséquent il est nécessaire d'appeler `isEmpty` ou `isFull` avant d'ajouter, de supprimer ou de consulter le sommer de la pile.

Les exigences qui correspondent à ce code sont les suivantes :

`push`. *Place a new item on top of the stack (does not check if the stack is full).*

`pop`. *Remove and return the top stack item (does not check if the stack is empty).*

`peek`. *Return top stack item without removing it (does not check if the stack is empty).*

`isEmpty`. *Returns true if the stack is empty, otherwise returns false.*

`isFull`. *Returns true if the stack is full, otherwise returns false.*

Questions

1. **Self Testing.** Implémenter les invariants et les contracts de la classe `BoundedStack` en insérant des assertions Java[4] (en utilisant le mot-clé `assert`) dans le corps des methodes. Il pourra être nécesssaire d'introduire des variables auxiliaires pour garder les anciennes valeurs (`old`) de certaines variables.

2. **Unit Testing.** Implémenter les invariants et les contracts de la classe `BoundedStack` sous forme de cas de test JUnit 5. Vérifiez que vous obtenez bien 100% de couverture des décisions (*branch coverage*) avec EclEmma.

2. La programmation par contrat a été introduite au départ pour le langage Eiffel en 1986. Elle a été intégrée à la norme du langage Ada en 2012 qui est utilisé dans l'industrie (principalement avionique et ferroviaire).

3. https://www.openjml.org

4. https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html

```
class BoundedStack {

    char[] elems;
    int top;

    // @invariant (top >= 0 && top <= elems.length);

    // @requires (capacity > 0);
    // @ensures (top == 0 && elems.length == capacity);
    BoundedStack(int capacity) {
        top = 0;
        elems = new char[capacity];
    }

    // @requires (size >= 0) && (size < array.length);
    // @ensures (top == size && elems.length == array.length);
    BoundedStack(char[] array, int size) {
        top = size;
        elems = array;
    }

    // @requires (top > 0);
    // @ensures \result == elems[top - 1];
    char peek() {
        return elems[top - 1];
    }

    // @requires (top < elems.length);
    // @ensures (top == \old (top) + 1) && elems[\old (top)] == item;
    void push(char item) {
        elems[top] = item;
        top = top + 1;
    }

    // @requires (top > 0);
    // @ensures (top == \old (top) - 1) && \result == elems[top];
    char pop() {
        top = top - 1;
        return elems[top];
    }

    // @ensures (\result == (top == elems.length));
    boolean isFull() {
        if (top == elems.length)
            return true;
        else
            return false;
    }

    // @ensures (\result == (top == 0));
    boolean isEmpty() {
        if (top == 0)
            return true;
        else
            return false;
    }
}
```