Generador automático de casos de prueba unitarios para Junit

Autores:

- Jaime Sáez Bertrand
- Fernando Rodríguez Acero

Directora

- Elvira Albert

Proyecto de Sistemas Informáticos,
Facultad de Informática,
Universidad Complutense de Madrid

Autorización Legal

Los abajo firrmantes, matriculados en la asignatura de Sistemas informáticos de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a los autores el presente trabajo de proyecto de Sistemas Informáticos: "Generador automatico de casos de prueba unitarios para Junit", realizado durante el curso académico 2009-2010, bajo la dirección de Elvira Albert (UCM), incluyendo la propia memoria, el código, la documentación y el prototipo desarrollado.
También autorizan a la biblioteca de la Universidad Complutense de Madrid, a depositarlo en el archivo institucional E-Prints Complutense, con el objeto de incrementar la difusión, uso e impacto del trabajo en internet y garantizar su preservación y acceso a largo plazo.

Fernando Rodriguez Jaime Sáez

Resumen

Este trabajo está muy ligado al proyecto Pet. El objetivo de este programa es la generación automática de casos de prueba partiendo de un *byte code* de java (JBC). El sistema recibe como entrada un JBC y devuelve un conjunto de casos de prueba. Inspeccionando dichos casos se pueden detectar las situaciones conflictivas y/o especiales del programa, bajo las cuales se puede determinar si la aplicación funciona. Este conjunto de *test cases* son devueltos de manera implícita en términos Prolog, indicando los valores de entrada, salida y sus *heaps* correspondientes.

Este proyecto también está relacionado con las pruebas unitarias y con Junit, *framework* que permite realizar la ejecución de clases Java de manera controlada, permitiendo evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.

El objetivo de este proyecto de fin de carrera radica en añadir a Pet la funcionalidad de crear automáticamente un archivo Junit con todos los métodos de prueba necesarios, automatizando por completo el trabajo del *testing* unitario y ahorrando mucho tiempo y esfuerzo al desarrollador del *software*. Para ello recoge los casos de prueba y demás información relevante de la base de conocimiento de Pet, analiza estos datos y genera un archivo con el código necesario para testear los métodos creando e inicializando todos los objetos necesarios.

Summary

This work is related to the PET project. The objective of PET is to automatically generate test cases from a Java byte code (JBC) program. The system receives as input a JBC and returns a set of test cases. From such test case, it is possible to spot erroneous situations and/or special program executions, from which the user can determine if the application works correctly. This set of test cases are returned in the form of Prolog terms, which contain the values of input, output and their respective heaps.

This project is also related to unit testing and thus Junit. This framework enables the execution of Java classes in a testing mode which in turn allows the user to evaluate whether the execution of each of the methods of the class behaves as expected.

The goal of this master thesis is to extend PET with the functionality to automatically create Junit class. This way we achieve complete automation of the unit testing process and save much time and effort to the software developer. The main contribution is taking the test cases and other relevant information from the database of PET and parsing this information in order to generate a Java file with the code to test the methods, creating and initializing all necessary objects.

Palabras clave:
Junit, Pet, Testing unitario, generador automático, Java, Reflection, Prolog

Indice

Introducción

En esta seccíon explicaremos cual es el contexto del trabajo, en que se basa, qué es lo que hace y mostraremos algunos ejemplos de que como es su salida.

1.1 - Objetivos del proyecto

No se puede empezar este apartado sin indicar que nuestro trabajo está intrínsecamente ligado al proyecto PET, desarrollado por Elvira Albert, Miguel Gómez-Zamalloa, Germán Puebla y José Miguel Rojas. En el apartado 3 se explica con detalle en qué consiste, pero de manera resumida puede decirse que el objetivo de esta aplicación es la generación automática de casos de prueba partiendo de un *byte code* de java (JBC). El sistema recibe como entrada un JBC y devuelve un conjunto de casos de prueba en el que quedan reflejadas las situaciones conflictivas y/o especiales del programa, bajo las cuales se puede determinar si la aplicación funciona. PET realiza este trabajo utilizando como unidad de trabajo un método de una clase. Este conjunto de *test cases* son devueltos de manera implícita en términos Prolog, indicando los valores de entrada, salida y sus *heaps* correspondientes. Dado que la comprensión de estos términos es muy poco intuitiva, PET incluye una serie de herramientas para mostrar de manera más clara la información devuelta. (ver sección 3.3).

Este proyecto de fin de carrera también está ligado al mundo del *testing* unitario y del JUNIT (ver apartado 2). Actualmente el principal problema de JUNIT y de la mayoría de los entornos de testing, es que el programador ha de realizar dos laboriosos trabajos de ingeniería:

1. Averiguar cuáles son los casos necesarios a probar para asegurarse que un método funciona, es decir tiene que analizar su código para extraer los *test cases*.
2. Programar los métodos de test de manera manual, creando el escenario adecuado para llamar al método a probar. Para ello normalmente ha de inicializar una gran cantidad de objetos y variables de entrada y/o salida.

Gracias a PET, el punto primero, queda resuelto, pero el segundo no, puesto que PET sólo describe como tiene que ser este escenario de prueba. El objetivo de este proyecto de fin de carrera radica en añadir a PET la funcionalidad de crear automáticamente métodos de Junit en java, automatizando por completo el trabajo del testing unitario y ahorrando mucho tiempo y esfuerzo al desarrollador del *software*.

1.2 - Salida de PET, el *test case*

Como es lógico, para poder crear los métodos de JUNIT, antes ha de ejecutarse PET para obtener los casos de prueba del método, una vez hecho esto es de vital importancia comprender bien la salida y ser capaz de extraer toda la información, para poder así crear, más adelante, las variables y objetos necesarios para el escenario del método de prueba.

Una vez ejecutado Pet, devuelve los casos de prueba para un determinado método. Estos datos están representados mediante un término prolog en el que se incluye la información relevante del caso, consta de 8 campos y tiene este aspecto:

```
test_case(Signatura, [[valores_de_entrada], heap_de_entrada],
[valor_salida, heap_salida, Exec], OP1, OP2, Num_caso)
```

Veamos de manera individual en qué consiste cada elemento:

- **Signatura del método**: En este campo queda reflejada la clase, el paquete y el nombre del método al que pertenece el caso de prueba, también se indica los tipos de los argumentos de entrada y de salida.
- **Lista de valores de entrada:** En esta lista se muestra cual es el valor de cada argumento de entrada. En caso de que el tipo sea primitivo, como enteros o booleanos, su valor aparece de manera absoluta indicándolo tal cual, sin embargo cuando el tipo del argumento es un objeto o un array el valor de este campo es una referencia al *heap*, donde se encuentra la información relativa a ese elemento.
- ***Heap* de entrada:** Es una lista que contiene todos los arrays y los objetos que influyen en el desarrollo del caso de prueba. En el caso de los arrays, cada dimensión se guarda en un elemento del *heap* y en el caso de los objetos, se indica su signatura y el valor de cada uno de sus campos.
- **Exec**: Indica si el caso de prueba implica el lanzamiento de una excepción. De ser así, el valor de este campo es una referencia al *heap* de salida en el que encuentra un objeto con los datos de la excepción correspondiente.
- **OP1** y **OP2** hace referencia a distintas opciones internas de Pet, irrelevantes en este apartado
- **Num_caso**: Indica el numero de caso al que hace referencia el termino prolog. A medida que se van creado casos de prueba este contador va aumentado, tomando cero como valor inicial.

1.3 - Ejemplos de *test case*

Para aclarar y facilitar la comprensión de cada uno de estos campos citados en el apartado anterior veamos, mediante unos ejemplos que información puede obtenerse de los *tests cases*.

En el primer ejemplo se analiza un *test case* muy sencillo en el que no intervienen los objetos. Es el resultado de ejecutar Pet para un método que calcula el mínimo común múltiplo de dos números. Si nos fijamos en este término:

```
public static int lcm(int x,int y){
        int gcd = gcd(x,y);
        return abs(x*y/gcd);
    }
```

```
test_case(ints/Arithmetic.lcm(II)I, [[-10, 0], heap(A, B)], [0,
heap(A, B), ok], [], [], 7)
```

Atendiendo al primer campo se deduce que el método se llama lcm, su clase es Arithmetic y su paquete es ints, que recibe como argumentos de entrada dos enteros y devuelve un entero. Si nos fijamos en la lista de valores de entrada y salida, vemos que las variables valen -10 y 0 respectivamente y que el valor de salida es 0. Como en los heaps no hay ningún objeto, podemos deducir que su estado es irrelevante. El *flag* Exec indica que no ha habido excepción y por último el campo num_caso indica que estamos ante el octavo test case.

Utilizando el mismo método del mínimo común múltiplo si analizamos este caso:

```
test_case(ints/Arithmetic.lcm(II)I, [[0, 0], heap(A, B)], [C, heap([
(0, object(java/lang/ArithmeticException, []))|A],B), exception(0)],
[], [], 9)
```

Vemos que si ambos argumentos de entrada valen 0, el valor de salida es indeterminado y que el campo Exec vale *exception*(0). Eso quiere decir que se ha producido una excepción cuyo tipo se encuentra referenciado en la posición 0 del heap de salida. En esta posición hay almacenado un objeto de la clase `java.lang.ArithmeticException`, que nos indica el tipo de excepción producida.

Otro elemento con el que tambien se trabaja en este proyecto son los arrays. En este caso de prueba:

```
public   int v1(int[] aa){
      if (aa[1] < 0) return -1;
      else return aa[1];
}
```

```
test_case(spooky/spookytypes.v1([I)I, [[ref(0)], heap([ (0, array(I,
2, [-10, 0|B]))|C], D)], [0, heap([ (0, array(I, 2, [-10, 0|B]))|C],
D), ok], [], [], 0)
```

Vemos que se tiene un método que recibe como argumento de entrada un array de enteros (representado por el tipo '[I'). El valor de entrada es una referencia al elemento 0 del *heap*. Ambos *heaps* son iguales, lo que indica que durante la ejecución del método el *heap* de la aplicación no ha de cambiar. Los dos tienen en su interior la tupla (0, array(I, 2, [-10, 0|B]) esto indica que en la posición 0 hay un array de enteros con dos elementos -10 y 0.

Pet también trabaja con arrays multidimensionales. Cada objeto de este tipo implica crear tantos elementos en el *heap* como dimensiones tenga el array, así por ejemplo una matriz de dos dimensiones como esta:

```
ref(B)= {null, null, {-10, -10, -10}}
```

podría tener este aspecto en el *heap*:

```
heap([ (B, array(C, 3, [D, E, ref(F)|G])), (F, array(I, 3, [-10, -10,
-10|H]))|J], K)])
```

Hemos visto ejemplos de *test cases* sin *heap*, con excepciones, con vectores y con arrays, ahora sólo falta analizar como Pet representa los objetos en sus casos de prueba.

Supongamos una clase Fracción con dos atributos enteros (Numerador y Denominador) y un método *simplify* que se encarga de simplificar la fracción. Esta función modifica los atributos del objeto que invoca su llamada, es decir, simplifica el elemento this.

Vamos a analizar un posible caso de prueba para este método, pero como ese término prolog es algo más complejo que los anteriores se descompone para facilitar la comprensión:

```
public void simplify(){
        int gcd = ints.Arithmetic.gcd(num,den);
        num = num/gcd;
        den = den/gcd;
}
```

Signatura: heap/Fraccion.simplify()
Parámetros de entrada: [ref(B)]
Heap de entrada:heap([(B, object(heap/Fraccion, [field(heap/Fraccion.num:I, 10), field(heap/Fraccion.den:I, 5)|C]))|D], F)]
Valor de salida: none
Heap de salida:heap([(B, object(heap/Fraccion, [field(heap/Fraccion.num:I, 2), field(heap/Fraccion.den:I, 1)|C]))|D], F)]

Si nos fijamos en el *heap*, puede observarse como es la representación de los objetos. Estos constan de una tupla object(Tipo, lista_campos), en la que se dice cual es el tipo del objeto y su lista de atributos, esta lista su vez está formada por tuplas field(tipo, valor), en la que se detalla el nombre de cada campo, la que clase pertenece y su valor. Como puede observarse en la Signatura, el método *simplify* no tiene ningún argumento de entrada, pero sin embargo, en los parámetros si que hay un elemento, ref(B), esta referencia indica cual es el elemento del heap, que hace de this, es decir el

elemento que realiza la llamada al método y sobre el cual se hará la simplificación. Cabe mencionar que en todos los *test case*, Pet incluye un elemento this en el *heap* y lista de entrada, siempre y cuando el método no sea estático.

Dado que java es un lenguaje orientado a objetos, Pet trabaja con ellos y nuestro generador de JUNITs ha de poder crear todos los objetos que sean necesarios para probar el método, es por ello por lo que hemos tenido que trabajar muy duramente para conseguir un buen tratamiento de estas estructuras de datos.

1.4 - Características generales de JUNIT

Como se ha dicho en el apartado 1.1, el objetivo del proyecto es trascribir la información relativa a los casos de prueba, generada por Pet y hacer un programa java que permita crear los test cases de los métodos correspondientes. Para esta tarea nos hemos apoyado en una plataforma llamada Junit que es un conjunto de bibliotecas creadas por Erich Gamma y Kent con el objetivo de realizar pruebas unitarias de aplicaciones Java (ver apartado 2.5).

Junit proporciona una serie de métodos que se encargan de hacer comprobaciones, los llamados asserts. Tenemos métodos *assert* para ver si dos objetos son iguales (assertEquals...), para comprobar si un objeto es nulo (assertNull) o demostrar si dos variables apuntan a un mismo objeto (assertSame...)

La lógica de los *asserts* es simple, si devuelve cierto quiere decir que ha cumplido el test, en caso contrario lanza un FailedAssertionError, es decir un error en el testeo.

En nuestro programa sólo utilizamos assertEquals y assertArrayEquals. El primer método recibe dos objetos cualesquiera y dice si son iguales, para hacer esta comprobación recurre al método *equals* de la clase a la que pertenece, y de no existir simplemente comparará las referencias. AssertArrayEquals recibe dos arrays y comprueba que todos los elementos coinciden en ambos arrays, para ello también llama al equals del objeto.

1.5 - Estructura del Junit generado

Las clases de Junit generadas por el programa tienen una serie de características comunes, veamos cuales son:

El nombre del archivo generado es el mismo al de la clase a la que pertenece el método a probar, añadiendo "Test" al final. Por ejemplo si probamos un método de la clase heap.java, el archivo resultante se llamará heapTest.java.

El archivo generado pertenece al mismo paquete del método que se quiere probar. Como es propio de java, el nombre de la clase será el mismo que el nombre del fichero. Para poder acceder a los métodos de Junit estas clases

generadas extienden de otra llamada TestCase incluida en las librerías de Junit.

Si Pet devuelve varios casos de pruebas para un mismo método, todas esas pruebas son agrupadas bajo el mismo método de Junit.

Los nombres de los métodos generados tienen una sintaxis establecida:

```
public void test_N___R___P1_P2_P3 ()throws Exception{}
```

siendo N el nombre del método, R el tipo del valor devuelto y P1, P2, P3… los tipos de los argumentos de entrada. Por ejemplo el método generado para `public int lcm2(int x,int y)` se llamaría public void test_lcm2___int___int_int(){…}
Dado que el en nombre del método no se puede poner el carácter ` [ `, el tipo vector se representa por Array$Tipo_vector, así el procedimiento `public int[] fibSequence(int n)` seria `public void test_fibSequence___Array$int___int(){}`

Después de crear el archivo la clase y la cabecera del método queda lo más importante, escribir las instrucciones necesarias que reflejen los casos de pruebas devueltos por Pet.
Estas instrucciones son ejecutadas en un estricto orden: primero aparecen las declaraciones de variables, después se crean todos los objetos necesarios y a continuación se da valor a las variables y atributos. Una vez hecho esto, como ya se tiene creado el entorno necesario, se llama al método del que se quiere verificar su funcionamiento y para terminar, se ejecutan los asserts que comprueban si la salida devuelta es la esperada o no. En el apartado 1.7 pueden verse algunos ejemplos del código generado.

Los tests cases generados por Pet que devuelven excepciones son tratados de manera especial. En estos casos no hay que analizar el valor de salida como en los métodos normales, nuestro método de prueba a de provocar el lanzamiento de la excepción y ver si su tipo coincide con el indicado por pet. Es decir que nuestro *assertEquals* se hace sobre el tipo de la excepción y no sobre el valor devuelto. La estructura de los casos con excepción es muy parecida a la del restos pero con una gran diferencia: Dado que nuestro método de prueba lanzará y capturará un error de ejecución , cada *test case* ha de generarse en un método individual ya que solo se puede analizar una excepción en cada bloque catch. En el apartado 1.7.2 puede verse un ejemplo de esta situación.

1.6 – Opciones disponibles para la generación del Junit

El programa recibe dos parámetros que permiten adaptar el proceso de generación del archivo de pruebas. Estos son la posibilidad de poder acceder a atributos privados de una clase mediante *reflection* y brindar la posibilidad de añadir el estado del *heap* a los requisitos de satisfabilidad del método.

1.6.1 – Análisis completo

Es un parámetro *booleano* con el que se controla los elementos a los que se les aplica el predicado *assert* de Junit, es decir esta opción influye a la hora de establecer las variables que son necesarias analizar para determinar si la ejecución es satisfactoria o no.
Si la opción esta desactivada, los únicos elementos que se comparan son los que aparecen como argumentos en la salida y el objeto this, de esta manera el estado del *heap* después de la ejecución no se tiene en cuenta a la hora de determinar si el método es correcto.

Veámoslo con un ejemplo. Supongamos que se tiene un método para el que Pet devuelve esta salida, con dos casos de prueba:

In		Out	
Input args	**Input heap**	**Ret**	**Ouput heap**
ref(B),ref(C)	heap([object(B),object(C)]	9	heap([object(B),object(C)]
ref(B),ref(C)	heap([object(B),object(C)]	8	Heap([heap([object(B)])

Si el análisis no es completo, el código Junit contendría dos casos de prueba en los cuales se crearían los objetos B C, después se llamaría al método a testar pasándoles como argumentos B y C. Para terminar se comprobaría, mediante el predicado *assert*, que la salida es nueve u ocho respectivamente. Sin tener en cuenta el valor del *heap* de salida.

Esto puede ser un problema puesto que al usuario le puede interesar, aparte de comprobar que el método devuelve lo que tiene que devolver, que se asegure que todos los objetos del *heap* de salida mantiene los valores deseados después de la ejecución. Esto se consigue activando la opción de análisis completo, cuando se hace esto, el sistema se asegura que todos los objetos que están en el heap de entrada tienen el valor indicado en el *heap* de salida después de llamar al método que se quiere probar.
Así en el caso primero de nuestra tabla además de mirar si el método devuelve 8, comprobaría que los objetos B y C mantienen su valor. En el segundo caso del ejemplo habría que ver si la función retorna 8, si el objeto B no cambia y el C desaparece, es decir vale null.

Mantener la opción de análisis completo desactivada puede ser recomendable para casos en los que heap sea extremadamente grande, puesto que se reduce drásticamente el número de objetos generados y de instrucciones *asserts*. Para el resto de casos en los que el tiempo de ejecución no es importante se recomienda realizar el análisis completo.

1.6.2 Uso de Reflection

En muchas ocasiones es complicado crear una instancia de un objeto con unos determinados valores, mediante sus constructoras y sus métodos *setters*, ya que habitualmente el programador no define estas funciones, por esta razón la manera más sencilla de realizar esta tarea es crear el objeto vacío e ir asignado valores a los atributos accediendo directamente a ellos. Esta técnica tiene un grave problema ya que si el campo es privado no es posible acceder a él desde una clase externa. Este problema puede ser subsanado gracias a la técnica de *reflection*.
Java incluye en su API la librería *Reflection* la cual permite al código examinar y "reflexionar" sobre los componentes Java en tiempo de ejecución y también brinda la posibilidad de usar estos miembros reflexionados.
La Reflexión se utiliza para instanciar clases e invocar métodos usando sus nombres, un concepto que brinda a la programación de un gran dinamismo. Clases, interfaces, métodos, campos, y constructoras pueden ser descubiertos y usados en tiempo de ejecución.
Hemos aprovechado esta funcionalidad de Java para poder acceder y modificar, en tiempo de ejecución, los atributos privados de las clases, pudiendo así crear cualquier objeto indicado por Pet.

Relacionado con el uso del *reflection* el generador de Junit brinda la posibilidad de utilizarlo o no mediante una variable booleana. Su uso es muy recomendable cuando la clase a la que pertenece el método a probar contiene atributos privados, dado que si se accede al campo directamente, el archivo resultante contendrá errores de compilación.
Por otra parte el activar esta opción implica la escritura de una gran cantidad de líneas de código lo que supone que el archivo resultante pierde claridad y se hace más difícil de entender.

En el apartado 1.7.4 se muestra un ejemplo de su utilización.

1.7 - Ejemplos de Junit generados.

En esta sección se muestra, mediante ejemplos, como funciona el generador de Junits y sus posibles opciones, para ello se analizará la salida de pet y el código java resultante.

1.7.1 - Ejemplo sencillo sin *Heap*

Analicemos un caso sencillo en el que no intervienen los objetos para nada. Po ejemplo sea el método `public static int abs(int x)`, perteneciente al paquete `ints` y la clase `Arithmetic`, que se encarga de devolver el valor absoluto de un número.

```
public static int abs(int x){
        if (x >= 0) return x;
        else return -x;
    }
```

Si ejecutamos Pet vemos que devuelve dos casos de prueba, uno para números positivos y otro para negativos, ambos sin ninguna información relevante en el heap.

In		Out		
Input args	**Input heap**	**Ret**	**Exc flag**	**Ouput heap**
-10	heap(B, C)	10	ok	heap(B, C)
0	heap(B, C)	0	ok	heap(B, C)

De acuerdo con las normas indicadas en el apartado 1.5, el archivo generado para esta salida de Pet, contiene una clase llamada `ArithmeticTest` perteneciente al paquete `ints`, con un método en el que se incluyen los dos *test cases*. Si analizamos cada uno de los casos, vemos que primero crea las variables, luego las inicializa, después llama al método abs y por último comprueba que la salida se corresponde con la esperada mediante el predicado *assertEquals*.

```
package ints ;
public class ArithmeticTest extends TestCase{
   public void test_abs___int___int(){

     //--------Case 0 Without Exception--------
     int var_in_0_a;
     int var_out_1_a;
     int expected_2_a;
     var_in_0_a = -10;
     var_out_1_a = 10;
     expected_2_a = Arithmetic.abs(var_in_0_a);
     assertEquals("the result Expected",expected_2_a,var_out_1_a);

     //--------Case 1 Without Exception--------
     int var_in_3_b;
     int var_out_4_b;
     int expected_5_b;
     var_in_3_b = 0;
     var_out_4_b = 0;
```

```
     expected_5_b = Arithmetic.abs(var_in_3_b);
     assertEquals("the result Expected",expected_5_b,var_out_4_b);
   }
}
```

1.7.2 - Ejemplo de una excepción

Como se ha dicho en el apartado 1.5, las excepciones son tratadas de manera especial, puesto que lo importante no es valorar la salida del método, sino el tipo de la excepción devuelta.

Supongamos un método que calcula el módulo de dos números, perteneciente a la clase `ints.Arithmetic` y con la signatura `public static int mod(int x,int y)`. En este caso Pet devuelve cinco casos de prueba, de los cuales dos son excepciones:

```
public static int mod(int x,int y){
        if ((x < 0)||(y <= 0)) throw new ArithmeticException();
        while (x >= y)
            x = x - y;
        return x;
    }
```

In			Out		
Input args		**Input heap**	**Ret**	**Exc flag**	**Ouput heap**
10	10	heap(B, C)	D	ex(0)	heap([(0,object(java/lang/ArithmeticException,[])])
0	1	heap(B, C)	0	ok	heap(B, C)
1	1	heap(B, C)	0	ok	heap(B, C)
2	1	heap(B, C)	0	ok	heap(B, C)
0	10	heap(B, C)	D	ex(1)	heap([(1,object(java/lang/ArithmeticException,[])])

El archivo generado contiene un método en el que se incluiría todas las instrucciones necesarias para probar los casos dos, tres y cuatro realizándolo de la misma forma que en el ejemplo anterior. Para los *test cases* en los que se produce una excepción, se crean dos medos independientes en los que se llama a la función con los valores indicados en la tabla, se captura la excepción y se comprueba que sus tipos coinciden. Este sería el código generado para el quinto *test case*.

```
 public void test_mod___int___int_int___exception_4(){
    try{
     int var_in_16_a;
     int var_in_17_a;
     var_in_16_a = 0;
     var_in_17_a = -10;
     Arithmetic.mod(var_in_16_a,var_in_17_a);
    }
    catch(Exception ex){
       assertEquals("java.lang.ArithmeticException",ex.getClass().getName());
       return;
    }
    fail("Did`nt find expected exception");
   }
```

1.7.3 - Ejemplo en el que interviene el heap

A continuación analizaremos como es el código obtenido para un *test case* en el que intervienen objetos. Estas situaciones generan códigos mucho más grandes y complejos. El uso de las opciones del programa (*reflection* y Análisis completo), obligan a estudiar cada caso de manera independiente, dado que la salida cambia mucho si activan o no.

Supongamos una clase pública Rational encarga de gestionar y representar fracciones. Con dos atributos públicos y un método para multiplicar fracciones. Un posible código de esta sería:

```
public class Rational{
    protected int num;
    protected int den;
    public Rational mul(Rational r){
        return new Rational(num * r.num,den * r.den);
    }
}
```

Si ejecutamos Pet al método `mul` devuelve un caso de prueba el cual tiene en el *heap* de entrada dos fracciones; `10/10` , `2/2` y en el *heap* de salida están las dos iniciales y una tercera en la que se devuelve el producto.

In			Out	
Input args		**Input heap**	**Ret**	**Ouput heap**
ref(B)	ref(C)	heap([(B, object(heap/Rational,[field(num:I,10), field(den:I,10) |D])), (C, object(heap/Rational,[field(num:I,2), field(den:I,2) |E]))|F], G)	ref(0)	heap([(0,object(heap/Rational,[field(num:I, 20), field(den:I, 20)])), (B, object(heap/Rational, [field(num:I,10), field(den:I,10) |D])), (C, object(heap/Rational, [field(.num:I,2), field(.den:I,2) |E]))|F], G)

El archivo Junit correspondiente a este *test case*, suponiendo que la opción de Análisis completo esta desactivada, tendría el siguiente código:

```
public class RationalTest extends TestCase{
   public void test_mul___heap_Rational___heap_Rational(){
     //--------Case 0 Without Exception--------
     Rational expected;
     Rational var_in_2_a;
     Rational var_in_3_a;
     Rational var_out_0_a;
     Rational var_out_2_a;
     var_in_2_a = new Rational();
     var_in_3_a = new Rational();
     var_out_0_a = new Rational();
     var_out_2_a = new Rational();
```

```
    var_in_2_a.num = 10;
    var_in_2_a.den = 10;
    var_in_3_a.num = 2;
    var_in_3_a.den = 2;
    var_out_0_a.num = 20;
    var_out_0_a.den = 20;
    var_out_2_a.num = 10;
    var_out_2_a.den = 10;
    expected= var_in_2_a.mul(var_in_3_a);
    assertEquals(expected,var_out_0_a);
    assertEquals(var_in_2_a,var_out_2_a);
  }
```

En este programa puede verse que se crean dos objetos de entrada `var_in_2_a` y `var_in_3_a` con valores 10/10 y 2/2 respectivamente. Como elementos de salida se declaran dos instancias de Rational `var_out_2_a` y `var_out_0_a`, la primera con un valor de 10/10 que representa el objeto this después de hacer la llamada a `mul`. El segundo objeto de salida que se crea es `var_out_0_a` en el se guarda el resultado esperado por Pet, 20/20.

Después de inicializar todas las variables, se llama al método `mul` el cual se quiere probar, almacenando en `expected` el valor devuelto.

Dado que no se realiza un análisis completo, nuestro método sólo comprueba dos cosas mediante el predicado de assert. La primera es asegurarse que el valor devuelto por la ejecución del método `mul` es igual al esperado por pet, es decir que los objetos `expected_4_a` y `var_out_0_a` son iguales. La segunda es comprobar que el objeto this no cambia durante la ejecución de `mul` Para ello se hace el assert de `var_in_2_a` con `var_out_2_a`.

Si realizamos el mismo ejemplo activando la opción de análisis completo entonces el sistema se asegura que todos los objetos que están en el *heap* de entrada tienen el valor indicado en el *heap* de salida después de llamar al método que se quiere probar. El resultado para el ejemplo anterior del método `mul` sería este:

```
public class RationalTest extends TestCase{
  public void test_mul___heap_Rational___heap_Rational(){
    //--------Case 0 Without Exception--------
    Rational expected_4_a;
    Rational var_in_2_a;
    Rational var_in_3_a;
    Rational var_out_0_a;
    Rational var_out_2_a;
    Rational var_out_3_a;
    var_in_2_a = new Rational();
    var_in_3_a = new Rational();
    var_out_0_a = new Rational();
    var_out_2_a = new Rational();
    var_out_3_a = new Rational();
    var_in_2_a.num = 10;
    var_in_2_a.den = 10;
    var_in_3_a.num = 2;
    var_in_3_a.den = 2;
```

```
    var_out_0_a.num = 20;
    var_out_0_a.den = 20;
    var_out_2_a.num = 10;
    var_out_2_a.den = 10;
    var_out_3_a.num = 2;
    var_out_3_a.den = 2;
    expected_4_a= var_in_2_a.mul(var_in_3_a);
    assertEquals(var_in_2_a,var_out_2_a);
    assertEquals(expected_4_a,var_out_0_a);
    assertEquals(var_in_3_a,var_out_3_a);
  }
```

En el código puede verse que se realizan tres *asserts*, los dos primeros son los mismos que en el caso anterior, estos se encargan de asegurarse que el elemento this tiene el valor correcto y que coinciden el valor esperado con el valor devuelto. Activando la opción de análisis completo aparece un tercer *assert*, `assertEquals(var_in_3_a,var_out_3_a)`, que compara el valor del objeto C, este objeto representa la fracción por la que se multiplicaba el elemento this que ha de mantenerse constante durante la ejecución.

1.7.4 Ejemplo de *reflection*

Como hemos visto en ejemplos anteriores, la manera que se utiliza para inicializar los campos de los objetos es accediendo directamente al atributo, pero este sistema no es viable cuando el campo es privado. El citado problema se ha solucionado usando *reflection* (apartado 1.6.1). Analicemos en detalle cómo se aplica esta técnica para conseguir acceder a los campos privados, para ello se explicará mediante un ejemplo.
Supongamos que tenemos una clase foo, que contiene dos atributos (de los cuales uno es privado) y una constructora que inicializa sus campos.

```
package pack;
public class foo{
     public  int publicAtr;
     private int  privateAtr;
     public foo(){
            publicAtr= 3;
            privateAtr=7;
    }
}
```

Si queremos crear un Junit para probar que la constructora funciona, este método sería muy simple, bastaría con crear un objeto, darle el valor correcto a sus atributos, crear otro objeto y hacer el *assert* de los dos para asegurarse que la constructora inicializa correctamente sus argumentos, pero para acceder al campo privado hay que utilizar *reflection*. Este código sería el generado por nuestro programa para realizar el Junit del la contructora de la clase Foo:

```
1.  //This code  has been generated automatically
2.  package pack;
3.  import junit.fr amework.TestCase;
4.  import static org.junit.Assert.assertArrayEquals;
5.  import static org.junit.Assert.assertEquals;
6.  import java.lang.reflect.Field;
7.  public class FooTest extends TestCase{
8.  public void test_Foo___none___()throws Exception{
9.     //--------Case 0 Without Exception--------
10.    Foo out;
11.    out = new Foo();
12.    out.publicAtr = 3;
13.    //Use reflection to modify private field privateAtr
14.    final Field fields_out_privateAtr[]= Foo.class.getDeclaredFields();
15.    for (int i = 0 ;i < fields_out_privateAtr.length; i++){
16.       if ("privateAtr".equals(fields_out_privateAtr[i].getName())){
17.           Field f =fields_out_privateAtr[i];
18.           f.setAccessible(true);
19.           f.set(out,7);
20.           break;
21.       }
22.   }
23.   Foo in= new Foo();
24.   assertEquals("the result expected",in,out);
25.   }
26.}
```

En código superior la técnica del *reflection* es aplicada en las líneas 13 – 22. Gracias a esto se consigue fijar el valor de `out.prívateAtr= 7`.
En la línea 14 mediante el método `getDeclaredFields` de la case `java.lang.reflect` se consigue una lista de objetos reflejados, cada uno de estos objetos, representa un atributo de la clase `foo`, después se recorre esa lista buscando el elemento cuyo nombre sea el del atributo que nos interesa, en este caso, `privateAtr`, a continuación en la mediante el método `set`, fijamos su valor, no sin antes hacer el campo accesible con la función `setAccessible`. Esta tiene la habilidad de suprimir el control de acceso de java, pudiendo hacer publico un elemento privado, como es lógico esto solo es posible hacer con objetos reflejados.

2 - Generación de casos de prueba

2. 1 - *Testing*

Las pruebas de software, en inglés *testing* son los procesos que permiten verificar y revelar la calidad de un producto *software*. Son utilizadas para identificar posibles fallos de implementación, calidad, o usabilidad de un programa de ordenador o videojuego. Básicamente es una fase en el desarrollo de *software* consistente en probar las aplicaciones construidas y mediante técnicas experimentales descubrir que errores tiene.

Existen muchos tipos de pruebas como:

Las **pruebas de validación** que se encargan de asegurarse que el software producido cumple con las especificaciones y su cometido.

***Testing* de caja negra**: Se denomina caja negra a aquel elemento que es estudiado desde el punto de vista de las entradas que recibe y las salidas o respuestas que produce, sin tener en cuenta su funcionamiento interno.

***Testing* de caja blanca:** Se denomina cajas blancas a un tipo de pruebas de *software* que se realiza sobre las funciones internas de un módulo. Así como las pruebas de caja negra ejercitan los requisitos funcionales desde el exterior del módulo, las de caja blanca están dirigidas a las funciones internas.

Pruebas de regresión: Se denominan pruebas de regresión a cualquier tipo de pruebas de *software* que intentan descubrir las causas de nuevos errores, o divergencias funcionales con respecto al comportamiento esperado del programa. Producidos por cambios recientemente realizados en partes de la aplicación que antes del cambio no eran propensas a derteminados errores y ahora si.

Este tipo de errores puede ser debido a prácticas no adecuadas de control de versiones, falta de consideración acerca del ámbito o extensibilidad del error que fue corregido, o simplemente una consecuencia colateral del rediseño de la aplicación.

Por lo tanto, en la mayoría de las situaciones del desarrollo de *software* se considera una buena práctica que cuando se localiza y corrige un error en el código, se haga un test completo del programa con el error ya corregido, asi el desarrollador se asegura que al realizar la última correción el resto del programa no se ha visto alterado.

Pruebas funcionales: La pruebas funcionales se hacen mediante el diseño de modelos de prueba que buscan evaluar cada una de las opciones con las que cuenta el paquete informático.

Pruebas unitarias: Asegura que cada uno de los módulos del código funciona correctamente por separado, la idea es escribir casos de prueba

para cada función o método en el módulo de forma que cada caso sea independiente del resto.

Tanto Pet (ver apartado 3) como JUNIT trabajan con pruebas unitarias puebas, es por ello por lo que se explicarán con más profundidad más adelante

Pruebas integrales o **pruebas de integración**: Las pruebas de integración es la fase del testeo en el que los módulos de *software* son combinados y testeados como un grupo. Son las pruebas posteriores a las pruebas unitarias y preceden el testeo de sistema.

Aparte de estas pruebas, sin entrar a explicar cada una de ellas cabe mencionar las Pruebas de sistema, Pruebas de aceptación, Pruebas de carga, Pruebas de prestaciones, Pruebas de recorrido y las Pruebas de mutación.

2.2 - Testing en el campo empresarial

En la cadena de valor del desarrollo de un software específico, el proceso de prueba es clave a la hora de detectar errores. Conceptos como estabilidad, escalabilidad, eficiencia y seguridad se relacionan a la calidad de un producto bien desarrollado. Las aplicaciones de software han crecido en complejidad y tamaño y por consiguiente también en costos. Hoy en día es crucial verificar y evaluar la calidad de lo construido de modo de minimizar el costo de su reparación. Mientras antes se detecte un fallo, más barata será su corrección.

En la industria, inicialmente el trabajo de programar y validar el código era realizado por el mismo equipo humano, fue apartir de los años 80, con el auge de la ingenieria del sofware, cuando se separaron estas dos tareas, dado que el proceso de prueba es un proceso técnico especializado de investigación que requiere de profesionales altamente capacitados en lenguajes de desarrollo, técnicas de pruebas y herramientas especializadas. El conocimiento que debe manejar un ingeniero de prueba es muchas veces superior al del desarrollador de software.

A medida que ha ido aumentando la necesidad de programas más fiables, se ha reconocido la necesidad de establecer unas normas en de ingeniería del software. Muchas empresas y organizaciones han establecido normas relacionadas con el testing , estas son una contribución fundamental para asegurar la producción de software de calidad. En relación a las normas, los profesionales se encuentran con un problema fundamental: la dispersión de las normas relativas al testing que, con frecuencia, han sido creadas por organismos muy diversos, bajo enfoques distintos y destinadas a ámbitos de actuación diferentes. Muchas compañías, por su parte, se han visto obligadas a generar sus propias normas cuando no disponen de unas de ámbito general. De hecho muchas organizaciones desarrollan sus propios conjuntos de normas adecuándolas a sus fines específicos. La organizaación *IEEE* trata algunos estos aspectos en los siguentes estándares*:IEEE 1008* en la que se establece normas de trabajo con las pruebas unitarias, el *IEEE 1012* estandar para validación y verificación del software, *IEEE 1028* en la

que se habla acerca de las reglas para la inspección de sofware, *IEEE 1044* en la que se clasifican las anomalías en los proyectos*, el IEEE 1061* en la que se agrupan un conjunto de normas para asegurar la calidad del software y el *BS 7925-1* que inlcuye un vocabulario acerca de los terminos utilizados en el testing.

2.3 - Pruebas unitarias

Como se ha dicho en el apardo 2.1, una prueba unitaria es una forma de probar el correcto funcionamiento de un módulo, aislándolo del resto. Dado que nuestro proyecto trabaja con este tipo de test, vamos analizar este concepto con detalle.

Para relalizar estos tests es de vital importancia obtener los Casos de Prueba o *Test Cases*, estos son un conjunto de condiciones o variables bajo las cuáles el analista determinará si el requisito de una módulo es satisfactorio.

Se pueden realizar muchos casos de prueba para determinar que un requisito es satisfactorio. Con el fin de comprobar que todos los requisitos de una aplicación son revisados, debe haber al menos un caso de prueba para cada requisito. Algunas metodologías como RUP [1] recomiendan el crear por lo menos dos casos de prueba para cada requisito. Uno de ellos debe realizar la prueba positiva de los requisitos y el otro debe realizar la prueba negativa.

Como se ha dicho para tener una prueba bien diseñada es necesario conocer de antemano el resultado de su ejecución, para ello son de gran utilidad los oráculos. Un oráculo es un elemento o artefacto que provee las salidas esperadas de cada caso de prueba dado.

La utilización o no de determinados oráculos en las pruebas unitarias es una decisión que debe plantearse, ya que pueden conllevar actividades de preparación que es necesario acometer antes del diseño y/o la ejecución de las pruebas. En este sentido es necesario evaluar siempre el coste/beneficio de su utilización, la fiabilidad del oráculo y sus necesidades de mantenimiento, ya que el oráculo deberá tener al menos el mismo mantenimiento que el sistema.

Por ejemplo, es muy común que a la hora de verificar un sistema en el que está implicado un cálculo complejo, el equipo de pruebas se prepare como oráculo un documento en que se refleje las salidas esperadas para distintas entradas. Este método conlleva, entre otros, los siguientes inconvenientes:

1. Quien prepara el documento debe llegar a conocer con detalle toda la complejidad del cálculo.

[1] Rup: Proceso Unificado Racional (*Rational Unified Process*) en inglés, es un proceso de desarrollo de software y junto con el Lenguaje Unificado de Modelado UML, constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos.

2. Si la información que llega a quien lo implementa en el sistema y a quien implementa el oráculo no es exactamente la misma y no se interpreta de la misma forma, los resultados de ambos pueden no ser coincidentes.
3. Pueden producirse errores al realizar el documento
4. Un error en las especificaciones del cálculo, producirá el mismo error en el oráculo, por lo que pasará inadvertido

Para que una prueba unitaria sea *buena* se deben cumplir los siguientes requisitos:

1. **Automatizable**: No debería requerirse una intervención manual. Esto es especialmente útil para integración continua.
2. **Completas**: Deben cubrir la mayor cantidad de código.
3. **Repetibles o Reutilizables**: No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. También es útil para *integración continua*.
4. **Independientes**: La ejecución de una prueba no debe afectar a la ejecución de otra.
5. **Profesionales**: Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Aunque estos requisitos no tienen que ser cumplidos al pie de la letra, se recomienda seguirlos o de lo contrario las pruebas pierden parte de su función.

El objetivo de las pruebas unitarias es aislar cada parte del programa y mostrar que las partes individuales son correctas. Proporcionan un contrato escrito que el trozo de código debe satisfacer. Estas pruebas aisladas proporcionan cinco ventajas básicas:

1. **Fomentan el cambio**: Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura (lo que se ha dado en llamar refactorización), puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.
2. **Simplifica la integración**: Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente. De esta manera se facilitan las pruebas de integración.
3. **Documenta el código**: Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.
4. **Separación de la interfaz y la implementación** dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro, a veces usando objetos mock (*mock object*) para simular el comportamiento de objetos complejos.
5. **Los errores están más acotados y son más fáciles de localizar** dado que tenemos pruebas unitarias que pueden desenmascararlos.

Es importante darse cuenta de que las pruebas unitarias no descubrirán todos los errores del código. Por definición, sólo prueban las unidades por sí solas. Por lo tanto, no descubrirán errores de integración, problemas de

rendimiento y otros problemas que afectan a todo el sistema en su conjunto. Además, puede no ser trivial anticipar todos los casos especiales de entradas que puede recibir en realidad la unidad de programa bajo estudio. Las pruebas unitarias sólo son efectivas si se usan en conjunto con otras pruebas de *software*.

2.4 - Qué tecnologías existen en relación a las pruebas unitarias

El *testing* unitario puede ser aplicado a la gran mayoria de programas, dado que estos suelen estar construidos de manera modular (clases, funciones, métodos...) gracias a su modularidad este tipo de pruebas han tenido mucho auge y han aparecido una gran cantidad de sistemas para trabajar con *unit cases*.

Dado que estas puebas están intrínsicamente ligadas al lenguaje del programa a probar, casi todas las plataformas tienen uno o varios sistemas para hacer puebas unitarias. A continuación se citan algunos de estos entornos:

JUnit: Entorno de pruebas para Java creado por Erich Gamma y Kent Beck. Se encuentra basado en SUnit creado originalmente para realizar pruebas unitarias para el lenguaje *Smalltalk*, Dado que el proyecto trata sobre esta plataforma, más adelante profundizaremos en ella.

TestNG creado para suplir algunas deficiencias en Junit. SimpleTest entorno de pruebas para aplicaciones realizadas en PHP. CPPUnit, versión del *framework* para lenguajes C/C++. Nunit versión del *framework* para la plataforma.NET. MOQ *framework* para la creación dinámica de objetos simuladores (*mocks*). Otros lenguajes como Ada, Fortran, Delphi, Pascal, JavaScript y Haskell, también disponen de plataformas para casos de prueba unitarios.

2.5 - Qué es JUNIT

JUnit es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que como su nombre indica se encarga de crear pruebas unitarias de aplicaciones Java. Contiene un conjunto de clases (*framework*) que permiten realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

JUnit también permite controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

El propio *framework* incluye formas de ver los resultados (*runners*) que pueden ser en modo texto, gráfico (AWT o Swing) o como tarea en Ant.

En la actualidad las herramientas de desarrollo como NetBeans y Eclipse cuentan con *plug-ins* que permiten la generación de las plantillas necesarias para la creación de las pruebas de una clase Java, estan plantillas se realizan de manera automática, dejando al programador enfocarse en la prueba y el resultado esperado. Hay que indicar que estas herramientas no generan el test, sólo facilitan su escritura. Actualmente el programador tiene que realizar un laborioso trabajo de ingeniería para averiguar cuales son los casos necesarios a probar para asegurarse que un modulo funciona, es decir tiene que analizar su código extraer los *test cases* y crear métodos de test de manera manual. Esta falta de automatizacón es el principal problema de JUNIT y de la mayoria de los entornos de *testing*.

El objetivo de este proyecto trata de subsanar esta limitación de JUNIT, creando de manera automática un *junit test* completo para un método o conjuntos de métodos de una clase, liberando al programador de la laboriosa tarea de crear una clase de test para su módulo. Gracias al proyecto Pet (ver apartado 3) este proceso se realiza de manera completamente automática sin que el usuario tenga que escribir nada.

3 - PET

3.1. Qué es PET

PET es un proyecto de investigación cuyo objetivo es la generación automática de casos de prueba partiendo del *bytecode* de java, apoyándose en la técnica de la evaluación parcial, PET responde a las siglas de "*Partial Evaluation-based Test*". El sistema recibe como entrada un *byte code* de JAVA, un conjunto de parámetros opcionales y un criterio de cobertura, y devuelve un conjunto de casos de prueba que garantizan que el criterio de cobertura ha sido alcanzado. Estos casos de prueba pueden ser devueltos de manera simbólica, en código Prolog o generando un archivo de JUNIT.

Como ya mencionamos en el punto 2.4 el *testing* unitario es aquel que se encarga de probar los módulos de un programa de manera independiente (estos módulos normalmente son métodos). El principal problema de este tipo de pruebas, reside en que el proceso de encontrar los *test cases* que verifican que un método queda probado, es una tarea ardua, en la que se requiere mucho ingenio y tiempo. Dado que la unidad de trabajo de Pet es el método, este problema queda resuelto, automatizando por completo la tarea del *testing* unitario, sin que el programador tenga que invertir esfuerzo en crear sus test.

La licencia de Pet es GPL, eso quiere decir que su código está a disposición de cualquiera que tenga interés en conocer en detalle cómo funciona el sistema. También dispone de un interfaz web [3], desde donde se puede ejecutar Pet de manera remota sin necesidad de instar ningún tipo de *software*. En la página hay disponibles una amplia variedad de ejemplos, estos cubren un gran rango de posibilidades de clases y métodos java. También si se desea se puede subir cualquier archivo ".java" o ".class" y ver cuál es la salida de Pet para ese código.

Actualmente Pet es capaz de trabajar con enteros, booleanos, arrays multidimensionales, objetos, float, char, byte, long y short.

3.2 - Cómo genera Pet los casos de prueba

En esta sección se explica de forma muy resumida cómo funciona Pet de manera interna, si se desea ver con más profundidad cómo es este proceso recomendamos que se revisen algunas de sus publicaciones [5] [1] [2] *https://costa.ls.fi.upm.es/pet/pet.php*.

En la figura 1 puede verse un esquema general de la arquitectura de Pet y cuales son los puntos más importantes para generar estos casos. Hay dos grandes pasos, el primero es la transformación del *byte code* de java (JBC) a un programa CLP (*constraint logic programming*) equivalente y el segundo paso es la generación de los casos de prueba. Como entrada recibe el JBC del programa con el que se desea trabajar y una serie parámetros entre los que destaca el criterio de cobertura.

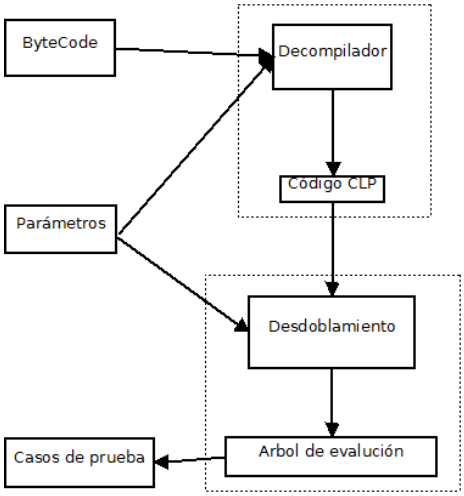

Figura 1

Paso 1: DECOMPILACON CLP:

Este proceso se encarga de transformar el JBC, recibido en la entrada a un programa PROLOG equivalente, gracias un decompilador CLP existente al que se han realizado algunas modificaciones [4].

Para mostrar como es este proceso de decompilación, se utiliza un ejemplo de un método Java que calcula el resultado de elevar una base a un exponente, este caso es muy representativo ya que tiene, excepciones, bucles y sentencias condicionales (Fig. 2).

Analizando las reglas PROLOG generadas (Fig. 2), puede observarse como el bucle ha sido detectado y convertido en una recursión y las sentencias condicionales han dado lugar sus correspondientes instrucciones CLP.

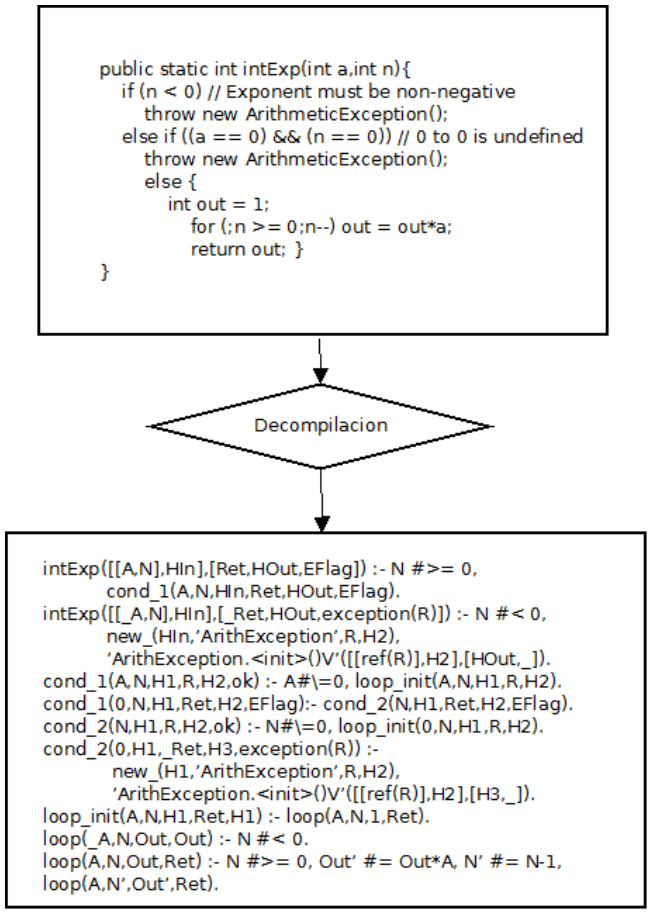

Figura 2

Paso 2: DESDOBLAMIENTO Y PROCESADO DEL Árbol

Esta tarea consiste en determinar cuáles son los distintos caminos de ejecución del programa CLP, Para ello Pet ejecuta el código de manera simbólica, asegurándose que esta tarea de desdoblamiento termina. Una vez finalizada la ejecución se analiza él árbol resultante, en esta estructura, cada rama no fallida representa un flujo de ejecución distinto y por cada rama satisfactoria se creará posteriormente un caso de prueba. La profundidad del árbol está ligada a la cobertura que se le pasa como parámetro, así se consigue definir el grado de refinamiento del proceso, asi por ejemplo si el valor de este criterio es dos, Pet realiza dos vueltas de cada bucle.

En nuestro ejemplo, si ejecutamos el código CPL de la figura , obtenemos el siguiente árbol (Fig. 3), en el que las ramas B1, B2 y B3 son los caminos que dan lugar a casos de prueba. Las propiedades de estos casos quedan reflejas en las transiciones de un nodo a otro.

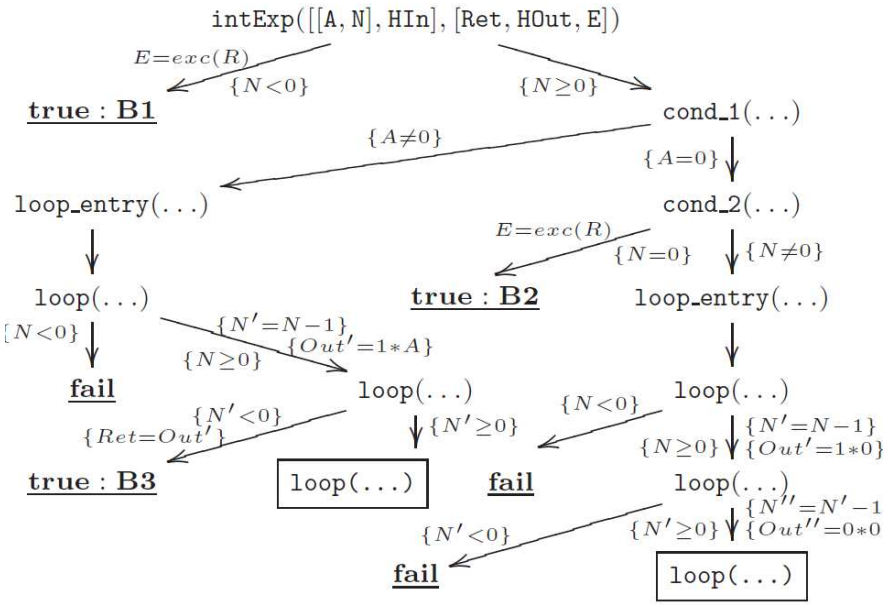

Figura 3

3.3 - Ejemplos de salida de Pet

Como se ha dicho en el apartado anterior, Pet admite una serie de parámetros de entrada, unos están destinados a dirigir la búsqueda de *test cases*, como "*Coverage criterion*" o "*Labeling strategy*", y otros permiten indicar como queremos que Pet nos muestre la salida, también es posible mostrar, aparte de los casos de prueba, más información, como la traza de ejecución. Con el fin de facilitar la comprensión, sólo se mostrarán los casos de prueba de manera numérica y simplificados. Si se desea ver en más profundidad las posibilidades de Pet, se aconseja visitar su página web [3] *https://costa.ls.fi.upm.es/pet/pet.php*.

Ejemplo 1:

Empecemos con el caso más sencillo. Supongamos un método que devuelve el valor absoluto de un número entero, su código podría ser este:

```
public static int abs(int x){
  if (x >= 0) return x;
  else return -x;
}
```

Si ejecutamos Pet, este nos devuelve dos *test cases* (ver tabla), en los que como argumentos de entrada se pasan un entero positivo y otro negativo. En este ejemplo en particular los valores del *heap* de salida y de entrada no son relevantes.

In		Out		
Input args	**Input heap**	**Ret**	**Exc flag**	**Ouput heap**
-10	heap(B)	10	ok	heap(B,)
10	heap(B)	10	ok	heap(B)

Ejemplo 2:

En el caso de probar un método iterativo para calcular módulo de dos números, dado un código como este:

```
public static int mod(int x,int y){
    if ((x < 0)||(y <= 0)) throw new ArithmeticException();
    while (x >= y)
        x = x - y;
    return x;
}
```

Pet devuelve 5 test cases, entre los que se detectan dos excepciones cuyos valores están almacenados en el heap de salida:

In			Out		
Input args		**Input heap**	**Ret**	**Exc flag**	**Ouput heap**
-10	-10	heap(B)	D	exc(0)	heap([(0, object(java/lang/ArithmeticException,[]))|B])
0	1	heap(B)	0	Ok	heap(B, C)
1	1	heap(B	0	Ok	heap(B, C)
2	1	heap(B)	0	Ok	heap(B, C)
0	-10	heap(B)	D	exc(1)	heap([(1 object(java/lang/ArithmeticException,[]))|B])

Ejemplo 3:

En este caso se prueba un método en se utiliza la recursión, para calcular el factorial de un número, cuyo código y salida de Pet se muestran a continuación:

```
public static int factRec(int x){
  if (x == 0) return 1;
  else return x*factRec(x-1);
    }
```

In		Out		
Input args	**Input heap**	**Ret**	**Exc flag**	**Ouput heap**
3	heap(B)	6	ok	heap(B)
2	heap(B)	2	ok	heap(B)
1	heap(B)	1	ok	heap(B)
0	heap(B)	1	ok	heap(B)

Los ejemplos anteriores son muy básicos dado que no interviene en ningún momento el *heap*, es decir no hay objetos. Como ya se ha dicho, Pet soporta este tipo de elementos así que vamos a ver algún ejemplo un poco más complejo.

Ejemplo 4:

Supongamos una clase Rational con dos argumentos: numerador y denominador y que tiene un método para simplificar una fracción pasada como argumento de entrada/salida.

```
public class Rational{
     protected int num;
     protected int den;
     public Rational(){
      num = 0;
      den = 1;
     }
     public void simplify(){
        int gcd = ints.Arithmetic.gcd(num,den);
        num = num/gcd;
        den = den/gcd;
      }
}
```

Si ejecutamos Pet vemos que devuelve 5 casos, con sus *heap* de entrada y salida correctamente tratados. En estos *heaps* puede verse representado el objeto de tipo Rational y el valor de sus dos atributos (num y den).

In		Out		
Input args	**Input heap**	**Ret**	**Exc Flag**	**Ouput heap**
ref(B)	heap([(B, object(Rational, [field(num: -10), field(den: 0)]])	none	ok	heap([(B, object(Rational, [field(num: -1), field(den: 0)]])
ref(B)	heap([(B, object(Rational, [field(num: 1), field(den: 0)]])	none	ok	heap([(B, object(Rational, [field(num: 1), field(den: 0)]])
ref(B)	heap([(B, object(Rational, [field(num: 0), field(den: 0)]])	none	ex(0)	heap([(0, object(java/lang/ArithmeticException, [])), (B, object(Rational, [field(num: 0), field(den: 0)]])
ref(B)	heap([(B, object(Rational, [field(num: -10), field(den: -10)]])	none	ok	heap([(B, object(Rational, [field(num: -1), field(den: -1)]])
ref(B)	heap([(B, object(Rational, [field(num: -10), field(den: 1)]])	none	ok	heap([(B, object(Rational, [field(num: -10), field(den: 1)]])

4 - TRANSFORMAR TERMINOS PROLOG DE Pet A JUNIT

Este capítulo trata sobre como generar un JUNIT a partir del término prolog que genera Pet, se analizaran todos los aspectos, la salida de Pet, el algoritmo de transformación y los casos especiales.

4.1 - Test_case:

El primer paso es comprender el *test case*, es la estructura básica sobre la que se trabaja, en el se encuentra toda la información necesaria para generar un fichero JUNIT que emule el estado del programa antes y después de la ejecución. Para facilitar la comprensión del algoritmo se ilustrarán todos los pasos con un ejemplo. El método que se ha escogido es el método insert de la clase SortedListInt, se trata de una lista enlazada cuyos elementos son SLNode.

```
public void insert(int[] dataArr){
               for (int i = 0;i < dataArr.length;i++)
                      insert(dataArr[i]);
}

public void insert(int data){
        SLNode curr, foll;

        if ((first == null) || (data <= first.data))
        first = new SLNode(data,first);
        else {
               curr = first; foll = first.next;
               while ((foll != null) && (foll.data < data)){
                      curr = foll; foll = foll.next;
               }
               curr.next = new SLNode(data,foll);
        }
}
```

El *test case* completo tiene este aspecto:

Method	In		Out		
	Input args	Input heap	Ret	Exc Flag	Ouput heap
heap/SortedListInt.insert([I)V	ref(B) ref(C)	heap([(C, array(I, 1, [0|D])), (B,object(heap/SortedListInt, [field(heap/SortedListInt.first:Lheap/SLNode;, null)|E]))|F], G)	none	ok	heap([(0, object(heap/SLNode, [field(heap/SLNode.data:I, 0), field(heap/SLNode.next:Lheap/SLNode;, null)])), (C, array(I, 1, [0|D])), (B, object(heap/SortedListInt, [field(heap/SortedListInt.first:Lheap/SLNode;, ref(0))|E]))|F], G)

Method: signatura del método al que está dirigido el test, contiene los tipos y el número de los parámetros de entrada y salida, el nombre de la función, la clase y el paquete.

```
Ej: heap/SortedListInt.insert([I)V
Heap: Nombre del paquete
SortedListInt: Nombre de la clase .
insert: Nombre del método.
([I)V : Argumentos de la función, un array de enteros de entrada
y void de salida.
```

Entrada: se compone de dos campos:

InValueList: lista de parámetros de entrada para este caso de prueba, estos pueden ser referencias a objetos que pueden existir o no en el *heap* de entrada, o si el tipo de los parámetros es primario simplemente el valor de entrada.

```
Ej: ref(B) ref(C) Estos son los parámetros de entrada, hay dos
porque el primero es el objeto que llama a la función, y el otro
es el argumento de entrada, cuando pone ref() significa que
estos dos objetos a los que se hace referencia se encuentran en
el heap de entrada.
```

InHeap: el *heap* de entrada, es decir, el estado de la memoria del programa antes de que se ejecute la función, se trata de una lista de objetos.

```
Ej:" heap([ (C, array(I, 1, [0|D])),
(B,object(heap/SortedListInt,[field(heap/SortedListInt.first:Lhe
ap/SLNode;, null)|E]))|F], G)"

Hay dos elementos en el heap, cada elemento está formado por una
tupla cuyo primer elemento es la referencia del objeto y la otra
es la información sobre el objeto.

"(C, array(I, 1, [0|D])) ":
'C' es la referencia del objeto, por la que lo vamos a localizar
mas tarde.
"array(I,1,[0|D]) " indica que el elemento es un array de
enteros con un elemento cuyo valor es "0".

"(B,object(heap/SortedListInt,[field(heap/SortedListInt.first:Lh
eap/SLNode;, null)|E])"
'B' es la referencia.
Object se compone de dos campos a su vez, el primero es la clase
del objeto, el segundo es información sobre sus campos, por lo
tanto:
"SortedListInt" es su clase.
"field(heap/SortedListInt.first:Lheap/SLNode;, null)" indica que
tiene un campo llamado "first" de tipo SLnode y cuyo valor es
null.
```

Salida: se compone de tres campos:

OutValue: valor de salida de la función si lo tiene.

```
Ej: "none" significa que la función no devuelve nada.
```

OutHeap: estado de la memoria después de la ejecución de la funcion.

```
Ej: heap([ (0, object(heap/SLNode, [field(heap/SLNode.data:I,
0), field(heap/SLNode.next:Lheap/SLNode;, null)])), (C, array(I,
1, [0|D])), (B, object(heap/SortedListInt,
[field(heap/SortedListInt.first:Lheap/SLNode;, ref(0))|E]))|F],
G)
Se observan dos diferencias con respecto al heap de entrada, el
campo first de el objeto 'B' ahora contiene ref(0) en vez de
null, y hay un nuevo objeto en el heap de tipo SLNode marcado
con la referencia '0' .
```

Exc Flag: Indica si la ejecución de la función causó una excepción o no, el tipo de la excepción se indica en el OutHeap.

```
Ej: "ok" significa que la función no produjo una excepción.
```

Num_Case: numero de caso, ya que para cada función se generan varios casos de prueba.

4.2 Algoritmo

El algoritmo recibe una lista de signaturas de métodos, todos los métodos pertenecen a la misma clase java, a partir de la primera signatura se escribe la cabecera del archivo, no es mas que los *imports* necesarios para poder ejecutar luego el JUNIT y la declaración de la clase que hace el test.

```
Ej:
package heap ;
import junit.framework.TestCase;
import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.assertEquals;
public class SortedListIntTest extends TestCase{
```

Una vez hecho esto empieza un proceso que se repite para cada signatura, se buscan los *test_case* generados para esa signatura y se separan entre los que tienen excepción y los que no, primero se tratan los que no generan ninguna excepción, así que se escribe la cabecera de la función de prueba, el nombre de la función lleva incluidos los parámetros de entrada para diferenciarla de otras funciones llamadas igual pero con parámetros diferentes.

```
Ej: public void test_insert___none___Array$int(){
```

Luego se repite el proceso que se detalla a continuación para cada *test_case* que generó Pet a partir de ese método.

Los casos con excepción deben estar cada uno en una función, así que para cada *test_case* se genera una nueva función, el código generado estará encerrado en un bloque try-catch que comprobará que la excepción generada sea la correcta, por lo demás el algoritmo es exactamente igual en los dos casos.

Cuando ya se han tratado todos los *test_case* para esa signatura, se coge la siguiente y se repite el proceso hasta que no quede ninguna.

4.3 - Asertar la información:

La primera parte del algoritmo se basa en coger toda la información que contiene el *heap* y prepararla para su posterior consulta.

Asertar la información significa meterla en la base del conocimiento, es información a la que podremos acceder fácilmente y en tiempo constante en cualquier punto del programa, se usa múltiples veces. A lo largo del algoritmo se utilizan las siguientes estructuras para asertar:

```
Heap_elem_in (referencia, valor, nombre en java): objeto del heap de
entrada.

Heap_array_in (referencia, valor, nombre en java, tipo):array del heap
de entrada.

Heap_elem_out(referencia, valor, nombre en java): objeto del heap de
salida.

Heap_array_out(referencia, valor, nombre en java, tipo):array del heap
de salida.

Heap_soul(referencia, valor, nombre en java, tipo, dirección):el valor
siempre es default ya que son objetos que a priori no existen, y en
este caso la dirección indica si es de entrada o de salida.

Input_elem(nombre en java): parámetro de entrada de la función.

Output_elem(nombre en java): parámetro de salida de la función.

This(referencia): indica que objeto es el this.

Action (función, nombre en java, clase, valor, numero de caso)
```

4.3.1 - Procesar el *heap* de entrada:

esta tarea se compone de tres pasos:

4.3.1.1 Quitar variables: cada elemento del *heap* sea array u objeto se compone de dos campos, el primero de ellos es la referencia, la cual es una variable de prolog, para que deje de serlo se le da un valor, cuando esto ocurre en todos los sitios donde apareciera la variable ahora aparece el nuevo valor, así el objeto ya es reconocible tanto en el *heap* de entrada como en el de salida y en los parámetros de entrada.

> Ej: En el heap de entrada había dos objetos el B y el C, a partir de ahora serán los objetos 27 y 28 respectivamente, también se notaran los cambios en el heap de salida y en los parámetros de entrada, así quedaría el heap:
>
> ```
> HeapIn: heap([ (28, array(I, 1, [0|D])), (27,
> object(heap/SortedListInt,[field(heap/SortedListInt.first:Lheap/
> SLNode;, null)|E]))|F], G)”
> ```

```
HeapOut: heap([(0,object(heap/SLNode,[field(heap/SLNode.data:I,
0), field(heap/SLNode.next:Lheap/SLNode;, null)])), (28,
array(I, 1, [0|D])), (27, object(heap/SortedListInt,
[field(heap/SortedListInt.first:Lheap/SLNode;, ref(0))|E]))|F],
G)

Parámetros de entrada:  ref(28) ref(27).
```

4.3.1.2 Quitar referencias: en esta fase se recorren todos los campos de los objetos y los valores de los arrays, cuando se encuentra uno que tenga como valor ref(X) se cambia por var_in_X que es el nombre que posteriormente tendrá en el código java para generar el JUNIT, si en el campo se encuentra una variable o una referencia a una variable significa que el objeto es necesario pero no esta en el *heap* así que se aserta un *heap_soul*, mas adelante se explicara como funciona, aparte también se cambia su nombre por su nombre en java.

```
Ej: En el ejemplo no hay ninguna referencia en los campos por lo
tanto no cambiaria nada.
```

4.3.1.3 Asertar elementos del *heap*: Se recorre toda la lista de elementos del *heap* y se aserta, si se trata de arrays se aserta heap_array_in(referencia, valor, nombre en java, tipo), si es un objeto heap_elem_in(referencia,valor,nombre en java).

```
Ej: Asi quedarían asertados:
heap_elem_in (27 , object(heap/SortedListInt,
[field(heap/SortedListInt.first:Lheap/SLNode;, null)]),
var_in_27)
heap_array_in(28,  array(I, 1, [0]) , var_in_28, int[])
```

Hasta ahora ya están asertados todos los elementos del *heap* de entrada así como los *heap soul* que también deben crearse.

4.3.2 - Procesar los argumentos de entrada

Antes de comenzar esta sección es conveniente explicar como funcionan las acciones, son aserciones de la forma action (función, nombre en java, clase, valor, numero de caso), cada una de estas acciones equivale a una instrucción java que se generará mas tarde, se explicará con mas detalle mas adelante.

Para cada argumento de entrada se asertará la información necesaria para que se cree y para poder localizarlo fácilmente mas tarde.

- Si es una variable: se aserta input_elem (var_in_X) y dependiendo del tipo se asertan las acciones necesarias para que se cree mas tarde:
    - Si es un array:
        - action (put_array, nombre en java, tipo, [], _)
    - Si es un objeto:
        - action (def_var, nombre en java, tipo, _, _)
        - action(new, nombre en java, tipo, _, _)
    - Si es un tipo primitivo:

        - action (def_var, nombre en java, tipo, _, _)
        - action(match, nombre en java, _, valor por defeto, _)

- Si es una referencia:

    - Si es this y esta en el *heap*: no se aserta nada.
    - Si es this y no esta en el *heap*:
        - action (def_var, nombre en java, clase, _, _)
        - action (new, nombre en java, clase, _, _)
    - Si no es this y esta en el *heap*, se aserta:
        - input_elem (nombre en java)
    - Si no es this y no esta en el *heap*, se aserta:
        - heap_soul (referencia, 'default', nombre en java, tipo, dirección)

- Si no es nada de lo anterior es que se trata de un valor, en este caso se asertan las acciones:

        - action (def_var, nombre en java, tipo, valor,_)
        - action (match, nombre en java, _, valor,_)

```
Ej: en el ejemplo los argumentos son: ref(27) ref(28) el primer
argumento es el objeto que llama a la función (this) y esta en
el heap por lo tanto no se aserta nada, el segundo argumento es
una referencia que esta en el heap así que se aserta:
input_elem(var_in_28) .
```

4.3.3 - Procesar el *heap* de salida

Es muy parecido al *heap* de entrada, quitar las variables ya no es necesario porque al darles valor en el *heap* de entrada también se actualizaron las de salida, así que esta vez solo es necesario cambiar las referencias por sus nombres en java y asertar los elementos del *heap*.

```
Ej: tras procesar el heap de salida esta es la información que
queda asertada :

heap_elem_out (0, object(heap/SLNode, [field(heap/SLNode.data:I,
0), field(heap/SLNode.next:Lheap/SLNode;, null)]),var_out_0)

heap_elem_out (27 , object(heap/SortedListInt,
[field(heap/SortedListInt.first:Lheap/SLNode;, var_out_0)]),
var_out_27)

heap_array_out(28,  array(I, 1, [0]) , var_in_28, int[])
```

4.3.4 - Procesar el argumento de salida

Tiene la misma filosofía que procesar el argumento de entrada, se crea la variable y se identifica como output_elem, aparte se crea otra variable llamada expected donde se asignará la salida de la función.

- Si es una variable: se aserta output_elem (var_in_X) y las acciones necesarias para que se cree:

- Si es un array:
  - action (put_array, nombre en java, tipo, [], _)
- Si no lo es:
  - heap_soul (referencia, 'default', nombre en java, tipo,dirección)

- Si es una referencia:

  - Si es this y esta en el heap: no se aserta nada
  - Si es this y no esta en el heap:
    - action (def_var, nombre en java, clase, _, _)
    - action (new, nombre en java, clase, _, _)
  - Si no es this y esta en el heap, se aserta:
    - output_elem (nombre en java)
  - Si no es this y no esta en el heap: no se aserta nada.

- Si no es nada de lo anterior es que se trata de un valor, en este caso se aserta:

  - output_elem(nombre en java)
  - action (def_var, nombre en java, tipo, valor,_)
  - action (match, nombre en java, _, valor,_)

  También se define una nueva variable llamada expected, es donde se guardara la salida de la función, para ello se aserta:

  - expected(nombre en java)
  - action(def_var, nombre en java, tipo, _, _)

  ```
  Ej: en el ejemplo la salida de la función es void por lo tanto
  no se procesa el argumento de salida y no se aserta nada nuevo.
  ```

4.3.5 - Asertar las acciones

Ahora que están asertados y tratados todos los elementos del *heap*, ya se puede comenzar a asertar las acciones que generarán los objetos en el código JUNIt, la estructura básica de todas las acciones es esta:

action (función, nombre en java, tipo, valor, numero de caso)

Como ya se comentó anteriormente, cada acción se corresponde con una línea de código en java, dependiendo del campo función, existen cinco opciones para el campo función:

- Def_var: Declarar la variable, su sintaxis esquemática sería esta:
  Tipo nombre_en_java;

- New: generar el objeto, siempre se utiliza la constructora por defecto:
  Nombre_en_java = new tipo();

- Def_var_match: declarar la variable y asignarle un valor:

Tipo nombre_en_java = valor;

- Put_array: declarar y asignar valores a un array:
  Tipo nombre_en_java = {v1,v2,v3,v4};

- Match: asignar un valor a una variable:
  Nombre_en_java = valor;

4.3.5.1 - Asertar acciones a objetos:

Se recorre la lista de objetos asertados con la forma heap_elem_in y heap_elem_out, y para cada uno de ellos se asertan las siguientes acciones:

action(def_var, nombre_en_java, tipo, _, _)
action(new, nombre_en_java, tipo, _, _)

Una vez hecho esto, se recorren sus campos internos y para cada uno de ellos se aserta:

Action(match, nombre_en_java, tipo del campo, valor, clase del campo)

En este caso el nombre en java ya incluye el nombre del campo, es decir tendría esta forma "nombre_en_java.atributo", con esta secuencia de acciones se define la variable que contendrá el objeto, se crea el objeto y se le asigna el valor correspondiente a cada uno de sus campos.

```
Ej: acciones generadas para cada elemento asertado:

Elemento '27' de entrada:
heap_elem_in(27, object(heap/SortedListInt,
[field(heap/SortedListInt.first:Lheap/SLNode;, null)]),
var_in_27)
action(def_var, var_in_27, SortedListInt, _G23439, _G23440)
action(new, var_in_27, SortedListInt, _G23439, _G23440)
action(match, var_in_27.first, Lheap/SLNode;, null,
heap/SortedListInt)

Elemento '0' de salida:
heap_elem_out(0, object(heap/SLNode, [field(heap/SLNode.data:I,
0), field(heap/SLNode.next:Lheap/SLNode;, null)]), var_out_0)
action(def_var, var_out_0, SLNode, _G23439, _G23440)
action(new, var_out_0, SLNode, _G23439, _G23440)
action(match, var_out_0.data, I, 0, heap/SLNode)
action(match, var_out_0.next, Lheap/SLNode;, null, heap/SLNode)

Elemento '27' de salida:
heap_elem_out(27, object(heap/SortedListInt,
[field(heap/SortedListInt.first:Lheap/SLNode;, var_out_0)]),
var_out_27)
action(def_var, var_out_27, SortedListInt, _G23439, _G23440)
action(new, var_out_27, SortedListInt, _G23439, _G23440)
action(match, var_out_27.first, Lheap/SLNode;, var_out_0,
heap/SortedListInt)
```

4.3.5.2 - **Asertar acciones a arrays:**

Se recorre la lista de arrays asertados, estos tiene la forma heap_array_in y heap_array_out, y para cada uno de ellos se aserta la siguiente acción:

action(put_array, nombre_en_java, tipo, elementos, _)

```
Ej: acciones generadas para cada array:
Elemento '28' de entrada:
heap_array_in(28, array(I, 1, [0]), var_in_28, int[])
action(put_array, var_in_28, int[], [0], _G23443)

Elemento '28' de salida:
heap_array_out(28, array(I, 1, [0]), var_out_28, int[])
action(put_array, var_out_28, int[], [0], _G23443)
```

4.3.5.3 - **Asertar acciones a elementos heap_soul:**

Los *heap soul* son los elementos mas complicados ya que no se sabe si son objetos o tipos primitivos y porque no tienen valor.

El primer paso es cacular su valor por defecto, si este es null significa que es un objeto, en este caso se asertarian las acciones:

action(def_var, nombre_en_java, tipo, _, _)
action(new, nombre_en_java, tipo, _, _)

Si por el contrario tiene valor por defecto, que en la mayoría de los casos será '0' se aserta la siguiente acción:

action(def_var_match, nombre_en_java, tipo, valor, _)

Esta acción declara la variable y le da valor en la misma instrucción, esto se hace así porque el orden en el cual se generaran luego las instrucciones es importante, mas tarde se vera el porqué.

```
Ej: en el ejemplo no hay objetos heap_soul por lo tanto no se
aserta ninguna acción.
```

4.4 - **Escribir el código java:**

Es el último paso del algoritmo, se encarga de generar el código necesario para probar el programa, esto incluye crear los objetos y variables, hacer la llamada al método, y comparar los resultados.

4.4.1 - **crear los objetos y variables:**

Una vez están todas las acciones necesarias asertadas solo falta recorrerlas en el orden correcto:

1. def_var, las declaraciones de variables.
2. new, se crean los objetos.

3. def_var_match, esta acción incluye darle valor a la variable porque si no tiene valor y la variable forma parte de un array al crear el array produce un error.
4. put_array, crea los arrays.
5. Match, asigna variables y valores a variables y a los campos de los objetos, cuando los campos son privados se usa *reflection*, es una técnica mediante la cual se consigue acceso a esos campos, se explicará con detalle en el apartado de casos especiales.

Según se escriben las instrucciones se les añade al nombre de la variable el numero de caso.

```
Ej: instrucciones generadas:
action(def_var, var_in_27, SortedListInt, _G23439, _G23440)
action(def_var, var_out_0, SLNode, _G23439, _G23440)
action(def_var, var_out_27, SortedListInt, _G23439, _G23440)
SortedListInt var_in_27_b;
SLNode var_out_0_b;
SortedListInt var_out_27_b;

action(new, var_in_27, SortedListInt, _G23439, _G23440)
action(new, var_out_0, SLNode, _G23439, _G23440)
action(new, var_out_27, SortedListInt, _G23439, _G23440)
var_in_27_b = new SortedListInt();
var_out_0_b = new SLNode();
var_out_27_b = new SortedListInt();

action(put_array, var_in_28, int[], [0], _G23443)
action(put_array, var_out_28, int[], [0], _G23443)
int[] var_in_28_b= {0};
int[] var_out_28_b= {0};

action(match, var_in_27.first, Lheap/SLNode;, null,
heap/SortedListInt)
action(match, var_out_0.data, I, 0, heap/SLNode)
action(match, var_out_0.next, Lheap/SLNode;, null, heap/SLNode)
action(match, var_out_27.first, Lheap/SLNode;, var_out_0,
heap/SortedListInt)
var_in_27_b.first = null;
var_out_0_b.data = 0;
var_out_0_b.next = null;
var_out_27_b.first = var_out_0_b;
```

4.4.2 - Hacer la llamada al método:

Ahora que ya se han creado todos los objetos antes y después de la ejecución de la función, ya se puede hacer la llamada al método, el código generado depende de si el método es estático y de si devuelve algún valor.

Información asertada con anterioridad necesaria para la llamada:

Expected: nombre de la variable donde se guarda el resultado, esta asertado de la forma expected(nombre).

Clase: La clase de la función, necesaria para llamadas desde métodos estáticos, se encuentra asertada de la forma class(nombre).

This: nombre del objeto que llama a la función, esta asertado de la forma this (indentificador).

Función: nombre de la función, esta asertado de la forma shortMethodName(nombre), lo asertó pet antes de la llamada al Junit generator.

Parámetros de entrada: nombres de las variables de entrada separadas por comas, esta asertado de la forma input_elem(nombre).

- Tiene elemento de salida:
  - Estatico:

    expected = clase.función(parámetros de entrada);

  - No estatico

    expected = this.función(parámetros de entrada);

- No tiene elemento de salida:
  - Estático

    clase.funcion(parámetros de entrada);

  - No estatico

    - Es conructora:

      This = new funcion(parámetros de entrada);

    - No es conructora:

      this.funcion(parámetros de entrada);

```
Ej: la función que se prueba no tiene elemento de salida,
el método no es estático y no se trata de una constructora
por lo tanto esta es la llamada al método:
var_in_27_b.insert(var_in_28_b);
```

4.4.3 - Comparar los resultados

Esta tarea se basa en hacer llamadas a *assertEquals*, es una función de JUNIT, comprueba si dos objetos o dos arrays son iguales y muestra el resultado por pantalla. Si el *test_case* tiene excepción no es necesario comparar los resultados simplemente se compara si la excepción producida es la que debería.

El JUNIT generator tiene dos opciones de evaluación:

Parcial: se hacen llamadas a *assertEquals* con el valor que devuelve la función y con el objeto this siempre que existan.

Completa: se hacen llamadas a *assertEquals* con el valor devuelto y con todos los objetos que estén en el *heap* de entrada.

```
Ej: si se elije evaluación parcial la llamada seria:
assertEquals("the object expected",var_in_27_b,var_out_27_b);
se comprueba solo el objeto this ya que la funcion no devuelve
nada.
La evaluación completa seria:
assertEquals("the object expected",var_in_27_b,var_out_27_b);
assertArrayEquals("the Array
expected",var_in_28_b,var_out_28_b);
son los dos elementos que se encontraban en el heap de entrada.
Finalmente el código java generado para este caso de prueba es:
SortedListInt var_in_27_b;
SLNode var_out_0_b;
SortedListInt var_out_27_b;
var_in_27_b = new SortedListInt();
var_out_0_b = new SLNode();
var_out_27_b = new SortedListInt();
int[] var_in_28_b= {0};
int[] var_out_28_b= {0};
var_in_27_b.first = null;
var_out_0_b.data = 0;
var_out_0_b.next = null;
var_out_27_b.first = var_out_0_b;
var_in_27_b.insert(var_in_28_b);
assertEquals("the object expected",var_in_27_b,var_out_27_b);
assertArrayEquals("the Array
expected",var_in_28_b,var_out_28_b);
```

4.5 - Casos especiales

4.5.1 – Arrays

Cuando en el *heap* hay un array de objetos su tipo no aparece indicado sin embargo es necesario que se cree para el correcto funcionamiento del programa. Al recorrer la lista de parámetros de entrada si se encuentra un array se busca el objeto asertado al que referencia y se le añade su tipo, ya que en la signatura del método están indicados todos los tipos, si el tipo es un multi-array se le añade el tipo a su vez a todos los elementos de ese array y así sucesivamente. Si el array no esta referenciado en los argumentos de entrada estará referenciado por algún otro objeto que lo contenga, y en el campo del objeto si vendrá indicado el tipo, una vez se tiene se repite el mismo proceso comentado anteriormente.

```
Ej: en el test case de esta función no aparece indicado el tipo de los
arrays:


public static int vvvv2(int[][][][] aa, int[][][][] bb){
       if (aa[1][2][3][4] == 9) return -1;
       else return bb[4][3][2][1];
   }
```

Method	In		Out		
	Input args	Input heap	Ret	Exc Flag	Ouput heap
spooky/spookytypes.vvvv2([[[[I[[[[I)I	ref(B),C	heap([(B, array(D, 2, [E, ref(F)|G])), (F, array(H, 3, [I, J, ref(K)|L])), (K, array(M, 4, [N, O, P, ref(Q)|R])), (Q, array(I, 5, [-10, -10, -10, -10, 9|S]))|T], U)	none	ok	heap([(B, array(V, 2, [E, ref(F)|G])), (F, array(W, 3, [I, J, ref(K)|L])), (K, array(X, 4, [N, O, P, ref(Q)|R])), (Q, array(I, 5, [-10, -10, -10, -10, 9|S]))|T], U)

Esta función recibe como argumentos dos arrays de enteros de cuatro niveles, se puede observar como el tipo no viene indicado en los arrays excepto en el de enteros. También se observa que el segundo argumento de entrada no esta en el heap, deberá ser creado con su valor por defecto.

Código generado:

```
 int var_out_65_b;
int expected_66_b;
int[][][][] var_in_49_b;
var_in_49_b = new int[][][][0];
int[] var_in_52_b= {-10,-10,-10,-10,9};
int[] var_in_55_b= {};
int[] var_in_54_b= {};
int[] var_in_53_b= {};
int[][] var_in_51_b=
{var_in_53_b,var_in_54_b,var_in_55_b,var_in_52_b};
int[][] var_in_57_b= {};
int[][] var_in_56_b= {};
int[][][] var_in_50_b= {var_in_56_b,var_in_57_b,var_in_51_b};
int[][][] var_in_58_b= {};
int[][][][] var_in_48_b= {var_in_58_b,var_in_50_b};
int[] var_out_52_b= {-10,-10,-10,-10,9};
int[] var_out_61_b= {};
int[] var_out_60_b= {};
int[] var_out_59_b= {};
int[][] var_out_51_b=
{var_out_59_b,var_out_60_b,var_out_61_b,var_out_52_b};
int[][] var_out_63_b= {};
int[][] var_out_62_b= {};
int[][][] var_out_50_b= {var_out_62_b,var_out_63_b,var_out_51_b};
int[][][] var_out_64_b= {};
int[][][][] var_out_48_b= {var_out_64_b,var_out_50_b};
var_out_65_b = -1;
expected_66_b = spookytypes.vvvv2(var_in_48_b,var_in_49_b);
assertArrayEquals("the Array expected",var_in_52_b,var_out_52_b);
assertArrayEquals("the Array expected",var_in_51_b,var_out_51_b);
assertArrayEquals("the Array expected",var_in_50_b,var_out_50_b);
assertArrayEquals("the Array expected",var_in_48_b,var_out_48_b);
assertEquals("the result Expected",expected_66_b,var_out_65_b);
```

Tambien es conveniente resaltar la importancia del orden al crear los arrays, de otro modo generarían errores.

4.5.2 - *Reflections*

Las acciones de tipo match son las que se encargan de dar valor a las variables o atributos de los objetos y son procesadas en el método write_match. Creando una instrucción java con esta forma objeto.atributo=valor.

Como se dijo en el apartado 1.6.2 esta instrucción no es correcta si el atributo es privado, en este caso ha de utilizarse la técnica de *Reflection*. Que consiste en acceder al campo utilizando una serie de predicados incluidos en la clase de Java Reflec.

Para aplicar esta técnica en el método prolog write_match, antes de procesar un atributo se analiza si este es privado, mediante el predicado is_public/2 y si lo es se accede al campo del objeto escribiendo este código, cuyo significado esta explicado en el apartado 1.7.4.

```
final Field fields_objeto[] = SearchTreeInt.class.getDeclaredFields();

for (int i = 0 ;i < fields_objeto.length; i++){
    if ("atributo".equals(fields_objeto[i].getName())){
          Field f =fields_objeto[i];
      f.setAccessible(true);
      f.set(objeto,valor);
      break;
   }
}
```

4.5.3 – Excepciones

En los *test_case* que producen una excepción el bloque de código se introduce en un bloque try-catch y se comprueba que la excepción producida sea la correcta, esto se hace en una función aparte para cada excepción.

```
Ej:

try{
     SortedListInt var_in_48_a;
     int[] var_in_49_a;
     SortedListInt var_out_48_a;
     var_in_48_a = new SortedListInt();
     var_out_48_a = new SortedListInt();
     var_in_49_a = null;
     var_in_48_a.insert(var_in_49_a);
}

catch(Exception ex){
        assertEquals("exception",
"java.lang.NullPointerException", ex.getClass().getName());
        return;
}

fail("Did`nt find expected exception");
```

5- Resultados experimentales

Para la realización de las pruebas de tiempo hemos utilizado la web de Pet [3]. Esta página permite ejecutar la aplicación controlando todas las opciones posibles, como el criterio de cobertura, el rango de valores a utilizar, si se desea crear el Junit y demás opciones relacionadas con el proceso de generación de los casos de prueba. Además incluye varios temporizadores destinados a medir el tiempo de ejecución.

La web también cuenta con una amplia batería de ejemplos, dividida en tres paquetes: Ints, Heap e Inheritance. Estos contienen una gran variedad de métodos y nos permite tener una batería de pruebas muy variada.

A continuación, para cada paquete se detalla un resumen de ejecución de todas las clases disponibles. Analizando las características del archivo de entrada, los parámetros de ejecución de Pet y los detalles de la salida generada. Estas pruebas han sido ejecutadas en una máquina Intel Celeron a 2.00 Ghz con 2 GB de memoria ram bajo Ubuntu 2.6.31-14-generic.

Para las llamadas a pet se ha utilizado un criterio de cobertura de Block-2, que garantiza que cada bucle del programa es recorrido dos veces. Un rango de valores que va de -10 a 10. Generando el junit de manera completa, es decir analizando todos los objetos del *heap* de salida y utilizando reflection si es necesario. El resto de opciones de Pet han sido desactivadas.

Los datos que hemos estudiado son: Números de métodos Java que ejecuta Pet, el número de líneas del archivo java a analizar, el tiempo empleado en traducir el *byte code* en un programa CLP equivalente, el número de casos de prueba generados por Pet y el tiempo en realizar esta tarea. A la hora de generar el Junit hemos recogido estos valores: Número de líneas creadas, número de métodos resultantes, número de *asserts* que contiene y el tiempo empleado en generar este archivo.

El primer paquete estudiado es el Ints, compuesto por 6 clases relacionadas con números enteros y vectores, no se manejan otros tipos de datos. En general son métodos simples y cortos, como el cálculo del mínimo común divisor, sucesión de Fibonacci, números primos, etc.

El segundo paquete analizado es el llamado Heap es el más completo porque incluye una gran cantidad de situaciones distintas en la que se ven involucrados objetos de todo tipo. Contiene 10 clases, casi todas son estructuras de datos simples creadas con el fin de comprobar las cualidades de Pet. Listas genéricas, listas de enteros, pilas, arboles de búsqueda son algunos ejemplos de estas clases.

Por último, el paquete Related también contiene estructuras de datos, pero estas han sido extraídas de librerías GPL como la de apache o Sun TreeMap, así se pone de manifiesto que Pet es compatible con los códigos de la vida real. Ejemplos de estas clases son las dobles listas enlazadas y los árboles rojinegros.

Los resultados experimentales de la ejecución de la batería de pruebas están disponibles en la tabla de la siguiente página, de la podemos sacar varias conclusiones.

Como es lógico, los tiempos empleados por Pet en crear el programa CLP y en generar los casos de prueba, están ligados al tamaño del archivo original y al número de *test cases* creados.

Si nos fijamos en los tiempos de generación de los casos de prueba en de creación del Junit, vemos que el segundo es casi siempre mayor. Analizando paquete por paquete vemos que en el Ints se tarda un 38 % más en crear el Junit, en el Heap esta relación es del 16% y en el Related del 99%, puede observarse que a medida que las estructuras de datos de los ejemplos se vuelven más complejas la velocidad del Junit tiende a aumentar con respecto a Pet.

En el archivo de salida, también se observa que el tiempo empleado en su generación es proporcional al número de líneas creadas y la cantidad de *asserts* incluidos.

Todas estas pruebas han sido realizadas con la opción de "análisis completo" activada (apartado 1.6.1), esto implica la creación de una gran cantidad de objetos, la inicialización de sus atributos y un aumento en el número de *asserts*, generando archivos muy grandes y como hemos visto con un coste de tiempo elevado. Si se desea simplificar este proceso es recomendable desmarcar esta opción.

Tabla Resúmen de la batería de pruebas

Paquete Ints	Clase ejecutada	Arithmetic.java	Combinatory.java	Factorial.java	Fibonacci.java	intExponential.java	PrimeNumbers.java
Archivo de Entrada	Número de métodos Java	9	5	9	5	4	4
	Lineas del archivo Java	78	48	67	43	47	36
Salida de Pet	Tiempo CLP (ms)	70	50	91	60	60	80
	Tiempo casos de prueba (ms)	39	10	19	10	10	1150
	Número de casos de prueba generados	21	19	23	10	16	112
Junit Generado	Número de lienas generadas	259	219	365	111	231	1039
	Números de Métodos junit	11	5	16	5	10	6
	Tiempo de creación del Junit (ms)	31	19	31	10	29	101
	Número de asserts	23	21	33	12	18	114
	Tiempo de ejecucion del Junit (ms)	0,002	0,002	0,004	0,028	0,002	0,005

Paquete Heap	Clase ejecutada	Aliasing.java	ArraySearch.java	Josephus.java	Rational.java	SearchTreeInt.jav a	SortedListInt.java	Sorting.java	StackInt.java
Archivo de Entrada	Número de métodos Java	5	6	2	13	5	5	6	7
	Lineas del archivo Java	331	145	162	977	1665	741	530	171
Salida de Pet	Tiempo CLP (ms)	60	70	71	250	80	198	240	50
	Tiempo casos de prueba (ms)	20	19	19	80	40	91	379	21
	Número de casos de prueba generados	19	10	4	50	16	26	33	7
Junit Generado	Número de lienas generadas	331	145	162	977	1665	741	530	171
	Números de Métodos junit	13	6	2	32	9	9	22	7
	Tiempo de creación del Junit (ms)	40	20	10	130	80	89	71	20
	Número de asserts	33	19	6	87	36	77	50	19
	Tiempo de ejecucion del Junit (ms)	0,003	0,097	0,028	0,012	0,007	0,009	0,011	0,002

Paquete Related	Clase ejecutada	DoublyLinkedList.Java	EntryRBT.java	NodeDLL.java	NodeJosephus.java	Triangle.java
Archivo de Entrada	Número de métodos Java	13	5	7	5	1
	Lineas del archivo Java	204	97	57	113	47
Salida de Pet	Tiempo CLP (ms)	231	39	80	140	60
	Tiempo casos de prueba (ms)	539	11	50	19	39
	Número de casos de prueba generados	116	5	19	9	14
Junit Generado	Número de lienas generadas	3837	93	450	159	201
	Números de Métodos junit	18	5	7	5	1
	Tiempo de creación del Junit (ms)	561	10	70	21	21
	Número de asserts	396	11	42	12	16
	Tiempo de ejecucion del Junit (ms)	0,007	0,002	0,007	0,002	0,001

6 – Conclusiones

A lo largo de este proyecto de fin de carrera hemos aprendido valiosas lecciones, en un principio comenzamos a programar directamente para resolver problemas sencillos, como el proyecto se desarrollaba en prolog tuvimos que aprender a manejarlo, tuvimos que comprender la estructura del programa Pet para poder acoplar nuestro generador de JUNIT, tras dos meses de trabajo nos dimos cuenta que la estructura que habíamos seguido hasta entonces no servía para todos los casos, y empezamos de cero de nuevo, pero ya con los conocimientos adecuados. Comprendimos que era mejor planificar el algoritmo antes de ponernos a programar, hicimos un nuevo algoritmo estudiando todas las posibilidades, este algoritmo era mucho más sencillo que el anterior y más versátil, sólo tuvimos que remodelarlo ligeramente para los casos especiales.

Aparte de las dificultades que superamos comprendimos el trabajo que lleva hacer *testing* y vimos la utilidad del programa que desarrollábamos, aunque lo mas difícil de este *software* ya estaba hecho nos gustó participar en esta iniciativa, y esperamos que algún dia se use de manera profesional.

7- Referencias

[1]. *M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. Journal of Information and Software Technology. Vol. 51, pages 1409-1427, Elsevier, October 2009.*
[2]. *E. Albert, J. Gallagher, M. Gómez-Zamalloa, and G. Puebla. Type-based Homeomorphic Embedding for Online Termination. Information Processing Letters, Vol. 109, pages 879-886, Elsevier, July 2009. .*
[3]. *https://costa.ls.fi.upm.es/pet/pet.php.*
[4]. *Y. Futamura. Partial Evaluation of Computation Process - AnApproach to a Compiler-Compiler. Systems, Computers, Controls,2(5):45–50, 1971.*
[5]. *E. Albert, M. Gómez-Zamalloa, and G. Puebla. Pet: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM), ACM Press, 25-28, January 2010. .*

8- Bibliografía

http://en.wikipedia.org/
http://es.wikipedia.org/
http://pruebassw.blogspot.com/2007/01/orculos.html
http://www.cc.uah.es/drg/adis2009/articles/adis-09-Torres-Oraculos.pdf
http://www.junit.org/
http://login.osirislms.com/offline/junit/index.htm
http://costa.ls.fi.upm.es/pet/pet.php
http://en.wikipedia.org/wiki/Java_bytecode
http://java.decompiler.free.fr/?q=jdeclipse
http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/
http://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas
http://www.dc.fi.udc.es/ai/tp/practica/junit/ppasos.htm
http://www.dcc.uchile.cl/~peortega/guiaprolog/leccion3.html
http://www.sics.se/sicstus/docs/4.0.3/html/sicstus/mpg_002dref_002dformat.html
http://www.programacion.com/java/articulo/jap_junit/#2_paso1
http://www.humbertocervantes.net/cursos/ingsoft/practica5/practica5.html
http://www.swi-prolog.org/pldoc/index.html
http://softwaredevscott.spaces.live.com/blog/cns!1A9E939F7373F3B7!524.entry
http://www.cse.psu.edu/~catuscia/teaching/prolog/Manual/sec-3.19.html
http://www.dosideas.com/wiki/Mock_Object
http://es.wikipedia.org/wiki/Objeto_simulado
http://www.mockobjects.com/
http://msdn.microsoft.com/es-es/library/bb932285.aspx#EHAA
http://www.easymock.org/EasyMock2_5_2_Documentation.html
http://cea-roberto-roldan.blogspot.com/2008/01/easymock.html
http://www.miltonpividori.com.ar/2008/02/06/mockobjects-con-rhinomocks/

9 - Apendice: Código del proyecto

Archivo junit_generator.pl

```
/** <module>
This module implements a series of predicates to generate Java code,

@author Fernando Rodriguez, Jaime Saez
@license GPL
*/
:-module(junit_generator,[junit_generator/1,
                  action/5,
                  this/1,
                  new_this/1,
                  output_elem/1,
                  input_elem/1,
                  heap_elem_in/3,
                  heap_array_in/4,
                  heap_elem_out/3,
                  heap_array_out/4,
                  heap_soul/5,
                  class/1,
                  inParams/1,
                  outParam/1,
                  getJMSig/1,
                  expected/1]).

:-use_module(pet(pet_driver),[pet/2]).
:-use_module(junit_utils,[is_static/0, get_pack_class/3,contains/2,
contains_str/2]).
:-use_module(make_asserts,[make_asserts/1]).
:-use_module(pet(test_cases_db),[test_case/6]).
:-use_module(pet(syntax_conversion),[jmsig_to_msig/2]).
:-
use_module(pet(pet_utils),[init_list/3,atom_concat/2,no_path_filename/2
, append_/3]).
:- use_module(pet(parameters_db),[paramdb_get/2]).
:- use_module(pet(out_filenames),[out_filename/3]).
:- use_module(pet(parameters_db),[paramdb_get/2]).
:-use_module(pet(jvml),[field_isPrivate/1,classname_fields/2]).

:-dynamic getJMSig/1.%Method signature
:-dynamic shortMethodName/1.%Java method name
:-dynamic action/5.
:-dynamic this/1.  % name of input this object
:-dynamic new_this/1. %name of output this object
:-dynamic output_elem/1.%output object
:-dynamic input_elem/1. %input objects
:-dynamic heap_elem_in/3.
:-dynamic heap_array_in/4.
:-dynamic heap_elem_out/3.
:-dynamic heap_array_out/4.
:-dynamic heap_soul/5.
:-dynamic heap_soul_array/5.
:-dynamic class/1. %class name
```

```
:-dynamic package/1.%package name
:-dynamic inParams/1. %input types list
:-dynamic outParam/1. % output type
:-dynamic expected/1. % return variable name
:-dynamic exception/1. %the case is an exception (true o false)
:-dynamic num_case/1. % num of current case
:-dynamic is_constructor/1. %the method is a constructor (true o flase)
:-dynamic use_reflection/1. %use reflection option
:-dynamic complete_test/1.%assert all heap option


%*****************************************
%Main Method
%*****************************************
%all signatures must be from the same package
junit_generator([]).
junit_generator([JMSig|JMSigList]):-
     %**** write file instructions
       paramdb_get(out_format,html),
       out_filename(junit,JMSig,FN),
       open(FN,write,S),
       set_output(S),
        %**************************
     jmsig_to_class(JMSig,Class),
     asserta(class(Class)),
     (paramdb_get(junit_test,complete) -
>asserta(complete_test(true));true),
     (paramdb_get(junit_test_reflection,yes) -
>asserta(use_reflection(true));true),
     jmsig_to_package(JMSig,Package),
     asserta(package(Package)),

     write_header,
     write_tests([JMSig|JMSigList]),

     write('}'),
        told,
     retract(class(_)),
     retract(package(_)).


write_header:-
     class(Class),
     package(Package),
     format('//This code  has been generated automatically ~n',[]),
     format('package ~w ;~n',[Package]),
     format('import junit.framework.TestCase;~n',[]),
     format('import static org.junit.Assert.assertArrayEquals;~n',[]),
     format('import static org.junit.Assert.assertEquals;~n',[]),
     (use_reflection(true)->format('import
java.lang.reflect.Field;~n',[]);true),
        format('public class ~wTest extends TestCase{ ~n',[Class]).

write_tests([]).
write_tests([JMSig|JMSigList]):-
     asserta(getJMSig(JMSig)),
```

```
      jmsig_to_ShortMethodName(JMSig,Method),
      class(Class),
      (Method=='<init>' ->
             Method2= Class,
             asserta(is_constructor(yes))
             ;
             Method2=Method,
             asserta(is_constructor(no))
             ),
      asserta(shortMethodName(Method2)),
      jmsig_to_msig(JMSig,methodSignature(_, _, Params,_)),
      asserta(inParams(Params)),
      jmsig_to_RetType(JMSig,Ret),
      asserta(outParam(Ret)),
      list_test_cases(JMSig,ListCases),
      list_test_exceptions(JMSig,ListExceptions),
      assert(exception(no)),
      write_test(JMSig,ListCases),
      retract(exception(_)),
      !,
      assert(exception(yes)),
      write_tests_exceptions(JMSig,ListExceptions),
      retract(exception(_)),
      retract(getJMSig(_)),
      retract(inParams(_)),
      retract(shortMethodName(_)),
      retract(is_constructor(_)),
      write_tests(JMSigList).

write_test(_,[]).
write_test(JMSig,ListCases):-
      jmsig_to_ShortMethodName(JMSig,M),
      class(Class),
      (M=='<init>' -> Method= Class; Method=M),
      jmsig_to_RetType(JMSig,Ret),
      jmsig_to_Params(JMSig,JMSParams),
      Tab1="  ",
      get_test_method_name(JMSParams,Ret,Method,Name),
      replace_chars(Name,'/','_',Name2),
      (use_reflection(true)->format('~s public void ~w()throws
Exception{~n',[Tab1,Name2])
                            ;
                            format('~s public void
~w(){~n',[Tab1,Name2])),
      write_cases(ListCases),
      format('~s }~n',[Tab1]).

write_tests_exceptions(_,[]).
write_tests_exceptions(JMSig,[Case|ListCases]):-
      write_test_exception(JMSig,Case),
      write_tests_exceptions(JMSig,ListCases).

write_test_exception(JMSig,Case):-
    jmsig_to_ShortMethodName(JMSig,Method),
    jmsig_to_RetType(JMSig,Ret),
    jmsig_to_Params(JMSig,JMSParams),
```

```
    Tab1="  ",
    Tab2="   ",
    Tab3="       ",
    get_test_method_name(JMSParams,Ret,Method,Name),
    replace_chars(Name,'/','_',Name2),
    Case=exc(InValueList,InHeap,_, heap([(_, object(ExType,
_))|_],_),Num_case),

Modified_Case=test_case(InValueList,InHeap,none,heap([],_),Num_case),
    asserta(num_case(Num_case)),
    format('~n~s //--------Case ~w With Exception-------
~n',[Tab1,Num_case]),
    format('~s public void
~w___exception_~w(){~n',[Tab1,Name2,Num_case]),
    format('~s try{~n',[Tab2]),
    make_asserts(Modified_Case),
    write_code,
    delete_asserts,
    retract(num_case(_)),
    format('~n~s }~n',[Tab2]),
    format('~s catch(Exception ex){~n',[Tab2]),
    remove_slash(ExType, ExType2),
    format('~s assertEquals("exception", "~s",
ex.getClass().getName());~n',[Tab3,ExType2]),
        format('~s return;~n',[Tab3]),
    format('~s }~n',[Tab2]),
    format('~s fail("Did`nt find expected exception");~n',[Tab2]),
    format('~s }~n',[Tab1]).


write_cases([]):-
     true.
write_cases([Case|ListCases]):-
     Case=test_case(_,_,_,_,Num_case),
     asserta(num_case(Num_case)),
     make_asserts(Case),
     format('\n     //--------Case ~w Without Exception--------
\n',[Num_case]),
     assert_value,
     write_code,
     retract(num_case(_)),
     delete_asserts,
     write_cases(ListCases).


%*****************************************
%these predicates write all instructions of java method
%*****************************************
write_code:-
     Tab="    ",
     write_def_var(Tab),
     write_new(Tab),
     write_def_var_match(Tab),
     write_put_array(Tab),
     write_match(Tab),
        write_method_call(Tab),
```

```
      write_assert_call(Tab).


write_method_call(Tab):-
      findall(X,input_elem(X),Xs),
      add_num_case(Xs,AA),
      list_to_str(AA,In_list),
      shortMethodName(Name),
      (output_elem(_)->%the method return something
         (is_static->
                % is static
           expected(Expected),
           add_num_case([Expected],[Expected2]),
           class(Class),
              format('~s ~w = ~w.~w(~w);\n',[Tab,Expected2,Class,Name,
In_list]),
           (complete_test(true)->write_complete_test(Tab);true)
             ;
                %is not static
           this(This),
           atom_chars(This,This2),
              append_(['var_in_'],This2,This_name_aux),
              atom_concat(This_name_aux,This_name_in),
              append_(['var_out_'],This2,This_name_aux2),
              atom_concat(This_name_aux2,This_name_out),
           expected(Expected),
           add_num_case([Expected ,This_name_in,This_name_out
],[Expected2,This_name_in2,This_name_out2]),
           format('~s ~w=
~w.~w(~w);\n',[Tab,Expected2,This_name_in2,Name,In_list]),
           (complete_test(true)->
              write_complete_test(Tab)
              ;
              format('~s assertEquals("the result
expected",~w,~w);\n',[Tab,This_name_in2,This_name_out2])))
      ;%the method return anything
         (\+is_static->
              this(This),
              atom_chars(This,This2),
              append_(['var_in_'],This2,This_name_aux),
              atom_concat(This_name_aux,This_name_in),
              append_(['var_out_'],This2,This_name_aux2),
              atom_concat(This_name_aux2,This_name_out),

add_num_case([This_name_in,This_name_out],[This_name_in2,This_name_out2
]),
                 (is_constructor(no)->
                     format('~s
~w.~w(~w);\n',[Tab,This_name_in2,Name,In_list])
                   ;
                   format('~s ~w= new
~w(~w);\n',[Tab,This_name_in2,Name,In_list]),
                    format('~s assertEquals("the result
expected",~w,~w);\n',[Tab,This_name_in2,This_name_out2])),
              (E==yes->true
               ;
```

```
               (complete_test(true)->
                   write_complete_test(Tab)
                   ;
                   format('~s assertEquals("the result
expected",~w,~w);\n',[Tab,This_name_in2,This_name_out2])))
          %is not static
          ;
              exception(E),
              class(Class),
              (E==yes->
                   format('~s ~w.~w(~w);\n',[Tab,Class,Name, In_list])
                ;
                format('~s ~w.~w(~w);\n',[Tab,Class,Name, In_list]),
               (complete_test(true)->
                    write_complete_test(Tab)
                    ;
                    write('//this method is static and return void,
nothing to assert.\n'))))).


write_assert_call(Tab):-
     (output_elem(X)->
            expected(Expected),
            add_num_case([Expected],[Expected2]),
            add_num_case([X],[X2]),
           (heap_array_out(_,_,X,_)->
            format('~s assertArrayEquals("the result
Expected",~w,~w);\n',[Tab,Expected2,X2])
           ;
            format('~s assertEquals("the result
Expected",~w,~w);\n',[Tab,Expected2,X2]))
     ;

      true).


write_complete_test(Tab):-
     compare_objects(Tab),
     compare_arrays(Tab).

compare_arrays(Tab):-
     heap_array_in(X,_,Name_in,_),
     heap_array_out(X,_,Name_out,_),
     add_num_case([Name_in,Name_out],[Name_in2,Name_out2]),
     format('~s assertArrayEquals("the Array
expected",~w,~w);\n',[Tab,Name_in2,Name_out2]),
     fail.
compare_arrays(_).
compare_objects(Tab):-
     heap_elem_in(X,_,Name_in),
     heap_elem_out(X,_,Name_out),
     add_num_case([Name_in,Name_out],[Name_in2,Name_out2]),
     format('~s assertEquals("the object
expected",~w,~w);\n',[Tab,Name_in2,Name_out2]),
     fail.
compare_objects(_).
```

```
write_def_var(Tab):-
      action(def_var,Name,Type,_,_),
         add_num_case([Name],[Name2]),
      format('~s ~w ~w;\n',[Tab,Type,Name2]),
      fail.
write_def_var(_).

write_new(Tab):-
      action(new,Name,Type,_,_),
      add_num_case([Name],[Name2]),
      (contains_str(Type,'[]')->
           %if Type= "class_name[]", convert to  new class_name[0]"
           put_array_num(Type, Type2),
           format('~s ~w = new ~w;\n',[Tab,Name2,Type2])
           ;
      format('~s ~w = new ~w();\n',[Tab,Name2,Type])),
      fail.
write_new(_).



write_put_array(Tab):-
    action(put_array,Name,Type,Value,_),
    add_num_case([Name],[Name2]),
    add_num_case(Value,Value2),
    format('~s ~w ~w=
{~@};\n',[Tab,Type,Name2,write_put_array_aux(Value2)]),
    fail.
write_put_array(_).
write_put_array_aux([]).
write_put_array_aux([X]):-write(X).
write_put_array_aux([X|Xs]):-
    write(X),
    write(','),
    write_put_array_aux(Xs).
write_def_var_match(Tab):-
      action(def_var_match,Name,Type,Value,_),
      add_num_case([Name],[Name2]),
      (Type == 'boolean' -> (Value == 0 -> Value2 ='false';
Value2='true');Value2=Value),
      format('~s ~w ~w= ~w;\n',[Tab,Type,Name2,Value]),
      fail.
write_def_var_match(_).
write_match(Tab):-
      action(match,Name,_,Value,PackageClass),
      %add num case to  var name
      add_num_case([Name],[Name2]),
      % if value is var  then add nun  case
      (is_var(Value)->add_num_case([Value],[Value2]);true),
      % if name is a boolean var  then put true or false
      (is_boolean(Name)->(Value==0->Value2='false';
Value2='true');true),
      (\+nonvar(Value2)->Value2=Value; true),

      (\+use_reflection(true)->
```

```
          format('~s ~w = ~w;\n',[Tab,Name2,Value2])
          ;
          (\+is_atr(Name2)->
              %is not atribute
              format('~s ~w = ~w;\n',[Tab,Name2,Value2])
              ;
              %is atribute
             get_var_atr(Name,Var,Atr),
             (is_public(PackageClass,Atr)->
                format('~s ~w = ~w;\n',[Tab,Name2,Value2])
                  ;
               %is private,
               action(def_var,Var, Class,_,_),%saco el tipo de la
variable
               add_num_case([Var],[Var2]),
               format('~s//Use reflection to modify private field
~w\n',[Tab,Atr]),
               format('~s final Field fields_~w_~w[] =
~w.class.getDeclaredFields();\n',
                    [Tab,Var2,Atr,Class]),
               format('~s for (int i = 0 ;i < fields_~w_~w.length;
i++){\n',[Tab,Var2,Atr]),
               format('~s   if
("~w".equals(fields_~w_~w[i].getName())){\n',[Tab,Atr,Var2, Atr]),
               format('~s       Field f =fields_~w_~w[i];\n',[Tab,
Var2,Atr]),
               format('~s       f.setAccessible(true);\n',[Tab]),
               format('~s       f.set(~w,~w);\n',[Tab, Var2,Value2]),
                    format('~s       break;\n',[Tab]),
               format('~s   }\n',[Tab,Tab]),
               format('~s }\n',[Tab])
           )        )
     ),
       fail.

write_match(_).

%*****************************************
% write all assert, this is only used to debug
%*****************************************
assert_value:-
     write('/*\n'),
     write('Current num case= '),
     num_case(U),
     write(U),
     write('\nIs  constructora: '),
     is_constructor(II),
     write(II),
     write('\nIs static: ' ),
     (is_static-> write('yes'); write('no')),
     write('\nThis object :'),
     this(X),
     write(X),
     write('\nInput objects: '),
     assert_value_entrada,
     write('\nInput Heap objects:\n'),
```

```
      assert_value_heap_in,
      assert_value_heap_array_in,
      write('\n'),
      write('\nObject don`t referenced in heap (heap_soul): \n'),
      assert_value_heap_soul,
      write('New_this Object:'),
      new_this(Y),
      write(Y),
      write('\nOutput elem: '),
      assert_value_salida,
      write('\nOutput Heap objects:\n'),
      assert_value_heap_out,
      assert_value_heap_array_out,
      write('\n'),
      write('Actions:\n'),
      assert_value_acciones,
      write('\n\nCode:\n'),
      write('*/').
assert_value_entrada:-
      input_elem(X),
      write(X),
      write(', '),
      fail.
assert_value_entrada.

assert_value_salida:-
      output_elem(X),
      write(X),
      write(', '),
      fail.
assert_value_salida.

assert_value_heap_in:-
      heap_elem_in(X,Y,Z),
      write(heap_elem_in(X,Y,Z)),
      write('\n'),
      fail.
assert_value_heap_in.

assert_value_heap_out:-
      heap_elem_out(X,Y,Z),
      write(heap_elem_out(X,Y,Z)),
      write('\n'),
      fail.
assert_value_heap_out.

assert_value_heap_array_in:-
      heap_array_in(X,Y,Z,W),
      write(heap_array_in(X,Y,Z,W)),
      write('\n'),
      fail.
assert_value_heap_array_in.

assert_value_heap_array_out:-
      heap_array_out(X,Y,Z,W),
      write(heap_array_out(X,Y,Z,W)),
```

```
      write('\n'),
      fail.
assert_value_heap_array_out.
assert_value_heap_soul:-
      heap_soul(X,Y,Z,W,M),
      write(heap_soul(X,Y,Z,W,M)),
      write('\n'),
      fail.
assert_value_heap_soul.

assert_value_acciones:-
      action(Q,W,E,R,S),
      write(action(Q,W,E,R,S)),
      write('\n'),
      fail.
assert_value_acciones.


%*****************************************
%deletes
%*****************************************
delete_asserts:-
        delete_heap_elems_in,
      delete_heap_elems_out,
      delete_heap_array_in,
      delete_heap_array_out,
      delete_actions,
      delete_input,
      delete_output,
      delete_this,
      delete_expected,
      delete_heap_soul.
delete_input:-
      retract(input_elem(_)),
      fail.
delete_input.

delete_expected:-
      retract(expected(_)),
      fail.
delete_expected.

delete_output:-
      retract(output_elem(_)),
      fail.
delete_output.

delete_actions:-
      retract(action(_,_,_,_,_)),
      fail.
delete_actions.

delete_heap_elems_in:-
      retract(heap_elem_in(_,_,_)),
      fail.
delete_heap_elems_in.
```

```
delete_heap_elems_out:-
      retract(heap_elem_out(_,_,_)),
      fail.
delete_heap_elems_out.
delete_heap_array_in:-
      retract(heap_array_in(_,_,_,_)),
      fail.
delete_heap_array_in.
delete_heap_array_out:-
      retract(heap_array_out(_,_,_,_)),
      fail.
delete_heap_array_out.
delete_this:-
      retract(this(_)),
      retract(new_this(_)),
      fail.
delete_this.
delete_heap_soul:-
      retract(heap_soul(_,_,_,_,_)),
      fail.
delete_heap_soul.



%*******************************************
%utils
%*******************************************
%list_test_cases(-A,+B)
list_test_cases(JMSig,ListCases2):-
      findall(test_case(InValueList,InHeap,
OutValue,OutHeap,Num_case),test_case(JMSig,[InValueList,InHeap],[OutVal
ue,OutHeap,ok],_,_,Num_case),ListCases),
      remove_types(ListCases, ListCases2).

%list_test_exceptions(-A, +B)
list_test_exceptions(JMSig,ListExceptions2):-
      findall(exc(InValueList,InHeap,OutValue,OutHeap,Num_exc),test_case
(JMSig,[InValueList,InHeap], [OutValue,OutHeap,
exception(_)],_,_,Num_exc),ListExceptions),
      remove_types(ListExceptions, ListExceptions2).



%this method return a valid test metod mame, ex
test_exp___int___int_int
%get_test_method_name(-A,-B,-C,+D)
get_test_method_name(JMSParams,Ret2,Method,Name):-
      params_to_str([Ret2],Ret),
      params_to_str(JMSParams,StringParams),
      append_(['test_'],[Method],Str1),
      append_(Str1,['___'],Str2),
      append_(Str2,[Ret],Str3),
      append_(Str3,['___'],Str4),
      append_(Str4,[StringParams],Name2),
```

```
      atom_concat(Name2,Name).
params_to_str(In,Out):-
      params_to_str_aux(In,N3),
      reverse(N3,N4),
      remove_first_elem(N4,'_',N5),
      reverse(N5,N6),
      atom_concat(N6,Out).
params_to_str_aux([],Ret):-Ret=[].
params_to_str_aux([primitiveType(X)|Xs],Ret):-
      params_to_str_aux(Xs,L),
      append_([X],['_'],Str3),
      append_(Str3,L,Ret).
params_to_str_aux([refType(objectType(X))|Xs],Ret):-
      params_to_str_aux(Xs,L),
      append_([X],['_'],Str3),
      append_(Str3,L,Ret).
params_to_str_aux([refType(arrayType(X))|Xs],Ret):-
      params_to_str_aux([X|Xs],L),
      append_(['Array$'],L,Ret).
params_to_str_aux([none], Ret):-
      Ret=['none'].


%get  metod name
jmsig_to_ShortMethodName(JMSig,ShortMethodName):-
      jmsig_to_msig(JMSig, methodSignature(_,ShortMethodName, _, _)).
%get input types
jmsig_to_Params(JMSig,Params):-
      jmsig_to_msig(JMSig,methodSignature(_,_ , Params,_)).
%get output type
jmsig_to_RetType(JMSig,Ret):-
      jmsig_to_msig(JMSig, methodSignature(_,_, _,Type)),
      (Type= none -> Ret = none;
       functor(Type,F,_ ),
       (F=refType(F1)->
        (F1=objectType(T)-> Ret=T;true);Ret=Type);true).
%get package name
jmsig_to_package(JMSig, Package):-
      jmsig_to_msig(JMSig, methodSignature(Name,_, _, _)),
      atom_chars(Name,L),
      reverse(L,L2),
      get_pack_class(L2,_,P),
      reverse(P,P2),
      atom_concat(P2,Package).
%get class name
jmsig_to_class(JMSig, Class):-
      jmsig_to_msig(JMSig, methodSignature(Name,_, _, _)),
      atom_chars(Name,L),
      reverse(L,L2),
      get_pack_class(L2,C,_),
      reverse(C,C2),
      atom_concat(C2,Class).



%remove first elem in the list, if method is static
```

```
%remove_this(-InList,+OutList):-
remove_this([],Ret):- Ret=[].
remove_this([X|Xs],Ret):-
     (is_static-> Ret = [X|Xs];
                 Ret=Xs).
%remove X only if X is at the beginning of  the  list
%remove_first_elem(-List,-X,+List)
remove_first_elem([],_,[]).
remove_first_elem([Elem|Xs],Elem,N ):- N= Xs.
remove_first_elem([Elem|Xs],_,N ):- N= [Elem|Xs].

%change in a String the symbol '/' by'.'
remove_slash(In, Out):-
     atom_codes(In,N2),
     replace_det2(N2, 47,46, N3), % 47 is the ascii code of '/' and 46
is '.'
     reverse(N3,N4),
     remove_first_elem(N4,46,N5),
     reverse(N5,Out).

%change in a String the symbol 'X' by'Y'
replace_chars(In,X,Y, Out):-
     atom_chars(In,N2),
     replace_det2(N2, X,Y, N3),
     atom_concat(N3,Out).
%Replace all  appearances of X by Y in the XS list an return YS list
%replace_det2(-XS,-X,-Y,+YS)
replace_det2([],_,_,[]).
replace_det2([A|B],A,Y,[Y|L]):- replace_det2(B,A,Y,L).
replace_det2([A|B],X,Y,[A|L]):- A\=X, replace_det2(B,X,Y,L).


%remove first elem of a list
remove_first([],Ret):- Ret=[].
remove_first([_|Xs], Ret):- Ret=Xs.

%list_to_str(InList,Str)
list_to_str(L,Out):-
     list_to_str2(L,O),
     atom_concat(O,Out).
list_to_str2([],Out):-Out=[].
list_to_str2([X|[]],Out):- Out=[X].
list_to_str2([X|Xs],Out):-
     list_to_str2(Xs,L),
     append_([X],[','],L2),
     append_(L2,L,Out).


%this method recive as input a list of strings, and add  the current
num_case to all elem of the list
%Ex: ['aa', 'bb','var.art']  return ['aa_1', 'bb_1', 'var_1.atr']
add_num_case([],Ret):- Ret=[].
add_num_case([''|Xs], Ret):-
        add_num_case(Xs, R),
        append_([''],R,Ret),!.
```

```
add_num_case(['null'|Xs], Ret):-
        add_num_case(Xs, R),
        append_(['null'],R,Ret),!.
add_num_case([X|Xs], Ret):-
     (number(X)->add_num_case(Xs, R),append_([X],R,Ret),!;fail).
add_num_case([X|Xs], Ret):-
     atom_chars(X,Y),
     (\+contains(Y,'.')->
          %X is not atribute
        add_num_case(Xs,L),
        num_case(Nc),
        atom_chars(X,X1),
        append_(X1,['_'],X2),
        append_(X2,[Nc],X3),
        atom_concat(X3,X4),
        append_([X4],L,Ret)
           ;
        %X is atribure:  X=var.atr
        add_num_case(Xs,L),
        num_case(Nc),
        get_var_atr(X,Var,Atr),
           atom_chars(Var,V),
        atom_chars(Atr,A),
        append_(V,['_'],D0),
        append_(D0,[Nc],D1),
        append_(D1,['.'],D2),
        append_(D2,A,D3),
        atom_concat(D3,D4),
        append_([D4],L,Ret)).



%return if Name is something like 'var...'
is_var(Name):-
     (nonvar(Name)->
         atom_chars(Name, [X,Y,Z|_]),
         (X== 'v', Y=='a', Z=='r' -> true;fail);
      fail).

%get_var_atr(-A, -B, +C)
%get_var_atr('var.atr', B, C). return  B= var, C=atr
get_var_atr(T,Var,Atr):-
     atom_chars(T,T2),
     get_var_atr2(T2,Var1,Atr1),!,
     atom_concat(Var1,Var),
     atom_concat(Atr1, Atr).

get_var_atr2([],Var,Atr):- Var=[], Atr=[].
get_var_atr2(['.'|Xs],Var, Atr):-Var=[], Atr=Xs.
get_var_atr2([X|Xs], Var,Atr):-
     get_var_atr2(Xs,V,Atr),
     append_([X],V,Var).

is_boolean(Name):-
     (action(def_var,Name,boolean,_,_)-> true; fail).
```

```
is_atr(Name):-
     atom_chars(Name,Y),
     (contains(Y,'.')-> true;fail).

%is_public(-A,-B)
is_public(Class,Atr):-
       classname_fields(Class, Fl),
       find_field(Fl,Class,Atr,F),
       (field_isPrivate(F)-> fail; true).


%find an atribure in a field list
find_field([],_,_,'error atr not found').
find_field([field(fieldSignature(T, N, _), _, _, Public,_)|Xs],Type,
Name, Field):-
     (T==Type , N==Name ->
          Field=field(fieldSignature(T, N, _), _, _, Public,_);
          find_field(Xs, Type,Name, Field)).

% receive this  'name[]',  and return  'name[0]'
put_array_num(Str,Ret):-
     atom_chars(Str, Str2),
     reverse(Str2, [_,_|Xs]),
     Str3=[']','0','['|Xs],
     reverse(Str3,Str4),
     atom_concat(Str4,Ret).

%remove types in InValueList and in OutValue
remove_types([],[]).
remove_types([test_case(InValueList,InHeap,OutValue,OutHeap,Num_case)|X
s], Ret):-
     remove_types2(InValueList, In),
     remove_types2([OutValue], [Out|_]),
     Y= test_case(In,InHeap,Out,OutHeap,Num_case),
     remove_types(Xs, Ys),
     Ret=[Y|Ys].
remove_types([exc(InValueList,InHeap,OutValue,OutHeap,Num_case)|Xs],
Ret):-
     remove_types2(InValueList, In),
     remove_types2([OutValue], [Out|_]),
     Y= exc(In,InHeap,Out,OutHeap,Num_case),
     remove_types(Xs, Ys),
     Ret=[Y|Ys].

%this method  convert a list like [int(0), boolean(A),ref(K)]  to [0,A,
ref(K)]
%remove_types(-A, +B)
remove_types2([],[]).
remove_types2([X|Xs],Ret):-
     (nonvar(X)->
     functor(X,A,N),
       (N==1->
          arg(1,X,Value),
         (A== ref->
            Y=X;
```

```
       Y=Value)
  ;Y=X)
;Y=X),
remove_types2(Xs,Ys),
Ret=[Y|Ys].
```

Archivo make_asserts.pl

```
/** <module>

This module implements a series of predicates to generate Java code,
@author Fernando Rodriguez, Jaime Saez
@license GPL
*/
:-module(make_asserts,[make_asserts/1]).
:-use_module(pet(pet_utils),[append_/3,atom_concat/2]).
:-
use_module(junit_utils,[is_static/0,get_pack_class/3,is_java_primitive/
1]).
:-use_module(pet(heap_operations),[new_ref/1]).
:-use_module(junit_generator,[ action/5,
                  this/1,
                  new_this/1,
                  output_elem/1,
                  input_elem/1,
                  heap_elem_in/3,
                  heap_array_in/4,
                  heap_elem_out/3,
                  heap_array_out/4,
                  heap_soul/5,
                  class/1,
                  inParams/1,
                  outParam/1,
                  expected/1]).

:-dynamic dir/1. %(input or output)
:-dynamic var_copy/2. % if two variables have the same mame

%*****************************************
%Make asserts
%this method assert all actions,
%*****************************************

make_asserts(test_case(InValueList, InHeap, OutValue, OutHeap,_)):-


       % put_ref method, replace varliables X by Ref(X), in fields
values and array elems
       % this replacements  imply  create a sould object
       asserta(dir('in')),
         put_ref(InHeap,InHeap2),
       retract(dir(_)),
       asserta(dir('out')),
          put_ref(OutHeap,OutHeap2),
       retract(dir(_)),

       put_ref_list(InValueList),
       put_ref_list([OutValue]),

       asserta(dir('in')),
```

```
           process_InHeap(InHeap2),
           assert_name_this(InValueList),
           process_InValueList(InValueList, InHeap2,0),
        retract(dir(_)),
        asserta(dir('out')),
           assert_name_new_this,
           process_OutHeap(OutHeap2),
           (\+OutValue==none-
>process_OutValue([OutValue],OutHeap2);true),
           assert_action_new_this(OutHeap2),
        retract(dir(_)),
        assert_actions.



%*****************************************
%process  heaps
%*****************************************

process_InHeap(InHeap):-
     replace_vars(InHeap),
     remove_ref(InHeap, NewInHeap),
     assert_heap_elems(NewInHeap).
process_OutHeap(OutHeap):-
     remove_ref(OutHeap, NewOutHeap),
     assert_heap_elems(NewOutHeap).

% Replace vars by numbers
replace_vars(heap([],_)):-true.
replace_vars(heap([(Ref,_)|Xs],_)):-
     (\+nonvar(Ref)->
            new_ref(I),
           Ref=I,
           replace_vars(heap(Xs,_));
      replace_vars(heap(Xs,_))).

%this method replace the 'ref(Var)' string  in heap by java name Var
%if var isn't in heap,  creates a heap soul
%remove_ref(-A,+B)
remove_ref(heap(Xs,B),heap(NewHeap,B)):-
     remove_ref2(Xs, NewHeap).
remove_ref2([],Ret):-Ret=[].
remove_ref2([(R, object(Name,Field_list))|Xs], Ret):-
     remove_ref2(Xs,Ret2),
     remove_ref_fields(Field_list,New_list),
     append_([(R,object(Name,New_list))],Ret2,Ret).
remove_ref2([(R, array(T,N,L))|Xs], Ret):-
       remove_ref2(Xs,Ret2),
       remove_ref_array(L,T,New_array),
       append_([(R, array(T,N,New_array))],Ret2,Ret).
remove_ref_array([],_,Ret):-Ret=[].
remove_ref_array([Xs],_, Ret):-
     (\+nonvar(Xs)-> Ret= []).
remove_ref_array([X|Xs],Type, Ret):-
     (nonvar(X)->
            functor(X,F,A),
```

```
            %If X is a Reference
            (A==1,F= ref -> arg(1,X,Val),
                %Si Val es una variable, es decir,  no esta en el heap
creo un heap_sould
                %If Val, is a variable, it isn't in heap, create a
heap_sould
                (\+nonvar(Val)->
                       new_ref(V),
                      get_java_name(V, Java_name),
                      dir(Dir),
                      asserta(heap_soul(V,'default',Java_name,_,Dir))
                  ;
                %If Val isn't  a variable, add 'var_in'
                get_java_name(Val, Java_name)),
                   remove_ref_array(Xs,Type,L),
                   append_([Java_name],L,Ret); true),
            %Is is a value  do anything
            (A==0->  remove_ref_array(Xs,Type,L),
                    append_([X],L,Ret);true)
      ;
      %if elem is a variable, create a heap_soul
      remove_ref_array(Xs,Type,L),
          new_ref(XX),
      get_java_name(XX, Java_name),
         concat('var_in_', XX, Java_name),
      dir(Dir),
      asserta(heap_soul(XX,'default',Java_name,_,Dir)),
      append_([Java_name],L,Ret)).


remove_ref_fields([], Ret):-Ret=[].
remove_ref_fields([Field|Xs], Ret):-
        %to avoid error initialize  Xs
     (\+nonvar(Xs)-> Xs=[]; true),
     %If isn't a varaible
     (nonvar(Field)->
          functor(Field,Y,_),
              %is a field
          (Y== field->
             arg(1,Field, Type),
             arg(2,Field, X),
             %if field has value
             (nonvar(X)->
                 functor(X,F,A),
                 %if this valuse is a reference
                 (A==1,F= ref -> arg(1,X,Val),
                       %If reference is a variable
                      (\+nonvar(Val)->
                       new_ref(Val),
                       get_java_name(Val,Java_name),
                       dir(Dir),
                       get_type(Type,Type2),
                       jmsig_java_type(Type2,Type3),

asserta(heap_soul(Val,'default',Java_name,Type3,Dir))
                   ;
```

```
                          %if reference isn't a variable
                    get_java_name(Val, Java_name),
                    (heap_soul(Val,_,_,_,_)->
                     dir(Dir),
                     get_type(Type,Type2),
                     jmsig_java_type(Type2,Type3),

asserta(heap_soul(Val,'default',Java_name,Type3,Dir))
                     ;
                     true)),

                         New_Field= field(Type,Java_name),
                         remove_ref_fields(Xs,L),
                         append_([New_Field],L,Ret);true),
                  %If isn't a reference
                  (A==0-> New_Field= field(Type,X),
                         remove_ref_fields(Xs,L),
                         append_([ New_Field],L,Ret);true)
          ;
          %if value has a variable
           new_ref(X),
          get_java_name(X, Java_name),
          dir(Dir),
          get_type(Type, Type2),
          jmsig_java_type(Type2,Type3),
             asserta(heap_soul(X,'default',Java_name,Type3,Dir)),
          Field= field(Type,Java_name),
             remove_ref_fields(Xs,L),
             append_([Field],L,Ret))
          );true).



assert_heap_elems(heap([],_)):-true.
assert_heap_elems(heap([(Ref,Value)|Xs],_)):-
    dir(Dir),
    (Dir=='in' ->
         (nonvar(Value)->
          functor(Value,Fun,_),
          (Fun=array-> Is_array=yes;true);true),
            concat('var_in_',Ref,Java_name),
          (Is_array == yes ->
                assertz(heap_array_in(Ref,Value,Java_name,_))
                ;
                assertz(heap_elem_in(Ref,Value,Java_name))),
           assert_heap_elems(heap(Xs,_))
    ;
        (nonvar(Value)->
         functor(Value,Fun,_),
         (Fun=array-> Is_array=yes;true);true),
         concat('var_out_',Ref,Java_name),
         (Is_array == yes ->
                assertz(heap_array_out(Ref,Value,Java_name,_));
                assertz(heap_elem_out(Ref,Value,Java_name))),
         assert_heap_elems_out(heap(Xs,_))).
```

```
assert_heap_elems_out(heap([],_)):-true.
assert_heap_elems_out(heap([(Ref,Value)|Xs],_)):-
      (nonvar(Value)->
       functor(Value,Fun,_),
       (Fun=array-> Is_array=yes;true);true),

      concat('var_out_',Ref,Java_name),
      (Is_array == yes ->
              assertz(heap_array_out(Ref,Value,Java_name,_));
              assertz(heap_elem_out(Ref,Value,Java_name))),
      assert_heap_elems_out(heap(Xs,_)).

%*****************************************
%process in value List
%*****************************************

process_InValueList([],_,_):-true,!.
process_InValueList([X|Xs],Heap,N):-
      N2 is N+1,
      (nonvar(X)->
           functor(X,Y,_),
             %If is  a ref
           (Y== ref->
              arg(1,X,Ref),
              this(This),
              %if is this elem
              (This==Ref ->
             %If is this and isn't in heap
                (\+find_heap(Ref,Heap)->
                      class(Class),
                   concat('var_in_',Ref,Java_name),
                        %if it has not been declared, declare it
                   (\+action(def_var,Java_name,Class,_,_)->
                        assertz(action(def_var,Java_name, Class,_,_)),
                        assertz(action(new,Java_name, Class,_,_));true);
                 % If this is in heap
                 true),
             (\+N==0->assertz(input_elem(Java_name));true)%add this as
input elem
              ;
            %ref isn't  this  and is in  heap
            (find_heap(Ref,Heap)->
            %if is array
            (is_array(Ref)-> % add array type
                  heap_array_in(Ref,Value,Java_n,_),
                     retract(heap_array_in(Ref,_,_,_)),
                  get_input_type(N2,Type),
                  Value=array(_,_,Elems),
                  put_type_multi_arrays_in(Type,Elems),
                  assertz(heap_array_in(Ref,Value,Java_n,Type))
                  ;true),

                concat('var_in_',Ref,Java_name),
                assertz(input_elem(Java_name))
                ;
```

```
            %ref isn't this and isn't in heap


            concat('var_in_',Ref,Java_name),
            assertz(input_elem(Java_name)),
            dir(Dir),
            get_input_type(N2,Type),
            %create a heap soul
            (\+heap_soul(_,_,Java_name,_,_)->

asserta(heap_soul(Ref,'default',Java_name,Type,Dir));true) ))
          ;
             %is a value
           new_ref(I),
          concat('var_in_',I,Java_name),
          get_input_type(N2,Type),
          assertz(input_elem(Java_name)),
          assertz(action(def_var, Java_name,Type, X,_)),
          assertz(action(match, Java_name,_,X,_)))
     ;
     %Is a variable
      new_ref(I),
     concat('var_in_',I,Java_name),
     assertz(input_elem(Java_name)),
     get_input_type(N2,Type),
     (var_copy(X,Java_name_copy)->
         assertz(action(def_var,Java_name,Type,_,_)),
         assertz(action(match,Java_name,_,Java_name_copy,_))
     ;

         (is_same_var(X,Xs)->
             assertz(var_copy(X,Java_name))
         ;
             true),

         (is_array_test(Type)->
             assertz(action(put_array,Java_name,Type,[],_))

         ;
             assertz(action(def_var,Java_name, Type,_,_)),
             get_def_val(Type,Def_val),
             (Def_val==null->
                 assertz(action(new, Java_name,Type,_,_))
             ;
                 assertz(action(match, Java_name,_,Def_val,_))))))),

     process_InValueList(Xs, Heap,N2).

is_array(Ref):-heap_array_in(Ref,_,_,_).
is_array(Ref):-heap_array_out(Ref,_,_,_).



%*****************************************
%process output  value
%*****************************************
```

```
process_OutValue([X|[]],Heap):-

      (nonvar(X)->
           functor(X,Y,_),
              %Is a ref
           (Y== ref->
              arg(1,X,Ref),
              this(This),
              %is  this
              (This==Ref ->
             %is  this and isn't in heap
                (\+find_heap(Ref,Heap)->
                      class(Class),
                         concat('var_out_',Ref,Java_name),
                      assertz(action(def_var,Java_name, Class,_,_)),
                      assertz(action(new,Java_name, Class,_,_));
                  %Is this and is in heap
                  true);
            %isn't this and isn't  in heap
            (find_heap(Ref,Heap)->
            get_output_type(Type),
            (is_array(Ref)-> %add array type
                  heap_array_out(Ref,Value,Java_n,_),
                     retract(heap_array_out(Ref,_,_,_)),
                  Value=array(_,_,Elems),
                  put_type_multi_arrays_out(Type,Elems),
                  assertz(heap_array_out(Ref,Value,Java_n,Type))
                  ;
                  true),

            Expected='expected_',
            asserta(expected(Expected)),
            assertz(action(def_var, Expected,Type, _,_)),
               concat('var_out_',Ref,Java_name),
               assertz(output_elem(Java_name));true);true)
           ;
              %is a  value
            new_ref(I),
           concat('var_out_',I,Java_name),
           assertz(output_elem(Java_name)),
           get_output_type(Type),
           Expected='expected_',
           asserta(expected(Expected)),
           assertz(action(def_var, Java_name,Type, _,_)),
           assertz(action(def_var, Expected,Type, _,_)),
           assertz(action(match, Java_name,_,X,_)))
      ;
      %is a variable
       new_ref(I),
      concat('var_out_',I,Java_name),
      assertz(output_elem(Java_name)),
      get_output_type(Type),
      (is_array_test(Type)->
          assertz(action(put_array,Java_name,Type,[],_))
```

```
      ;
            dir(Dir),
         asserta(heap_soul(I,'default',Java_name,Type,Dir))),

      Expected='expected_',
      asserta(expected(Expected)),
      assertz(action(def_var, Expected,Type, _,_))).



% put_ref method, replace varliables X by Ref(X), in fields values and
array elems
% this replacements  imply  create a sould object
put_ref(heap(Xs,B),heap(NewHeap,B)):-
     put_ref2(Xs, NewHeap).
put_ref2([], Ret):-Ret=[].
put_ref2([(R, object(Name,Field_list))|Xs], Ret):-
     put_ref2(Xs,Ret2),
     put_ref_fields(Field_list,New_list),
     append_([(R,object(Name,New_list))],Ret2,Ret).
put_ref2([(R, array(T,N,L))|Xs], Ret):-
       put_ref2(Xs,Ret2),
       put_ref_array(L,N,New_array),
       append_([(R, array(T,N,New_array))],Ret2,Ret).
put_ref_fields([], Ret):-Ret=[].
put_ref_fields([Field|Xs], Ret):-
        %to avoid errors  initialize xs
     (\+nonvar(Xs)-> Xs=[]; true),
     (nonvar(Field)->
          functor(Field,Y,_),
               %Is es un field
          (Y== field->
             arg(1,Field, Type),
             arg(2,Field, X),
             %i  el valor del campo es una variable
             (\+nonvar(X)->
                 new_ref(N),
               get_java_name(N, Java_name),
               dir(Dir),
               get_type(Type, Type2),
               jmsig_java_type(Type2,Type3),
                  asserta(heap_soul(N,'default',Java_name,Type3,Dir)),
               Field2= field(Type,Java_name),
                  put_ref_fields(Xs,L),
                  append_([Field2],L,Ret)
          ;
          %si el valor del campo  no es una variable
              put_ref_fields(Xs,L),
              append_([Field],L,Ret));true);true).

put_ref_array([],_,Ret):-Ret=[],!.
put_ref_array(_,0,Ret):-Ret=[],!.
put_ref_array([X|Xs],N, Ret):-
          N1 is N-1,
       put_ref_array(Xs,N1,L),
```

```
      %si el elememto es una variable lo cambio por una  ref, si no lo
dejo igual
     (\+nonvar(X)-> append_([ref(X)],L,Ret) ; append_([X],L,Ret)).


%asert the name of this object by the imput args list

assert_name_this([]):-asserta(this('the method is static')),!.
assert_name_this([X|_]):-
     (is_static -> asserta(this('the method is static'));
         (\+nonvar(X)-> asserta(this('the method is static'));
              functor(X,F,_),
              (F= ref -> arg(1,X,Val),
                       (\+nonvar(Val)->
                          new_ref(I),
                        Val=I;true),
                        asserta(this(Val));true))),
     !.
assert_name_this(_):- write('error al poner  el this').
assert_name_new_this:-this(Val),asserta(new_this(Val)),!.
assert_name_new_this:-asserta(new_this('the method is static')).

assert_action_new_this(OutHeap):-
     this(This),
     (\+This=='the method is static'->
        (\+find_heap(This,OutHeap)->
           class(Class),
           concat('var_out_',This,Java_name),
           assertz(action(def_var,Java_name, Class,_,_)),
           assertz(action(new,Java_name, Class,_,_));

           true);
        true).



%*****************************************
%assert actions
%%*****************************************


assert_actions:-
     assert_object_actions,
     assert_array_actions,
     assert_object_actions_out,
     assert_array_actions_out,
     assert_heap_soul.


assert_heap_soul:-
     heap_soul(Num,_,Java_name,Type,_),
     assert_actions_heap_soul(Num,_,Java_name,Type,_),
     fail.
assert_heap_soul.

assert_array_actions:-
```

```
      heap_array_in(Ref,Value,Java_name,Type),
      assert_actions_array(Ref,Value,Java_name,Type),
      fail.
assert_array_actions.

assert_object_actions:-
      heap_elem_in(Ref,Value,Java_name),
      assert_actions_object(Ref,Value,Java_name,in),%in, indica que  es
de entrada
      fail.
assert_object_actions.

assert_array_actions_out:-
      assert_array_actions_out_aux,
      assert_array_actions_out_aux2.

assert_array_actions_out_aux:-
      heap_array_out(Ref,Value,Java_name,_),
      heap_array_in(Ref,_,_,New_Type),
      retract(heap_array_out(Ref,_,_,_)),
      Value=array(_,_,Elems),
      asserta(heap_array_out(Ref,Value,Java_name,New_Type)),
      put_type_multi_arrays_out(New_Type,Elems),
      fail.
assert_array_actions_out_aux.

assert_array_actions_out_aux2:-
      heap_array_out(Ref,Value,Java_name,Type),
      assert_actions_array_out(Ref,Value,Java_name,Type),
      fail.
assert_array_actions_out_aux2.


assert_object_actions_out:-
      heap_elem_out(Ref,Value,Java_name),
      assert_actions_object(Ref,Value,Java_name,out),%out, indica que es
de salida
      fail.
assert_object_actions_out.

%assertar acciones de heap_soul

assert_actions_heap_soul(_,_,Java_name,Type,_):-
          (\+is_java_primitive(Type)->
                  assertz(action(def_var,Java_name, Type,_,_)),
                  assertz(action(new,Java_name, Type,_,_))
                 ;
                 get_def_val(Type,Value),
                 assertz(action(def_var_match,Java_name,
Type,Value,_))).
      .


%assertar acciones de arrays

assert_actions_array(_,array(_,_,Elems),Java_name,Type):-
```

```
      (nonvar(Type)->
            assertz(action(put_array,Java_name,Type,Elems,_))
      ;
            true).
assert_actions_array_out(_,array(_,_,Elems),Java_name,Type):-
      (nonvar(Type)->
            assertz(action(put_array,Java_name,Type,Elems,_))
      ;
            true).


%assertar acciones de objetos
% el campo Dir indica si estoy procesando los de entrada o de salida
assert_actions_object(_,object(Type, Xs),Java_name,Dir):-
       %saco el nombre de la clase
      atom_chars(Type,L),
      reverse(L,L2),
      get_pack_class(L2,L3,_),
      get_class_interna(L3,C,_),
      reverse(C,C2),
      atom_concat(C2,New_type),
       %asserto las aciones
      assertz(action(def_var,Java_name, New_type,_,_)),
      assertz(action(new,Java_name, New_type,_,_)),
      assert_actions_fields(Xs, Java_name,Dir),
      !.

assert_actions_object(_,_,_,_). %para que no falle cuando no es objeto

assert_actions_fields([],_,_).
assert_actions_fields([X|Xs],Java_name,Dir):-
      (\+nonvar(X)->
            true
      ;
            assert_actions_fields_aux(X,Java_name,Dir),
            assert_actions_fields(Xs,Java_name,Dir)).
assert_actions_fields_aux(field(Type,Value),Java_name,Dir):-
      get_atr(Type,Atr),
      (\+Atr=='this$0'->
         (Dir == in ->
            (heap_array_in(X,Value,Y,Z)->
               retract(heap_array_in(X,Value,Y,Z)),
               get_type(Type,T),
               jmsig_java_type(T,T2),
               %funcion que pone tipo a los arrays heredados(en este
hueco)
               Value=array(_,_,Elems),
             put_type_multi_arrays_in(T2,Elems),
               asserta(heap_array_in(X,Value,Y,T2));true);true),
         (Dir == out ->
            (heap_array_out(X,Value,Y,Z)->
               retract(heap_array_out(X,Value,Y,Z)),
               get_type(Type,T),
               jmsig_java_type(T,T2),
               %funcion que pone tipo a los arrays heredados(en este
hueco)
```

```
              Value=array(_,_,Elems),
            put_type_multi_arrays_out(T2,Elems),
              asserta(heap_array_out(X,Value,Y,T2));true);true),
      atom_concat(Java_name,'.',N),
      atom_concat(N,Atr,Name),
      get_type(Type, Atr_type),
      get_class(Type,Atr_class),
      assertz(action(match,Name,Atr_type , Value,Atr_class));true).


%*****************************************
%Utils
%*****************************************

find_heap(_,heap([],_)):- fail.
find_heap(X,heap([(Y,_)|Ys],_)):-
        (X == Y -> true;
         find_heap(X,heap(Ys,_))).
find_heap(_,heap(_,_)):- fail.



get_input_type(N,Type):-
      inParams(P),!,
      (\+is_static->N1 is N-1; N1=N),
      get_elem(N1,P,Elem),
      jmsig_type_to_java(Elem,Type),!.
get_output_type(Type):-
      outParam(P),
      (nonvar(P)-> jmsig_type_to_java(P,Type);fail).

%This method recive a list  like
[refType(objectType('heap/SortedListInt')), primitiveType(int),
refType(arrayType(primitiveType(int)))] and return SortedListInt or int
or int[].

jmsig_type_to_java(none,Ret):-
      Ret='none'.
jmsig_type_to_java(primitiveType(X),Ret):-
      Ret= X.
jmsig_type_to_java(refType(arrayType(X)),Ret):-
      arrayType_to_java(refType(arrayType(X)),O),
      Ret=O.
jmsig_type_to_java(refType(objectType(X)),Ret):-
      atom_chars(X,L),
      reverse(L,L2),
      get_pack_class(L2,C,_),
      reverse(C,C2),
      atom_concat(C2,Ret).

%transform a array in JMSig sintax to a java sintax,
% Ej:refType(arrayType(refType(arrayType(primitiveType(int)))))
return 'int[][]'
arrayType_to_java(primitiveType(X),Out):-
      Out= X.
arrayType_to_java(refType(objectType(X)),Ret):-
```

```
      atom_chars(X,L),
      reverse(L,L2),
      get_pack_class(L2,C,_),
      reverse(C,C2),
      atom_concat(C2,Ret).
arrayType_to_java(refType(arrayType(X)),Out):-
      arrayType_to_java(X,L),
      append_([L],[[]],L2),
      atom_concat(L2,Out).

get_elem(N,Xs,Ret):-
      get_elem2(N,0,Xs,Ret).
get_elem2(N,_,[],Ret):-
      format('\n Error, elem ~w not found in list ~w',[N, Ret]).
get_elem2(N,Step,[X|Xs],Ret):-
      Step2 is Step +1,
      (N==Step2 -> Ret=X;
        get_elem2(N,Step2, Xs,Ret)).

%Jaime di que es lo que hace este método
get_class_interna([],C,P):- C=[], P=[],!.
get_class_interna(['$'|Xs],C,P):- C=[],P=Xs,!.
get_class_interna([X|Xs],C,P):-
      get_class_interna(Xs,C1,P),
      append_([X],C1,C).

%dado algo como esto t= 'nodo/aol.derecho:tipo' devuelve A=derecho
get_atr(T,Ret):-
      atom_chars(T,T2),
      reverse(T2,T3),
      get_atr2(T3,Ret2),!,
      reverse(Ret2,Ret3),
      get_atr3(Ret3,Ret4),!,
      atom_concat(Ret4,Ret).
%dado algo como esto t= 'nodo/aol.derecho:tipo' devuelve A=tipo
get_type(T,Ret):-
      atom_chars(T,T2),
      reverse(T2,T3),
      get_atr3(T3,T4),!,
      reverse(T4,T5),
      atom_concat(T5,Ret).

%dado algo como esto t= 'nodo/aol.derecho:tipo' devuelve A=nodo/aol
get_class(T,Ret):-
         atom_chars(T,T2),
         get_atr2(T2,T3),
      atom_concat(T3,Ret).

%funcion auxiliar de get_atr
get_atr2(['.'|_], R):- R=[].
get_atr2([],R):- R=[].
get_atr2([X|Xs],R):-
      get_atr2(Xs, R1),
      append_([X], R1, R).
%funcion auxiliar de get_type
get_atr3([':'|_], R):- R=[].
```

```
get_atr3([],R):- R=[].
get_atr3([X|Xs],R):-
      get_atr3(Xs, R1),
      append_([X], R1, R).


%Trasforma el tipo jmsig a tipo de java, -- '[[I' -> 'int[][]' --
'Lheap/SLNode;' ->'SLNode'
jmsig_java_type(In, Out):-
      atom_chars(In, I1),
      jmsig_java_type2(I1,O1),
      atom_concat(O1,Out).

jmsig_java_type2([], R):- R=[].
jmsig_java_type2(['['|Xs], R):-
      jmsig_java_type2(Xs,R1),!,
      append_(R1,['[]'],R).
jmsig_java_type2(['I'|_], R):- R= ['int'].
jmsig_java_type2(['Z'|_], R):- R= ['boolean'].
jmsig_java_type2(['L'|Xs], O):-
      reverse(Xs,Ys),
      jmsig_java_type_object(Ys,Zs),!,
      reverse(Zs, O).
%Dado una lista con estos caracteres 'heap/SLNode;' devuelve 'SLNode'

jmsig_java_type_object([';'|Xs], R):- jmsig_java_type_object(Xs,R).
jmsig_java_type_object(['/'|_], R):- R=[].
jmsig_java_type_object([], R):- R=[].
jmsig_java_type_object([X|Xs], R):-
      jmsig_java_type_object(Xs,R1),
      append_([X],R1,R).



test(Type):-
      atom_chars(Type, Xs),
      num(Xs, '[', N),
      (N < 2 -> true; fail).
is_array_test(Type):-
      atom_chars(Type, Xs),
      num(Xs, '[', N),
      (N > 0 -> true; fail).
num([],_,A):-A=0.
num([X|Xs],Y,A):-
      num(Xs,Y,A1),
      (X==Y-> A is A1+1;A is A1).


%comprueba si el tipo es multi array y pone el tipo a los arrays
heredados(para heap_in)
put_type_multi_arrays_in(_,[]).
put_type_multi_arrays_in(Type,[X|Xs]):-
      (\+test(Type)->
         put_type_multi_arrays_in_aux(Type,X),
         put_type_multi_arrays_in(Type,Xs)
      ;
```

```
         true).

put_type_multi_arrays_in_aux(Type,X):-
      (\+X==null->
          (heap_array_in(I,Value,X,_)->
              quit_one_array_level(Type,New_type),
              retract(heap_array_in(I,_,_,_)),
              asserta(heap_array_in(I,Value,X,New_type)),
              Value=array(_,_,Elems),
              put_type_multi_arrays_in(New_type,Elems)
           ;
               (heap_soul(I,_,X,_,in)->
               quit_one_array_level(Type,New_type),
                  retract(heap_soul(I,_,X,_,_)),
                  asserta(heap_array_in(I,array(_,_,[]),X,New_type))

            ;
               true))

       ;
          true).

quit_one_array_level(Type,New_type):-
      atom_chars(Type,Xs),
      reverse(Xs,Xsr),
      quit_one_array_level_aux(Xsr,Ysr,2),
      reverse(Ysr,Ys),
      atom_concat(Ys,New_type).

quit_one_array_level_aux(Xs,Xs,0).
quit_one_array_level_aux([_|Xs],Ys,N):-
      M is N-1,
      quit_one_array_level_aux(Xs,Ys,M).


put_type_multi_arrays_out(_,[]).
put_type_multi_arrays_out(Type,[X|Xs]):-
      (\+test(Type)->
         put_type_multi_arrays_out_aux(Type,X),
         put_type_multi_arrays_out(Type,Xs)
      ;
         true).

put_type_multi_arrays_out_aux(Type,X):-
      (\+X==null->
          (heap_array_out(I,Value,X,_)->
              quit_one_array_level(Type,New_type),
              retract(heap_array_out(I,_,_,_)),
              asserta(heap_array_out(I,Value,X,New_type)),
              Value=array(_,_,Elems),
              put_type_multi_arrays_out(New_type,Elems)
           ;
              (heap_soul(I,_,X,_,out)->
               quit_one_array_level(Type,New_type),
                  retract(heap_soul(I,_,X,_,_)),
                  asserta(heap_array_out(I,array(_,_,[]),X,New_type))
```

```
            ;
               true))
       ;
          true).

%Devuelve  en función el tipo un valor por defecto
get_def_val('int', Ret):- Ret='0',!.
%get_def_val('I', Ret):- Ret='0',!.
get_def_val('boolean', Ret):- Ret='true',!.
%get_def_val('z', Ret):- Ret='true',!.
get_def_val('byte', Ret):- Ret='0',!.
get_def_val('double', Ret):- Ret='0',!.
get_def_val('float', Ret):- Ret='0',!.
get_def_val('long', Ret):- Ret='0',!.
get_def_val('short', Ret):- Ret='0',!.

get_def_val(_, Ret):- Ret='null',!.%si no es nada, yo (Fer) pondría
default, en vez de null


%esta funcion  dado X devuelve var_in_x, o var_out_x, segun corresponda
get_java_name(Var, Ret):-
      (dir(Dir)->true; Dir='unknow_dir'),
      concat('var_',Dir,R1),
      concat(R1,'_',R2),
      concat(R2, Var,Ret).

is_same_var(_,[]):-fail.
is_same_var(X,[Y|Ys]):-
      (X==Y -> true ; is_same_var(X,Ys)).


put_ref_list([]).
put_ref_list([X|Xs]):-
      %si X  es una Variable pongo ref(num)
      (\+nonvar(X)->
          new_ref(I),
         X=ref(I);true),

      %si X es una Ref(variable) pongo ref(num)
      functor(X,Y,_),
        (Y== ref->
             arg(1,X,Ref),
             (\+nonvar(Ref)-> new_ref(Ref);true);true),

      put_ref_list(Xs).
```

Archivo junit_utils.pl

```
/** <module>

This module implements a series of predicates to generate Java code,
@author Fernando Rodriguez, Jaime Saez
@license GPL
*/
:-module(junit_utils,[is_static/0,get_pack_class/3,contains/2,
contains_str/2,is_java_primitive/1]).
:-use_module(pet(pet_utils),[append_/3]).
:-use_module(junit_generator,[getJMSig/1]).
is_static:-
      getJMSig(JMSig),
      syntax_conversion:jmsig_to_msig(JMSig,MSig),
      jvm_eval:methodSignature_class(MSig,CN),
      jvm_eval:program_class(CN,Cl),
      jvm_eval:class_method(Cl,MSig,Method),
      (jvml:method_isStatic(Method)-> true; fail).

%dado algo como esto 'aa/bb/cc/dd' devuelve  Claas='dd'
      Pack='aa/bb/cc'
get_pack_class(['/'|Xs],C,P):- C=[],P=Xs,!.
get_pack_class([],C,P):- C=[], P=['Error no package in signature'],!.
get_pack_class([X|Xs],C,P):-
      get_pack_class(Xs,C1,P),
      append_([X],C1,C).

%dice si una  lista contine un valor
contains([],_):-fail.
contains([X|Xs],Y):-
      (X==Y-> true; contains(Xs,Y)).

%dice si un string contiene una cadena
contains_str(Str,In):-
      atom_chars(Str,Str2),
      atom_chars(In, In2),
      sublist(In2,Str2).


prefix([],_).
prefix([X|Xs], [X|Ys]):- prefix(Xs,Ys).

suffix(Xs,Xs):-!.
suffix(Xs, [_|Ys]):- suffix(Xs,Ys).
sublist(Xs,Ys):-prefix(Xs,Gs), suffix(Gs,Ys).
is_java_primitive('int'):-true,!.
is_java_primitive('boolean'):-true,!.
is_primitive(_):-fail,!.
```