Departamento de Cs. e Ingeniería de la Computación
Universidad Nacional del Sur

Métodos Formales para Ingeniería de Software

Trabajo Práctico N° 3

Especificación - JML

Segundo Cuatrimestre de 2018

Importante: El objetivo de este trabajo práctico es ejercitar la especificación formal en JML. Por lo tanto, para la resolución de los ejercicios de este práctico no deberán implementarse las clases o porciones de código Java suministrados.

Ejercicios

1. Considere dos arreglos de enteros, `b` y `c`, cuyo tamaño es n. Sean j y k dos índices tales que $0 \leq j < k < n$. Además, suponga que `b[j..k]` denota el segmento del arreglo `b` comprendido entre los índices j y k, siendo esta notación análoga para segmentos del arreglo `c`. Observación: esta notación para identificar segmentos no es válida en JML.

   Escriba expresiones en JML que describan precisamente las siguientes condiciones:

   a) Todas las componentes en el segmento `b[j..k]` tienen valor cero.
   b) Todas las componentes que tienen valor cero en `b[j..k]` también tienen valor cero en el segmento `c[j..k]`.
   c) Si una componente tiene valor cero en `b`, entonces se encuentra en el segmento `b[j..k]` y también tienen valor cero en el segmento `c[j..k]`.
   d) No es el caso que todas las componentes que tienen valor cero en `b` están en el segmento `b[j..k]`.
   e) El arreglo `b` contiene al menos dos componentes con valor cero.
   f) El arreglo `b` contienen exactamente dos componentes con valor cero.

2. Considere la clase `Lista`, con el siguiente *esqueleto*:

```
public class Lista {
    Object cabeza;
    Lista cola;

    Lista(Object cabeza, Lista cola) {
        this.cabeza = cabeza;
        this.cola = cola;
    }

    int largo() {
        // ...
    }
}
```

   Brinde especificaciones JML para el método `largo`.

3. Considere los siguientes *esqueletos* de clases, las cuales modelan la clasificación de libros en categorías:

```
public class Category {
    private Book[] books;

    //...

    public boolean belongsToCategory(Book b) {
        // Determina si el libro b pertenece a la categoria (pertenencia a books)
    }

    public void addBook(Book b){
        /*
            Si el libro b posee la categoría actual, lo añade a la colección de libros.
            Además, en caso de añadirlo, le asocia la categoría actual a b.
        */
    }
}

public class Book {
    private Category cat;
    private Book[] similar;

    // ...

    public Category viewCategory() {
        // Retorna la categoria del libro
    }

    public boolean isSimilar(Book b) {
        // Determina si el libro b es similar al actual
    }

    public void addCategory(Category c){
        /*
            Si el libro actual no tiene categoría, le asocia c.
            Además, en caso de éxito, agrega el libro actual al conjunto de libros de c.
        */
    }
}
```

Brinde especificaciones JML para resolver los siguientes incisos:

a) Defina invariantes que capturen las siguientes restricciones:

- Un libro `b` posee la categoría `c` sí y sólo sí `b` pertenece al conjunto de libros de la categoría `c`.
- Si dos libros se encuentran en relación de similitud, entonces poseen la misma categoría.
- Un libro no puede estar en relación de similitud consigo mismo.
- Las categorías son disjuntas, es decir, no poseen libros en común.

b) Defina contratos para los métodos de ambas clases, respetando su comportamiento.

4. Considere el siguiente *esqueleto* de la clase `Cola`, la cual implementa la estructura de datos homónima mediante un arreglo.

```
public class Cola {
    Object[] arreglo;
    int long;     // Cantidad de elementos
    int primero;  // Indice del primer elemento
    int proximo;  // Indice donde se insertaria el proximo elemento

    Cola(int tope) {
        // ...
    }

    public int largo(){
        // ...
    }

    public void agregarElemento(Object x){
        // ...
    }

    public Object sacarElemento(){
        // ...
    }
}
```

   Brinde especificaciones JML para la clase provista (invariantes necesarios y contratos para cada método).

5. Considere una clase que implementa un laberinto mediante un arreglo de dos dimensiones (por lo tanto, cada fila tiene la misma cantidad de componentes). La clase `Laberinto` declara los siguientes elementos:

   - Constantes de la forma `MOVER_X` que codifican las diferentes direcciones en las cuales puede moverse un jugador que intenta salir del laberinto (arriba, abajo, izquierda o derecha).
   - Constantes `SALIDA`, `LIBRE` y `PARED`, que codifican el tipo de cada celda del laberinto.
   - Dos campos de tipo entero, `jugadorFila` y `jugadorColumna`, que identifican la posición actual del jugador dentro del laberinto.

   Teniendo en cuenta esto, considere el siguiente *esqueleto* para la clase `Laberinto`:

```
public class Laberinto {

    // Constantes de Movimiento
    public final static int MOVER_ARRIBA = 0;
    public final static int MOVER_ABAJO = 1;
    public final static int MOVER_IZQ = 2;
    public final static int MOVER_DER = 3;

    // Constantes de Celda
    public final static int LIBRE = 0;
    public final static int PARED = 1;
    public final static int SALIDA = 2;
```

```
    /*  Campo de Juego:
        El campo de juego esta dado por un rectángulo donde las paredes se
        representan mediante entradas con valor Laberinto.PARED (es decir,
        con valor '1'), la salida (la cual es única) es una entrada con valor
        Laberinto.SALIDA ('2') y todas las celdas restantes que no sean pared
        tienen valor Laberinto.LIBRE ('0').
        Dada una posicion del arreglo, el primer valor determina la fila y el
        segundo la columna (comenzando desde cero).
    */
    private int[][] lab;

    /*
        Posicion del jugador:
        La posicion del jugador debera siempre estar dentro del laberinto, y no
        puede haber una pared en esa misma posicion.
    */
    private int jugadorFila, jugadorColumna;

    /*
        Laberinto (constructor):
        Inicializa el laberinto y pone al jugador en la posicion inicial indicada.
    */
    public Laberinto(int[][] lab, int filaInicial, int columnaInicial) {
        this.lab = lab;
        this.jugadorFila = filaInicial;
        this.jugadorColumna = columnaInicial;
    }

    /*
        gano:
        Determina si el jugador ha llegado a la salida.
    */
    public boolean gano() {
        // ...
    }

    /*
        movimientoPosible:
        Determina si es posible realizar el movimiento a la fila y columna indicadas.
        Un movimiento a la posicion (nuevaFila,nuevaColumna) es posible si y solo si
        la celda esta dentro del laberinto y esta libre. Ademas, la posicion de la
        celda destino debera ser adyacente a la posicion actual del jugador.
    */
    public boolean movimientoPosible(int nuevaFila, int nuevaColumna) {
        // ...
    }

    /*
        mover:
        Toma una direccion y ejecuta el movimiento si es posible.
        Sino, el jugador permanence en la misma posicion.
        La direccion debe ser una de las definidas por las constantes MOVER_X.
        El valor retornado indica si el movimiento tuvo exito.
    */
    public boolean mover(int direccion) {
        // ...
    }
}
```

Brinde especificaciones JML para resolver los siguientes incisos:

a) Defina invariantes que capturen las siguientes restricciones:
   - El laberinto no es nulo. Es decir, el campo de juego en sí y sus componentes son no nulas.
   - Cada fila del laberinto tiene la misma cantidad de columnas.
   - La posición ocupada por el jugador no es una pared.
   - Cada celda del laberinto es o bien una pared, la salida, o está libre.
   - Hay exactamente una salida.

b) Especifique contratos para cada método, capturando el comportamiento indicado en su descripción.