Université Grenoble Alpes — UFR IM2AG
Master 2 Génie Informatique — année 2016-2017

JML - TD 3

Yves LEDRU

Janvier 2017

Les fichiers décrits dans ce TD sont déposés sur le site Moodle du cours:
`http://imag-moodle.e.ujf-grenoble.fr/course/view.php?id=93`

1 Classe des nombres premiers

L'objectif de cette première partie est de compléter la classe ci-dessous qui contient une variable entière correspondant à un nombre premier.

```
1  public class Prime{
       private /*@ spec_public @*/ int p;
       /*@ public invariant
         @ (* A COMPLETER *);
5        @*/

       public Prime(){
           p = 3;
       }
10
       //@ requires (* A COMPLETER *);
       //@ ensures (* A COMPLETER *);
       public Prime(int x){
           p = x;
15     }

       //@ requires (* A COMPLETER *);
       //@ ensures (* A COMPLETER *);
       public void set_p(int x){
20         p = x;
       }

       //@ ensures (* A COMPLETER *);
       public /*@ pure @*/ int get_p(){
25         return p;
       }

       /*@ ensures \result == true <==>
         @   (n > 1 ) && (\forall int d; 2<= d && d<= n-1; n % d != 0);
30       @*/
       public static /*@ pure helper @*/ boolean is_prime(int n){
           return true; // A MODIFIER
        }
   }
```

1.1 Question 1

Exécutez la suite de test du fichier `TestPrimeJUnit4`. Elle comprend deux tests de la fonction `is_prime` et deux tests du constructeur de la classe. A ce stade, nous ne tenons compte que des deux

premiers tests. L'échec de `testSequence_1` nous indique que l'implémentation de la fonction `is_prime` n'est pas conforme à sa spécification.

1.2 Question 2

Insérez une implantation de la méthode `is_prime` qui renvoie vrai si et seulement si son argument est un nombre premier.

1.3 Question 3

Testez cette méthode avec la classe `TestPrimeJUnit4` y ajoutant des tests sur des nombres premiers et des nombres non premiers.

1.4 Question 4

Complétez l'invariant et les pré- et post-conditions des autres opérations de la classe. Calculez les pré-conditions pour qu'elles empêchent un appel d'opération qui invaliderait l'invariant. Complétez la classe `TestPrimeJUnit4` pour y tester les diverses méthodes et constructeurs de la classe.

2 Test paramétré et test aléatoire

Note importante : pour utiliser toutes les fonctionnalités du test paramétré en JUnit, vous devez utiliser la version 4.12. Attention! Avec cette version, il faut aussi charger explicitement `hamcrest-core-1.3.jar` . Ces versions sont notamment disponibles sur `junit.org` .

2.1 Question 5

Le fichier `TestPrimeJUnit4.java` contient deux tests paramétrables, et deux valeurs de paramètres. Testez votre classe avec 10 valeurs de paramètres en ajoutant comme paramètres des nombres premiers et des nombres non premiers. Vous constaterez que l'insertion d'un nombre non-premier mène à un verdict inconclusif.

2.2 Question 6

Comme la classe `Prime` prend des paramètres entiers, elle est bien adaptée à du test aléatoire. Modifiez le code de la méthode *params()* pour qu'elle génère aléatoirement 5 entiers (en utilisant la méthode `nextInt()` de la classe `Random`). Exécutez ces tests et constatez que les valeurs aléatoires sont très différentes des valeurs que vous avez sélectionnées à la question précédente. Recommencez en tirant 10 valeurs entre 0 et 10000 (en utilisant la méthode `nextInt(int n)`). Vous pouvez constater qu'une grande majorité des valeurs tirées au sort ne sont pas des nombres premiers et mènent à un verdict inconclusif.

3 Tableau de nombres premiers

La classe `PrimeArray` contient un tableau d'éléments de la classe `Prime` et une variable `size`. Ce tableau a les propriétés suivantes:

- Les `size` premiers éléments sont rangés par ordre croissant.
- Deux éléments consécutifs sont des nombres premiers consécutifs (par exemple, si l'élément `i` est 5, alors l'élément `i+1` est forcément 7 et ne peut pas être 11 ou 13).

Cette classe ne comprend qu'une méthode `grow` qui augmente la variable `size` d'une unité et met à jour le tableau en y ajoutant un nombre premier.

```
1   public class PrimeArray{

        private /*@ spec_public @*/ Prime[] l;
        private /*@ spec_public @*/ int size = 0;
5

      /*@ public invariant // Premier invariant
        @ (* Les size premiers elements sont ranges en ordre croissant *);
        @*/
10    /*@ public invariant // Deuxième invariant
        @ (* Les entiers compris entre deux elements consecutifs,
        @ parmi les size premiers elements, ne sont pas premiers *);
        @*/

15      /*@ ensures size == 0;
          @*/
        public PrimeArray(){
            l = new Prime[100];
            size = 0;
20      }

        /*@ ensures \old(size)+1 == size;
          @*/
        public void grow(){
25       }
    }
```

3.1 Question 7

Complétez le fichier ci-dessus en exprimant formellement les deux propriétés manquantes de l'invariant, et en implantant la méthode `grow`.

3.2 Question 8

Testez votre classe.

3.3 Question 9

Introduisez des erreurs dans votre implémentation et testez la pour montrer que l'invariant de la classe détecte ces erreurs.

3.4 Si il vous reste du temps

Spécifiez et implémentez un constructeur qui prend en entrée un tableau d'entiers et le charge dans `PrimeArray`. Générez aléatoirement des tableaux d'entiers: l'un d'entre eux satisfait-il les contraintes de l'invariant?