ASERCIONES

Los contratos de software se especifican mediante la utilización de expresiones lógicas denominadas aserciones. En el Diseño por Contratos se utilizan tres tipos de aserciones:

- Precondiciones
- Poscondiciones
- Invariante de Clase.

Tripleta de Hoare (Hoare 1969)

Sea **A** alguna computación y **P**, **Q** aserciones, entonces la siguiente expresión:

{ P } A { Q }

representa lo que se llama fórmula de corrección: cualquier ejecución de **A** que comience en un estado en el cual se cumple **P** dará como resultado un estado en el cual se cumple **Q**. Por ejemplo

{ x > 10 } x := x /2 { x >= 5}

Si se parte de un estado en el cual **x** es mayor que 10 y luego se aplica la computación x := x/2 entonces como resultado x será mayor o igual a 5. A la aserción **P** se la llama **precondición** y a **Q** se la llama **poscondición**. En el ejemplo la precondición es x > 10 y la poscondición x >=5.

Precondiciones: 'require'

- La precondición se especifica al comienzo del cuerpo del método
- La precondición permite establecer bajo que condicione el método puede ser invocado.
- La precondición se especifica mediante la cláusula ***require***.

```
class buffer {

 public void addElement (Object o) {
 /** require !isFull(); o != null; **/
 ...
}
```

- El método *addElement* solo puede ser invocado si el buffer no esta lleno y el parámetro formal es un objeto válido (no es una referencia nula)
- En la precondición se utilizan parámetros y atributos, no variables locales.
- El método *isFull* debe ser declarado public para satisface la condición.

Postcondiciones: 'ensure'

- La postcondición se especifica al final del método.
- Una postcondición establece los estados las propiedades que debe garantizar cuando la rutina finalice.

```
public void addElement (Object o) {
…
/** ensure !isEmpty() && contains(o); **/
}
```

- Luego que se ejecuta el método **addElement** el buffer no puede estar vacío y el objeto ha sido insertado.

La postcondición puede contener 3 constructores especiales:

- *Old*
- *changeonly*
- *Result*.

El estado del objeto al comienzo del método es almacenado en una variable especial ***Old***. Con esta variable se puede especificar relaciones entre los estados de entrada y salida.

```
public void addElement Object o) {
...
/** ensure !isEmpty() && contains(o);
 Old.count == count-1; **/
}
```

Para usar esta facilidad se debe implementar el método *clone* sin lanzar ninguna excepción.

```
public class Buffer implements Cloneable {

 protected Object clone() {
    Object b = null;
    try {
      b = super.clone();
    }
    catch (CloneNotSupportedException e){;}
    return b;
  }
```

La interface java.lang.Cloneable debe ser implementada. Así se genera una copia del objeto.

La cláusula ***changeonly*** seguida de una lista de atributos es utilizada en la especificación de la poscondición, para aquellos datos a los que se les permite

cambiar su estado. Los atributos no listados deben mantenerse constantes. La lista vacía establece que ***nada cambia***.

```
public void addElement (Object o) {
...
/** ensure !isEmpty() &&  contains(o);
           Old.count == count-1;
           changeonly{count,buffer}; **/
}
```

Para realizar este chequeo una copia del objeto (generado con clone), es comparado con el objeto luego de la ejecución del método. Los arreglos son manejados de otra forma.

Cuando un método retorna un valor (no void) ***Result*** permite identificar dicho valor de retorno (simple u objeto). Se usa en la postcondición.

```
public boolean empty() {
    return set.length == 0;
  /** ensure changeonly{}; Result == (set.length == 0); **/
 }

public String toString()   {
   String s = "";
   for(int i = 0; i < size(); ++i) {
      s += list.elementAt(i).toString() + " ";
    }
   return "[ " + s + "]";
 /** ensure changeonly{};Result != null; **/
}
```

Invariante de Clase: 'invariant'

- Los invariantes de clase sirven para expresar propiedades globales de las instancias de una clase, mientras que las pre y poscondiciones describen las propiedades de rutinas particulares.
- Cláusula: ***invariant*** y se establece al final de la clase.
- La invariante de clase incluye todas las aserciones que contienen condiciones que se mantienen para la clase completa, no para un método.

```
public class Buffer {
...
/** invariant 0 <= dim && dim <= buffer.length; **/
}
```

define la cantidad de elementos mínima y máxima del buffer.

Los atributos y parámetros pueden ser usados en la invariante de clase.

La invariante de clase puede ser agregada explícitamente a cada precondición y postcondición. La semántica es la misma, pero no la intención como condición invariante.

Semántica de las condiciones de las aserciones.

kind of assertion	type of expressions	usage of fields and methods	usage of local variables	usage of formal parameters	usage of Old and Result
precondition	boolean	special	not allowed	allowed	not allowed
postcondition	boolean	allowed	not allowed	allowed	allowed
class invariant	boolean	allowed	not allowed	not allowed	not allowed

Expresiones Forall y exists

- Con el uso de **forall** y **exists** se puede expresar propiedades que deben ser válidas para todos los elementos de un conjunto finito o por al menos un elemento del conjunto.

```
public class SimpleCinema {
  private boolean[] seats; // seats are "true" if reserved,
                           // false" if free.

  private int seatCount;  // counts the number of seats that are
                          // reserved AND already used.

  public SimpleCinema (int n) { // Constructs a cinema of given
                               // size. All seat are free after
                               //initialisation. @param n size of
                               // cinema

  /** require n > 0; **/
      seats = new boolean[n]; // false by default
  /** ensure forall i : {0 .. seats.length-1} # !seats[i]; **/
  }

  public boolean hasFreeSeats () {// Returns if there are free
                                  // seats in the cinema.
                                  // true if there are free
                                  // seats, false otherwise

     for (int i = 0; i < seats.length; i++)
      if (!seats[i]) return true;
     return false;
     /** ensure Result == (exists i : {0 .. seats.length-1} #
                    !seats[i]); **/
   }
```

Corchetes

```
/** require [buffer_not_full] !isFull();
            [object_is_valid] o != null; **/
```

Las etiquetas establecidas entre corchetes delante de la aserción, identifican a la aserción en los mensajes de error.

Ejemplo Completo

```
class Pila {
   final static int maxpila = 100; // atributos
   Object [] elementos;
   int cima;

Pila() // Constructor de la clase Pila
{
  elementos = new Object[maxpila];
  cima = -1 ;
}

public boolean estaVacia() // verifica si la pila esta vacia
{
  return (cima == -1);
/** changeonly{ }; **/
}

public boolean estaLlena() // verifica si la pila esta llena
{
return (cima == maxpila-1);
/** changeonly{ }; **/
}

public void meter(Object o) // agrega un elemento en la pila
{
 /** require !estaLlena(); o != null; **/
 cima++;
 elementos[cima]=o;
/** ensure !estaVacia();
    Old.cima == cima-1;
    changeonly{cima,elementos}; **/
}

public Object sacar() // saca un elemento del tope de la pila
{
/** require !estaVacia(); **/
 Object aux;
 aux= elementos[cima];
 cima--;
 return (aux);
/** ensure !estaLlena();
    Old.cima == cima+1;
    changeonly{cima,elementos}; **/
}
}
```