Université Toulouse III – Paul Sabatier
L3 Informatique

Tests et maintenance logiciel
24 février 2014
Semestre 6

TP n° 2 & 3 — Analyse de couverture de coe

Antoine de ROQUEMAUREL (Groupe 1.1)

1 Algorithme

L'algorithme à été développé en java. Afin de créer la liste, deux classes ont été créés : `List` et `Cell`.

1.1 `Cell`

`Cell` est une cellule de la liste, elle contient donc la cellule précédente, la suivante et la valeur concernée [1].

Cette classe possède la visibilité package afin que celle si ne soit pas utilisée à mauvais escient.

```
1  package list;
2
3  class Cell {
4    private int _data;
5    private Cell _previous;
6    private Cell _next;
7
8    public Cell(int data, Cell next) {
9      _data = data;
10     _previous = null;
11     _next = next;
12   }
13
14   public int getData() {
15     return _data;
16   }
17
18   public Cell getNext() {
19     return _next;
20   }
21
22   public void setNext(Cell _next) {
23     this._next = _next;
24   }
25 }
```

Listing 1 – Classe `Cell`

1.2 `List`

La classe `List` qui est publique, permet à l'utilisateur de créer une liste triée et d'y ajouter des éléments [2].

L'algorithme proposé à donc été implémenté dans la méthode `add`.

1. La valeur est de type `int`
2. Possibilité d'ajouter un élément depuis la méthode `add` ou depuis le constructeur

```java
package list;

public class List {
  private Cell _head;

  public List(int... data) {
    for(int d : data) {
      add(d);
    }
  }

  public void add(int data) {
    Cell current = _head;
    Cell previous = null;
    Cell newCell;
    boolean notEnd = current != null;

    while(notEnd && current != null) {
      if(current.getData() > data) {
        notEnd = false;
      } else {
        previous = current;
        current = current.getNext();
      }
    }
    newCell = new Cell(data, current);

    if(previous == null) {
      _head = newCell;
    } else {
      previous.setNext(newCell);
    }
  }

  @Override
  public String toString() {
    Cell current = _head;
    String ret = "";
    while(current != null) {
      ret += current.getData()+", ";
      current = current.getNext();
    }

    return ret;
  }
}
```

Listing 2 – Classe `List`

2 Instructions couvrant une insertion en position intermédiaire

2.1 Ajout des deux premiers entiers

Figure 2.1 est disponible la couverture pour l'insertion des deux premiers entiers, avant l'insertion de l'entier qui nous intéresse.

```
public void add(int data) {
    Cell current = _head;
    Cell previous = null;
    Cell newCell;
    boolean notEnd = current != null;

    while(notEnd && current != null) {
        if(current.getData() > data) {
            notEnd = false;
        } else {
            previous = current;
            current = current.getNext();
        }
    }
    newCell = new Cell(data, current);

    if(previous == null) {
        _head = newCell;
    } else {
        previous.setNext(newCell);
    }
}
```

Figure 1 – Ajout des entiers 1 et 5 dans une liste vide

2.2 Ajout de l'entier 4

Afin d'insérer l'entier 4, les entiers 1 et 5 devaient être déjà présents dans la liste, afin d'éviter que le rapport d'emma soit faussé par ces deux insertions, une nouvelle méthode à été créé insérant ces deux entiers.

```
public void add(int data) {
    Cell current = _head;
    Cell previous = null;
    Cell newCell;
    boolean notEnd = current != null;

    while(notEnd && current != null) {
        if(current.getData() > data) {
            notEnd = false;
        } else {
            previous = current;
            current = current.getNext();
        }
    }
    newCell = new Cell(data, current);

    if(previous == null) {
        _head = newCell;
    } else {
        previous.setNext(newCell);
    }
}
```

Figure 2 – Ajout de l'entier 4 dans une liste possédant 1 et 5

Comme le montre la figure 2.2, l'insertion du 4 ne passe que par une portion du code, en effet, les instructions d'insertion en tête ou en queue ne sont pas nécessaire ici, c'est ainsi que la condition `current != null` du while et le test `previous == null` ne sont pas nécessaire pour l'insertion intermédiaire. Nous pouvons observer avec la figure 2.1 que l'insertion en tête ou en queue à l'inverse passe dans ces deux instructions mais n'a pas besoin du test vérifiant si `current.getData() > data`.

On peut donc en déduire que c'est ce test qui permet à l'insertion intermédiaire de fonctionner : ça permet d'insérer notre élément à une place correct dans la liste triée.

3 Jeu de test minimal

Afin de pouvoir tester toutes les instructions du programme, il faut utiliser un jeu de test gérant tous les cas possibles.

Pour l'insertion dans une liste, 3 cas ressortent et sont donc représentés dans le code : insertion en début de liste, insertion en fin de liste, insertion en position intermédiaire. Ainsi, notre jeu de test devra comporter 3 entiers afin de tester ces trois cas.

Insertion en début 1

Insertion en queue 20

Insertion en position intermédiaire 4

L'insertion consécutive de ces 3 entiers ci-dessous couvriront l'intégralité du code comme le montre la figure 3

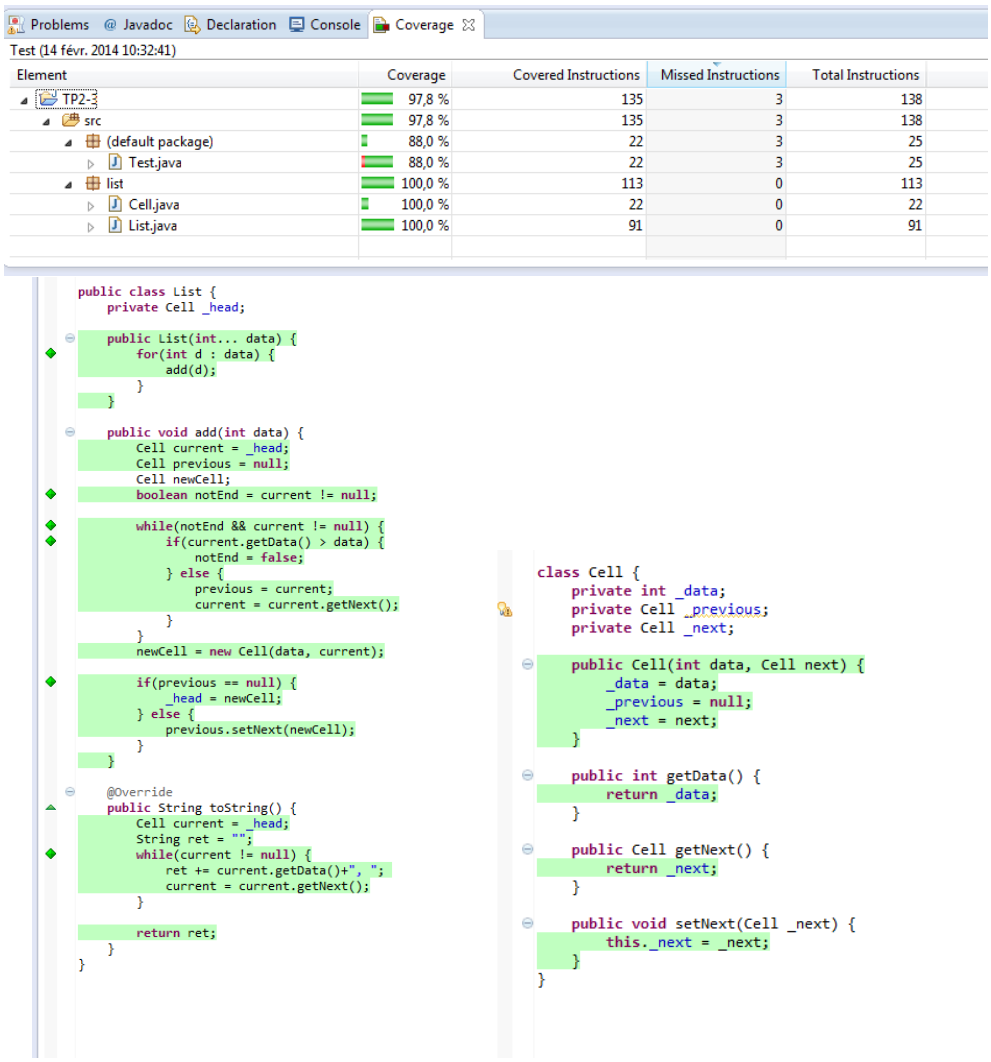

Figure 3 – Tests de couverture avec un jeu de test minimal

R Le fichier de Test ne possède pas 100% de couverture car nous n'utilisons que le `main` et pas la classe qui le contient.
Cependant, cela n'est pas important car nous testons le package `list`, celui-ci possède bien 100%.

4 Cas d'insertion et instructions

Oui, la couverture de toutes les instructions permettent ici de tester les cas possible.

En effet, notre fonction d'insertion possède différentes conditions en fonction du placement du chiffre :

Insertion en début Nous ne rentrons pas dans la boucle, et passons dans le cas `previous == null` afin d'affecter la tête à courant.

Insertion en position intermédiaire C'est le seul cas permettant de vérifier que nous sortons bien de la boucle avant la fin : ce qui nous permet d'insérer ensuite notre entier au milieu de la liste. Ceci est fait grâce à la condition `current.getData() > data`.

Insertion en queue C'est le cas ou l'intégralité de la boucle est parcourue, afin de vérifier que nous sortons bien de la boucle une fois la fin de liste atteint. C'est l'instruction `current != null` qui est dans le test de la boucle.

Nous testons tous les cas possibles, cependant nos tests ne permettent pas de dire si la liste en sortis est bien triée, en effet, rien ne nous permet de dire si les entiers sont insérés au bon endroit.

5 Graphe de contrôle et PLCS

5.1 Graphe de contrôle

Figure 4 est disponible le graphe de contrôle du programme, en utilisant le constructeur permettant une insertion multiple.

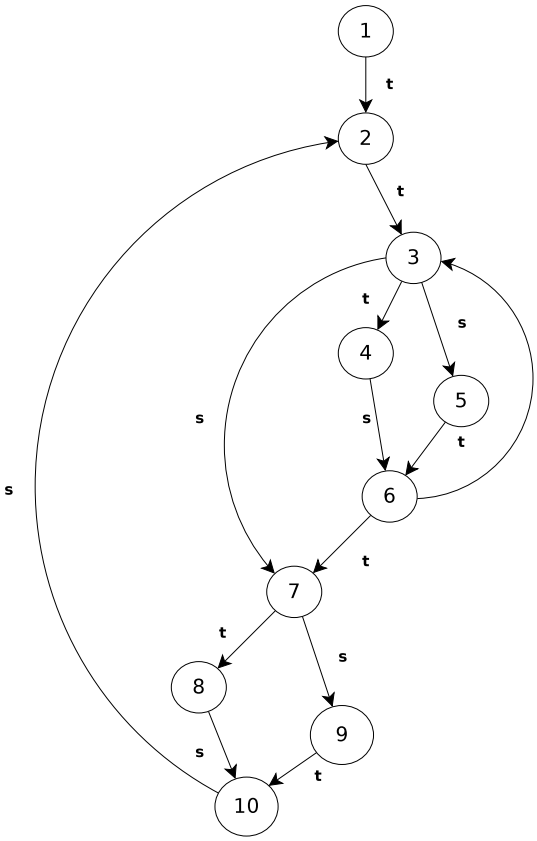

Figure 4 – Graphe de contrôle

5.2 PLCS

- 1&2&3&4,6
- 1&2&3, 5
- 1&2&3,7
- 2&3, 5
- 2&3,7
- 2&3&4,6
- 3,7
- 3&4,7
- 3,5
- 5&6&7,9
- 5&6,3
- 5&6&7&8,10
- 6&7&8,10
- 6&7,9
- 6,3
- 7,9
- 7&8,10
- 9&10,2
- 9&10,-1
- 10, -1