Ingeniería del Software II. Curso 2004/2005.

Enunciado de la práctica del segundo parcial.

Puede hacerse individualmente, por parejas o en grupos de tres (la nota final dependerá de la calidad del trabajo y del número de personas que hayan colaborado).

Se desea disponer de implementaciones Java de los tipos *Collection*, *Set*, *Bag* y *Sequence*, y ya se ha escrito el código de gran parte de las dos primeras (ver anexo 2).

Se pide:

1) Anote con invariantes OCL las clases *Set*, *Bag* y *Sequence* y sus operaciones. Anote con precondiciones y postcondiciones OCL las operaciones de las clases *Set*, *Bag* y *Sequence*. **Hasta 1,5 puntos.**
2) Si lo desea, implemente las operaciones que faltan de la clase *Set*. En caso de hacerlo, implemente una clase OclExpression, que será el tipo de los argumentos pasados a las operaciones *select*, *reject*, *exists*, *forAll*, etc. **Hasta 3 puntos, dependiendo de cuánto implemente.**
3) Aplicando el proceso de pruebas descrito en el anexo 3 y utilizando los "operadores tradicionales" de la herramienta Mujava, pruebe la clase *Set* que se adjunta. Indique cuántos mutantes se han generado y cuáles de ellos son funcionalmente equivalentes. Construya casos de prueba para matar tantos mutantes como sea posible. **Hasta 2 puntos, dependiendo del porcentaje de mutantes muertos. Hasta 1 punto más si prueba las operaciones que haya escrito en el apartado 2.**
4) Utilizando desarrollo dirigido por las pruebas y la herramienta JUnit, implemente las clases *Bag* y *Sequence*, que deben ser necesariamente especializaciones de la clase *Collection* del anexo 2 **Hasta 2 puntos.** Si, a partir de *Collection*, construye una clase de prueba abstracta (*TestCollection*, por ejemplo) que sirva como base para crear las clases de prueba de *Bag* y *Sequence*, añada **1 punto más.**

En papel se debe entregar:

Entregable P1. El ejercicio 1.

Entregable P2. Si ha resuelto todo o parte del ejercicio 2, el código de la clase *OclExpression* y el código de las clase *Set*.

Entregable P3. Listado de los mutantes funcionalmente equivalentes (el código fuente no, sólo los nombres que les asigna Mujava), breve informe con los resultados obtenidos (y una interpretación) y una breve descripción de la técnica o heurística utilizadas para generar los casos de prueba.

Entregable P4. Código fuente de las clases *Bag* y *Sequence* y de las clases de prueba. Resultados de la ejecución de los casos de prueba.

En CD o disquete, entregue todo lo anterior más el código fuente de todos los mutantes, salvaguardando la estructura de directorios de Mujava.

ANEXO 1.

Para escribir las invariantes de clases, tenga en cuenta las siguientes descripciones:

Collection: ninguna.

Set: contiene elementos de cualquier tipo sin repetir. Entendemos que dos elementos están repetidos si, al compararlos con *equals*, el resultado es *true*.

Bag: contiene elementos de cualquier tipo que pueden estar repetidos.

Sequence: contiene elementos de cualquier tipo que pueden estar repetidos; los elementos son accesibles usando un indexador de tipo entero.

Restricciones sobre operaciones.

La implementación de las operaciones en *cursiva* es optativa (corresponde al ejercicio 2).

a) Bag

Operaciones de *Bag(T)* (*T* es el tipo de los elementos de la bolsa; con r nos referimos al resultado)	
Operación	**Postcondiciones**
union(s : Set) : Bag	r contiene todos los elementos de los dos conjuntos unidos, y r no contiene elementos repetidos
union(b : Bag) : Bag	r contiene todos los elementos del conjunto y de la bolsa unidos, pudiendo haber elementos repetidos
equals (s : Bag) : boolean	True si los dos conjuntos contienen los mismos elementos (equivale al operador = de Bag)
intersection(s : Set) : Set	r contiene los elementos que están en ambos conjuntos
intersection(b : Bag) : Bag	r contiene los elementos que hay en el conjunto y la bolsa
including(x : Object) : Bag	r contiene los elementos que tenía antes de ejecutar la operación y, si x no pertenecía al conjunto, también a x
excluding(x : Object) : Bag	r contiene los elementos que tenía antes de ejecutar la operación y, si x pertenecía al conjunto, excepto x
count(x : Object) : int	r devuelve el número de elementos de la bolsa
asSequence() : Sequence	
asSet() : Set	
iterate(expr : OclExpression)	
select(expr : OclExpression) : Bag	*r contiene los elementos que cumplen la condición expr*
reject(expr : OclExpression) : Bag	*r contiene los elementos que no cumplen la condición expr*
collect(expr : OclExpression) : Bag	*r contiene los elementos que cumplen la condición expr (ojo, porque ahora r es una Bag)*

b) Sequence

Operaciones de *Sequence(T)* (*T* es el tipo de los elementos de la secuencia; con *r* nos referimos al resultado de la operación)	
Operación	**Postcondiciones**
count(x : Object) : int	
= (s : Sequence) : boolean	True si ambas secuencias son iguales
union(s : Sequence) : Sequence	r contiene todos los elementos de la secuencia sobre la que se ejecuta la operación y, a continuación, los elementos de la secuencia pasada como parámetro
append(x : Object) : Sequence	r contiene los elementos que tenía antes, y el elemento pasado como parámetro como último elemento
preprend (x : Object) : Sequence	
subsequence(inf : int, sup:int) : Sequence	r contiene los elementos desde la posición inf hasta la posición sup, ambas inclusive. **Precondiciones:** inf<=sup; inf>=1; inf<=count; sup>=1; sup<=count
at(pos : int) : Object	**Precondiciones:** pos>=1 y pos<=count
first() : Object	**Precondiciones:** count>0
last() : Object	**Precondiciones:** count>0
including(x : Object) : Sequence	Igual que append
excluding(x : Object) : Sequence asBag() : Bag	r contiene los mismos elementos que antes, pero se han quitado todas las apariciones de x. Las posiciones eliminadas no quedan vacías, sino que se habrán eliminado. Los elementos están en el mismo orden que antes.
asSet() : Bag *iterate(expr : OclExpression)*	
select(expr : OclExpression) : Sequence *reject(expr : OclExpression) : Sequence*	*r contiene los elementos que cumplen la condición expr* *r contiene los elementos que no cumplen la condición expr*
collect(expr : OclExpression) : Sequence	*r contiene los elementos que cumplen la condición expr (ojo, porque ahora r es una Bag)*

ANEXO 2. Disponible en la página web de la asignatura.

Collection.

```
package ocl;
import java.util.ArrayList;

public abstract class Collection
{
  protected ArrayList mE;

  public abstract int size();
  public abstract boolean includes(Object x);
  public abstract int count(Object x);
  public abstract boolean includesAll(Collection c);
  public abstract boolean isEmpty();
  public abstract boolean notEmpty();
  public abstract double sum() throws Exception;
  public abstract boolean exists(OclExpression expr);
  public abstract boolean forAll(OclExpression expr);
  public abstract Object iterate(OclExpression expr);
}
```

OclExpression.

```
package ocl;

public class OclExpression
{
}
```

Set.

```
package ocl;

import java.util.ArrayList;

public class Set extends Collection
{
  /*** Operaciones propias de Set.
   * Más abajo, aparecen las heredadas de Collection
   * ***/

  public Set()
  {
    mE=new ArrayList();
  }

  public Set union(Set s)
  {
    Set result=new Set();
    for (int i=0; i<size(); i++)
      result.mE.add(mE.get(i));
    for (int i=0; i<s.size(); i++)
      if (!result.includes(s.mE.get(i)))
        result.mE.add(s.mE.get(i));
    return result;
  }

  public boolean equals(Object o)
  {
    if (!(o instanceof Set))
      return false;
    Set s=(Set) o;
    if (size()!=s.size())
      return false;
    for (int i=0; i<size(); i++)
      if (!s.includes(mE.get(i)))
        return false;
    return true;
  }

  public Set intersection(Set s)
  {
    Set result=new Set();
    for (int i=0; i<size(); i++)
      result.including(mE.get(i));
    return result;
  }

  public Set difference(Set s)
  {
    Set result=new Set();
    for (int i=0; i<size(); i++)
      result.including(mE.get(i));
    for (int i=0; i<result.size(); i++)
      if (s.includes(result.mE.get(i)))
        result.excluding(mE.get(i));
    return result;
  }

  public Set including(Object x)
  {
    if (!this.includes(x))
      mE.add(x);
    return this;
  }

  public Set excluding(Object x)
  {
    if (this.includes(x))
      mE.remove(x);
    return this;
  }

  public Set symmetricDifference(Set s)
  {
    Set result=new Set();
    for (int i=0; i<size(); i++)
      if (!s.includes(mE.get(i)))
        result.includes(mE.get(i));
    for (int i=0; i<s.size(); i++)
      if (!includes(s.mE.get(i)))
        result.includes(s.mE.get(i));
    return result;
  }

  /*** Las cabeceras de las operaciones select,
reject, collect, asSequence y
   * asBag aparecen más abajo comentadas y sin
implementación
   * ***/

  /*** Implementación de las operaciones heredadas
de Collection ***/
  public int size()
  {
    return mE.size();
  }

  public boolean includes(Object x)
  {
    return (mE.indexOf(x)!=-1);
  }
```

```
public int count(Object x) {
  return (includes(x) ? 1 : 0);
}

public boolean includesAll(Collection c)
{
  for (int i=0; i<c.size(); i++)
    if (!this.includes(c.mE.get(i)))
      return false;
  return true;
}

public boolean isEmpty()
{
  return (this.size()==0);
}

public boolean notEmpty()
{
  return !isEmpty();
}

public double sum() throws Exception
{
  double result=0;
  for (int i=0; i<size(); i++) {
    Object o=mE.get(i);
    if (o instanceof Number)
    {
      result+=((Number) o).doubleValue();
    } else
        throw new Exception("Hay al menos un
elemento que no es número");
    }
    return result;
  }

  public boolean exists(OclExpression expr) {
    return false;
  }

  public boolean forAll(OclExpression expr) {
    return true;
  }

  public Object iterate(OclExpression expr);
    return null;
  }

  //public Set select(OclExpression expr) { }

  //public Set reject(OclExpression expr) { }

  //public Set collect(OclExpression expr) { }

  //public Set union(Bag b) { }

  //public Set intersection(Bag b) {  }

  //public Sequence asSequence() { }

  //public Bag asBag() { }
}
```

ANEXO 3.

Proceso de pruebas al usar mutación.

1) Construir casos de prueba y pasarlos a la clase que se está probando (*class under test* o *CUT).* Recuerde que es muy conveniente escribir correctamente un método *public String toString()* y un método *public boolean equals(Object)* en la CUT.
2) Si los casos de prueba encuentran fallos, corregir la CUT y volver a pasar los casos, hasta que no se descubran fallos en la CUT.
3) Generar los mutantes. Elimine los mutantes en los que la mutación haya afectado al método *equals* o al *toString*. Elimine los mutantes funcionalmente equivalentes que vaya encontrando.
4) Ejecutar los casos de prueba contra la CUT y contra los mutantes. Si la ejecución de algún mutante "cuelga" el ordenador, revise el mutante porque puede haber ocurrido que la mutación haya hecho, por ejemplo, que la variable contador de un *for* se decremente en el cuerpo del *for.* Por mutación débil, considere que estos mutantes están muertos y elimínelos del conjunto. Es muy probable que, entre los mutantes que han permanecido vivos, haya mutantes funcionalmente equivalentes, que debe eliminar. Si descubriera fallos en la CUT, debe retroceder al paso 1.
5) Si mata el 100% de los mutantes, ya ha terminado. Si no, debe construir más casos de prueba hasta conseguir matar todos los mutantes.

Dudas, ampliación de información, etc.; en horario de tutorías. 926.295300 ext. 3730. Macario.polo@uclm.es