Test et Validation du Logiciel

Property Based Testing

Un *framework* simple pour le PBT avec JUnit 5

Le but de ce TP est d'implementer un *framework* simple pour le test basé sur les propriétés (PBT) avec JUnit 5. Le test basé sur les propriétés combine l'approche *design-by-contracts* avec la génération aléatoire de cas de test. Plus précisément, étant donnée une méthode munie d'un contrat, cette méthode est testée comme suit :

- un générateur est utilisé pour créer une série de valeurs aléatoires du bon type,
- la pre-condition est utilisée pour filtrer les données aléatoires en entrée de la méthode,
- la post-condition est enfin utilisée pour vérifier le résultat de la méthode.

Le générateur, combiné avec le filtre pour la pre-condition, est souvent appelé le *provider* et la post-condition est appelée simplement la « propriété ».

Des *frameworks* dédiés au PBT avec Java comme jqwik (https://jqwik.net), ScalaCheck (https://www.scalacheck.org) ou encore jetCheck (https://github.com/JetBrains/jetCheck), fournissent une manière confortable d'exprimer les propriétés ou d'implanter de nouveaux générateurs.

Toutefois, l'annotation JUnit 5 `@ParameterizedTest` est suffisante dans beaucoup de cas : l'annotation `@MethodSource` permet de facilement référencer un *provider* particulier. La série de données aléatoires en entrée peut être implantée sous forme de `Collection`, ou plus efficacement, de `Stream`.

Par conséquent, pour utiliser JUnit 5 comme un *framework* simple de PBT, nous avons juste besoin d'implémenter quelques générateurs pout les principaux types et collections de Java, et de fournir en plus au testeur une API pour developper ses propres générateurs.

Implémentation

Un générateurs pour un type `T` peut généralement s'implémenter directement comme une méthode statique (en s'appuyant sur la classe `Random` de la JDK). En utilisant la notation lambda, une telle méthode statique peut ensuite être transformée en un `Supplier<T>`[1] à chaque fois qu'il faut la passer en paramètre, par exemple à un autre générateur.

La class `Arbitrary`, donnée ci-dessous, fournit des générateurs pour les types standard et les `Collections` de Java, definies comme des méthodes statiques (`arbitraryInteger`, `arbitraryString`, `arbitraryList`, etc). Un générateur pour une classe particulière peut ensuite être créé en combinant ces méthodes avec un constructeur.

1. https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html

```java
package net.lecnam.info;

import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.function.Supplier;

class Arbitrary {

    // Returns an arbitrary integer
    static Integer arbitraryInteger() { // TODO }

    // Returns an arbitrary integer between 0 (inclusive) and 'bound' (exclusive)
    // throws IllegalArgumentException if bound is less than or equal to 0
    static Integer arbitraryInteger(int bound) { // TODO }

    // Returns an arbitrary integer between 'start' and 'end' (both inclusive)
    // throws IllegalArgumentException if 'start' is greater than or equal to 'end'
    static Integer arbitraryInteger(int start, int end) { // TODO }

    // Returns either 'true' or 'false'
    static Boolean arbitraryBoolean() { // TODO }

    // Returns an arbitrary character between 'start' and 'end' (both inclusive)
    // throws IllegalArgumentException if 'start' is greater than or equal to 'end'
    static Character arbitraryCharacter(char start, char end) { // TODO }

    // Returns an arbitrary character that occur in the given string
    // throws IllegalArgumentException if the given string is empty
    static Character arbitraryCharacterFrom(String possibleChars) { // TODO }

    // Returns an arbitrary upper case ASCII letter ('A'-'Z')
    static Character arbitraryUppercaseLetter() { // TODO }

    // Returns an arbitrary lower case ASCII letter ('a'-'z')
    static Character arbitraryLowercaseLetter() { // TODO }

    // Returns an arbitrary decimal digit ('0'-'9')
    static Character arbitraryDigit() { // TODO }

    // Returns a string of arbitrary size containing arbitrary chars
    static String arbitraryString(Supplier<Integer> arbitrarySize,
            Supplier<Character> arbitraryCharacter) { // TODO }

    // Returns an arbitrary value taken from the given list
    // throws IllegalArgumentException if the given list is empty
    static <T> T arbitraryFrom(List<T> values) { // TODO }

    // Returns a list of arbitrary size containing arbitrary items
    static <T> List<T> arbitraryList(Supplier<Integer> arbitrarySize,
            Supplier<T> arbitraryItem) { // TODO }

    // Returns a map of arbitrary size mapping arbitrary keys to arbitrary values
    static <K, V> Map<K, V> arbitraryMap(Supplier<Integer> arbitrarySize,
            Supplier<K> arbitraryKey, Supplier<V> arbitraryValue) { // TODO }

    // Returns an array of arbitrary size containing arbitrary strings
    static String[] arbitraryArray(Supplier<Integer> arbitrarySize,
            Supplier<String> arbitraryString) { // TODO }
}
```

Générateurs et *streams*

A générateur peut être transformé en *stream* en utilisant la méthode `Stream.generate` de la classe `Stream`[2] (qui prend aussi un `Supplier<T>` en argument). Par exemple, cette expression construit un *stream* de caractères choisis arbitrairement entre `'1'` and `'Z'` :

```
Stream.generate(() -> arbitraryCharacter('1', 'Z'))
```

où `arbitraryCharacter` est une des méthodes de la classe `Arbitrary`. On peut par exemple utiliser cette expression pour définir un *provider* de 20 données de test pour une méthode :

```
static Stream<Character> testCaseProvider() {
    return Stream.generate(() -> arbitraryCharacter('1', 'Z'))
            .filter(c -> Character.isLetter(c)) // pre-condition
            .limit(20);
}
```

Exercices

1. **Implémentation**. Compléter l'implémentation de la classe `Arbitrary` en fournissant le corps des méthodes statiques (à la place des commentaires `TODO`). La spécification de chaque méthode est donnée comme un commentaire informel dans la classe.

2. **Unit Testing.** Utiliser JUnit 5 `@ParameterizedTest` avec l'annotation `@MethodSource` annotation pour tester vos générateurs.

3. **Exemple : add.** Utiliser la méthode `arbitraryInteger` pour la tester la méthode `add` avec 20 entiers aléatoires compris entre 0 et 100.

4. **Exemple : shorter.** Utiliser la méthode `arbitraryString` pour la tester la méthode `shorter` avec 20 triplets aléatoires de `String` (constituées de lettres minuscules et de tailles inférieures à 10). On utilisera la méthode `Arguments.of`[3] de JUnit pour construire ces triplets.

5. **Exemple : sort.** Définir un contrat pour méthode statique de tri suivante et utiliser ensuite la méthode `arbitraryArray` pour la tester sur 20 tableaux aléatoires (où les `String` auront la même forme que pour la question précédente) :

   ```
   static void sort(String[] array) {
       boolean flag = true; // set flag to true to begin first pass
       while (flag) {
           flag = false; // set flag to false awaiting a possible swap
           for (int j = 0; j < array.length - 1; j++) {
               if (array[j].compareTo(array[j + 1]) > 0) { // ascending sort
                   String temp = array[j]; // swap elements
                   array[j] = array[j + 1];
                   array[j + 1] = temp;
                   flag = true; // a swap occurred
               }
           }
       }
   }
   ```

6. **Exemple : BoundedStack.** Définir les générateurs permettant le test aléatoire de la classe `BoundedStack`.

2. https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
3. https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests-sources-MethodSource