IN201 : 3 – JML

Solution

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

Résumé

Ce TP a pour but de vous faire manipuler la programmation par contrat (invariants, préconditions, postconditions) au travers de la conception et de l'implantation d'une classe `EnsembleEntier` en utilisant le langage de modélisation JML.

1 Contenu

Ce corrigé regroupe les réponses au TP sur la programmation par contrat. Les sources des fichiers Java sont disponibles sur le site.

2 Manipulation de la classe `Date` et de JML

Récupérer la classe `Date` sur le site ainsi que les classes `TestDate` et `TestDateInc` dont les listings sont présentés respectivement sur les listings 1 et 2.

Listing 1– La classe `TestDate`

```
public class TestDate {
    public static void main (String args []) {
        // Construire les dates
        Date d1 = new Date(20, 5, 1989);
        Date d2 = new Date(13, 2, 1993);
        Date d3 = new Date(31, 6, 2001);

        // Afficher les dates
        System.out.println("d1 = " + d1);
        System.out.println("d2 = " + d2);
        System.out.println("d3 = " + d3);
    }
}
```

Listing 2– La classe `TestDateInc`

```
public class TestDateInc {
    public static void main (String args []) {
        // Construire une date
        Date d1 = new Date(20, 5, 1989);

        // Incrementer la date
        d1.augmenter(6);

        // Afficher la date
        System.out.println("d1 = " + d1);
    }
}
```

1. compiler les classes avec `javac` et les exécuter avec `java`. Que constatez-vous ?

Solution :

Le résultat obtenu pour l'exécution de `TestDate` est le suivant :

```
d1 = 20/5/1989
d2 = 13/2/1993
d3 = 31/6/2001
```

Évidemment, le fait que la troisième date ne soit pas valide n'est signalée ni à la compilation, ni à l'exécution. On pourrait donc continuer à travailler sur `d3`.

Le résultat obtenu pour l'exécution de `TestDateInc` est le suivant :

```
d1 = 23/5/1989
```

Là encore, on a essayé d'ajouter 6 jours à la date et on s'aperçoit que seuls 3 jours sont ajoutés. Le programme ne s'arrête pas pour autant.

2. compiler maintenant l'application avec `jmlc`. On peut choisir de placer les *bytecodes* instrumentés non pas dans le répertoire `classes` (qui sera réservé pour le code non instrumenté), mais dans un répertoire `classesJML`. Il suffit de faire : `jmlc -classpath $CLASSPATH:../classesJML -d ../classesJML Date.java TestDate.java`.

   Exécuter le programme avec `jmlrac`. Que constatez-vous ?

   Solution :

   Le résultat pour l'exécution de `TestDate` est le suivant :

```
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInternalPreconditionError:
 by method Date.Date regarding specifications at
File "src/Date.java", line 56, character 37 when
        'j' is 31
        'm' is 6
        'a' is 2001
        at TestDate.main(TestDate.java:322)
```

   Cette fois-ci, l'instrumentation du contrat de la classe `Date` avec *JML* permet de détecter une violation de ce contrat à l'exécution. Le message nous précise que la précondition du constructeur de `Date` a été violée (le constructeur de `Date` est la « méthode » `Date.Date`). Le client de la classe (i.e. la personne qui a écrit `TestDate`) doit donc vérifier que les paramètres qu'il utilise pour créer une date respectent bien le contrat (par exemple en utilisant la méthode publique statique `estValide`).

   Le résultat pour l'exécution de `TestDateInc` est le suivant :

```
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInternalNormalPostconditionError:
 by method Date.augmenter regarding specifications at
File "src/Date.java", line 226, character 38 when
        'nbJours' is 6
        '\old(getNbJoursDepuisAn0())' is 726242
        'this' is 23/5/1989
        at Date.augmenter(Date.java:3621)
        at TestDateInc.internal$main(TestDateInc.java:11)
        at TestDateInc.main(TestDateInc.java:309)
```

   Là encore, l'instrumentation nous permet de détecter une violation du contrat de la classe sur une postcondition. Cela nous permet de dire que la faute est due à l'implantation de la classe `Date` qui a été faite.

3 Présentation du problème

On cherche à modéliser un ensemble d'entiers sous la forme d'un objet. Pour cela on dispose d'une spécification d'une classe `EnsembleEntierTab` qui stocke les entiers dans un tableau. Un nouvel élément est ajouté à la fin du tableau. Cette modélisation va être utilisée pour implanter le crible d'Ératosthène.

4 Étude de la spécification de `EnsembleEntierTab`

Récupérer le source de la classe `EnsembleEntierTab` sur le site. À partir des spécifications écrites « en JML », répondre aux questions suivantes[1] :

1. peut-on ajouter un élément dans l'ensemble s'il est déjà présent ?

   Solution :
   La seule précondition dans le contrat de la méthode `ajouter` est :

```
121     //@ requires !contient(x) ==> getCardinal() < getCapacite();
```

   Cette précondition n'impose pas que l'élément ne soit pas déjà dans l'ensemble.

2. peut-on toujours ajouter un élément dans l'ensemble ?

   Solution :
   Là encore, regardons la seule précondition du contrat de `ajouter` :

```
121     //@ requires !contient(x) ==> getCardinal() < getCapacite();
```

   C'est une implication qui signifie : si x n'est pas dans l'ensemble, alors le nombre d'éléments de l'ensemble doit être inférieur à la capacité de l'ensemble. Donc si on essaye d'ajouter un élément non déjà présent dans l'ensemble, il faut qu'il « reste de la place » dans le tableau.

3. si l'on ajoute trois fois l'élément 1 dans l'ensemble et qu'on le supprime (ôte) une fois, l'élément est-il encore présent dans l'ensemble ?

   Solution :
   Regardons les postconditions du contrat de `oter` :

```
136     //@ ensures !contient(x);   // element supprime
137     //@ ensures !\old(contient(x)) ==> getCardinal() == \old(getCardinal());
138     //@ ensures \old(contient(x)) ==> getCardinal() == \old(getCardinal()) - 1;
```

   La postcondition `//@ ensures !contient(x);` nous assure que l'élément est bien enlevé. On travaille bien avec un ensemble et non un multi-ensemble.

4. est-ce que la manière d'utiliser le tableau est bien spécifiée par les invariants de `EnsembleEntierTab` ? On se contentera d'une explication intuitive, informelle.

   Solution :
   Examinons les invariants un à un :

```
41      //@ private invariant (\forall int i; 0 <= i && i < getCardinal(); contient(tab[i]));
```

   Cet invariant exprime le fait que les éléments compris entre les indices 0 et `getCardinal()-1` font partie du tableau.

```
46    //@ private invariant (\forall int i; 0 <= i && i < getCardinal();
47                           (\forall int j; 0 <= j && j < i; tab[i] != tab[j]));
```

   Cet invariant exprime le fait que deux éléments de l'ensemble sont toujours différents.
   On a donc une inclusion de `{tab[0],..., tab[getCardinal()-1]}` dans l'ensemble et comme on a même cardinal, l'ensemble est donc constitué exactement des éléments de `{tab[0],..., tab[getCardinal()-1]}`.

1. Les spécifications apparaissent dans la documentation javadoc et y sont plus lisibles.

5 Implantation de la classe `EnsembleEntierTab`

La classe `EnsembleEntierTab` est partiellement implantée. On vous fournit trois classes qui vont permettre de tester l'ensemble d'entiers (`TestEnsembleEntierTab`) et de calculer les nombres premiers (`CribleEntier` et `NombresPremiersEntier`). Ces classes n'ont a priori pas à être modifiées.

1. compiler et exécuter la classe `TestEnsembleEntierTab` avec le JDK (la classe `EnsembleEntierTab` peut être compilée même si le code réel des méthodes a été omis). Que constatez-vous ? Est-il facile de savoir d'où viennent les erreurs qui doivent être corrigées ?

   Solution :

   Les programmes compilent correctement.

   La première exécution à travers `java TestEnsembleEntierTab` donne le résultat suivant :

   ```
   Le min est : 0  (ok)
   Le min est : 0  (ok)
   Le min est : 0  ERREUR ! Resultat attendu : 1
   Le min est : 0  ERREUR ! Resultat attendu : 2
   Le min est : 0  ERREUR ! Resultat attendu : 3
   Le min est : 0  ERREUR ! Resultat attendu : 3
   Le min est : 0  ERREUR ! Resultat attendu : 3
   Le min est : 0  ERREUR ! Resultat attendu : 3
   Le min est : 0  ERREUR ! Resultat attendu : 7
   Le min est : 0  ERREUR ! Resultat attendu : 7
   Le min est : 0  ERREUR ! Resultat attendu : 7
   ```

   Il n'y a pas d'erreur à l'exécution. Par contre le résultat est anormal.

   La seconde exécution à travers `java NombresPremiersEntier 20` donne le résultat suivant :

   ```
   Nombres premiers : 1
   ```

   Là encore, pas d'erreur à l'exécution, mais le résultat est anormal.

   Dans les deux cas, seul l'affichage (et donc un « opérateur » humain !) permet de vérifier que les programmes ne fonctionnent pas correctement. De plus, il est très difficile de trouver les erreurs.

2. recommencer, mais cette fois-ci en utilisant `jmlc` et `jmlrac`. Que constatez-vous ?

   Solution :

   Les compilations se déroulent normalement. L'exécution de `jmlrac TestEnsembleEntier` donne le résultat suivant :

   ```
   Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInternalNormalPostconditionError:
    by method EnsembleEntierTab.EnsembleEntierTab regarding specifications at
   File "squel/EnsembleEntierTab.java", line 67, character 30 when
           'taille_' is 20
           'this' is
           at TestEnsembleEntierTab.main(TestEnsembleEntierTab.java:739)
   ```

   Une *erreur* est levée (le programme ne se déroule pas normalement). On apprend que cette erreur a été générée par une violation de postcondition dans le constructeur de `EnsembleEntierTab`.

   L'exécution de `jmlrac NombresPremiersEntier 20` donne le résultat suivant :

   ```
   Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInternalNormalPostconditionError:
    by method EnsembleEntierTab.EnsembleEntierTab regarding specifications at
   File "squel/EnsembleEntierTab.java", line 67, character 30 when
           'taille_' is 20
           'this' is
           at NombresPremiersEntier.main(NombresPremiersEntier.java:331)
   ```

   Là encore, une erreur est levée (la même, les postconditions du constructeur ne sont pas respectées, la capacité du tableau n'étant pas initialisée).

   Remarque : à la compilation de `EnsembleEntierTab` avec JML, on obtient le message suivant :

   ```
   File "src/EnsembleEntierTab.java", line 23, character 53
         warning: This quantifier is not executable.
   ```

On obtient en fait ce message (un *warning* et non une erreur) trois fois. Il concerne le même type de spécifications JML. Si on regarde la spécification concernée par le message précédent, il s'agit de :

```
23      //@ public invariant !estVide() ==> getMin() == (\min int x; contient(x); x);
```

Le message nous indique que cette spécification n'est pas exécutable. Examinons la. Cette spécification d'invariant nous indique que si l'ensemble n'est pas vide, alors la valeur renvoyée par `getMin` est égale au minimum des éléments contenus par l'ensemble calculé *par JML* (expression de type `min`).

Or l'expression est `\min int x ; contient(x); x` : on cherche donc le minimum parmi l'ensemble des entiers qui vérifient la propriété `contient(x)`. Il faut donc pour JML parcourir l'ensemble des entiers possibles, vérifier si chacun des entiers vérifie `contient(x)` et renvoyer le minimum des entiers vérifiant la propriété. Cette spécification n'est donc pas exécutable, car il faudrait parcourir un ensemble de valeurs complet. Il aurait fallu restreindre l'ensemble d'entiers pouvant être introduits dans le tableau.

3. compléter et/ou corriger le corps des méthodes de la classe `EnsembleEntierTab` ;

   Solution :

   Le source est disponible sur le site. Il n'y avait pas de difficulté particulière.

4. tester votre implantation en s'appuyant sur les deux programmes `TestEnsembleEntierTab` et `NombresPremiersEntier`. On activera bien entendu la vérification dynamique des contrats ;

   Solution :

   Les traces d'exécution sont disponibles sur le site. Tout marche parfaitement maintenant.

   Il faut bien remarquer que les invariants d'une classe doivent être vérifiés *avant et après* tout appel à une méthode de la classe. En particulier, dans les méthodes `oter` et `ajouter`, les éventuels appels à `getCardinal` dans une boucle pouvaient être problématiques, car les invariants doivent être respectés avant tout appel à une méthode. Dans ce cas, on pouvait (devait...) utiliser `nb`.

   Par exemple, supposons que l'on écrive la méthode `oter` de la façon suivante :

```
    public void oter(int x) {
        int i = 0;
        while(i < this.nb && this.tab[i] != x) {
            i++;
        }
        if (i < this.nb) {
            for (int j = i; j < this.getCardinal() - 1; j++) {
                this.tab[j] = this.tab[j+1];
            }
            this.nb--;
        }
    }
```

   Dans ce cas, à l'exécution de `TestEnsembleEntierTab`, on obtient l'erreur suivante :

```
L'ensemble contient 6 elements
Le plus petit est : -3
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInvariantError:
 by method EnsembleEntierTab.getCardinal@pre<File "src/inv/EnsembleEntierTab.java", line 87, character 11
File "src/inv/EnsembleEntierTab.java", line 46, character 27 when
        'tab' is [I@66848c
        'this' is EnsembleEntierTab@8813f2
        at EnsembleEntierTab.checkInv$instance$EnsembleEntierTab(EnsembleEntierTab.java:1637)
        at EnsembleEntierTab.getCardinal(EnsembleEntierTab.java:2087)
        at EnsembleEntierTab.internal$oter(EnsembleEntierTab.java:141)
        at EnsembleEntierTab.oter(EnsembleEntierTab.java:3511)
        at TestEnsembleEntierTab.internal$testerEnsemble(TestEnsembleEntierTab.java:33)
        at TestEnsembleEntierTab.testerEnsemble(TestEnsembleEntierTab.java:544)
        at TestEnsembleEntierTab.internal$main(TestEnsembleEntierTab.java:66)
        at TestEnsembleEntierTab.main(TestEnsembleEntierTab.java:725)
```

Cette erreur nous indique que nous avons une erreur d'invariant (JMLInvariantError), avant un appel à `getCardinal()` (ligne 87, donc dans la méthode `oter`) et que l'invariant en question est ligne 46. Il s'agit de l'invariant précisant que le tableau ne peut pas contenir deux éléments identiques (car il s'agit d'un ensemble).

L'erreur est normale : lorsque l'on rentre une première fois dans la boucle, on recopie `tab[j+1]` sur `tab[j]` et donc, comme `nb` n'a pas été modifié, on a deux éléments identiques dans le tableau. Comme le test de fin de boucle fait appel à `getCardinal`, on a une erreur (car avant tout appel de méthode les invariants doivent être vrais).

5. comparer le temps de calcul pour afficher les nombres premiers entre 2 et 100 avec ou sans instrumentation du code.

Solution :

Il suffit d'utiliser l'instruction `time` sous Unix. Avec instrumentation du code, le résultat est le suivant :

```
Nombres premiers : 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, ...
 jmlrac NombresPremiersEntier 100: 2.16s used
```

Sans instrumentation, le résultat est le suivant :

```
Nombres premiers : 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, ...
 java NombresPremiersEntier 100: 0.02s used
```

Le fait d'instrumenter le code est donc pénalisant en terme de performances (et c'est bien normal). Une version d'exploitation d'un logiciel devra donc « tourner » sans instrumentation. Par contre, le test de l'implantation se fera grâce à la vérification dynamique de contrats.

6 Utilisation de `jmlunit`

NB : les explications données dans cette section sont assez avancées. Il est donc « facultatif » d'utiliser `jmlunit`.
On peut utiliser le framework de test JUnit avec JML pour générer des classes de test utilisant les assertions de JML. On peut se référer à [1] pour plus d'explications.
Supposons que nous souhaitions générer une classe de test utilisant JUnit pour la classe `Date`. Il suffit d'appeler `jmlunit Date.java` qui va générer deux classes :
– `Date_JML_Test` qui contient les tests ;
– `Date_JML_TestData` qui va contenir les données pour les tests.
Important : ces deux classes doivent être compilées avec `javac` et pas `jmlc`.

6.1 La classe `Date_JML_Test`

Cette classe contient le « mécanisme » de test de la clase `Date`. *On n'a donc pas a priori à la modifier*. Le principe mis en œuvre pour effectuer un test sur une méthode de `Date` est le suivant :
– pour chaque paramètre de la méthode, on utilise les *stratégies* de la classe `DateJML_TestData` pour obtenir un ensemble de valeurs (cf. section 6.2). On fera autant de tests que de combinaisons possibles entre les valeurs des différents paramètres ;
– si une valeur possible pour un paramètre lève une exception pour une précondition, alors le test n'a pas de sens. On dit qu'il est *meaningless* : on ne peut pas tester le comportement d'une méthode si ses préconditions sont fausses, car on ne garantit absolument rien sur le résultat ;
– si une exception concernant une postcondition est levée durant l'exécution, le test échoue ;
– si une exception concernant une précondition d'une méthode appelée à l'intérieur de la méthode testée est levée, le test échoue.
Le mécanisme est donc simple : on fournit un ensemble de valeurs pour chaque type de paramètres, et on appelle la méthode avec toutes les combinaisons possibles de ces valeurs.

6.2 La classe `Date_JML_TestData`

La classe `Date_JML_TestData` permet de fournir au test un jeu de données pour chaque type dont les méthodes de la classe vont avoir besoin. On peut remarquer que les mêmes valeurs sont réutilisées pour chaque méthode. Cela ne pose pas de problème, car les préconditions permettent de « filtrer » les valeurs.
Pour chaque type nécessaire (*y compris les objets sur lesquels on appelle les méthodes*), on définit un *itérateur* et une *stratégie*. Ces deux notions sont en fait des *design patterns* [3, 2], nous ne étendrons pas plus sur le sujet.

Par exemple, dans la classe `Date_JML_TestData`, on trouve un attribut `vintStrategy` qui est défini comme un objet de type `org.jmlspecs.jmlunit.strategies.IntStrategyType`. C'est cet objet qui va nous permettre de rajouter nos propres valeurs à l'ensemble des données de test pour le type **`int`**. Pour les dates, on utilisera l'attribut `vDateStrategy`.
Vous trouverez sur le site les classes de tests de `Date` pour mieux comprendre ces exemples.
Utiliser ce *framework* va vous permettre d'automatiser les tests en utilisant les assertions, mais également d'affiner vos préconditions. En effet, *framework* utilise par défaut des jeux de tests qui contiennent par exemple `Double.NaN` pour le type **`double`**, ce que l'on oublie souvent de tester dans les préconditions (`Double.NaN` est obtenu dans le cas d'une division par zéro par exemple).
L'utilisation de `jmlunit` est bien sûr facultative. La lecture de [1]vous permettra de mieux comprendre son fonctionnement.

Références

[1] Y. Cheon et G.T. Leavens. *The JML and JUnit way of unit testing and its implementation*. Rap. tech. 04-02a. Department of Computer Science, Iowa State University, 2004. URL : http://www.cs.iastate.edu/~leavens/JML/papers.shtml.

[2] E. Gamma et al. *Design Patterns - Catalogue de modèles de conception réutilisables*. Traduction de *Design Patterns - Elements of reusable object-oriented software*, Addison-Wesley, 1994. Vuibert, 1999.

[3] E. Gamma et al. *Design Patterns - Elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1994.