Aplicación de JML a las prácticas de programación con Java

Antonio David Gómez Morillo, Antonio Menchén Peñuela
Departamento de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla
Avda. Reina Mercedes s/n
adavidg@supercable.es , menchen@lsi.us.es

Resumen

JML (Java Modeling Language), es una herramienta desarrollada para Java que permite especificaciones pre-post, así como expresiones muy potentes para colocar asertos en cualquier línea de programa. Con escasamente 5 años, JML aún resultará desconocido para la mayoría de los que nos dedicamos a la enseñanza universitaria. Este trabajo pretende ser uno de los vehículos propulsores de esta herramienta. En el mismo se introducen los elementos básicos del lenguaje JML, y se desarrollan algunos ejemplos. Pensamos que la herramienta es cómoda de instalar y de usar, pudiendo mejorar claramente el código de los programas que escriben los alumnos en las clases de prácticas de las asignaturas de programación.

1. Palabras Clave

Especificación pre-post, interfaces, herencia, Tipos Abstractos de Datos, eficiencia.

2. Introducción

Incluso si nos limitamos al contexto de una programación muy elemental (por ejemplo, algoritmos de ordenación o de búsqueda utilizando arrays de objetos), el alumno suele cometer muchos errores al escribir su código en Java. Estos errores pueden provenir de que los algoritmos no están verificados, o que la trascripción a Java no se ha realizado correctamente o, más a menudo, de despistes propios de seres humanos. De manera que los programas compilan pero o bien los resultados no son los esperados, o se eleva una excepción sin conocerse la causa, o los programas se quedan "colgados". En muchos casos, podremos acudir al depurador para poder detectar las líneas de código donde se producen los errores. Sin embargo, si el algoritmo no es correcto un depurador difícilmente podrá ayudarnos. Un depurador quizá pueda servirnos para darnos cuenta, por ejemplo, que de la ejecución de tal instrucción alternativa no se obtiene el caso esperado, pero esta información suele ser incompleta. En los casos en que se elevan excepciones o se queda colgado el programa, un depurador puede ayudarnos bastante, sin embargo, por experiencia sabemos que arreglamos el problema en un punto y aparece un nuevo error en otro sitio. En tal caso, la técnica más utilizada es el parcheo y la colocación de banderas, imprimiendo, por ejemplo, el valor local de ciertos parámetros. Al final de un arduo proceso el alumno respira tranquilo ¡el programa funciona! Más tarde, cuando se le realiza una prueba no prevista, saltan los errores de nuevo. JML (eso esperamos), podrá acabar con estos métodos pedestres.

En la actualidad se está trabajando en otras herramientas como ESC/Java, ESC/Java2, JCon-

tract y Jass. JML ha asumido alguna de las características de las citadas herramientas. Y en algunos medios es muy comparado con ESC/Java, que recibe sus siglas de "Extended Static Checking for Java", pero, a diferencia de JML, tan solo admite especificaciones del tipo `Lightweight`[1]. Su sintaxis está muy próxima a la de JML, aunque con menos extensiones al lenguaje Java. Además de lo citado, ESC/Java realiza un análisis estático, es decir, que verifica si nuestro código puede lanzar excepciones como `ArrayOutOfBoundException`, y nos informa mediante un mensaje de error. Los errores más frecuentes que detecta según sus autores son: referencias nulas a objetos, desbordamientos de arrays y castings incorrectos o falta de los mismos.

En lo que sigue, introducimos el lenguaje JML y presentamos varios ejemplos, que permiten entender el papel determinante que puede realizar esta herramienta en el desarrollo futuro de prácticas de programación con Java.

3. Extensiones al lenguaje Java

Una de las primeras cuestiones que nos llaman la atención cuando comenzamos a trabajar con JML es la cantidad de extensiones que se han ido añadiendo al lenguaje de programación Java.

Para comenzar, nuestras especificaciones serán colocadas a modo de anotaciones o comentarios, muy similares a los de Javadoc, de la siguiente forma: `/*@ especificación @*/`, o bien si lo preferimos `//@ especificación`.

Una vez que tenemos claro dónde debemos insertar nuestras precondiciones y postcondiciones, el paso siguiente es descubrir que en JML disponemos de dos grandes tipos de especificaciones. Los autores de JML, les han dado los nombres de `Lightweight` y `Heaveweight`, haciendo distinción entre una especificación "ligera" (incompleta) y otra "robusta" (completa). Por nuestra parte, hemos comprobado que una especificación aplicada a Tipos Abstractos de Datos ha de ser lo más completa posible para evitar olvidos desagradables que, más tarde, pueden pagarse caro.

Para comprender mejor la distinción entre los dos comportamientos, obsérvese la siguiente especificación:

```
//@ requires dato > 0;
public int metodo (int dato);

-----------------------------------

/*@ public behavior
requires dato >0;
diverges \not_specified;
assignable \not_specified;
accessible \not_specified;
when \not_specified;
working_space \not_specified;
duration \not_specified;
ensures \not_specified;
signals \not_specified;
@*/
public int metodo (int dato);
```

Especificación 1.

Tal y como puede observarse en el lado superior disponemos de una especificación `lightweight` y en el lado inferior una `heaveweight`. Ambas hacen lo mismo. La diferencia fundamental está en que la primera requiere menos código, y la segunda es más fácil de modificar a posteriori.

Comparando la parte superior con la inferior, diríamos que mientras que en la primera ahorramos tiempo y código, en la segunda el comportamiento del método queda mucho mejor expuesto al programador. En dicha parte superior, lo único que podemos asegurar es que el parámetro de entrada es estrictamente positivo, pero el programador no sabrá si dicho método lanza o no una excepción, ni otros muchos comportamientos que quedan determinados en la especificación de la parte inferior.

Así, en la especificación `Heaveweight`, no solo encontramos la cláusula `requires`, sino que disponemos de toda una serie de cláusulas propias de JML, con las que podremos, por ejemplo, decirle al programador que dicho método lanza tal o cual excepción especificada en la cláusula `signals` (postcondición de excepción).

Sin embargo, existen diferencias más importantes que el número de cláusulas JML que usemos. Ambas especificaciones están claramente diferenciadas desde la primera línea de código. Así, la sentencia `public behavior` es la cabecera de la especificación. Además, disponemos de otras cabeceras, como `normal_behavior`, que es exactamente igual que la anterior, solo que conlleva implícita la sentencia: `signals (ja-`

[1] Ver siguiente apartado.

va.lang.Exception) false;. Por lo que dicha cabecera es apropiada cuando el método en cuestión **no** lanza ninguna excepción. También, en contraposición a ésta última disponemos de la cabecera exceptional_behavior, en la que supondremos la postcondición : ensures false;. Y en lugar de la postcondición normal, utilizaremos la postcondición de excepción signals.

Es interesante saber, por otro lado, que en una especificación lightweight cuando omitimos por ejemplo la precondición (requires) el compilador no le asocia ningún valor por defecto, es decir, que no toma ni verdadero, ni falso. Simplemente se le transmite al compilador la extensión \not_specified.

Una vez visto cómo debe comenzar una especificación, veamos cuales son los añadidos más útiles que podemos encontrar en JML.

Por un lado tenemos los cuantificadores. Disponemos de todos los más usuales como son: \forall, \exists, \min, \max, \sum, \product, \num_of. También veremos más adelante ejemplos de especificaciones que usan algunos de los cuantificadores relacionados. También serán palabras reservadas en JML: \result, que representa el valor devuelto por una función, y \old(expresión), que representa el valor de una expresión antes de ser invocado el procedimiento o función. A continuación mostramos un ejemplo de especificación donde puede verse la utilización de algunas de estas palabras reservadas:

```
/*@ ensures(
        (\forall int j; (j>=1) && j<i);
        lista.get(j).equals(\old(lista.get(j)))
        );
*@/
```

Especificación 2.

El significado de la misma es que el programador está obligado a asegurar al cliente que después de llamar a la función o procedimiento de que se trate, ninguno de los objetos almacenados en la lista, desde la posición 1 a la i, ha cambiado de estado.

La mayoría de estos cuantificadores funcionan todos de igual forma, así, para el caso de \forall dispondremos de tres campos separados por punto y coma. En el primero se hace la declaración de la variable que determina el dominio, en el segundo el dominio de dicha variable que debe ser un booleano en su expresión final y por último otra expresión booleana que sería el aserto que se debe cumplir.

Así mismo, caben en JML comentarios del estilo a: (* los elementos de la lista no deben cambiar *), que puede añadirse con && al código mostrado en la última especificación. De esta forma el cliente podrá aclarar al programador, más si cabe, el contenido de dichas líneas.

Otra cuestión interesante, es la posibilidad de enriquecer nuestras especificaciones con métodos diseñados por nosotros mismos o por otros implementadores (incluidos los de JDK[2]). JML permite incluir cualquier método siempre que lo declaremos pure, de manera que su invocación no puede provocar efectos laterales. La forma de declararlos es colocar la línea: /*@ pure */ delante del nombre del método:

```
public String /*@ pure @*/ getNombre()
{
        //código
}
```

Especificación 3.

Por lo demás, y aunque ya se ha hecho uso del concepto, debemos tener en cuenta que para la precondición en JML utilizaremos la palabra reservada requires, y para la poscondición ensures, aunque debemos aclarar que contaremos con dos tipos de poscondiciones. Por un lado tenemos lo que podríamos llamar poscondición normal, y por otro la poscondición de excepción, en esta última se utiliza la palabra reservada signals, como se muestra en algún ejemplo más adelante.

Hasta aquí las cuestiones más elementales para comenzar a trabajar con JML. Conceptos más avanzados son los de atributos y métodos model. A grosso modo podríamos definirlos como atributos y métodos cuya existencia está estrechamente relacionada con la especificación, no existiendo desde el punto de vista del intérprete de Java.

Aquí cabría destacar que el aspecto más relevante, en el caso de un método declarado como model, es que puede contener código, ¡aún cuando se encuentre dentro de una interfaz!

[2] La versión de JDK debe ser igual o superior a la versión 1.4. Las pruebas han sido desarrolladas con la versión 1.4.2_02

Como ejemplo de atributo `model`, a continuación se declara una cadena de caracteres (`String`), a la que se le exige que no puede ser nula, la característica principal que nos interesa es que tan solo existirá para propósitos de especificación.

```
//@ public model non_null String nombre;
```

Especificación 4.

Otra cuestión importante es que al igual que existe la herencia en Java, también podremos hacer que se hereden las especificaciones. Para ello se hace necesario informar a la especificación de que existe otra en una superclase o interfaz situada en un nivel superior en la jerarquía de herencia. Esto lo obtendremos en JML con la palabra reservada `also` al comienzo de la especificación.

Para concluir, sirva el siguiente ejemplo completo, que forma parte de la especificación del TAD Lista al que nos hemos ya referido.

Supóngase que disponemos de una jerarquía de interfaces similar a la siguiente, y que al mismo tiempo deseamos especificar el método `void añadir(Object x)`.

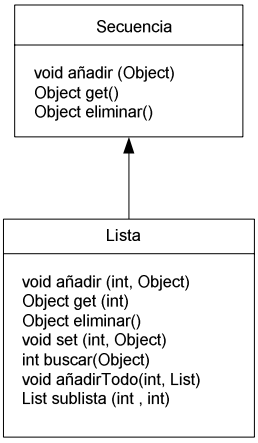

Figura 1.

La semántica del método podría ser la que sigue: “inserta el objeto `x`, en el lugar apropiado de la secuencia”. Ahora bien, el comportamiento semántico de una lista no es el mismo que el de una pila, luego dicho método habrá de ser refinado en la interfaz `Lista`.

```
/*@
@public normal_behavior
@requires (x!=null);
@ensures (
@(*El objeto se añade en el lugar
adecuado*)
@&&
@seq.tamaño()==\old(seq.tamaño()+1)
@);
@*/
public void añadir(Object x);
```

Especificación 5.

La especificación anterior ilustra lo ya comentado, se trata de un comportamiento genérico para todas las secuencias, sin especificar ninguna en particular.

Una mejor aproximación a la secuencia lista, debería contener una especificación más exhaustiva. Tal podría ser la que sigue, en la que se ha colocado la palabra `also`, al comienzo de la misma para denotar la herencia.

```
/*@
@also
@public normal_behavior
@ensures
@(
@(\forall int i; (i>=1)&& (i<lista.tamaño());
@lista.get(i).equals(\old(lista.get(i))))
@&&
@lista.get(lista.tamaño()).equals(x)
@);
@*/
void añadir(Object x);
```

Especificación 6.

4. Aplicación de JML a Tipos Abstractos de Datos

4.1. Justificación

Durante las prácticas de programación con Java, se pide a menudo al alumno que implemente un TAD que se comporte de una forma determinada. A priori este no sabrá si su implementación se ajusta a la especificación facilitada, a lo sumo puede garantizar una compilación correcta de lo que se le está pidiendo. Pero, ¿no sería mejor disponer de un lenguaje que le informe claramente de lo que tiene que hacer, permitiéndole comprobar si las operaciones que está implementando son correctas? Esto puede suponer una gran ventaja, ya que el alumno no solo es consciente de que me-

diante la especificación, tiene "en manos" el comportamiento, sino que además puede someter la implementación a la especificación desde la primera compilación, comprobando si lo que ha estado realizando se ajusta a las exigencias de los profesores de la asignatura (en el papel de clientes).

Para que todo lo dicho surta el efecto esperado, lo primero que hay que asegurarse es de que la especificación sea lo más completa posible, es decir, el profesor debe entregar al alumno una especificación que reúna todos los requisitos de comportamiento de los métodos del TAD o algoritmo. Por su parte, el alumno deberá de aprender el lenguaje JML, pero esto le resultará bastante fácil a aquéllos que estén familiarizados con el lenguaje Java, ya que las extensiones al lenguaje resultan cómodas y fáciles de utilizar.

Resumiendo lo dicho, diríamos que se trata de que el alumno conozca exactamente lo que debe hacer. Seguramente este escogerá la implementación que le resulte más familiar, pero, en cualquier caso, dicha implementación cumplirá con los requisitos de especificación. **La ruptura de la ambigüedad es una constante con JML**.

4.2. Compilación y ejecución

Otra cuestión interesante a tener en cuenta son los tiempos de ejecución. En principio, JML nos proporciona un compilador (jmlc) y una máquina virtual (jmlrac). El compilador funciona como el que incluye JDK, es decir, crea los archivos de extensión `class` que serán interpretados, en su caso, por la máquina virtual. Sin embargo, también podremos interpretar nuestros programas con la máquina virtual incluida en JDK. Para ello basta con añadirle la librería `jmlruntime.jar`, que encontraremos en el directorio `bin` de JML, al `CLASSPATH` del entorno que estemos utilizando.

Por último, referirnos a las posibles extensiones de los archivos empleados en JML, según podemos ver [1], estas extensiones son: '`.jml`', '`.jml-refined`','`.java-refined`','`.refines-jml`', '`.refined-java`'. Y cómo no, además de las citadas la extensión propia de los archivos de java '`.java`', es también admitida.

La razón de los archivos con extensión '`refined`', es bien sencilla. JML ofrece la oportunidad de separar el cuerpo del método de su especificación. Si disponemos de una clase 'Lista' en un archivo '`Lista.java`', en el que se encuentra una posible implementación, nosotros podemos disponer de otro archivo '`Lista.refined-java`', en el que sólo insertaremos su especificación seguida de la cabecera del método. Este comportamiento sería muy similar a especificar una interfaz de Java.

5. Aplicación a un ejemplo concreto: el TAD Lista

Nada mejor que un ejemplo concreto para ver cómo trabaja JML en la práctica. En éste caso particular recurrimos a una lista. Con el propósito de aprovechar al 100% las características de Java, vamos a suponer que disponemos de una interfaz que llamaremos Lista. En dicha interfaz colocamos los métodos que el programador debe construir, es decir, la cabecera de los métodos cuyos cuerpos irán en la clase correspondiente.

Como ejemplo tomaremos el método añadir, cuyo perfil: `void añadir (int i, Object x)`, con el que añadiremos a la lista el objeto x en la posición i-ésima de la misma, desplazando el objeto que estaba en dicha posición hacia la "derecha", es decir, que pasaría a ocupar la posición (i+1), procediéndose igualmente con el resto de elementos hasta el final de la lista.

En la siguiente figura se muestra esta idea más claramente:

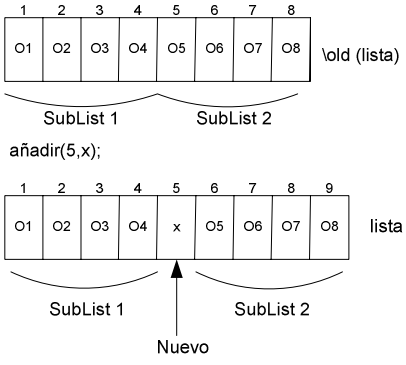

Figura 2.

Teniendo en cuenta esto, una especificación para la operación indicada podría ser:

```
/*@
@public normal_behavior
@requires ((i>=1 && i<=(tamaño()+1)) && x!=null );
@ensures
@       (
@(\forall int j; (j>=1)&& (j<i);lista.get(j).equals(\old(lista.get(j))))
@&&
@(\forall int k; (k>i) && (k<=size());lista.get(k).equals(\old(lista.get(k-
1))))
@&&
@(lista.get(i)).equals(x)
@&&
@(lista.tamaño()== \old(lista.tamaño()+1))
@       );
@also
@public exceptional_behavior
@requires (i<1 || i> (lista.tamaño()+1));
@signals(IndexOutOfBoundsException)true;
@*/
void añadir(int i, Object x) throws IndexOutOfBoundsException;
```

Especificación 7.

Tal y como se puede observar, el programador dispondrá de una información sumamente valiosa para él, siendo el lenguaje utilizado muy cercano al de programación, y en el que puede leer sin dificultades los comportamientos del método que ha de desarrollar.

5.1. Traducir la especificación.

Lo primero que encontramos en la anterior especificación, es la sentencia `public normal_behavior` y más abajo en el mismo código `public excepcional_behavior`. Inmediatamente el programador podrá deducir que la función lanzará una excepción en unas condiciones determinadas, y además, cuáles son dichas condiciones así como de qué excepción se trata.

Cuando el programador lea la precondición del método (`requires`) sabrá que debe cumplir que el primer parámetro (i) debe encontrarse dentro de los límites establecidos en la especificación, dado que la primera posición de una lista es 1, el nuevo objeto estará entre 1 y el tamaño de la lista más uno (podemos colocarlo al final de la misma).

Pero la información no termina aquí.

Lo siguiente es leer la postcondición, (`ensures`). El hecho de que JML sea una extensión de Java, facilita la lectura de lo que hace el operador universal (`\forall`), que como es usual significa "para todo…".

Para terminar, el caso en el que se eleva la excepción está también determinado claramente en el código anterior.

Así pues, no sólo estamos ante un comunicado estático de lo que un programador debe realizar, sino además ante un código dinámico que *servirá de ayuda, antes, durante y después de la programación*. **Con JML todo está en la especificación: Comportamiento y posterior verificación de la codificación.**

¿Cuánto tiempo hemos ahorrado en pruebas de compilación?

Con la especificación a priori, tal y como la realiza JML, se pueden suprimir muchos inconvenientes que aparecen en la programación real. Por un lado sabemos lo que tenemos que hacer. Por otro la prueba va a consistir en someter lo que hemos hecho al código que nos han pasado durante la especificación. Si la especificación es correcta y el código la pasa, la probabilidad de haber cometido un error en tiempos de diseño será mucho menor.

6. Aplicación a problemas de algoritmia

En Algoritmia tenemos siempre planteadas dos cuestiones fundamentales: resolver algorítmicamente un problema y obtener soluciones lo más eficientes posible. Con JML, el cliente podrá indicarle al programador qué solución debe encontrar el algoritmo, y, además, podría acotar el tiempo de ejecución (`\duration`) y el espacio en memoria de programa (`\working_space`), de los algoritmos (aunque estas características están aún en fase de desarrollo).

Un ejemplo muy sencillo nos lo ofrece el problema del intercambio de dos componentes de un array `a` (supongamos de enteros), cuya especificación en JML podría ser:

```
/*@ requires (i >= 0) && (j >= 0) &&
             (i!=j) &&
             (a!=null)&&
             (a.length>i)&&(a.length>j);

   ensures (* intercambia los valores
                de a[i] y a[j] *) &&
             (a[i] == \old(a[j])) &&
             (a[j] == \old(a[i]));
      signals (IndexOutOfBoundsException)
      (0>i||0>j||a.length<=i||a.length<=j);
      signals (NullPointerException)
      (a==null);
      signals (IndicesIgualesExcepcion)
      (i==j);
  */
```

Especificación 8.

De manera que a la vista del siguiente código para la operación de intercambio:

```
a[i]+=a[j];
a[j]=a[i]-a[j];
a[i]-=a[j];
```

¿Quien podría afirmar sin dudar que los valores de `a[i]` y `a[j]` se intercambian?

Por otro lado, el primer algoritmo que se nos ocurre a todos es utilizar una variable temporal que almacene el valor de una componente para que no se pierda en el intercambio:

```
int temp=a[i];
  a[i]=a[j];
  a[j]=temp;
```

Pudiendo comprobarse que las dos soluciones son válidas desde el punto de vista de la especificación, sólo que la primera es más eficiente al no requerir memoria auxiliar.

7. Trabajos futuros

En el presente, JML no dispone de ningún entorno "amigable" en el que podemos trabajar. Bien es cierto que podemos utilizarlo con herramientas como Eclipse, EditPlus, etc., que nos facilitarán la tarea de compilación y ejecución de nuestras pruebas.

En la actualidad sólo disponemos de ficheros *.bat que ejecutados en MS-DOS nos presentan en pantalla los resultados que esperamos (aunque en las últimas versiones de la herramienta, se ha mejorado el aspecto con ventanas tipo de las que ofrecen los paquetes `java.swing` y `java.awt`). Una cuestión interesante y que tenemos el propósito de desarrollar como aportación al proyecto de Leavens [5], es la construcción de un IDE para JML.

En el futuro confiamos que JML disponga de entornos que hagan de esta herramienta un elemento de ayuda a la programación tanto a nivel docente como empresarial.

Son muchas las razones que podemos aportar para apuntar la necesidad de un IDE, pero como la práctica es la que nos está abriendo caminos, después de haber especificado el TAD Lista, y de haber realizado muchas pruebas de compilación y ejecución, hemos observado que cuando tenemos un error sobre todo sintáctico en JML su localización y la naturaleza del mismo, suele ser, hasta el presente, una actividad algo tediosa.

8. Conclusiones

No nos cabe la menor duda de que JML aporta mucho. Tanto al cliente como al programador. Desde un punto de vista didáctico, cuando el alumno no hace exactamente lo que le pide, el profesor podrá decirle que no ha hecho lo que estaba especificado. Por lo ya comentado, JML aporta desde nuestro punto de vista:

- Precisión, especificaciones de comportamiento no ambiguas.
- Ahorro de tiempo en la codificación.
- Ahorro de tiempo en las pruebas.
- Curva de aprendizaje suave. (Se trata de código Java.)
- Documentación de la especificación y del código Java (con "jmldoc").
- La posibilidad de que el cliente indique al programador la eficiencia de los algoritmos que este debe implementar.

Por último, debemos tener en cuenta que JML está todavía en desarrollo y que puede sufrir variaciones notables, tal y como nos han comentado sus mismos autores.

9. El nuevo Sistema de Transferencia de Créditos Europeo (ECTS)

Entre una de las consecuencias del replanteamiento metodológico que conlleva la implantación del ECTS, está la de que **el volumen de trabajo del estudiante deberá ser estimado y revisado periódicamente** ([6]). En este sentido, debido a las

características de JML apuntadas en el presente trabajo, pensamos que se trata de una herramienta que favorece el acercamiento entre el profesor y el alumno, ya que el segundo puede entender con más precisión los requerimientos del primero. De esta forma, uno de sus efectos podría ser que el **aprendizaje por el aprendizaje** [6] del alumno se vea favorecido al permitir un aumento de las horas de tutoría o de seminarios, en detrimento de las tradicionales clases presenciales, efectos que se esperan conseguir al implantarse el nuevo sistema. Por lo que la efectiva aplicación de JML en el desarrollo de las prácticas de programación, permitiría que el profesor lleve un control de las prácticas que fueran realizando sus alumnos, permitiendo esto que con el tiempo desaparezca la defensa de una única práctica obligatoria en el periodo de exámenes finales.

Agradecimientos

A Yoonsik Cheon, colaborador y desarrollador en el proyecto JML que tan gentilmente nos atendió por correo electrónico.

Referencias

[1] Gary T. Leavens, Eric Poll, Curtis Clifton, Yoonsik Cheon, Clyde Rudy, *JML Referente Manual (revisión 1.65)* 28.01.2004 "Se trata de un buen manual, aunque en la actulidad, (o al menos la versión empleada) está incompleta."

[2] Gary T. Leavens, Albert L. Barker and Clyde Rudy. *Preliminary Design of JML* (Septiembre 2003) "Más específico que el anterior, pero require conocimientos previos, al menos básicos de JML".

[3] Yoonsik Cheon, *A Runtime Assertion Checker for the Java Modeling Language* (Abril 2003) "Se trata del desarrollo teórico de JML, útil para futuras ampliaciones"

[4] Gary T. Leavens and Yoonsik Cheon *Design by Contract with JML* (6.10.2003). "El mejor sitio donde comenzar a crecer con JML, recomendado para principiantes.".

[5] El mejor sitio donde buscar documentación es la página del autor: http://www.jmlspecs.org

[6] Juan José Iglesia Rodríguez. Vicerrector de la universidad de Sevilla. Jornada de trabajo para la experiencia pilota para la implantación de ECTS. http://www.us.es/include/frameador2.php?url=/us/temasuniv/espacio-euro